Automated Methods for Creating Diversity in Computer Systems

by

Elena Gabriela Barrantes Sliesarieva

Bachiller, Universidad de Costa Rica, 1990
M.S., Florida Atlantic University, 1995

DISSERTATION

Submitted in Partial Fulfillent of the
Requirements of the Degree of

Doctor of Philosophy
Computer Science

The University of New Mexico

Albuquerque, New Mexico

June, 2005

Acknowledgments

I would like to thank my advisor, Professor Stephanie Forrest for her continuous support of this project. My thanks also to Dennis Chao and Christina Warrender who kept pushing me to write, read very rough drafts and helped give form to this document. Finally, a thank-you to the adaptive group members, who were always willing to hear a practice talk or discuss the implications of an obscure implementation point.

Automated Methods for Creating Diversity in Computer Systems

by

Elena Gabriela Barrantes Sliesarieva

ABSTRACT OF DISSERTATION

Submitted in Partial Fulfillent of the

Requirements of the Degree of

Doctor of Philosophy

Computer Science

The University of New Mexico

Albuquerque, New Mexico

June, 2005

Automated Methods for Creating Diversity in Computer Systems

by

Elena Gabriela Barrantes Sliesarieva

Bachiller, Universidad de Costa Rica, 1990
M.S., Florida Atlantic University, 1995
Ph.D., Computer Science, University of New Mexico, 2005

Abstract

The pervasive homogeneity of computer systems attached to the Internet, combined with the ease of attacking multiple identical systems once one machine is compromised, represents a serious security threat. A possible response to this situation can be found using biological diversity as inspiration. In nature, diversity provides a defense against unpredictable threats by maximizing the probability that some individuals will survive and replenish the population with a defense against that particular threat. Diversity in computer systems could confer security benefits by protecting against attacks that rely on known regularities.

A diversity defense can render a standard attack ineffective or slow it down, depending on its placement and implementation. Diminishing the uniformity in existing systems is, however, a non-trivial task, as standardization must be maintained at many interface points in any given system. This dissertation intends to assess the costs and benefits of adding

diversity to existing computer systems by implementing diversity at different levels. Diversification in computer systems can be accomplished at the interface or the implementation level. In general, an interface diversification changes function labels making them unique to a given system. In contrast, an implementation diversification modifies function behavior to prevent locking into idiosyncratic states. Three techniques to introduce automated diversity in existing systems are presented: one at the interface and two at the implementation levels. Their effectiveness at stopping or slowing down attacks is studied.

The first diversity scheme presented is an interface diversification: a machine language randomization, named Randomized Instruction Set Emulation (RISE). RISE is intended as a protection against the threat of code-injection attacks, which insert malicious machine-language code into programs. Code-injection attacks constitute one of the most prevalent threats on current networks, and its most famous examples are the so-called *buffer overflow* attacks. RISE protects against all code-injection attacks regardless of their point of entry by creating a unique machine code per process. The current RISE implementation runs over an emulator, and maps all executable bytes of the process to a random mask. When injected code attempts to execute, its code is also mapped to the mask, but given that it was not correctly encoded, it is 'decoded' to random bytes. Though the attack will not execute as intended, there is a small probability that the attack will manage to execute some random instructions. This work also offers an analysis of the risks associated with the execution of random instructions.

Many Denial of Service (DoS) attacks exploit a system's implementation rather than its interface. Two approaches to diversify an implementation are explored. The first approach randomizes internal protocol parameters within acceptable ranges. It is tested against an attack targeting one of the TCP congestion control parameters. The diversification achieves the objective of keeping a portion of the hosts in the attacked network operating at larger bandwidths than if they were all using the same standard parameter values.

The second implementation diversification targets Denial of Service attacks by re-

source exhaustion. The diversity solution used creates a unique filter per host that, for a given attack traffic pattern, passes most of the legitimate traffic and blocks attacking requests. Filters are created using Genetic Programming on different attack patterns. Current results suggest that it could be effective when used as a front-end for resource managers that are non-preemptive in nature.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The current number of computers with Internet access is on the order of hundreds of millions [27]. It is likely that each one of them has been subjected to at least a probe testing for security weaknesses, or to an actual attack. For almost any vulnerability found in an application, it is relatively easy to scan and find several hundred systems that are susceptible to the same flaw. This is not an abstract possibility but an ongoing problem over at least ten years and one that is increasing [20].

One of the factors enabling this proliferation of attacks is the uniformity of computer systems at most levels: architecture, operating systems, languages, applications and so on. Because of the combination of pressure for compatibility and interoperability, and the reduction in the number of software companies in the market, program diversity in computer systems has steadily decreased. At the same time, the decrease in the costs of computer equipment and network access, has made the Internet accessible to an ever increasing segment of the population, which naturally has expanded the number of malicious users.

In the current network environment, specific attacks are usually met with a swift engineered response, in the form of a patch or a filter. Although certainly necessary, this is a reactive approach, that can only respond to attacks that have already been identified. The

challenge is to create defenses that can block undiscovered attack avenues. For a partial solution to this problem, in the present work, I draw inspiration from diversity in nature.

Organisms must deal with evolving pathogens, varying environmental conditions and so on. Just as in computer systems, natural systems have specific defenses against known dangers, but also have a more general mechanism in place, which provides a defense against new forms of attack. This mechanism is diversity, which increases the probability that at least some members of the population at any given level (e.g. cells, animals, herds) would be able to withstand the attack because of morphological, chemical or metabolic differences.

In computer systems, more diversity would mean that automated scanning and attack propagation would be more difficult and not as attractive for amateur criminals. Most attacks are very fragile and work only under very constrained conditions. The proliferation of unmodified script attacks is a testimony to the homogeneity of many computers. Using the natural diversity metaphor, the goal is to apply automated procedures to diversify the internal environment of a system to provide at least some defense against unknown threats.

Diversity can also be seen in the spirit of installing locks and closing the door at night: the automated diversification could provide enough disturbance in the environment so that vanilla attacks are rendered ineffective. This will force attackers to deploy concentrated attacks on each host, elevating the costs enough to discourage all but the most determined attackers. Clearly, for a diversification to have applicability, the external standard interfaces must be preserved, so interoperability would not be affected

An important aspect of diversity is that it constitutes a population level defense. The expected result of an attack on a diverse group is that some members of the population will resist, but some will fail. Therefore, this class of defenses are viewed with suspicion because they cannot offer any guarantees to a particular individual. However, two considerations must be made: (a) By definition, the user cannot know if she will be the survivor

against a new threat, and some probability of survival is better than a 100% probability of failure; and (b) In the long run, even those individuals that were vulnerable during the first wave of the attack, benefit from the engineered solutions that are developed from the surviving members. Furthermore, in some environments, the shutdown of a number of components only causes a slowdown of the whole system, as opposed to a complete stop in operations if they were completely homogeneous.

The cost of a diversification can be relatively high, given the necessity of maintaining interoperability. Although it would be easy to protect a system completely making all of its interfaces different, it would cut entirely its contact with the world, which is unacceptable. Therefore, mechanisms to 'translate' interfaces and to differentiate among internal and external agents have to be put in place, which could add considerable overhead. A diversification should be undertaken only if the benefits clearly exceed the costs, although the quantification of both is difficult to make.

In a more practical vein, the introduction of automated diversity could be done at either the interface or the implementation [31]. Interface adaptations work on the surface of the code being protected. They modify the layout or access controls to interfaces, without changing the implementation of the core code that the interfaces are giving access to. Implementation adaptations, do not alter the interface. Instead they modify implementation of portions of the system to make them resistant to attacks. The case for diversity could be made stronger by including both categories in the diversification of a system.

There is also the question of the level at which the system is to be protected: network, host, application and so on. For a population defined as processes inside a host,the survival of some of them because of the diversification still, still leaves the higher-level user with the ability of function. This makes host-level diversity defenses particularly attractive.

Instead of attempting to do an exhaustive, and possibly infeasible, exploration of all interface and implementation diversification for a given operating system, I chose to explore

three diversifications that sample areas different enough to make broad predictions about the safety and effectiveness of this approach in computer security. This work describes the implementation and evaluation of those three schemes. These diversifications also target two important classes of threats: code injection and and Denial of Service.

The background and related work for the unifying theme, *automated diversity for security*, is explored in Chapter 2. This chapter charts the story of the use of diversity in computer science, from the fault-tolerance field to security. The use of diversity for security is not a new idea, and part of Chapter 2 is dedicated to present the reasearch on the original concept, and to describe work done either previously or in parallel with the present one. After a long period of dormancy, the diversity approach has bloomed in the last two years, and I recount some of the recent development in this chapter. In the case of approaches close to the ones described in this work, they are presented in the appropriate chapters.

In Chapter 3, I describe the design and implementation of an instruction set randomization tool for IA32, which builds randomized instruction set support into a version of the Valgrind IA32-to-IA32 binary translator [90, 81]. Instruction set randomization is a host-level, interface randomization against the ubiquitous threat of injected code. The tool is named RISE (Randomized Instruction Set Emulation). RISE is composed by a randomizing loader for Valgrind and an interceptor for the fetch cycle. The loader scrambles code sequences loaded into emulator memory from the local disk using a hidden random key. Then, Valgrind's emulated instruction fetch cycle is intercepted and fetched instructions are first unscrambled, yielding the unaltered IA32 machine code sequences of the protected application. The RISE design makes few demands on the supporting emulator and could be easily ported to any binary-to-binary translator for which source code is available. The effectiveness and efficiency of RISE is discussed at the end of Chapter 3. It reports empirical tests of the prototype and confirms that RISE successfully disrupts a range of actual code injection attacks against otherwise vulnerable applications. In addition, it comments

on performance issues.

RISE is effective in the context of its threat model, the injection of binary code. However, this is just a subset of the memory corruption family of attacks, and memory corruption can be also exploited using data-only attacks. By adding some relatively simple interface randomizations in the data memory layout the protection offered by RISE can be significantly expanded. Chapter 3 explains the mechanisms used for these additional defenses and explores their contributions and limitations.

A basic property of the RISE defense mechanism is that if an attack manages to inject code by any means, essentially random machine instructions will be executed. Even if the intended attack is thwarted, there is always a possibility that random bits could create valid instructions and instruction sequences. The execution of this random bytes could conceivable damage other program structures or mask the existence of an attempted attack. Chapter 4 investigates the likely effects of such an execution in several different execution contexts. I present a general model of failure from the execution of randomized code. Experimental and theoretical models of risk evaluation are built on top of the general failure model. Experimental results are reported and theoretical analyses are given for two different architectures.

The empirical data suggests that the majority of random code sequences will produce an address fault or illegal instruction quickly, causing the program to abort. Most of the remaining cases throw the program into a loop, effectively stopping the attack. The theoretical model is in agreement with the experimental data, but also suggests that there is always a non-zero probability that the random execution could not be detected. In any case, the code originally intended by the attacker is never executed.

Interface randomizations could be very effective, but it is also important to explore implementation diversity defenses for attacks that do not require interface knowledge but exploit implementation errors, misinterpretations or even features. Denial of Service (DoS)

attacks that target network protocol vulnerabilities serve as the defense targets for the implementation diversifications explored in this thesis. Chapter 5 presents an instance of a straightforward way of creating implementation changes in an automated way: the randomization of protocol parameters. Specifically, it randomizes some congestion control state variables in TCP to create a network-level defense against the *shrew* attack [65]. This attack is one of a class that exploits emergency states in communication protocols. The purpose of these attacks is to trick the protocol into determining that an emergency situation has arisen (for example, congestion in the network), which usually causes a fall back into a slow mode. These attacks use the well-known, mostly arbitrary, parameters that implementations of the protocol use to classify environmental conditions, to distort the data and/or the timing so as to force the implementation to take the wrong decision about sending traffic into the network. The goal of the attacker is to cause the maximum slowdown at the victim with the least possible investment, and the attacker uses knowledge about the implementation to achieve this. Chapter 5 describes three different code modifications and tests them with a range of parameter values, to establish the usefulness of the parameter randomization as a defense mechanism.

The last implementation diversity tested can trace its roots to N-Version diversity. Its target threat model the resource exhaustion DoS. These attacks use protocol or implementation features or errors to consume some group of resources, which can cause either a program collapse or the repudiation of legitimate requests. This vulnerability can arise from deficient resource allocation policies, misjudgment of possible request rates, authentication problems and so on. The diversification solution I implemented is based on intercepting all requests to a protected object and pass them through a filter based on policies constructed using Genetic Programming [62]. I present the implementation, a generic resource simulation evaluation and the results of protecting against the SYN Flood resource exhaustion attack.

Finally, I summarize all the results and their contribution to the case of automated

diversity for security in Chapter 7. It will show that automated diversity is a useful and important defense mechanism against attacks that exploit regularities in existing systems.

Chapter 2

Background and Related Work

The research presented in this thesis is part of a rich tradition of using diversity in Computer Science, implicitly or explicitly. This chapter shows the history and current uses of diversity, in order to situate my work in the larger context. As stated before, I use three separate threat/defense pairs to sample the large space of diversification possibilities. In consequence, I will discuss the specific background and related work for each one of them on their specific chapters. Only background and related work common to all three techniques will be treated in this chapter.

The organization of this chapter is as follows: Section 2.1 describes the use of diversity for reliability, Section 2.2 reviews general uses of diversity for security and Section 2.3 presents the use of diversity for deception.

2.1 Diversity for reliability

The idea of using diverse methods to improve the overall reliability of a task is very old. As early as 1837 Babbage was speculating that a particularly complicated calculation,

could be done in two or more distinct ways and the result accepted only if it were the same in all of them [8]. However, it was not until the early 1970s that several papers appeared suggesting the use of multiple versions to improve fault tolerance [6]. The two current systematic approaches to software design diversity appeared in the same decade: Recovery Blocks in 1975 [87] and N-version Programming in 1977 [7].

A fault-tolerant system using software diversity is composed of:

a. $N \geq 2$ distinct member units (the versions of the program);

b. An execution environment (EE) to execute and choose member units; and

c. A choosing algorithm (used by the EE).

The member units are assumed to be correct to within the limits of the testing procedures available. The failures that might happen should originate from unforeseen inputs and environmental conditions.

In Recovery Blocks (RB), the member units are called alternates. An RB EE is required to store an image of the last known correct system state at all times (the "Recovery Block"). The EE starts with an initial state, selects one alternate, and executes it until a predefined checkpoint. At that point, the EE applies an acceptance test that attempts to establish if the system is still in a "correct" state. The acceptance test is part of the RB choosing algorithm. If the state is still acceptable, the current state is used to generate another image, which replaces the previous one as most current. The EE then resumes execution of the current alternate towards the next checkpoint. If the state is not acceptable, the system state is restored to the last correct state seen, which is done using the stored image. After that, the choosing algorithm selects a different alternate from the pool of member units, and executes this new one until the checkpoint at which the problem was detected is reached again. At this point, the process repeats.

In N-Version Programming, the EE executes all versions (member units) in parallel. When reaching a checkpoint, the EE analyzes the results obtained by the versions and chooses one by majority vote (or by any other such method). This is the result given out as the output of the system at the given checkpoint. The versions might or not update their state with the state of the winner version. The main difference between the two approaches is the strategy that they apply to use the results of the different versions.

The usefulness of N-Version programming has been in dispute for a number of years. Fault-tolerant diversity has serious problems under the original evaluation criteria: that the software fail independently from one another for a given problematic input. The experiments first published by Knight and Leveson in 1986 [60], and several others, show that versions do not fail independently, although it does not make it clear if this is due to dependencies in the versions. In support of these results, the Eckhardt and Lee model [36] predicts that even if it were possible to generate perfectly independent versions for a given task, their failure for some inputs would not be independent.

More recently, however, it has been shown that this simple independence criterion is not directly relevant to the evaluation of N-Version. The important quantifier is the probability that a N-Version system will detect a fault when compared with a system running a single version. In that sense, it has been consistently shown that diverse systems do detect faults, and, in experimental settings, a large proportion of 3-version sets detect a fault relative to a single version [71]. Indirect evidence for the usefulness of N-Version diversity is that the techniques are used successfully in industry [72], although in-depth analysis of these systems are seldom published. The problem of equivocal evaluation techniques seem to be ubiquitous for implementation diversity solutions, and will be explored further in chapters 5, and 6.

In the diversity techniques discussed above, the versions are created manually. It is believed that, the cost associated with the creation of the versions, has been one of the main obstacles to more widespread use of software design diversity [72]. Not surprisingly,

some of the proposals to overcome this problem have to do with modularization, and creating smaller 'units of diversity', that could be independently developed and then put together in a larger system [104].

An additional approach being explored by some [40, 52], is the use of Genetic Programming [62] to automatically generate diverse versions. Although the target systems are small, the results are encouraging as they display the expected fault tolerance improvement in the multi-version systems against single-version comparisons. Combined with modular design it could prove an important tool for cheaper generation of diverse software. This is a technique that is adapted in the present work to create automated diversity for security.

2.2 Diversity for security

Diversity in software engineering is quite different from diversity for security. In software engineering, the basic idea is to generate multiple independent solutions to a problem (e.g., multiple versions of a software program) with the hope that they will fail independently, thus greatly improving the chances that some solution out of the collection will perform correctly in every circumstance. The different solutions may or may not be produced manually, and the number of solutions is typically quite small, around ten.

Diversity in security is introduced for a different reason. Here, the goal is to reduce the risk of widely replicated attacks, by forcing the attacker to redesign the attack each time it is applied. For example, in the case of a buffer overflow attack, the goal is to force the attacker to rewrite the attack code for each new computer that is attacked. Typically, the number of different diverse solutions is very high, potentially equal to the total number of program copies for any given program. Manual methods are thus infeasible, and the diversity must be produced automatically.

Cohen [23] published the first paper advocating the need to make computer systems less homogeneous to improve security via automated methods. Although specifically targeted as a defense against computer viruses, it could be considered the predecessor of the idea of diversity. He is also the first to state the main premise of diversity for security: that the objective is to drive up the cost of an attack until it becomes no longer worth performing, as opposed to creating a demonstrably impenetrable defense. The problem to solve in order to implement such a defense is how to reduce coherence in the system, to force the attacker to customize the attack for each system. The proposed solution was a process he named 'evolution', in which systems change themselves progressively, building from equivalent but different sets of primitives, diverging from each other farther and farther in each step. Modifying steps could be triggered by system calls or asynchronously called at random times.

Another contribution to the field of diversity in [23] is an analysis of some techniques to produce semantically equivalent but morphologically different code starting from a base implementation. The results of a test implementation are presented, and while the transformations are definitely decidable, some of them are difficult to identify. It is suggested to use randomization in spots to make the result of the evolution even more unpredictable. What is unclear is the types of attacks that this code evolution could prevent directly, although it is an excellent way to make the structure of the evolved system difficult to scan for patterns.

In 1997 Forrest et al. published a paper that argued for a more general view of the possibilities of diversity for security [41], and introduced the word 'diversity' in the context of computer security. In this paper, application of diversity in computer systems was explored as a defense against a wide range of possible attacks, based on biological diversity principles. Additionally, it allowed for the diversification of data as well as code. Besides investigating targets where uniformity could be diminished in existing systems, it also presented an operational diversification defense against a real threat, that did not depend on

code but data layout randomization.

The automated introduction of diversity in a system can be considered a security adaptation. Cowan et al. introduced a classification for this class of defenses [31]. It partitions them based on what is being adapted: either the *interface* or the *implementation*. Interface adaptations work on the surface of the code being protected. They modify the layout or access controls to interfaces, without changing the implementation of the core code that the interfaces are giving access to. Implementation adaptations, do not alter the interface. Instead they modify implementation of portions of the system to make them resistant to attacks. The case for diversity could be made stronger by including both categories in the diversification of a system. However, all published diversifications use only one of them.

Since the publication of the seminal diversity papers, the need for diversity has been felt more acutely, and the concept of diversity for security has been explored in different ways. It was even broght forward to the general public attention by the publication of a report by a group of prominent computer scientists that denounced the security dangers of the monoculture introduced by Microsoft [44].

In this section, I will review some earlier approaches, and some of the new ones that are not directly related to the diversifications explored in this thesis. The rest are going to be presented in the relevant chapters.

A group that could be loosely considered to be a subset of implementation diversification includes derivatives of research into dynamic or static optimization, which implicitly introduces diversifications in the code and behavior of the programs. In this category the following two projects could be found: Synthetix [86], which is a toolkit for kernel optimization and Cactus [48], which is a micro protocols framework. While these projects did indeed diversify the code of the kernel, the effects of these particular adaptations on known attacks were almost nonexistent, as attacks tend to use higher level constructs [31].

Another approach is to modify a single component with a specialized function to

choose dynamically among a restricted number of options, and maybe randomize parts of the function execution. This diversification form was used by the SecComm component of Cactus [47] and the network protocol heterogeneity proposal in [107].

The SecComm component of the Cactus Project at the University of Arizona allows users to choose among a range of cryptographic protocols and parameters for those protocols [47], so ideally each communication in a protected channel will use a different encryption method. The purpose of this design is to make decryption attacks impractical, because the guesswork process would have to be repeated for each communication channel. This system concentrates on protecting privileged communication channels instead of end systems. The reported results concentrate on feasibility and efficiency, while its impact against attacks is only analyzed theoretically. This is a common situation in security research, because it is difficult to establish what kind of experiments would prove that a given defense is 'effective' and in what sense.

Zhang et al. [107] propose an approach similar in spirit to SecComm, in the context of networking protocols. Both could be seen as being in a 'path diversity' class. The idea here is to chose a path through the network layers with different protocols in each communication. That is, instead of using, say (TCP, IP, IP, TCP) for a connection, the system in theory could chose to use (UDP, IPX, IP, TCP), so an exploit expecting a TCP exchange would be frustrated. Apart from the difficulty of coordinating different protocols, this approach has the disadvantage of having a very small search space for path options. No more work has been reported on this proposal, so it is being included just for completeness.

The most promising way to incorporate diversity for security seems to be interface diversity, where the goal is to somehow randomize a vulnerable interface without changing the implementation. It is making the labels difficult to guess, without chaging the function to which they are attached. Some of the work in this area of diversification is directly related to ours, and has been published at about the same time than the research presented in this thesis. It will be reviewed in Chapter 3. The rest are discussed below.

The Morphing File System (MFS), part of the Blinded project [31], was an implementation of file system diversification. MFS stored core files at non standard locations, while preserving the normal file system structure with false files. Programs with need-to-know were told the location of the correct files, while intruders were left to wander in the forest of false files. According to the same paper, the implementation was successful in avoiding attacks, but was very complex and brittle. Unfortunately, this project was not continued, and is not posted for download and testing, so further analysis is difficult. In conversations with one of the authors (Calton Pu) I found that the project is currently being revisited, so more information could be forthcoming.

Chew and Song [21] proposed a method that combines kernel and loader modification on the system level with binary rewriting at the process level to provide system call number randomization, random stack relocation, and randomization of standard library calls. This work has not been completely evaluated to my knowledge.

Transparent Runtime Randomization (TRR) [105] randomize the positions of the stack, shared libraries and heap. The main difference between the two is the implementation level. ASLR is implemented in the kernel while TRR modifies the loader program. Consequently, TRR is more oriented to the end-user.

Bhatkar et al. [16] describe a method that randomizes the addresses of data structures internal to the process, in addition to the base address of the main segments. Internal data and code blocks are permuted inside the segments and the guessing range is increased by introducing random gaps between objects. The current implementation instruments object files and ELF binaries to carry out the required randomizations. No access to the source code is necessary, but this makes the transformations extremely conservative. This technique nicely complements that of RISE, and the two could be used together to provide protection against both code injection and return-into-libc attacks simultaneously.

PointGuard [30] uses automated randomization of pointers in the code and is imple-

mented by instrumenting the intermediate code (AST in GCC).

2.3 Diversity for deception

The basic strategy of deception is to leave false services in the open and carefully hide real ones. Deception has been used with two purposes: to capture or investigate attacker behavior and to protect systems [26]. While both goals use similar techniques, diversity is more apparent in deception for protection. In the case of protective deception, when the attacker queries for service information, it is given false leads, which may or may not contain intrusion detection monitors. But the general idea is to confuse the attacker and make him waste resources attacking nonexistent services, and make real ones difficult to distinguish among the false ones.

Using this approach are Cohen's Deception Tool Kit (DTK) [25] and the HoneyNet Project [97]. All of them target standard network services. Deception technologies have proved successful [24], but used alone, they constitute reactive diversity strategies, as they require detailed knowledge of attacks and services to mount plausible dummy scenarios. A system using deception for protection will still be vulnerable to a new attack that targets a service previously considered as safe and hence not hidden in a forest of false servers. Furthermore, it is always open to blind and brute force attacks: that is, attacks that assume certain vulnerability and just try to exploit it on all possible locations, without checking for its viability beforehand.

2.3.1 Hardware encryption

Hardware components to allow decryption of code and/or data on-the-fly have been proposed since the late 1970's [14, 15] and implemented as microcontrollers for custom systems (for example the DS5002FP microcontroller [34]). The two main objectives of

these cryptoprocessors are to protect code from piracy and data from in-chip eavesdropping. An early proposal for the use of hardware encryption in general-purpose systems was presented by Kuhn for a very high threat level where encryption and decryption were performed at the level of cache lines [64]. This proposal adhered to the model of protecting licensed software from users, and not users from intruders, so there was no analysis of shared libraries or how to encrypt (if desired) existing open applications. A more extensive proposal was included as part of TCPA/TCG [96]. Although the published TCPA/TCG specifications provide for encrypted code in memory, which is decrypted on the fly, TCPA/TCG is designed as a much larger authentication and verification scheme and has raised controversies about Digital Rights Management (DRM) and end-users' losing control of their systems [3, 4]. RISE contains none of the machinery found in TCPA/TCG for supporting DRM. On the contrary, RISE is designed to maintain control locally to protect the user from injected code.

Chapter 3

Instruction Set Randomization

An interesting level of interface diversification is that of a language specification. In particular, the working mechanism of many attacks, such as buffer overflows, is to insert malicious binary code into the memory of an executing process. An instruction-set diversification provides a strong defense against this type of attack by making the code of the attack unrecognizable by the execution engine. This chapter describes Randomized Instruction Set Emulation (RISE) [1], a diversity solution that randomizes the code of a process, making it unique to each execution.

Section 3.1 describes the code injection threat, and justifies the use of code randomization against this particular threat. Section 3.2 presents the strategy used to diversify the native instruction set, and Section 3.3 gives the details of the actual implementation and explains the operation of RISE. In Section 3.4 important issues discovered with the design and implementation of RISE are discussed. Section 3.5 and Section 3.6 report on the attacks tested against RISE, and an evaluation of its performance. A discussion of the results is given in Section 3.7. A very thorough background and related work analysis is presented in Section 3.8. Some future work suggestions are given in Section 3.9, and a

[1] Large sections of this chapter were adapted from the comprehensive description of RISE to be published in [11].

summary of the chapter is given in Section 3.10.

3.1 Target threat

The target threats of a binary code randomization are code injection attacks. Code-injection is currently the preferred means of exploiting a memory corruption vulnerability [103, 20]. In a code-injection attack, hostile machine code is written in the memory space of an executing process, with immediate or delayed activation. The reason why diversifying the instruction is effective against such an attack is that the malicious code is written in the standard machine language, but the process is using an individualized version of the language. From the point of view of the execution engine on which the process is running, the attack contains random binary data instead of well-formed instructions. Therefore, the attack is unable to execute. A restriction on the threat model is that only remote attacks are considered. Local attacks unfortunately can learn the randomization patterns and adjust its shape accordingly.

There are three factors that characterize a language that is a good candidate for randomization; the native instruction set complies with all of them:

- **Vulnerability expectation:** There should be a reasonable expectation that it is possibly to develop an attack using the language to be randomized. For example, diversifying the microcode for the operations in a given instruction set, but leaving the operation interface unchanged (same names, operator order and so on), is useless, as the only way an attacker could use the microcode is through the operations, and those remain the same. On the contrary, there are hundreds of attacks using standard machine code as payload [20].

- **Ease of discrimination** There should be at least one identification criteria such that it would be possible to distinguish with high certainty between trusted and untrusted

programs. The recognition mechanism should have a clear and simple target, preferably without any human interactive intervention. For example, it is easy to distinguish programs expressed in a given format (e.g. ELF [98]) already on disk versus those arriving over the Internet as raw fragments. If the policy is that the former are considered safe while the latter are not, the recognition mechanism is straightforward to implement and will not generate false positives. Similarly, in memory-corrupting attacks, disk content is considered safe before the attack, so it is easy to decide which code to convert to the customized language.

- **Critical mass** Most programs must be legitimate when the original randomization is performed. This is an issue not unlike the false-positive problem in intrusion detection: Making decisions about code deemed illegitimate by the recognition algorithm, but that could be safe, must be kept to a minimum. Otherwise, users overloaded with false positives will approve everything, ultimately leading to the execution of malicious code. It is due to this aspect that web scripting languages (e.g. Javascript, Activex) are bad candidates, as most code that must be executed is coming from external, hardly unverifiable sources. Binary code randomization clearly has critical mass as *all* executable programs on disk can be safely randomized.

Binary code is therefore an excellent candidate for diversification. It is also very effective: it against any attack that *requires* executing binary code during its deployment. Although attacks in this threat model are only a subset of of memory-corrupting attacks, they are the most prevalent of the group [103].

Further, attacks that only overwrite data are excluded from the threat model. They can be addressed using different interface randomizations [16, 82]. An important consideration is that attacks expressed in a language different from the native instruction set will also not be stopped by RISE. Although this seems clear for injected Perl code, for example, it also applies to attacks such as 'return into libc' [80], which are erroneously grouped together with binary code injection attacks. In reality, they use another language level, the language

defined by addressable functions in the process memory, which should be considered by another level of code randomization.

In the following section, the approach used to design the instruction set randomization is described.

3.2 Diversifying an instruction set

This section justifies the choices of a particular instruction set and operating system, and presents the diversification strategy.

3.2.1 Infrastructure

The architecture chosen was Intel's IA32, which in addition to wide popularity, has the characteristic of being one of the easiest to attack, because of its very complex, CISC design. For of these two reasons, most code-injection attacks are designed for this architecture.

The operating system used was Linux. Although many attacks target applications in Linux, the main reason to choose it was the fact that it is licensed under the GPL, and that all required tools are available under the same license. In the next subsection the diversification strategy is presented.

3.2.2 Diversification strategy

The principal difficulty is the modification of the processor to accept a customized machine language. Given the evident problem of attempting to modify the IA32 instruction set in hardware, it was decided to use a software version instead. The generic tool to be able

to execute a diversified instruction set is a binary (native-to-native) emulator. Although there are several open-source emulators, Valgrind [90, 81] was chosen because it is has an excellent record of successfully emulated real applications.

Next, it was necessary to decide about the location and mechanism for the diversification creation and interpretation. It was determined that the location for the new mapping creation and interpretation would be inside the emulator, at runtime. Delaying the diversification as late as possible has several advantages: A new language is created in each process execution, the mapping between the standard and the diversified language cannot be guessed from analyzing static files, and information about the mapping gained during one process run cannot be used for the next.

Finally, the diversification mechanism itself had to be chosen. Several choices existed, from creating a per-instruction mapping to complex cryptographic schemes. Because it is essential to have a large randomization space, but reasonable performance, randomization using XOR with a large random mask was chosen.

The randomization operates in two phases: First, it randomizes the executable at process initialization creating a unique mapping for the life of the process; and second, it de-randomizes instructions at fetch, interpreting correctly the new mappings.

The name given to the randomization strategy just described was Randomized Instruction Set Emulation (RISE). In the next section the implementation and operation of RISE are described in detail.

3.3 Implementation and operation

As stated above, RISE delays the randomization until process initialization. More specifically, it waits until the loader passes control to Valgrind, and then proceeds to encode all executable sections. The identification of executable portions depends on the format of

the binary file. Although Linux supports several representations for executables , Valgrind only uses the Executable and Linking Format (ELF) [98]. In ELF, executable sections are clearly separated from data, making their randomization a relatively easy task.

The randomization is accomplished by creating a random mask and combining it with the code using XOR. The result is written to the original code location. Figure 3.3 illustrates this process. Each byte in the original code is XORed with a byte in the mask, and the result written back. The random mask is created using pseudo-randomness from the Linux `/dev/urandom` device [100], which uses a secret pool of true randomness to seed a pseudo-random stream generated by feedback through SHA1 hashing. The mask can be as large as the whole executable space or be configured in size by the user.

Given that the architecture is CISC, the emulator has to proceed with instruction decoding byte-to-byte. Consequently, for each byte fetched, the mask corresponding to its address is retrieved and XOR-ed with it. The cleartext byte is then passed to the instruction decoder. Figure 3.3 shows the fetch process. Given that Valgrind is not a pure emulator, decoded linear blocks of code are stored in a cache and later used as needed. The more expensive fetch function just described will only be invoked on new or cache-evicted code.

Under attack, each byte of the injected code is combined with its 'corresponding' random mask. Given that the attack was not pre-randomized, the result is a stream of random bytes that either express a few lines of random machine code or results in process death. This process is illustrated in Figure 3.3.

The operation of RISE in Valgrind is depicted in Figure 3.3. RISE intercepts the next byte fetch, XORs it with the randomization mask an passes it along to Valgrind's transformations. Valgrind first translates interpreted IA32 code to UCode, a risk-like intermediate representation, then (optionally) optimizes and instruments the UCode block, translates it back to IA32, stores the code block into the emulator's cache and eventually Valgrind's dispatcher sends it for processing in the CPU. This intermediate processing is necessary

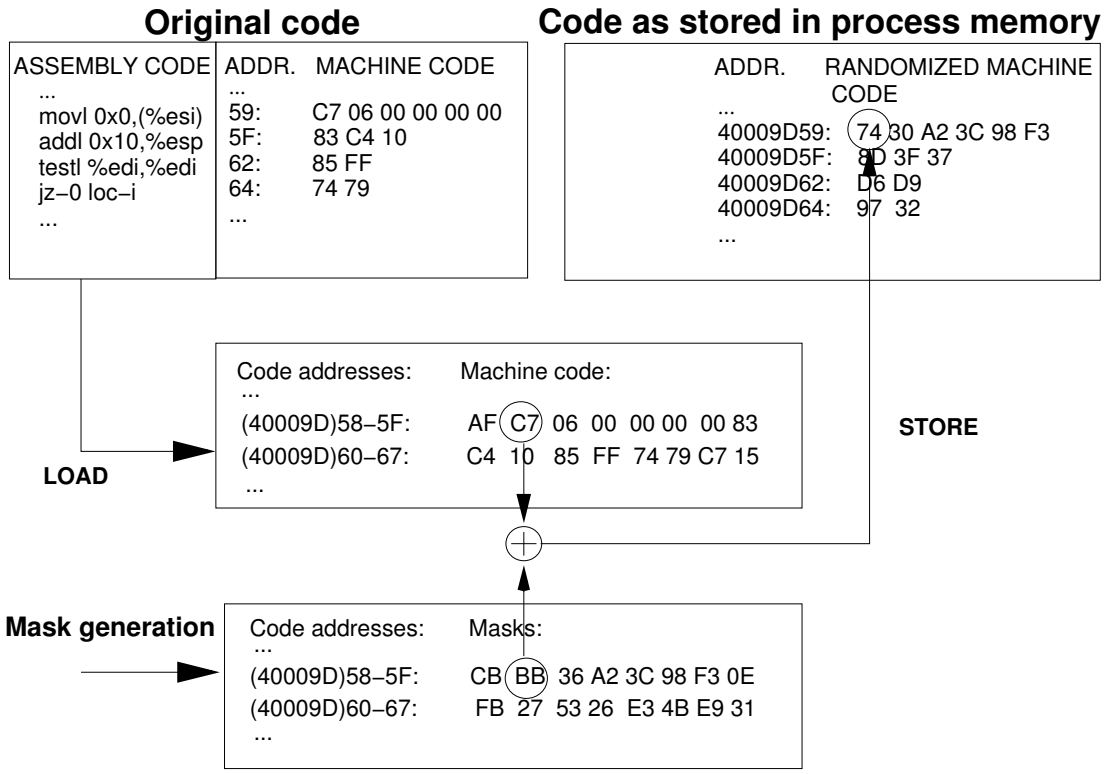


Figure 3.1: Creating a unique, randomized version of the process code just loaded into memory with RISE. For each byte in the executable sections, a mask byte is retrieved. The result is stored back in the original memory position.

for the many debugging functions of Valgrind, but adds unnecessary cost to RISE.

3.4 Design and implementation issues

There are several issues that complicate the implementation of RISE: the handling of shared libraries, the protection of emulator and cache code, the management of self-modifying code, the use by the emulator of calls defined in client-process space, data present inside code, and self-emulation. These difficulties are reviewed in this section.

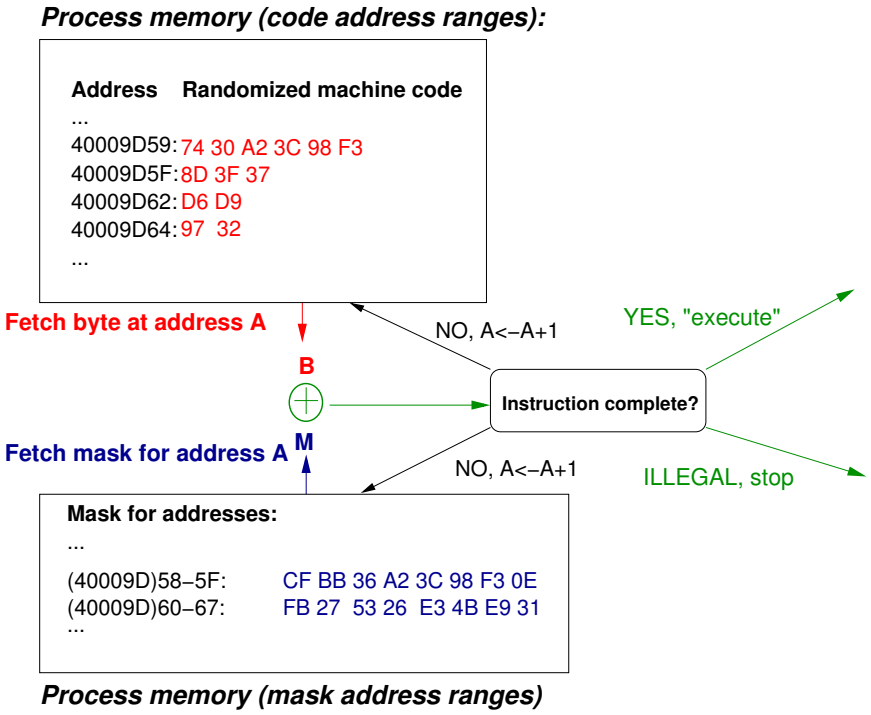


Figure 3.2: Interpreting the new language with RISE. When interpreting a byte from address A, the mask byte corresponding to address A is retrieved and XORed with the code byte. The cleartext result is passed to the instruction interpreter.

Shared libraries constitute a difficult problem for a code randomizations. There are several possible solutions: leaving shared libraries unprotected, encoding them with a shared mask, linking processes statically, or encoding them differently for each process. The first two solutions open vulnerabilities, while the third is impractical for most applications, and the fourth replicates the libraries for each process' child. While private shared libraries require extra memory, the cost is manageable by modern systems and it is the approach that provides the best protection. The duplication occurs as a side effect of copying the randomized version of the shared library bytes, as modern operating systems do a copy-on-write when a shared library is modified by a process.

Protecting the plaintext instructions inside Valgrind is a second concern. As Valgrind

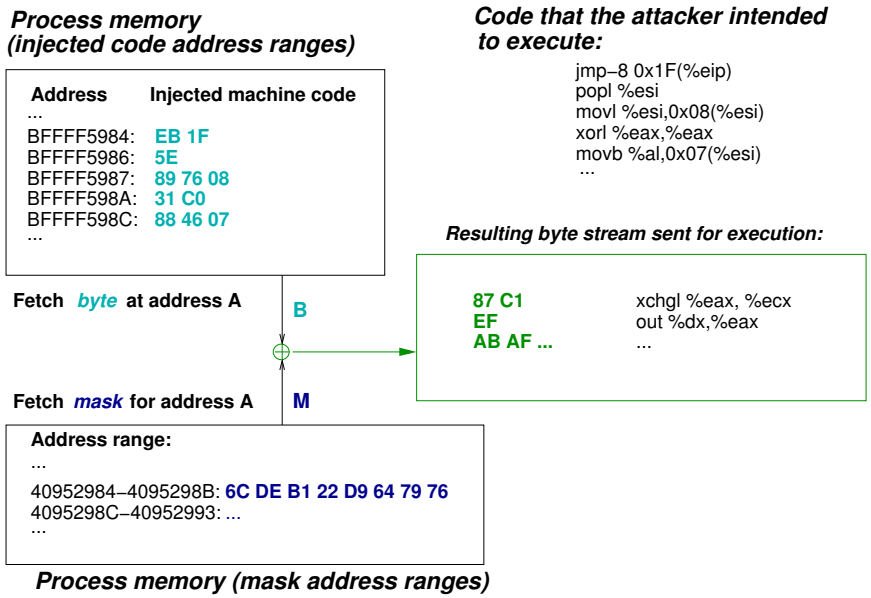


Figure 3.3: RISE operation under attack. As bytes from the mask are XORed with the attack code, the resulting random bytes are passed to the instruction interpreter. As could be seen to the right, the result is still executable code for a few instructions, but clearly not the intended code.

simulates the operation of the CPU, during the fetch cycle when the next byte(s) are read from program memory, RISE intercepts the bytes and unscrambles them; the scrambled code in memory is never modified. Eventually, however, a plaintext piece of the program (semantically equivalent to the block of code just read) is written to Valgrind's cache. From a security point of view, it would be best to separate the RISE address space completely from the protected program address space, so that the plaintext is inaccessible from the program, but as a practical matter this would slow down emulator data accesses to an extreme and unacceptable degree. For efficiency, the interpreter is best located in the same address space as the target binary, although this introduces some security concerns. A RISE-aware attacker could aim to inject code into a RISE data area, rather than that of the vulnerable program. This is problematic because the cache cannot be encrypted. To protect the cache, its pages are kept as read-and-execute only. When a new translated ba-

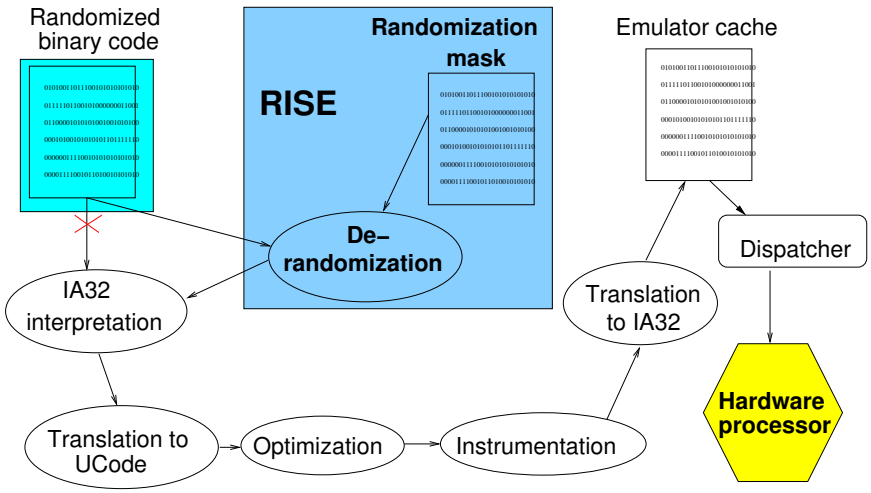


Figure 3.4: Global RISE operation in Valgrind. The randomized binary code is passed to RISE. RISE uses the randomization mask to decrypt the fetched byte, and passes it to the IA32 interpretation. From then on, several transforms create the corresponding binary code that eventually is execute in the real processor (in the standard binary language.).

sic block is ready to be written to the cache, RISE marks the affected pages as writable, execute the write action, and restore the pages to their original non-writable permissions. A more principled solution would be to randomize the location of the cache and the fragments inside it, a possibility for future implementations of RISE.

The current implementation does not handle self-modifying code, but it has a primitive implementation of an interface to support dynamically generated code. Arbitrary self-modifying code is an undesirable programming practice. However, it is desirable to support legitimate dynamically generated code, and it is intended to eventually provide a complete interface for this purpose.

For design clarity, an emulator should not use the same shared libraries as the process being emulated. Valgrind deals with this issue by adding its own implementation of all library functions it uses, with a local modified name (for example, `VGplain_printf` in-

stead of `printf`). A regular emulator can function in the presence of symbols incorrectly resolving to client space. When using RISE however, client code is stored randomized, so when the emulator's own (on-CPU) execution attempts to use client code, it crashes. Although disconcerting, this actually shows that the approach works, and that attempts to execute unauthorized code is meet with failure. These dangling unresolved references were resolved one-by-one by adding more local functions to Valgrind and renaming affected symbols with local names (e.g., `rise_umoddi` instead of `'%'` (the modulo operator)).

A more subtle problem arises because the IA32 does not impose any data and code separation requirement, and some compilers insert dispatch tables directly in the code. In those cases, the addresses in internal tables are scrambled at load time (because they are in a code section), but are not de-randomized at execution time (because they are read as data). Although this does not cause an illegal operation, it causes the emulated code to jump to a random address and fail inappropriately. To avoid this behavior, RISE looks for code sequences that are typical for jump-table referencing and adds machine code to check for in-code references into the block written to the cache. If an in-code reference is detected when the block is executing, our instrumentation de-scrambles the data that was retrieved and passes it in the clear to the next (real) instruction in the block. This scheme could be extended to deal with the general case of using code as data by instrumenting every dereference to check for in-code references. However, this would be computationally expensive, so it is not implemented it in the current prototype. Code is rarely used as data in legitimate programs except in the case of virtual machines, which is addressed separately.

An additional difficulty was discovered with Valgrind itself. Valgrind's thread implementation and memory inspection capabilities require Valgrind to emulate itself at certain moments. To avoid infinite emulation regress it has a special workaround in its code to execute some of its own functions natively during this self-emulation. This case was handled by detecting Valgrind's own address ranges and treating them as special cases. This

issue is specific to Valgrind.

In general, all implementation issues were solvable, and some will not be present at all in a hardware implementation. In the next section, a description of the experiments done to test RISE's effectiveness and performance is presented.

3.5 Experiments

The results reported in this section were obtained using the RISE prototype, available under the GPL from `http://cs.unm.edu/~immsec`. RISE's ability to run programs successfully under normal conditions and its ability to disrupt a variety of machine code injection attacks were tested. The attack set contained twenty synthetic and fifteen real attacks.

The synthetic attacks were obtained from two sources. Two attacks, published by Fayolle and Glaume [39], create a vulnerable buffer—in one case on the heap and in the other case on the stack—and inject shellcode into it. The remaining eighteen attacks were executed with the attack toolkit provided by Wilander and Kamkar and correspond to their classification of possible buffer overflow attacks [103] according to technique (direct or pointer redirection), type of location (stack, heap, BSS or data segment) and attack target (return address, old base pointer, function pointer and longjump buffer). Without RISE, either directly on the processor or using Valgrind, all of these attacks successfully spawn a shell. Using RISE, the attacks are stopped.

The real attacks were launched from the CORE Impact attack toolkit [28]. The fifteen attacks tested satisfied the following requirements of our threat model and the chosen emulator: the attack is launched from a remote site; the attack injects binary code at some point in its execution; and, the attack succeeds on a Linux OS. Because Valgrind runs under Linux; the focus was on Linux distributions, reporting data from Mandrake 7.2 and versions of RedHat from 6.2 to 9.

3.6 Results

All real (non-synthetic) attacks were tested on the vulnerable applications before retesting with RISE. All of them were successful against the vulnerable services without RISE, and they were all defeated by RISE (Table 3.1).

Based on the advisories issued by CERT in the period between 1999 and 2003, Xu et al. [105] classify vulnerabilities that can inject binary code into a running process according to the method used to modify the execution flow: buffer overflows, format string vulnerabilities, malloc/free and integer manipulation errors. Additionally, the injected code can be placed in different sections of the process (stack, heap, data, BSS). The main value of RISE is its imperviousness to the entry method and/or location of the attack code, as long as the attack itself is expressed as binary code. This is illustrated by the diversity of vulnerability types and shellcode locations used in the real attacks (columns 3 and 4 of Table 3.1).

The available synthetic attacks are less diverse in terms of vulnerability type. They are all buffer overflows. However, they do have attack code location variety (stack, heap and data), and more importantly, they have controlled diversity of corrupted code address types (return address, old base pointer, function pointer and longjump buffer as either local variable or parameter), and offer either direct or indirect execution flow hijacking (see [103]). All of Wilander's attacks have the shellcode located in the data section. Both of Fayolle and Glaume's exploits use direct return address pointer corruption. The stack overflow injects the shellcode on the stack and the heap overflow locates the attack code on the heap. All synthetic attacks are successful (spawn a shell) when running natively on the processor or over unmodified Valgrind 2.0.0. All of them are stopped by RISE (column 5 of Table 3.2).

These results confirm that RISE is successfully implemented, and that a randomized instruction set prevents injected machine code from executing, without the need for any

Attack	**Linux Dist.**	**Vulnerability**	**Loc. of injected code**	**Stopped by RISE**
Apache OpenSSL SSLv2	RedHat 7.0 & 7.2	Buffer Overflow & malloc/free	Heap	√
Apache mod php	RedHat 7.2	Buffer Overflow	Heap	√
Bind NXT	RedHat 6.2	Buffer Overflow	Stack	√
Bind TSIG	RedHat 6.2	Buffer Overflow	Stack	√
CVS flag insertion heap exploit	RedHat 7.2 & 7.3	malloc/free	Heap	√
CVS pserver double free	RedHat 7.3	malloc/free	Heap	√
PoPToP Negative Read	RedHat 9	Integer error	Heap	√
ProFTPD _xlate_ascii _write off-by-two	RedHat 9	Buffer overflow	Heap	√
rpc.statd format string	RedHat 6.2	Format string	GOT	√
SAMBA nttrans	RedHat 7.2	Buffer overflow	Heap	√
SAMBA trans2	RedHat 7.2	Buffer overflow	Stack	√
SSH integer overflow	Mandrake 7.2	Integer error	Stack	√
sendmail crackaddr	RedHat 7.3	Buffer overflow	Heap	√
wuftpd format string	RedHat 6.2–7.3	Format string	Stack	√
wuftpd glob "~{"	RedHat 6.2–7.3	Buffer overflow	Heap	√

Table 3.1: **Results of attacks against real applications executed under RISE.** Column 1 gives the exploit name (and implicitly the service against which it was targeted). The vulnerability type and attack code (shellcode) locations are included (columns 3 and 4 respectively). The result of the attack is given in column 5.

knowledge about how or where the code was inserted in process space. In the following section, the issue of RISE's cost is explored.

3.6.1 Performance

RISE was originally designed as a proof-of-concept prototype. The emulator on which it is based is not particularly fast (although it is not one of the slowest), as being efficient

Type of overflow	**Shellcode Location**	**Exploit origin**	**Number of $\neq$ pointer types**	**Stopped by RISE**
Stack direct	Data	[103]	6	6 (100%)
Data direct	Data	[103]	2	2 (100%)
Stack indirect	Data	[103]	6	6 (100%)
Data indirect	Data	[103]	4	4 (100%)
Stack direct	Stack	[39]	1	1 (100%)
Stack direct	Heap	[39]	1	1 (100%)

Table 3.2: **Results of the execution of synthetic attacks under RISE.** Type of overflow (Column 1) denotes the location of the overflowed buffer (stack, heap or data) and the type of corruption executed: *direct* modifies a code pointer during the overflow (such as the return address), and *indirect* modifies a data pointer that eventually is used to modify a code pointer. Shellcode location (column 2) indicates the segment where the actual malicious code was stored. Exploit origin (column 3) gives the chapter from which the attacks were taken. The number of pointer types (column 4) defines the number of different attacks that were tried by varying the type of pointer that was overflowed. Column 5 gives the number of different attacks in each class that were stopped by RISE.

was never an objective. In fact, Valgrind's implementation is peppered with error detection code meant for debugging, but inconvenient for a security tool such as RISE. It has been proved that RISE is effective, now it is going to be argued that the costs are acceptable for some applications, even with the current implementations.

RISE could be implemented with hardware support, in which case much better performance could be achieved, given that coding and decoding could be performed directly in registers rather than executing two different memory accesses for each fetch.

The size of each RISE-protected process is increased from three sources: the data structures native to Valgrind, including the code cache, which should be large for better run-time performance; the space occupied by the mask, which will depend on the mask size used, and can be as large as double the executable space; and the private copies of shared libraries.

From the point of view of runtime slowdowns, RISE amortizes interpretation cost by

storing translations in a cache, which allows native-speed execution of previously interpreted blocks. If RISE's cache size is large enough for the most frequently used sections of the program, longer-running process could execute at near native speed.

Valgrind is much slower than binary translators [9, 18] because it converts the IA32 instruction stream into an intermediate representation before creating the code fragment. However, some evidence will be given that long-running, server-class processes can execute at reasonable speeds and these are precisely the processes for which RISE is most needed.

As an example of this effect, Table 3.3 provides one data point about the long-term runtime costs of using RISE using the Apache web server in the face of a variety of non-attack workloads. Classes 0 to 3, as defined by SPEC [93], refer to the size of the files that are used in the workload mix. Class 0 is the least I/O intensive (files are less than 1 KB long) and class 3 uses the most I/O (files up to 1000 KB long). As expected, on I/O bound mixes, the throughput of Apache running over RISE is closer to Apache running directly on the processor [2]. Table 3.3 shows that the RISE prototype slows down by a factor of no more than three, and sometimes by as little as 5%, compared with native execution, as observed by the client. These results should not be taken as a characterization of RISE's performance, but as evidence that cache-driven amortization and large I/O and network overheads make the CPU performance hit of emulation just one (and possibly not the main) factor in evaluating the performance of this scheme.

By contrast, short interactive jobs are more challenging for RISE performance, as there is little time to amortize mask generation and cache filling. For example, I measured a slowdown factor of about 16 end-to-end when RISE-protecting all the processes invoked to make this chapter from LaTeX source.

Results of the Dynamo project suggest that a custom-built dynamic binary translator

[2]The large standard deviations are typical of SPECweb99, as web server benchmarks have to model long-tailed distributions of request sizes [93, 77].

can have much lower overheads than Valgrind, suggesting that a commercial-grade RISE would be fast enough for widespread use. In fact, in long-running contexts where performance is less critical, even our proof-of-concept prototype might be practical.

Mix type	Native Execution		Execution over RISe		RISE /
	Mean(ms.)	Std.Dev.	Mean(ms.)	Std.Dev.	Native
class 0	177.32	422.22	511.73	1,067.79	2.88
class 1	308.76	482.31	597.11	1,047.23	1.93
class 2	1,230.75	624.58	1,535.24	1,173.57	1.25
class 3	10,517.26	3,966.24	11,015.74	4,380.26	1.05
total	493.80	1,233.56	802.63	1,581.50	1.62

Table 3.3: Comparison of the average time per operation between native execution of Apache and Apache over RISE. Presented times were obtained from the second iteration in a standard SPECweb99 configuration (300 seconds warm up and 1200 seconds execution).

3.7 Discussion

The preceding sections describe a prototype implementation of the RISE approach and evaluate its effectiveness at disrupting attacks. In this section, I discuss some important issues about the current RISE implementation, and code randomization in general.

A valid concern when evaluating RISE's security is its susceptibility to key discovery, as an attacker with the appropriate randomizing mask information could inject prepared code that will be accepted by the emulator. However, RISE is resilient against brute force attacks due to the size of the mask (never less than 4 Kb), and the fact that random sequences are not reused. In a brute force attack, a failure means the demise and final death of the attacked process, or a re-spawn with a new mask. Any guesswork carried from the previous iterations is be useless.

An alternative path for an attacker is to try to inject arbitrary address ranges of the

process into the network, and recover the key from the downloaded information. The download could be part of the key itself (stored in the process address space), scrambled code, or unscrambled data. Unscrambled data does not give the attacker any information about the key. Even if the attacker could obtain scrambled code or pieces of the key (they are equivalent because it can be assumed that the attacker has knowledge of the program binary), using the stolen key piece might not be feasible. If the key is created eagerly, with a key for every possible address in the program, past or future, then the attacker would still need to know where the attack code is going to be written in process space to be able to use that information. However, in my implementation, where keys are created lazily for code loaded from disk, the key for the addresses targeted by the attack might not exist, and therefore might not be discoverable. The keys that do exist are for addresses that are usually not used in code injection attacks because they are write protected. In summary, it would be extremely difficult to discover or use a particular encoding during the lifetime of a process.

Another potential vulnerability is RISE itself. I RISE would be difficult to attack for several reasons. First, a network-based threat model is used (attack code arrives over a network) and RISE does not perform network reads. In fact it does not read any input at all after processing the run arguments. Injecting an attack through a flawed RISE read is thus impossible.

Second, if an attack arises inside a vulnerable application and the attacker is aware that the application is being run under RISE, the vulnerable points are the code cache and RISE's stack, as an attacker could deposit code and wait until RISE proceeds to execute something from these locations. Although RISE's code is not randomized because it has to run natively, the entire area is write-protected, so it is not a candidate for injection. The cache is read-only during the time that code blocks are executed, which is precisely when this hypothetical attack would be launched, so injecting into the cache is infeasible.

Another possibility is a *jump-into-RISE* attack. Three ways in which this might happen

are considered: [3]

a. The injected address of RISE code is in the client execution path cache.

b. The injected address of RISE code is in the execution path of RISE itself.

c. The injected address of RISE code is in a code fragment in the cache.

In case a, the code from RISE will be interpreted. However, RISE only allows certain self-functions to be called from client code, so everything else will fail. Even for those limited cases, RISE checks the call origin, disallowing any attempt to modify its own structures.

For case b, the attacker would need to inject the address into a RISE data area in RISE's stack or in an executable area. The executable area is covered by case c. For RISE's data and stack areas have introduced additional randomizations have been introduced. The most immediate threat is the stack, so its start address is randomized. For other data structures, the location could be randomized using the techniques proposed in [16], although this is unimplemented in the current prototype. Such a randomization would make it difficult for the attacker to guess its location correctly. An alternative, although much more expensive, solution would be to monitor all writes and disallow modifications from client code and certain emulator areas.

It is worth noting that this form of attack (targeting emulator data structures) would require executing several commands without executing a single machine language instruction. Although such attacks are theoretically possible via chained system calls with correct arguments, and simple (local) attacks have been shown to work ([80]), they are not a common technique [103]. In the next version of RISE it is planned to include full data structure address randomization, which would make these rare attacks extremely difficult to execute.

[3] It is assumed that RISE itself does not receive any external input once it is running.

Case c is not easily achieved because fragments are write-protected. However, an attacker could conceivably execute an *mprotect* call to change writing rights and then write the correct address. In such a case, the attack would execute. This is a threat for applications running over emulators, as it undermines all other security policies ([59]). In the current RISE implementation the solution used in [59], is borrowed monitoring all calls to the *mprotect* system call by checking their source and destination and not allowing executions that violate the protection policy.

Although this chapter illustrates the idea of randomizing instruction sets at the machine-code level, the basic concept could be applied wherever it is possible to (1) distinguish code from data, (2) identify all sources of trusted code, and (3) introduce hidden diversity into all and only the trusted code. A RISE for protecting `printf` format strings, for example, might rely on compile-time detection of legitimate format strings, which might either be randomized upon detection, or flagged by the compiler for randomization sometime closer to runtime. Certainly, it is essential that a running program interact with external information, at some point, or no externally useful computation can be performed. However, the recent SQL attacks illustrate the increasing danger of expressing running programs in externally known languages [46]. Randomized instruction set emulators are one step towards reducing that risk.

An attraction of RISE, compared to an approach such as code shepherding, is that injected code is stopped by an inherent property of the system, without requiring any explicit or manually defined checks before execution. Although divorcing policy from mechanism (as in code shepherding) is a valid design principle in general, complex user-specified policies are more error-prone than simple mechanisms that hard-code a well-understood policy.

3.8 Background and Related Work

Many approaches have been developed for protecting programs against particular methods of code injection, including: static code analysis [102, 66, 35] and run-time checks, using either static code transformations [32, 101, 22, 78, 84, 37, 38, 106, 29, 10, 99, 5, 69, 57, 88], dynamic instrumentation [10, 59] or hybrid schemes [79, 56]. In addition, some methods focus on protecting an entire system rather than a particular program, resulting in defense mechanisms at the operating system level and hardware support [82, 74, 106]. Instruction-set randomization is also related to hardware code encryption methods explored in [64] and those proposed for TCPA/TCG [96].

The automated diversity project that is closest to RISE is the system described in [58], which also randomizes machine code. There are several interesting points of comparison with RISE, and two of them are described: (1) per-system (whole image) vs. per-process randomization; (2) Bochs [19] vs. Valgrind as emulator. First, in the Kc et al. implementation, a single key is used to randomize the image, all the libraries, and any applications that need to be accessed in the image. The system later boots from this image. This has the advantage that in theory, kernel code could be randomized using their method although most code-injection attacks target application code. A drawback of this approach lies in its key management. There is a single key for all applications in the image and the key cannot be changed during the lifetime of the image. Key guessing is a real possibility in this situation, because the attacker would be likely to know the cleartext of the image. However, the Kc et al. system is more compact because there is only one copy of the libraries. On the other hand, if the key is guessed for any one application or library, then all the rest are vulnerable. Second, the implementations differ in their choice of emulator. Because Bochs is a pure interpreter it incurs a significant performance penalty, while emulators such as Valgrind can potentially achieve close-to-native efficiency through the use of optimized and cached code fragments.

A randomization of the SQL language was proposed in [17]. This technique is essentially the same one used in the Perl randomizer [58], with a random string added to query keywords. It is implemented through a proxy application on the server side. In principle, there could be one server proxy per database connection, thus allowing more key diversity. The performance impact is minimal, although key capture is theoretically possible in a networked environment.

3.8.1 Comparison of RISE to other defenses against code injection

Other defenses against code injection (sometimes called "restriction methods") can be divided into methods at the program and at the system level. In turn, approaches at the program level comprise static code analysis and runtime code instrumentation or surveillance. System level solutions can be implemented in the operating system or directly through hardware modifications. Of these, only the methods most relevant to RISE are reported.

Program-level defenses against code injection

Program-level approaches can be seen as defense-in-depth, beginning with suggestions for good coding practices and/or use of type-safe languages, continuing with automated analysis of source code, and finally reaching static or dynamic modification of code to monitor the process progress and detect security violations. Comparative studies on program-level defenses against buffer overflows have been presented by Fayolle and Glaume [39], Wilander and Kamkar [103] and Simon [91]. Several relevant defenses are briefly discussed below.

The StackGuard system [32] modifies GCC to interpose a a canary word before the return address, the value of which is checked before the function returns. An attempt to

overwrite the return address via linear stack smashing will change the canary value and thus be detected.

StackShield [101], RAD [22], install-time vaccination [78], and binary rewriting [84] all use instrumentation to store a copy of the function return address off the stack and check against it before returning to detect an overwrite. Another variant, Propolice [37, 38] uses a combination of a canary word and frame data relocation to avoid sensible data overwriting. Split Control and Data Stack [106] divides the stack in a control stack for return addresses and a data stack for all other stack-allocated variables.

FormatGuard [29] used the C preprocessor (CPP) to add parameter-counting to printf-like C functions and defend programs against format print vulnerabilities. This implementation was not comprehensive even against this particular type of attacks.

A slightly different approach uses wrappers around standard library functions, which have proven to be a continuous source of vulnerabilities. Libsafe [10, 99], TIED, and LibsafePlus [5], and the type-assisted bounds checker proposed by Lhee and Chapin [69] intercept library calls and attempt to ensure that their manipulation of user memory is safe.

An additional group of techniques depends on runtime bounds checking of memory objects, such as the Kelly and Jones bound checker [57] and the recent C Range Error Detector (CRED) [88]. Their heuristics differ in the way of determining if a reference is still legal. Both can generate false positives, although CRED is less computationally expensive.

The common theme in all these techniques is that they are specific defenses, targeting specific points-of-entry for the injected code (stack, buffers, format functions, etc.). Therefore, they cannot prevent an injection arriving from a different source or an undiscovered vulnerability type. RISE, on the other hand, is a generic defense that is independent of the method by which binary code is injected.

There is also a collection of dynamic defense methods which do not require access to

the original sources or binaries. They operate directly on the process in memory, either by inserting instrumentation as extra code (during the load process or as a library) or by taking complete control as in the case of native-to-native emulators.

Libverify [10] saves a copy of the return address to compare at the function end, so it is a predecessor to install-time vaccination [78] and binary rewriting [84], with the difference that it is implemented as a library that performs the rewrite dynamically, so the binaries on disk do not require modification.

Code Shepherding [59] is a comprehensive, policy-based restriction defense implemented over a binary-to-binary optimizing emulator. The policies concern client code control transfers that are intrinsically detected during the interpretation process. Two of those types of policies are relevant to the RISE approach.

Code origin policies grant differential access based on the source of the code. When it is possible to establish if the instruction to be executed came from a disk binary (modified or unmodified) or from dynamically generated code (original or modified after generation), policy decisions can be made based on that origin information. The model described in this chapter implicitly implements a code origin policy, in that only unmodified code from disk is allowed to execute. An advantage of the RISE approach is that the origin check cannot be avoided—only properly sourced code is mapped into the private instruction set so it executes successfully.

Also relevant are restricted control transfers in which a transfer is allowed or disallowed according to its source, destination, and type. Although a restricted version of this policy is used to allow signal code on the stack, in most other cases the RISE language barrier should ensure that injected code will fail.

System-level defenses against code injection

System level restriction techniques can be applied in the operating system, hardware, or both. This section briefly reviews some of the most important system-level defenses.

The non-executable stack and heap as implemented in the PAGEEXEC feature of PaX [82] is hardware-assisted. It divides allocation into data and code TLBs and intercepts all page-fault handlers into the code TLB. As with any hardware-assisted technique, it requires changes to the kernel. RISE is functionally similar to these techniques, sharing the ability to randomize ordinary executable files with no special compilation requirements. Our approach differs, however, from non-executable stacks and heaps in important ways. First, it does not rely on special hardware support (although RISE pays a performance penalty for its hardware independence). Second, although a system administrator can choose whether to disable certain PaX features on a per-process basis, RISE can be used by an end-user to protect user-level processes without any modification to the overall system.

A third difference between PaX and RISE is in how they handle applications that emit code dynamically. In PaX, the process emitting code requires having the PAGEEXEC feature disabled (at least), so the process remains vulnerable to injected code. If such a process intended to use RISE, it could modify the code-emitting procedures to use an interface provided by RISE, and derived from Valgrind's interface for Valgrind-aware applications. The interface uses a validation scheme based on the original randomization of code from disk. In a pure language randomization, a process emitting dynamic code would have to do so in the particular language being used at that moment. In this approximation, the process using the interface scrambles the new code before execution. The interface, a RISE function, considers the fragment of code as a new library, and randomizes it accordingly. In contrast to non-executable stack/heap, this does not make the area where the new code is stored any more vulnerable, as code injected in this area will still be expressed in

non-randomized code and will not be able to execute except as random bytes.

Some other points of comparison between RISE and PaX include:

a. Resistance to return-into-libc: Both RISE and PaX PAGEEXEC features are susceptible to return-into-libc attacks when implemented as an isolated feature. RISE is vulnerable to return-into-libc attacks without an internal data structure randomization, and data structure randomization is vulnerable to injected code without the code randomization. Similarly, as the PaX Team notes, LIBEEXEC is vulnerable to return-into-libc without ASLR (Automatic Stack and Library Randomization), and ASLR is vulnerable to injected code without PAGEEXEC [82]. In both cases, the introduction of the data structure randomization (at each corresponding granularity level) makes return-into-libc attacks extremely unlikely.

b. Signal code on the stack: Both PaX and RISE support signal code on the stack. They both treat it as a special case. RISE in particular is able to detect signal code as it intercepts all signals directed to the emulated process and examines the stack before passing control to the process.

c. C trampolines: PaX detects trampolines by their specific code pattern and executes them by emulation. The current RISE implementation does not support this, although it would not be difficult to add it.

StackGhost [42] is a hardware-assisted defense implemented in OpenBSD for the Sparc architecture. The return address of functions is stored in registers instead of the stack, and for a large number of nested calls StackGhost protects the overflowed return addresses through write protection or encryption.

Milenković et al. [74] propose an alternative architecture where linear blocks of instructions are signed on the last basic block (equivalent to a line of cache). The signatures

are calculated at compilation time and loaded with the process into a protected architectural structure. Static libraries are compiled into a single executable with a program, and dynamic libraries have their own signature file loaded when the library is loaded. Programs are stored unmodified, but their signature files should be stored with strong cryptographic protection. Given that the signatures are calculated once, at compile time, if the signature files are broken, the program is vulnerable.

Xu et al. [106] propose using a Secure Return Address Stack (SRAS) that uses the redundant copy of the return address maintained by the processor's fetch mechanism to validate the return address on the stack.

3.9 Future Work

As suggested in the chapter there are two obvious improvements that should be tested. The first one is to explore a hardware implementation of RISE. The second one is to port RISE to an optimizing emulator.

In terms of extending the coverage offered by RISE against memory-corruption attacks that do not inject binary code, additional randomizations have been undertaken. The first is stack padding for the client process calls, which is accomplished by instrumenting the prologue and epilogue code. The second one is heapification of static data structures, of which there are many instances in Valgrind. Stable versions of these extra randomizations will provide less targets for dangerous data modifications. However, as previously discussed, non code-injection attacks are using another language level. A much more interesting direction would be to explore the randomization of the overlayed process-functions language. Some steps in this direction has already been done in [16, 82], but a more extensive randomization is needed.

Another important endeavor is the development of synthetic RISE-aware attacks to

create a better testing suite.

3.10 Summary

This chapter described the concept of a randomized instruction set emulator as a defense against binary code injection attacks. The feasibility and utility of this concept was demonstrated with a proof-of-concept implementation based on Valgrind. This implementation successfully scrambles binary code at load time, unscrambles it instruction-by-instruction during instruction fetch, and executes the unscrambled code correctly. The implementation was successfully tested on several code-injection attacks, some real and some synthesized, which exhibit common injection techniques.

Chapter 4

Risk analysis of a language randomization

Code diversification techniques such as RISE rely on the assumption that random bytes of code are highly unlikely to execute successfully. When binary code is injected by an attacker and executes, it is first de-randomized by RISE. Because the attack code was never pre-randomized, the effect of de-randomizing is to transform the attack code into a random byte string. This is invisible to the interpretation engine, which will attempt to translate, and possibly execute, the string. If the code executes at all, it clearly will not have the effect intended by the attacker. However, there is some chance that the random bytes might correspond to an executable sequence, and an even smaller chance that the executed sequence of random bytes could cause damage. In this chapter I measure the likelihood of these events under several different assumptions, and develop theoretical estimates.

The approach presented here is to identify the possible actions that randomly formed instructions in a sequence could perform and then to calculate the probabilities for these different events. There are several broad classes of events to be considered: illegal instructions that lead to an error signal, valid execution sequences that lead to an infinite loop

or a branch into valid code, and other kinds of errors. Because of subtle complications involved in the calculations simplifying assumptions are made. The simplifications lead to a conservative estimate of the risk of executing random byte sequences.

A theoretical analysis is important for several reasons. Diversified code techniques of various sorts and at various levels are likely to become more common. It is necessary to understand how much protection they confer. In addition, it will be helpful to predict the effect of code diversity on new architectures before they are built. For example, analysis allows us to predict how much increase in safety could be achieved by expanding the size of the instruction space by a fixed amount.

Section 4.1 presents a classification of the behaviors of random sequences. Based on this classification model, Section 4.2 documents the experiments executed to determine the possible outcomes of the execution of random sequences, and Section 4.3 presents theoretical models for the behavior of executing random sequences. Finally, Section 4.4 summarizes the results of this chapter.

4.1 Possible Behaviors of Random Byte Sequences

I start the risk assessment for the execution of random bytes by characterizing the possible events associated with a generic processor or emulator attempting to execute a random symbol. The term *symbol* is used to refer to a potential execution unit, because a symbol's length in bytes varies across different architectures. For example, instruction length in the PowerPC architecture is exactly four bytes and in the IA32 it can vary between one and seventeen bytes. Thus, we adopt the following definitions:

a. A *symbol* is a string of l bytes, which may or may not belong to the instruction set. In a RISC architecture, the string will always be of the same length, while for CISC it will be of variable length.

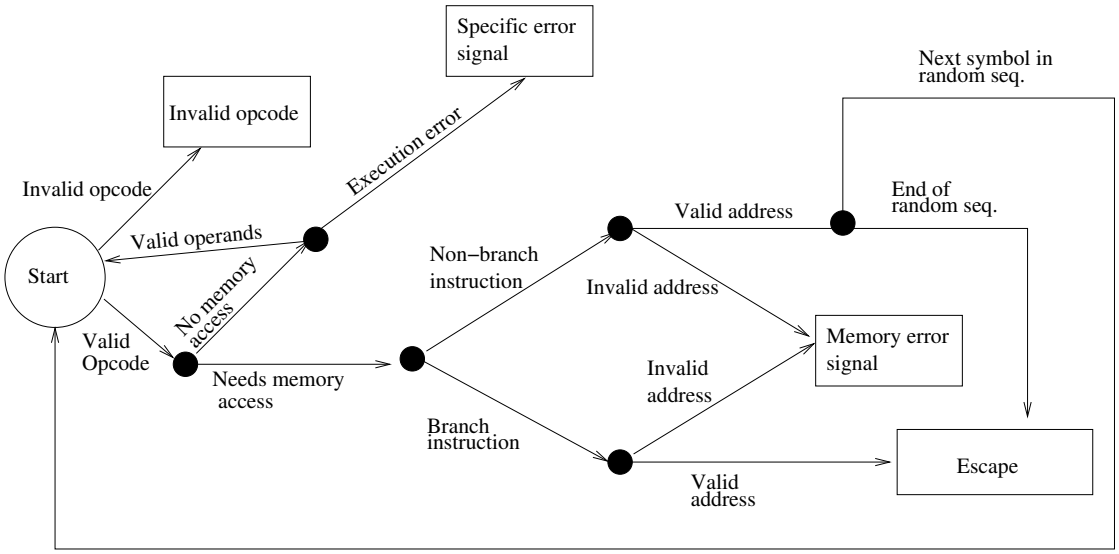


Figure 4.1: State diagram for random code execution. The graph depicts the possible outcomes of executing a single random symbol. For variable-length instruction sets, the Start state represents the reading of bytes until a non-ambiguous decision about the identity of the symbol can be made.

b. An *instruction* is a symbol that belongs to the instruction set.

In RISE there is no explicit recognition of an attack, and success is measured by how quickly and safely the attacked process is terminated. Process termination occurs when an error condition is generated by the execution of random symbols. Thus, the relevant questions are:

a. How soon will the process *crash* after it begins executing random symbols? (Ideally, in the first symbol.)

b. What is the probability that an execution of random bytes will branch to valid code or enter an infinite loop (*escape*)? (Ideally, 0.)

Figure 4.1 illustrates the possible outcomes of executing a single random symbol. There are three classes of outcome: an error which generates a signal, a branch into ex-

ecutable memory in the process space that does not terminate in an error signal (called *escape*), and the simple execution of the symbol with the program pointer moving to the next symbol in the sequence. Graph traversal always begins in the *start* state, and proceeds until a terminating node is reached (*memory error signal*, *instruction-specific error signal*, *escape*, or *start*).

The term *crash* refers to any error signal (the states labeled *invalid opcode*, *specific error signal*, and *memory error signal* in Figure 4.1). Error signals do not necessarily cause process termination due to error, because the process could have defined handlers for some of the error signals. For the purpose of this analysis, it is assumed that protected processes have reasonable signal handlers, which terminate the process after receiving such a signal. This outcome is included in the event *crash*.

The term *escape* describes a branch from the sequential flow of execution inside the random code sequence to any executable memory location. This event occurs when the Instruction Pointer (IP) is modified by random instructions to point either to a location inside the executable code of the process, or to a location in a data section marked as executable even if it does not typically contain code.

An *error signal* is generated when the processor attempts to decode or execute a random symbol in the following cases:

a. Illegal instruction: The symbol has no further ambiguity and it does not correspond to a defined instruction. The per-symbol probability of this event depends solely on the density of the instruction set. An illegal instruction is signaled for undefined opcodes, illegal combinations of opcode and operand specifications, reserved opcodes, and opcodes undefined for a particular configuration (e.g., a 64-bit instruction on a 32-bit implementation of the PowerPC architecture).

b. Illegal read/write: The instruction is legal, but it attempts to access a memory page to which it does not have the required operation privileges, or the page is outside the

process' virtual memory.

c. Operation error: Execution fails because the process state has not been properly prepared for the instruction; e.g., division by 0, memory errors during a string operation, accessing an invalid port, or invoking a nonexistent interrupt.

d. Illegal branch: The instruction is of the control transfer type and attempts to branch into a non-executable or non-allocated area.

e. Operation not permitted: A legal instruction fails because the rights of the owner process do not allow its execution, e.g., an attempt to use a privileged instruction in user mode.

There are several complications associated with branch instructions, depending on the target address of the branch. The assumption is that the only dangerous class of branch is a correctly invoked system call. The probability of randomly invoking a system call in Linux is $\frac{1}{256} \times \frac{1}{256} \approx 1.52 \times 10^{-5}$ for IA32, and at most $\frac{1}{2^{32}} \approx 2.33 \times 10^{-10}$ for the 32-bit PowerPC. The IA32 calculation stems from the fact that a system call requires two different values to execute: a system call number (for which one byte is allocated), and an interrupt number (usually INT 0x80 in Linux) for which another, independent byte is allocated. This probabilities do not even take into account that the arguments be reasonable. Alternatively, a process failure could remain hidden from an external observer, and it will be shown that this is the likeliest event.

A branch into the executable code of the process (ignoring alignment issues) will likely result in the execution of at least some instructions, and will perhaps lead to an infinite loop. This is an undesirable event because it hides the attack attempt even if it does not damage permanent data structures. Successful branches into executable areas (random or non-random) are modeled as always leading to the *escape* state in Figure 4.1. This conservative assumption allows us to estimate how many attack instances will not be immediately detected. These 'escapes' do not execute hostile code. They are simply attack

instances that are likely not to be immediately observed by an external process monitor. The probability of a branch resulting in a crash or an escape depends at least in part on the size of the executing process, and this quantity is a parameter in the calculations.

Different types of branches have different probabilities of reaching valid code. For example, if a branch has the destination specified as a full address constant (*immediate*) in the instruction itself, it will be randomized, and the probability of landing in valid code will depend only on the density of valid code in the total address space, which tends to be low. A return takes the branching address from the current stack pointer, which has a high probability of pointing to a real process return address.

These many possibilities are modeled by dividing memory accesses, for both branch and non-branch instructions into two broad classes:

a. Process-state-dominated: When the randomized exploit begins executing, the only part of the process that has been altered is the memory which holds the attack code. Most of the process state (e.g., the contents of the registers, data memory, and stack) remains intact and consistent. However, I do not have good estimates of the probability that using these values from registers and memory will cause an error. So, I arbitrarily assign probabilities for these values and explore the sensitivity of the system to different probabilities. Experimentally I know that most memory accesses fail (see Figure 4.2).

b. Immediate-dominated: If a branch calculates the target address based on a full-address size immediate, it can be assumed that the probability of execution depends on the memory occupancy of the process, because the immediate is just another random number generated by the application of the mask to the attack code.

This classification is later used in empirical studies of random code execution (Section 4.2). These experiments provide evidence that most processes terminate quickly when

random code sequences are inserted. Models for the execution of random IA32 and PowerPC instructions (Section 4.3) are presented, which allows us to validate the experiments and provides a framework for future analysis of other architectures.

4.2 Empirical testing

Two kinds of experiments were performed: (1) execution of random blocks of bytes on native processors, and (2) execution of real attacks in RISE on IA32.

4.2.1 Executing blocks of random code

I wrote a simple C program that executes blocks of random bytes. The block of random bytes simulates a randomized exploit running under RISE. The program was then tested for different block sizes (the 'exploit') and different degrees of process space occupancy. The program allocates a pre-specified amount of memory (determined by the *filler size* parameter) and fills it with the machine code for no operation (NOP). The block of random bytes is positioned in the middle of the filler memory.

Figure 4.2 depicts the observed frequency of the events defined in Section 4.1. There is a preponderance of memory access errors in both architectures, although the less dense PowerPC has an almost equal frequency of illegal instructions. Illegal instructions occur infrequently in the IA32 case. In both architectures, about one-third of legal branch instructions fail because of an invalid memory address, and two-thirds manage to execute the branch. Conditional branches form the majority of branch instructions in most architectures, and these branches have a high probability of executing because of their very short relative offsets.

Because execution probabilities could be affected by the memory occupancy of the

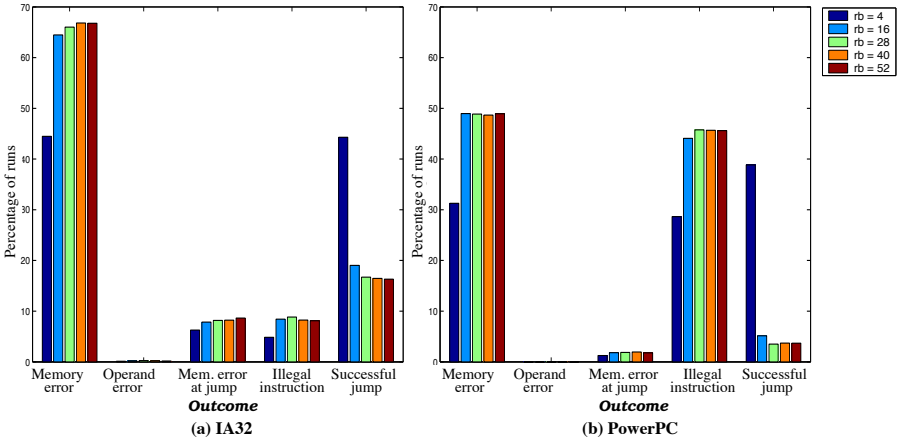


Figure 4.2: Executing random blocks on native processors. The plots show the distribution of runs by type of outcome for (a) IA32 and (b) Power PC. Each color corresponds to a different random block size (*rb*): 4, 16, 28, 40, and 52 bytes. The filler is set such that the total process density is 5% of the possible 2^{32} address space. The experiment was run under the Linux operating system.

process, different process memory sizes were tested. The process sizes used are expressed as fractions of the total possible 2^{32} address space (Table 4.1).

Process memory density (as a fraction of 2^{32} bytes)	0.00029	0.00360	0.01023	0.02349	0.05

Table 4.1: Process memory densities (relative to process size): Values are expressed as fractions of the total possible 2^{32} address space. They are based on observed process memory used in two busy IA32 Linux systems over a period of two days.

Each execution takes place inside GDB (the GNU debugger), single-stepping until either a signal occurs or more than 100 instructions have been executed. Information is collected about type of instruction, addresses, and types of signals during the run. This scenario was ran with 10,000 different seeds, 5 random block sizes (4, 8, 24, 40, and 56 bytes), and 5 total process densities (see Table 4.1), both for the PowerPC and the IA32.

Figure 4.3 plots the fraction of runs that *escaped* according to the definition of escape

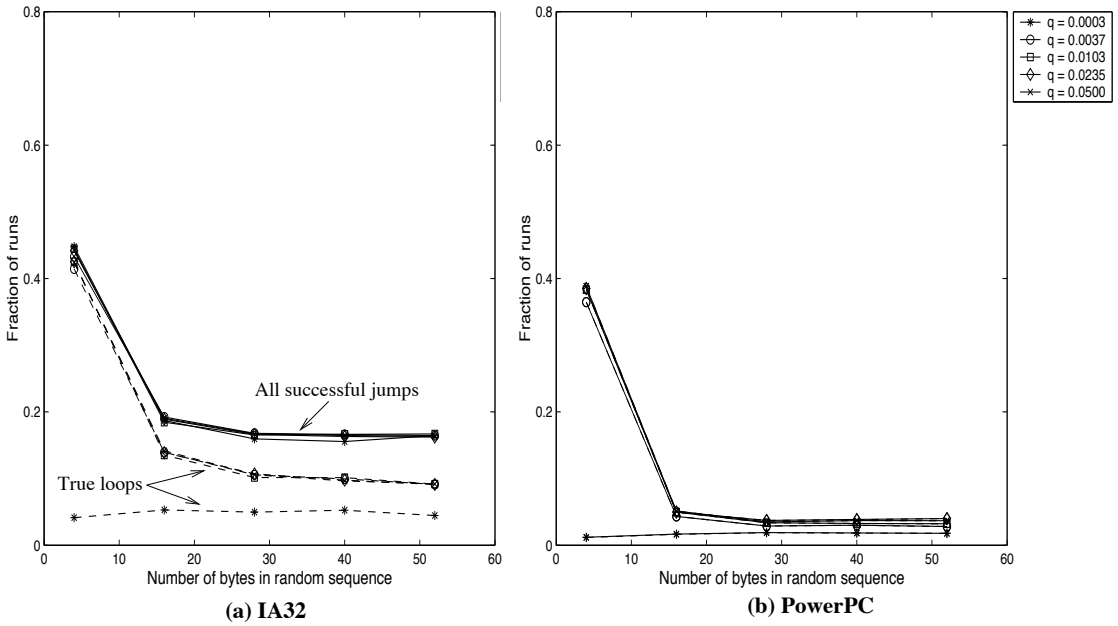


Figure 4.3: Probability that random code escapes when executed for different block sizes (the x-axis) for (a) IA32 and (b) Power PC. Block size is the length of the sequence of random bytes inserted into the process. Each set of connected points represents a different memory density (q). Solid lines represent the fraction of runs that escaped under the definition of escape, and dotted lines show the fraction of 'true' escaped executions (those that did not fail after escaping from the exploit area).

(given in Section 4.1) for different memory densities. An execution was counted as an escape if a jump was executed and did not fail immediately (that is, it jumped to an executable section of the code). In addition, it shows the proportion of escapes that did not crash within a few bytes of the exploit area ('true' escapes: for example when the execution is trapped into an infinite loop). Escapes that continued executing for more than 100 instructions were terminated. The figure shows that for realistic block sizes (over 45 bytes), the proportion of true escapes is under 10% (IA32). In the Power PC case, although the fraction of escaped runs is smaller, most of the escapes do not fail afterwards, so the curves overlap.

A second observation (not shown) is that memory density has a negligible effect on

the probability of escape, even though the environment was specially designed to maximizes successful escapes. This is likely because the process sizes are still relatively small compared to the total address space and because only a minority of memory accesses are affected by this density (those that are immediate-dominated).

Figure 4.4 shows the proportion of failed runs that die after executing exactly *n* instructions. On the right side of the graph, the proportion of escaped vs. failed runs is shown for comparison. Each instruction length bar is comprised of five sub-bars, one for each simulated attack size. All are plotted to show that the size of the attack has almost no effect on the number of instructions executed, except for very small sizes. On the IA32, more than 90% of all failed runs died after executing at most six instructions and in no case did the execution continue for more than 23 instructions. The effect is even more dramatic on the Power PC, where 90% of all failed runs executed for fewer than three instructions, and the longest failed run executed only ten instructions.

4.2.2 Executing real attacks under RISE

I ran several vulnerable applications under RISE and attacked them repeatedly over the network, measuring how long it took them to fail. I also tested the two synthetic attacks from [39]. In this case the attack and the exploit are in the same program, so they were directly executed in RISE for 10,000 times each. Table 4.2 summarizes the results of these experiments. The real attacks fail within an average of two to three instructions (column 4). Column 3 shows how many attack instances we ran (each with a different random seed for masking) to compute the average. As column 5 shows, most attack instances crashed instead of escaping. The synthetic attacks averaged just under two instructions before process failure. No execution of any of the attacks was able to spawn a shell.

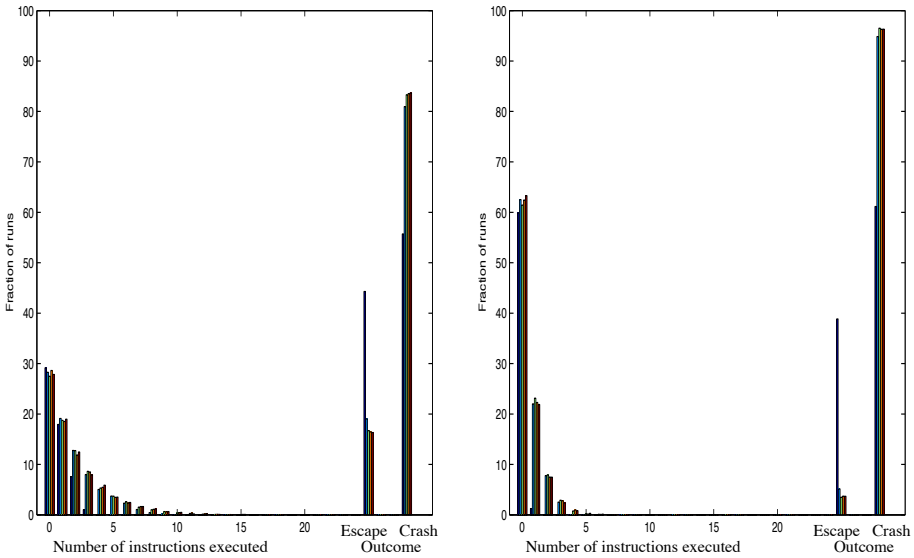


Figure 4.4: Proportion of runs that fail after exactly n instructions, with memory density 0.05, for (a) IA32 and (b) PowerPC. On the right, the proportion of escaped vs. crashed runs is presented for comparison. Each instruction length bar is composed by five sub-bars, one for each random block (simulated attack) sizes 4, 16, 28, 40 and 52 bytes, left to right.

4.3 RISE Safety: Theoretical Analysis

This section develops theoretical estimates of RISE safety and compares them with the experiments reported in the previous section. In the case of a variable-size instruction set, such as the IA32, the aggregate probabilities are calculated using a Markov chain. In the case of a uniform-length instruction set, such as the PowerPC, the probabilities can be computed directly.

4.3.1 IA32 Instruction Set

For the IA32 instruction set, which is a CISC architecture, I used the published instruction set specification [54] to build a Markov chain used to calculate the escape probability of

Attack Name	**Application**	**No. of attacks**	**Avg. no. of insns.**	**Crashed before escape**
Named NXT Resource Record Overflow	Bind 8.2.1-7	101	2.24	85.14%
rpc.statd format string	nfs-utils 0.1.6-2	102	2.06	85.29%
Samba trans2 exploit	smbd 2.2.1a	81	3.13	73.00%
Synthetic heap exploit	N/A	10,131	1.98	93.93%
Synthetic stack exploit	N/A	10,017	1.98	93.30%

Table 4.2: Survival time in executed instructions for attack codes in real applications running under RISE. Column 4 gives the average number of instructions executed before failure (for instances that did not 'escape'), and column 5 summarizes the percentage of runs crashing (instead of 'escaping').

a sequence of m random bytes (with byte-length $b = 8$ bits). The analysis is based on the graph of event categories shown in Figure 4.1, but it is specialized to include the byte-to-byte symbol recognition transitions. A transition is defined as the reading of a byte by the processor, and the states describe either specific positions within instructions or exceptions. Appendix A.1 provides the specifics of this particular Markov chain encoding.

Apart from the complexity of encoding the large and diverse IA32 instruction set, the major difficulty in the model is the decision of what to do when a symbol crosses the boundary of the exploit area. It is conceivable that the result of the interpretation is still ambiguous at the byte that crosses the border. However, the model needs to decide what happens to the execution at that moment. This his situation is modeled using both extremes: A *loose escape* declares the execution as an escape if the bytes interpreted up to the crossing point have not caused a crash; a *strict escape* declares that the execution of the partial instruction ends in crash. A characterization of the states in terms of the Markov chain is in Appendix A.1.

Figure 4.5 shows the probability of escape as a function of increasing sequence length for both loose and strict criteria of escape for a fixed memory density (0.05), and for different probabilities of a process state-dominated memory access to execute. The plot reveals several interesting phenomena.

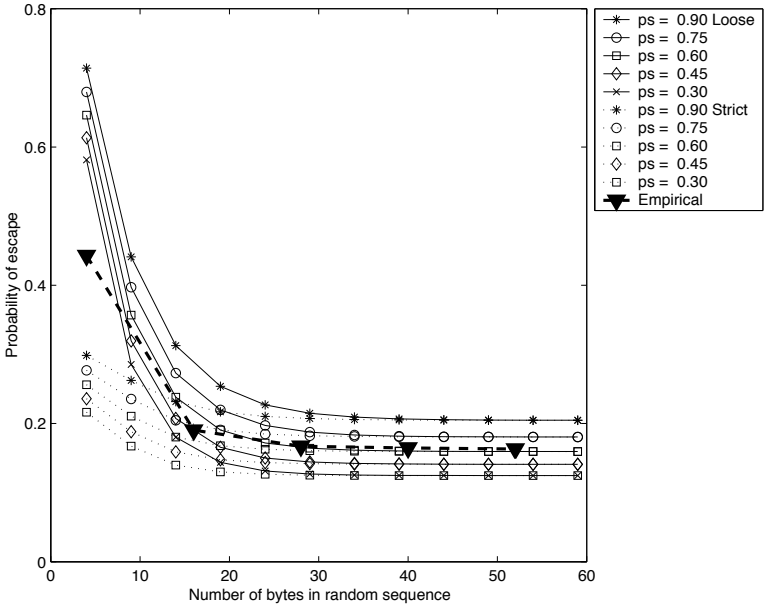


Figure 4.5: Theoretical analysis of IA32 escape probability: The x-axis is the number of bytes in the random sequence, and the y-axis is the probability of escaping from a random string of m bytes. Each connected set of plotted points corresponds to one assumed probability of successfully executing a process-state-dominated memory access (p_s), with either Strict or Loose criterion of escape. The memory density is fixed at 0.05. For comparison with empirical data, the dashed line with triangles marks the observed average frequency of successful jumps (data taken from Figure 4.3 for the IA32 and memory density $q = 0.05$).

First, the plots converge to a steady state quickly—in less than twenty bytes. This is consistent with the empirical data. Notably, the probability of escape converges to a non-zero value. This means that independently of exploit or process size, there will always be a non-zero probability that a sequence of random code will escape.

A second observation revealed by the plot is the relatively small difference between the loose and strict criteria for escape. The main difference between both cases is how

to interpret the last instruction in the sequence if the string has not crashed before the exploit border. Not surprisingly, as sequences get longer, the probability of reaching the last symbol diminishes, so the overall effect of an ambiguous last instruction in those few cases is respectively smaller.

A third observation (data not shown in the figure), is that for different memory densities, the escape curves are nearly identical. This means that memory size has almost no effect on the probability of escape at typical process memory occupancies. In part, this reflects the fact that most jumps use process-state-dominated memory accesses. In particular, immediate-dominated memory accesses constitute a very small proportion of the instructions that use memory (only four out of more than 20 types of jumps).

The fourth observation concerns the fact that the first data point in the empirical run (block size of 4 bytes) differs markedly from all the strict and loose predicted curves. Both criteria are extreme cases and the observed behavior is in fact bounded by them. The divergence is most noticeable during the first 10 bytes, as most IA32 instructions have a length between 4 and 10 bytes. As noted before, the curves for loose and strict converge rapidly as the effect of the last instruction becomes less important, and so a much closer fit can be seen with regard to the predicted behavior after 10 bytes, as the bounds become tighter.

The final observation is that the parameter p_s varies less than expected. I was expecting that the empirical data would have an ever-increasing negative slope, given that in principle the entropy of the process would increase as more instructions were executed. Instead, a close fit with $p_s = 0.6$ is found after the first 20 bytes. This supports the assumption that the probability of execution for process-state dominated instructions could be modeled as a constant.

4.3.2 Uniform-length instruction set model

The uniform-length instruction set is simpler to analyze because it does not require conditional probabilities on instruction length. Therefore, The probabilities can be estimated directly without resorting to a Markov chain. The analysis generalizes to any RISC instruction set, but the PowerPC [51] is used as an example.

Set name	Type of instructions in set
U	Undefined instructions.
P	Privileged instructions.
B_{SR}	Small offset, relative branch
L_D	Legal instructions with no memory access and no branching. All branches require memory access, so L_D only contains linear instructions.
L_{MI}	Legal no-branch instructions with immediate-dominated memory access.
B_{MI}	Legal branch instructions with immediate-dominated memory-access.
L_{MP}	Legal no-branch instructions with process-state dominated memory-access.
B_{MP}	Legal branch instructions with process-state dominated memory access.

Table 4.3: Partition of symbols into disjoint sets.

Let all instructions be of length b bits (usually $b = 32$). I calculate the probability of escape from a random string of m symbols $r = r_1 \ldots r_m$, each of length b bits (assumed to be drawn from a uniform distribution of 2^b possible symbols). All possible symbols can be partitioned into disjoint sets with different execution characteristics. Table 4.3 lists the partition chosen. Figure A.1 in Appendix A.3 illustrates the partition in terms of the classification of events given in Section 4.1. $S = U \cup P \cup B_{SR} \cup L_D \cup L_{MI} \cup B_{MI} \cup L_{MP} \cup B_{MP}$ is the set of all possible symbols that can be formed with b bits. $|S| = 2^b$. The probability that a symbol s belongs to any given set I (where I can be any one of $U, P, B_{SR}, L_D, L_{MI}, B_{MI}, L_{MP}$ or B_{MP}) is given by $P\{s \in I\} = P(I) = \frac{|I|}{2^b}$.

If there are a bits for addressing (and consequently the size of the address space is 2^a): E_I is the event that a symbol belonging to set I executes; M_t is the total memory space allocated to the process; M_e is the total executable memory of the process; and p_s is the probability that a memory access dominated by the processor state succeeds, then the probabilities of successful execution for instructions in each set are:

For illegal and privileged opcodes, $P(E_U) = P(E_P) = 0$.

For the remaining legal opcodes, $P(E_{L_D}) = P(E_{B_{SR}}) = 1$; $P(E_{L_{MI}}) = \frac{M_t}{2^a}$; $P(E_{B_{MI}}) = \frac{M_e}{2^a}$; $P(E_{L_{MP}}) = p_s$ and $P(E_{B_{MP}}) = p_s$.

The probability of a successful branch (escape) out of a sequence of n random bytes is very important. Let X_n denote the event that an execution escapes at *exactly* symbol n. This event requires that $n-1$ instructions execute without branching and that the n-th instruction branches successfully. In consequence, $P(X_n) = (P(L))^{n-1}P(E)$, where $P(L) = P(L_D) + P(L_{MI})P(E_{L_{MI}}) + P(L_{MP})$ is the probability that a symbol executes a successful linear instruction, and $P(E) = P(B_{MI})P(E_{B_{MI}}) + P(B_{MP}) + P(B_{SR})$ is the probability that a symbol executes a valid branch.

If $X_n^\star$ is the event that the execution of a random string $r = r_1 \ldots r_n$ escapes, its probability $P(X_n^\star)$ is given by (Appendix A.3):

$$P(X_n^\star) = P(E)\frac{1-P(L)^{n+1}}{1-P(L)} + P(L)^n$$

$P(X_n^\star)$ is plotted in Figure 4.6 for different values of p_s, increasing random code sizes and a given memory density (0.05 as in the IA32 case). The comparable data points from the experiments are shown for comparison. I did not plot results for different memory

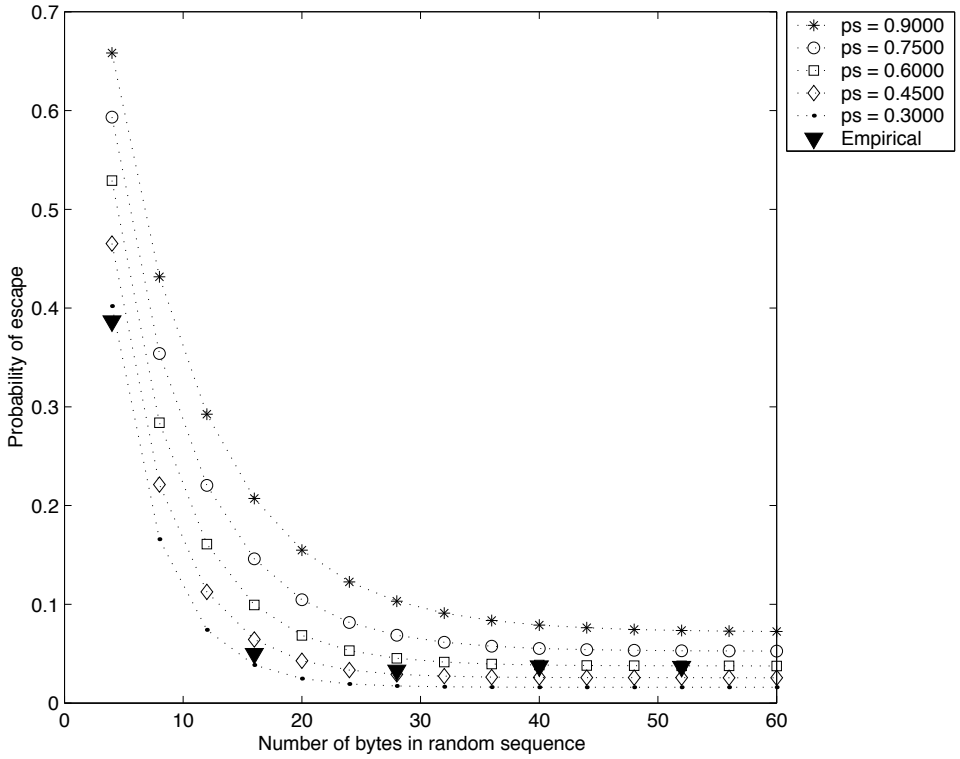


Figure 4.6: Theoretical probability of escape for a random string of n symbols. Each curve plots a different probability of executing a process-state-determined memory access (p_s) for the PowerPC uniform-length instruction set. Process memory occupancy is fixed at 0.05. The large triangles are the measured data points for the given memory occupancy (data taken from Figure 4.3 for the PowerPC and memory density $q = 0.05$), and the dotted lines are the predicted probabilities of escape.

densities because the difference among the curves is negligible. The figure shows that the theoretical analysis agrees with the experimental results. The parameters were calculated from the published documentation of the PowerPC instruction set [51], for the 32-bit case: $b = 32$, $a = 32$, $P(L_D) \approx 0.25$, $P(L_{MI}) = 0$, $P(L_{MP}) \approx 0.375$, $P(B_{MI}) \approx 0.015$, $P(B_{MP}) \approx 0.030$, $P(B_{SR}) \approx 0.008$.

It can be seen that the probability of escape converges to a nonzero value. For a uniform-length instruction set, this value can be calculated as:

$\lim_{n\to\infty} P(X_n^\star) = \frac{P(E)}{1-P(L)}$

The limit value of $P(X_n^\star)$ is the lower bound on the probability of a sequence of length n escaping. It is independent of n, so larger exploit sizes are no more likely to fail than smaller ones in the long run. It is larger than zero for any architecture in which the probability of successful execution of a jump to a random location is larger than 0.

4.4 Summary

In this chapter I proposed a model for the possible actions that random bytes could take when executed in a processor. Based on this model, I obtained two different theoretical approximation to the probabilities of the dangerous classes of failures. A somewhat surprising result is that the probability of a random sequence of bytes not crashing (therefore leaving the process suspended for no apparent reason) never goes below some constant that varies with the instruction set. The chapter also documents a large number of experiments executing random bytes on live processors, and monitoring the length of bytes executed before failure, the type of failure, and the percentage of cases where the sequence entered an infinite loop and left the process hanging without an explicit faiulre. It was shown that there is correspondence between the theoretical predictions and the observed results, thus validating the models.

Chapter 5

Protocol Parameter Randomization

The extensive interface randomization discussed in the previous two chapters is an effective host-level diversification. In contrast, this chapter and the next will focus on implementation diversity to protect the network level, a population-level diversification. The technique described in this chapter achieves implementation diversity by modifying protocol parameters. In particular, the chapter reports on a Transmission Control Protocol (TCP) parameter randomization designed to mitigate the class of Denial of Service attacks (DoS) that slow down communication protocols by causing them to retreat into recovery states.

Section 5.1 describes the target threat and the feasibility of exploiting fixed parameters. Section 5.2 explains the TCP congestion control algorithm and the calculation of the retransmission timeout, to illustrate how the fixed point is exploited by the *shrew* attack. The *shrew* is an example of a general class of attacks on TCP parameters. The attack itself is described in Section 5.3. In Section 5.4 the defense hypothesis is stated and all relevant parameters are examined. Specifically, a discussion is presented about how, when and what values to randomize. The methodology employed to obtain the data is presented in Section 5.5. The criteria for claiming success or failure are stated in Section 5.6, and

the analysis of the results with respect to these criteria is presented in Section 5.7. A brief recount of background and related work is presented in Section 5.8. Suggestions for future work are discussed in Section 5.9. Finally, Section 5.10 summarizes the findings of this chapter.

5.1 Target threat

A communication protocol defines the set of rules that two or more entities use to interact. In practice, most protocols are explicitly or implicitly defined as extended finite-state machines (FSMs), with state-by-state rules defining: how to interpret received messages (or lack of them), what messages should be sent, how to update state variables, and under what conditions the FSM transitions to another state. Ideally, the conditions should refer to the state of the real environment, but in practice, the real situation is estimated through a series of proxies which rely on imperfect measurements and require artificial mechanisms to recover from known proxy failures. Most parameters used by the estimators are somewhat arbitrary. Even when these parameters are adjustable, they often remain set to default values. At times, implementations go further and hardcode default values, which makes their tuning extremely difficult.

These parameters can create blind spots in the estimators which are exploitable. The fact that the parameters are fixed makes it much easier to fine-tune attacks by adjusting timing or content to deceive the estimators. There are several ways to exploit fixed parameters. The focus here is on a particular type of attack that forces the FSM to enter suboptimal states. Given that any communication protocol requires emergency states in order to recover from problematic situations in the environment, the goal of the attacker is to modify the environment in such a way that the estimators signal a non-existent emergency situation. This causes the protocol to go incorrectly into one of the slower, recovery-mode states. A slight variation of this attack is to cause the protocol to continuously switch

between states, or to restart any one of them. This type of attack is a form of Denial of Service (DoS) attack. Although these attacks do not gain control of the attacked hosts, they are a serious problem because they deny legitimate users access to resources.

There are many arbitrary parameters in the implementation of any protocol, but to prove the usefulness of a randomization defense it is necessary to use a real attack. I use the *shrew* exploit presented by Kuzmanovic and Knightly [65], which uses carefully synchronized traffic surges to force TCP to enter the *slow-start* mode repeatedly. The parameter being exploited in this case is the minRTO parameter, which is part of the TCP timeout (RTO) calculation and was originally designed as part of the congestion control mechanism. I will first describe the TCP congestion control algorithm, with emphasis on the calculation of the RTO, followed by the specifics of the attack.

5.2 Congestion Control in TCP

The Transmission Control Protocol (TCP) [2] is a reliable, transport layer data transfer protocol. It uses adjustable timeouts to determine when data has been lost, and when to retransmit. After a connection has been established, and as long as a connection is not considered lost or finished, it sends and receives packets, transitioning between the following four implicit states: *slow start* (SS), *congestion avoidance* (CA), *fast retransmit* (FRT) and *fast recovery* (FRE). We are only going to be concerned with the SS and CA states here.

At a minimum, a windowing protocol must have two state variables: the size of the sending window, and an approximate round trip time to determine the size of the timeouts. These parameters can be dynamic or static, but must be defined. In TCP, they are both dynamic and are maintained by using several state variables, which are described below.

After a connection is established, TCP sets the timeout clock (RTO) to some fixed

value (3 s in most implementations), starts the transmission window with a constant value given by the Initial Window (IW) parameter (usually one or two full segments [1]), and goes into slow start mode (SS). The initial connection values for RTO and the transmission window are used exactly once, after the connection is established they are of no interest for the problem being stated.

Once a valid round trip time [2] (RTT) measurement is made, the state variable of interest becomes the Congestion Window (CW), which starts its life with a value of 1 segment. During SS, TCP increases the congestion window exponentially as long as data is not lost or the limit is not reached. Each time a segment is acknowledged, the window grows by one. Given that all segments in a window are sent in bursts, in the absence of errors, this amounts to duplicating the size of the window each RTT.

In the absence of errors, this exponential growth stops when the transmitting window reaches either (1) the threshold between slow start and congestion avoidance (SSTRESH) or (2) the receiver-advertised window (RWND). Most implementations, in fact start SSTRESH at the same value of RWND. Once this point is reached, TCP transitions into the congestion avoidance (CA) state. During CA, the transmission window is incremented by one, each RTT (not at each ACK), thus increasing the window size only linearly. CA stops increasing the transmission window once it reaches RWND.

At any point, if a timeout happens, the value of SSTRESH is cut in half [3], and, more importantly, CW is set to LW, the Loss Window (as opposed to the original IW when the connection starts), which can never be larger than one segment. After the un-acknowledged

[1] TCP uses a transmission buffer measured in bytes, but in reality all variables refer to it in units of segment size (TCP packet size) being used, i.e., a window size of m means that it could contain m segments, *not* m bytes. If the segment size is r bytes, then the window contains $m \times r$ bytes. I use the convention of the size measured in segments when referring to window sizes.

[2] TCP measures the RTT as the length of time starting at packet transmission and finishing at its acknowledgement packet (ACK) reception. Only ACKs for non-retransmitted packets are considered valid for a RTT measurement.

[3] The calculation is slightly more complicated, but this approximation meets the needs here.

packets are retransmitted, TCP returns to the SS state. It is important to note that this is caused only by a timeout. Another way of detecting loss in TCP is the receipt of four identical ACKs without any intermediate packets. This particular class of loss is managed by the FRT and FRE states, which are not pertinent for my attack scenario. They eventually return to SS, the state in which we are interested.

An additional consequence of a timeout is that RTO, the retransmission timer is doubled. The return to an RTO reflecting a measured RTT happens only after a valid measurement, that is, the reception of an ACK for new data. The recording of a valid measurement (RTT_{sample}) is used to keep an up-to-date estimate of the round trip time using the smoothed round trip time (SRTT) and the round trip variance (RTTVAR), as follows:

$$SRTT \leftarrow (1-\alpha)SRTT + \alpha RTT_{sample} \tag{5.1}$$

$$RTTVAR \leftarrow (1-\beta)RTTVAR + \beta|SRTT - RTT_{sample}| \tag{5.2}$$

$$RTO \leftarrow max(minRTO, SRTT + max(G, k*RTTVAR)) \tag{5.3}$$

where $\alpha = \frac{1}{8}, \beta = \frac{1}{4}, k = 4$, and $minRTO = 1s$ as recommended in the relevant RFCs [4] ([2, 83]). The parameter G refers to the clock granularity in the host.

In summary, a timeout causes a return to SS, and a dramatic window decrease. If it were possible somehow to guess when the lost packets were being retransmitted and cause them to be dropped, then the affected data flow could be kept from sending any new data forever. This is the basis of the *shrew* attack which is described next.

[4] All Internet protocols are described in RFCs *Request for Comments* documents, which are generated and maintained by the Internet community.

5.3 The *shrew* attack

The critical observation made by Kuzmanovic and Knightly is because most round trip times on the Internet are below 1 s, Equation 5.3 returns minRTO most of the time. By inducing losses with a period of minRTO, it is possible to force most flows to continuously reset into the slow-start (SS) state. This can bring the throughput to zero as the flow will continue to attempt to resend the packets it had in flight when the first loss happened, and will not move on to new data. In a rarified environment with only one flow at a time, the attack can bring the bandwidth to zero even if the attacker keeps decreasing exponentially the frequency of the attack after each estimated timeout, given TCP's exponential backoff. In a more realistic environment with several flows in different timescales and many packets in flight, a frequency of one burst per minRTO is necessary and sufficient to implement the attack.

The attack is thus based on finely synchronized, high-rate low-duration packet bursts. The rate of the burst must be enough to induce loss, so it has to be equal to the wire rate (e.g. 100 Mbps in a Fast Ethernet network). The length of the burst (b) is chosen to be larger than most real RTTs in the target system, to ensure that at least one packet of most flows will be dropped. Finally, the burst is sent each T seconds ($T >> b$). T is referred to as the *period of the attack*. In a homogeneous minRTO system, if RTT_{real} is the real round trip time of a flow, then all flows with $RTT_{real} < b$ will suffer an extreme reduction in their throughput by this attack, as long as the T is synchronized with the minRTO.

Figure 5.1 depicts the mechanism of the attack. It can be seen that flow 1, whose RTT (real round trip time) is 9 units, partially evades the attack. Given that its ACKs take more than the burst duration to arrive, the transmissions are unsynchronized enough so that some packets evade loss. In contrast, flow 2 is forever synchronized to the burst because the ACK for packet 1 keeps getting lost, as the RTT of the flow is short enough so when the ACK is being transmitted the attack burst is active. Finally, flow 3 exemplifies

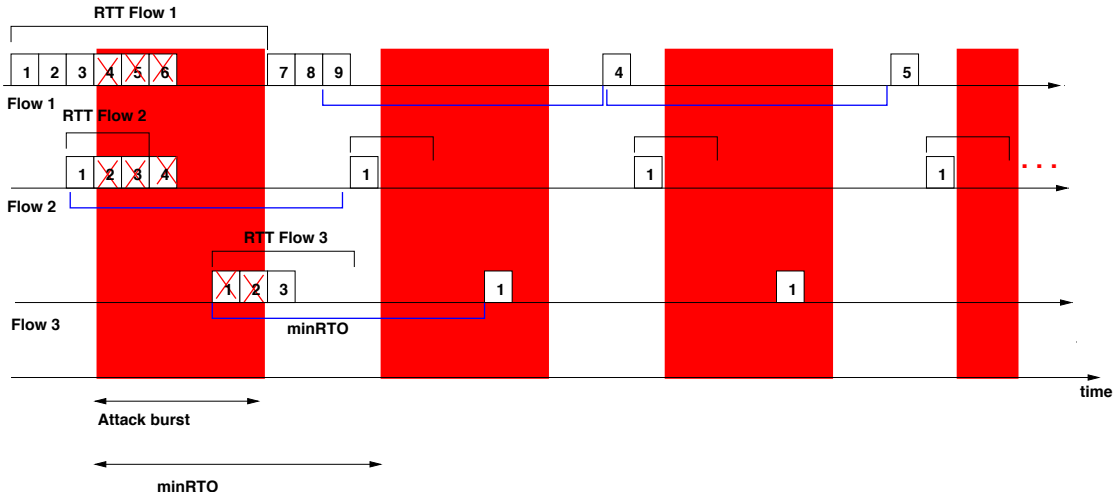


Figure 5.1: The *shrew* attack in action. Shown are three flows with slightly different traffic characteristics. The attack bursts (shadowed areas) cause any packets (and ACKs – not shown) in transit at that moment to be dropped. The RTTs of flows 1,2 and 3 are 9, 3 and 5 units, respectively. The burst duration is 6 units and minRTO is 10.

the direct action of the minRTO synchronization: segment 1 is sent (during a burst) and is lost, so the timer expires at minRTO, which places it again inside a burst, given that it has the exact same period (minRTO). Flows 2 and 3 are effectively stopped.

Throughput is defined as the amount of useful work that can be done, or the amount of data that can be delivered, in a period of time. Normalized throughput (usually denoted ρ) changes the representation to a proportion of the maximum available bandwidth. The normalized, cumulative throughput for several flows using a given minRTO under attack was derived in [65] *for the standard TCP RTO calculation* (Equation 5.3) and is given by:

$$\rho(T) = \frac{\lceil \frac{minRTO}{T} \rceil T - minRTO}{\lceil \frac{minRTO}{T} \rceil T} \tag{5.4}$$

where, T is the period of the attack. As could be expected, it holds for all flows where:

- $b \geq RTT_i$; and
- $minRTO > SRTT_i + 4RTTVAR_i$

The normalized throughput given by Equation 5.4 is going to be used as basis for comparison against the three Linux kernel implementations of the RTO calculations. It will be referred to as `standard TCP (model)` in the text.

The reader is referred to the original *shrew* paper [65] for a stronger proof that the attack works, and that it does so under a wide variety of environments. The paper also explores minRTO randomization, but under conditions that leave the authors unconvinced of its effectiveness. In the next section I will show that, under the correct conditions, a randomization of some of the TCP parameters used for congestion control mitigates the effects of this attack.

5.4 Randomizing TCP congestion control parameters

In order to mitigate the effects of the *shrew* attack, I propose to use a single minRTO per host, which could be changed infrequently (e.g. once a day). A per-flow, rapidly changing random minRTO has problems as shown in [65], and potentially could create the problem of depleting local good-quality sources of randomness. The analysis cited focused on the average composite throughput. In general, I would expect that population-level diversity defenses would not show many benefits in the average. However, I believe that the average is not the only way to evaluate the effectiveness of the defense. If the premise is that it would be better to have some individuals survive even if others are incapacitated, instead of spreading the misery, then the benefits should be also evaluated in terms of how many individuals survive and how different are their responses.

There are three parameters besides minRTO that could be changed in the congestion-

control algorithm, and I will explore their utility as well, but in order for them to have any effect, some changes have to be made in the calculation of the retransmission timeout. It could be observed from Equation 5.3 that all the work to keep the estimate of the round trip time accurate via the SRTT and RTTVAR is useless if the true round trip time of a flow is below minRTO, as the value assigned to the timeout clock will be minRTO. Most round trip times on the Internet are in fact below the recommended 1 s [55]. However, just discarding minRTO or making it very small is not an option for keeping congestion under control as shown in [1].

In the Linux TCP implementation (for kernels 2.4.2x), at least two significant modifications to the TCP specifications for congestion control are made. The first is to use a minRTO of 20 ms (although it is stated in RFC2988 [83] that the 1 s value is a *'should'* for a complying TCP implementation). The second Linux modification changes Equation 5.3 in the RTO calculation from

$$RTO \leftarrow max(minRTO, SRTT + max(G, k * RTTVAR))$$

to

$$RTO \leftarrow SRTT + max(minRTO, k * RTTVAR) \tag{5.5}$$

The TCP clock granularity, G is considered at the start of the calculation. If the measurement is '0', that is, below 10 ms, as $G = 10ms$ in Linux, then the measurement is reset as $RTT_{measurement} \leftarrow G$.

Therefore, in Linux, at any moment, the RTO is always going to be larger than minRTO, varying by the estimated average RTT time. This implementation gives some voice to the calculation of RTT, and modifies the environment for the evaluation of the randomization impact: in theory at least, it would make it more difficult for the attacker to find

effective attack periods. I will use the label `Kernel 1` to refer to a kernel using Equation 5.5 for the RTO calculation.

The *shrew* attack preys on the grouping around a maximum, which makes all timeouts synchronize in periods around $minRTO$. The goal of the randomization defense is to spread the timeouts at the same time as keeping a reasonably accurate estimate of the round trip time. Given that (a) the standard calculation, and the Linux implementation both have a maximum grouping as part of the equation, and (b) groupings could be exploited, even if the actual amount used as maximum is varied; it was important to explore an implementation that did not have a grouping. To accomplish this, I modified the Linux implementation of the retransmission timeout (RTO) as described in Equation 5.6.

$$RTO \leftarrow SRTT + k * RTTVAR + minRTO \tag{5.6}$$

This formula also permits different values of α, β and k, which are almost completely smothered in the `kernel 1` configuration (and would be never used in the standard calculation). I will refer to a configuration with Equation 5.6 and the *standard* values of α, β and k as `kernel 2`. A kernel with calculation 5.6 but modified α, β and k is called `kernel 3` (the values of the parameters are then made explicit). Table 5.1 summarizes the RTO calculations to be evaluated, and the labels for the kernels which incorporate them. The common part of the calculation is repeated here for convenience:

$$\begin{aligned} SRTT &\leftarrow (1-\alpha)SRTT + \alpha RTT_{sample} \\ RTTVAR &\leftarrow (1-\beta)RTTVAR + \beta|SRTT - RTT_{sample}| \end{aligned}$$

For minRTO, the range utilized includes all integer values between 0 and 2000 ms,

Label	**RTO calculation**	α	β	k	**Source of data**
`Standard TCP (Model)`	$RTO \leftarrow max(minRTO,$ $SRTT + max(G, k * RTTVAR))$	$\frac{1}{8}$	$\frac{1}{4}$	4	Model (theoretical)
`Kernel 1`	$RTO \leftarrow SRTT +$ $max(minRTO, k * RTTVAR)$	$\frac{1}{8}$	$\frac{1}{4}$	4	Standard Linux Kernel 2.4.2x, via (`iperf`).
`Kernel 2`	$RTO \leftarrow SRTT +$ $k * RTTVAR + minRTO$	$\frac{1}{8}$	$\frac{1}{4}$	4	Modified Linux Kernel 2.4.2x, via (`iperf`).
`Kernel 3`	$RTO \leftarrow SRTT +$ $k * RTTVAR + minRTO$	Varies			Modified Linux Kernel 2.4.2x, via (`iperf`).

Table 5.1: Different versions of RTO calculation being evaluated. Column 1 gives the label to be used through the text for each variant, column 2 shows the formulas used, columns 3 through 5 present the values of the α, β and k parameters used, and the last column explains the source of the data presented in the chapter for each variant.

thus making the network minRTO average equal to a compliant 1 s. The parameters α, β and k control the RTT estimate, and the hypothesis is that they will add some degree of randomness to the RTO times around the fixed (per-host) value of minRTO. In [1], Allman and Paxson examine the feasible ranges for TCP congestion control parameters. I chose to use their ranges, even though some of them are outside the recommended or mandatory ones in the current TCP specification, because according to [1] they deliver reasonable performance. Although the range for minRTOs is chosen so that the average is the required 1 s, the other three parameters are chosen without such constraint. Table 5.2 summarizes the values used for each parameter.

Based on the considerations expressed in this section, the following are the two hypotheses that are going to be tested in this chapter:

- For any of the three explored RTO calculations, a *minRTO* randomization will protect a network by allowing some hosts to have higher throughputs than others by

Parameter name	Recommended	Min	Max
RTO_{min} (s)	1	0	2
α (powers of 2)	$\frac{1}{8}$	0	1
β (powers of 2)	$\frac{1}{4}$	0	1
k (integer)	4	2	16

Table 5.2: Ranges for parameter randomization, used in the TCP congestion control algorithm. Column 1 gives the canonical parameter name, column 2 offers the recommended value and columns 3 and 4 define the minimum and maximum limits for the parameter

unsynchronizing their RTOs from the attack period.

- The proposed kernel modification creates useful RTO diversity, apart from the one achieved solely my modifying the minRTO.

Next section describes the methodology used to test these hypotheses.

5.5 Methodology

The network setup used to conduct the experiments is depicted in Figure 5.2. The attacker induces temporary outages on the pipeline from senders to receivers. Although the attacker here is depicted as being on the same network as the senders, the attack is equally effective with the attacker positioned any one of the sub-networks along the pipeline. It is important to remember that all the attacker needs to do is clog **at least one** of the (finite) queues of the routers and/or switches along the path for sufficient time to induce the loss of all packets for a given RTT.

The 2.4.27 Linux kernel sender TCP code was modified, so as to make each of the four parameters, accessible and modifiable, for each of the kernel versions. The original minRTO was the easiest as it was already a constant in the code. However, α and β were hardcoded *in the implementation*, so I first had to recover the original numerical form,

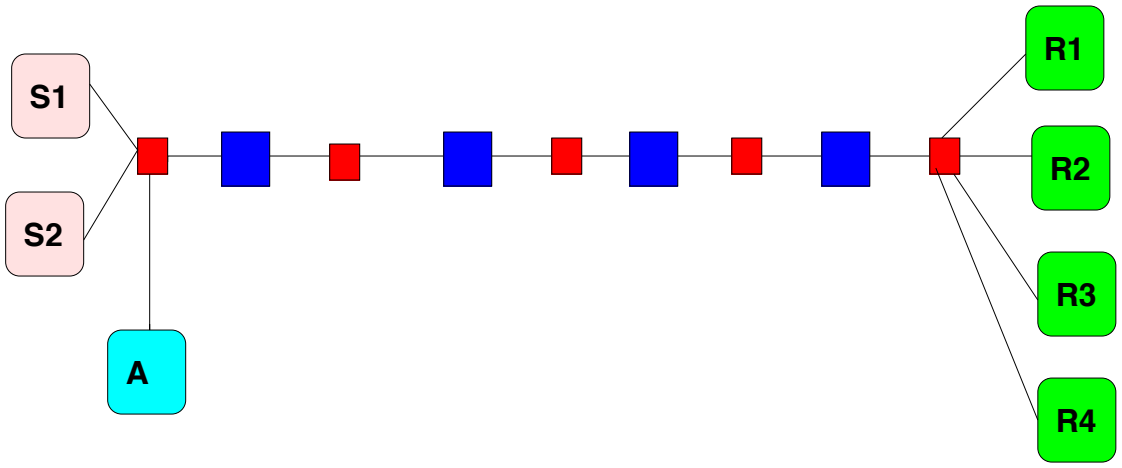


Figure 5.2: Network setup for the minRTO-related DoS attacks. Red squares represent switches, dark blue are routers, green are receiver hosts (R_i), pale pink are sender (victim) hosts (S_i) and the attacker is depicted in cyan (A).

before setting them as modifiable variables. Finally, k was used as a direct constant in the formulas, so I just found all its occurrences and replaced them with a variable. All four parameter variables were made public, and a kernel module was created to modify them as needed, using the `/proc` filesystem. The file is called `/proc/net/tcpParams`, and each parameter is shown as a line. The module operation is depicted in Figure 5.3. The randomization is achieved by retrieving four bytes from `/dev/random`, through the internal kernel interface.

The attack tools were provided by Alexander Kuzmanovic, and he has made them available at his web site `http://www.ece.rice.edu/networks/shrew`. Each attack is defined by: (a) the inter-burst period, (b) the burst duration, and (c) the burst rate. The Internet experiments described in [65] were done at 10 Mbps, while these experiments were done at 100 Mbps. Scaling was performed to create an equivalent set-up. Bursts were 20 ms in length with attacks ranging from 0.2 to 1.5 s inter-burst periods. The *average* bandwidth of an attack is defined by $\frac{bR}{T}$ where b is the burst rate, R is the wire rate (100 Mbps in Fast Ethernet), and T is the attack period. Table 5.3 shows the average

```
gbarrant@victim:~$ ls /proc/net/
arp         drivers   netlink  pfkey  rpc            snmp          tcp        unix
dev         igmp      netstat  raw    rt_cache       sockstat      tcpParams
dev_mcast   mcfilter  packet   route  rt_cache_stat  softnet_stat  udp
```

(a) Location of module in the /proc filesystem.

```
gbarrant@victim:~$ cat /proc/net/tcpParams
Alfa: 1/8
K: 4
Beta: 1/4
minRTO: 20
```

(b) Checking parameter state.

```
victim:~# echo "minRTO=100" > /proc/net/tcpParams
victim:~# cat /proc/net/tcpParams
Alfa: 1/8
K: 4
Beta: 1/4
minRTO: 100
```

(c) Change of one parameter (minRTO i.n the example).

```
victim:~# echo "randomize" > /proc/net/tcpParams
victim:~# cat /proc/net/tcpParams
Alfa: 1/32
K: 8
Beta: 1/128
minRTO: 54
```

(d) Parameter randomization.

Figure 5.3: Kernel module for modifying the four parameters in the randomization. (a) shows the location of the module, (b) presents an inquiry about the state of the parameters, (c) modifies the minRTO, and (d) executes a randomization.

bandwidth of the attacks used for different burst lengths. The rate at which the burst is sent is always 100 Mbps as it has to be the wire rate.

The use of bursts with length 20 ms could be considered insufficient. In the original paper, evidence is presented that the attack operates as a high-pass filter for flows with RTTs (round trip times) larger than the burst length. A burst size of 20 ms would seem to imply that only flows at distances less or equal than 20 ms from the sender would be affected by the attack. Nonetheless, the use of these shorter bursts is legitimate, because distances inside an institutional network tend to be short, and the usage that is most important to protect is the local, high-volume traffic.

T	$b = 20$ **ms**	$b = 100$ **ms**	$b = 200$ **ms**
0.2	10.00	50.00	-
0.3	6.67	33.00	66.67
0.4	5.00	25.00	50.00
0.5	4.00	20.00	40.00
0.6	3.33	16.67	33.33
0.7	2.86	14.29	28.57
0.8	2.50	12.50	25.00
0.9	2.22	11.11	22.22
1.0	2.00	10.00	20.00
1.1	1.82	9.09	18.18
1.2	1.67	8.33	16.67
1.3	1.54	7.69	15.38
1.4	1.43	7.14	14.29
1.5	1.33	6.67	13.33

Table 5.3: Average attack bandwidths (in Mbps) for different burst lengths and attack periods. Burst rate is fixed at 100 Mbps.

An additional reason to use 20 ms bursts is that the testbed runs at 100 Mbps, and for that bandwidth the volume of traffic generated at larger burst sizes triggers load-balancing behaviors at the routers, which make the results of the attacks difficult to interpret. Furthermore, it is against the best interests of the hypothetical attacker to use larger bursts in this environment, as the resulting high bandwidth from a single source can make it relatively easy to detect.

The effects of the attack on legitimate flows are measured using `iperf` between the endpoints to send two different types of traffic. The first type is long transmissions of 200 s with throughput measurements scheduled each 30 s. The second type is a mix of the following simultaneous flows: a 'long' transmission of 200 s, 'medium' length serial transmissions of 10 s each, and 'short' serial transmissions of 1 s per-transmission. Each TCP flow attempts to transmit at the maximum wire rate and the TCP congestion control balances the bandwidth among all simultaneous transmissions. The reported throughput

figures correspond to averages over five iterations for each attack period and type of traffic mix. The throughputs are reported in units of 100 Mbps, hence they range between 0 and 1.

To conduct the attacks, a set of tools were written to synchronize senders, receivers and the attacker using an ad-hoc communications protocol whose control resides at the attacker. Given that the kernel parameters have to be changed at the sender, the sender agent executes as root. The master controller signals the legitimate sender about: the kernel parameters to use, the type of traffic to send, and the traffic destination. It also notifies the receiver about the attack period, mix and parameters (so it saves the measurements correctly). After those preparations, the tool starts the attack and notifies the sender to start the legitimate flow. Notice that the attack has to start before, and end after the legitimate transmission, in order to not record high throughput during non-attack intervals at the beginning and/or end of the attack. The master tool iterates through all the periods and kernel parameters requested in its command line.

The attacks were executed using each of the three kernel versions, with several arbitrary values for α, β and k in the case of `kernel 3`. For each kernel version, the following minRTOs were tested: $\{0.2, 0.3, \ldots, 1.5\}$, with the attack periods as described in Table 5.3, and both traffic types were sent. For each (kernel, traffic mix, minRTO, period) the throughput obtained by each data flow was recorded. This data is analyzed using the criteria for success defined in the next section.

The next section outlines a variety of performance metrics which will be used to analyze the results in Section 5.7.

5.6 When to declare success?

In order to prove that a defense is *effective* against an attack which throttles legitimate throughput, it is necessary to show that the throughput achieved by the system under attack is better than it is without the defense. Additionally, it is desirable that the defense would be *efficient*. If S_D denotes a system using a given defense and S_O defines the same system in its original, undefended condition, then it must be true that **under attack**:

$$\rho(S_D) > \rho(S_O) \tag{5.7}$$

and that under **normal conditions**:

$$\rho(S_D) \sim \rho(S_O) \tag{5.8}$$

where ρ is the standard notation for normalized throughput, the amount of useful data being delivered by the system divided by the available bandwidth.

The classic defense evaluation criteria represented by Equations 5.7 and 5.8 require a definition of the abstract system being studied. For example, it is possible to maximize the throughput of a flow with some characteristics, an application, a host, or a network. In the case of a network, the entity of interest in this chapter, there is an implicit assumption that each node in the network obtains an equal share of the bandwidth. The reason is that network throughput is a difficult quantity to measure as it is an end-system statistic. Until a packet is delivered (inside a host) there is no throughput to talk about. To make models manageable, generally only the *aggregate* traffic is considered, and for experimental data, the throughput is measured at endpoints, and the network composite is estimated from there. The equal distribution holds for TCP, since it does dynamic load-balancing; hence

for n hosts, each host obtains $\frac{1}{n}$ of the total bandwidth. The total throughput of the network can be considered a theoretical $n\frac{1}{n} = 1$.

Under attack, in a homogeneous environment, given that all hosts respond in the same way, the throughput at each host is reduced by some proportion q. Therefore, the network throughput reduces to $n(q\frac{1}{n}) = q$. However, if each node responds differently to the attack, it is not clear what the network bandwidth would be. Given that this is central to showing that the defense works, I use the following approximation. If each host reduces its throughput by some individual q_i, then I assume that the **network throughput estimate** becomes:

$$\rho(S_D) = \sum_i q_i \frac{1}{n} \tag{5.9}$$

where S_D is the network. $\rho(S_D)$ can be less than, equal to, or larger than q, depending on the distribution of q_i.

Equation 5.9 is a rather conservative estimate of the network throughput under attack. When some hosts under-utilize the bandwidth, hosts with a larger q_i occupy it (constrained to each q_i limit). The real throughput would therefore be larger than this estimate. Unfortunately these interactions are difficult to measure or estimate, so when studying the throughput gain I will use the conservative estimate.

To take into account the bandwidth scavenging just described, I have added an additional measure, the dispersion quantifier. Well dispersed q_i will allow for better scavenging and an increase in bandwidth for most hosts. Given that an attack is defined by its period T, I define the per-period (T) dispersion quantifier, $dispersion(T)$, as the standard deviation (σ_T) of the throughputs achieved by different minRTOs during the attack, divided by the standard deviation of the uniform distribution (the optimal dispersion) on the [0,1] range ($\sigma_{uniform_{[0,1]}}$):

$$\begin{aligned} dispersion(T) &= \frac{\sigma_T}{\sigma_{uniform_{[0,1]}}} \\ &= \frac{\sigma_T}{\frac{1}{2\sqrt{3}}} \end{aligned} \tag{5.10}$$

The function $dispersion(T)$ is a measure of how close the distribution of throughputs is to a uniform distribution. A value close to 1 indicates well-distributed values across a significant range. Values larger than 1 denote separated clusters.

Another evaluation measure is based on the throughput of an unrandomized host. For the *reference throughput* defined as the throughput a host would have had under attack in the absence of the randomization (q), the *percentage of survivors* is defined as the percentage of hosts in which the throughput under attack is larger than the throughput in the absence of a randomization. A host h_i is denoted a *survivor* if in the presence of an attack with period T:

$$\rho(h_{iD}) > \rho(h_{iO}) \tag{5.11}$$

Based on Equations 5.9, 5.10, and 5.11, for a given attack, as defined by its period T, I define the following metrics for evaluating a randomization:

- **Average increase in throughput (gain):** $\frac{\rho(S_D)-q}{q}$;
- **Percentage of survivors:** $\frac{m}{n}$ where m is the number of hosts that comply with Equation 5.11 and n is the number of hosts in the network; and
- **Dispersion:** $dispersion(T) = \frac{\sigma_T}{\frac{1}{2\sqrt{3}}}$.

For a randomization to be labeled as moderately useful,the minimum expectation is that the gain and percentage of survivors would be larger than zero. In the absence of another defense with which to compare, an increase of 50% or more in throughput, a percentage of survivors at or above 20%, and a $dispersion(T)$ over 0.6 will be considered a good outcome. These parameters were arbitrarily chosen.

The next section presents the performance results of the randomization under attack, and discusses some of the implications.

5.7 Results and discussion

In this section I evaluate the effect of the proposed randomizations using the criteria established in the previous section. I will focus on minRTO randomizations, and then briefly discuss the effect of modifying the α, β and k parameters. All results shown for `Kernel 3` were obtained using $\alpha = \frac{1}{2}, \beta = \frac{1}{16}, k = 8$.

The efficiency of the randomization (the compliance with Equation 5.8) was tested using all kernel varieties and the traffic mixes described in Section 5.5. The differences in throughput with the defense in place without attack were under 1% (not shown). From the point of view of the host, the differences in the calculations are minimal. Therefore, I conclude that *the cost of the randomization is negligible*. Although it is almost certain that under network congestion the implementation modifications would make a difference in terms of the larger network throughput, this requires more extensive testing, and is considered future work. In [1] there is an important observation that the congestion control algorithm is relatively insensitive to different values of the three extra parameters. I speculate that this result may be the case as well with the different kernel implementations tested.

The throughput values for different values of the kernel parameters were collected by

running attacks against hosts in the testbed, for each of the three Linux kernel implementations discussed above. To compare with the standard TCP recommendation, I also show the normalized throughput as given in [65]. I collected data on the three kernels for nine minRTOs: 200, 250, 400, 500, 750, 800, 1000, 1250 and 1500 ms. Therefore, the experimental analysis of the kernels is based on the same values. In addition, I used the model to show that the larger-range, 0.1 to 2.1 s randomization is also feasible. The measured throughput averages are counted towards the average throughput in proportion to their relative presence in the population. Appendix B gives the presence distribution used, which allows an average minRTO of 1 s for the network. For the standard TCP model, I assume that the minRTO is chosen between 0.1 and 2.1 s.

Table 5.4 summarizes the throughput data per kernel, per attack period. It can be seen that the randomization obtains a survival percentage larger than zero for *all* attack periods in all kernels. This means that with a high probability there is always at least one host able to use some percentage of the bandwidth. Furthermore, there are only four instances where the percentage of survivors is below 20% (attack periods 0.4, 1.1 and 1.2 in `Kernel 2`, and attack period 1.1 in `Kernel 3`), and *no* instances where the percentage of survivors is below 10%. The network average throughput for the randomization is larger than the non-randomized (reference) throughput for 10 out of 14 attack periods in the case of `Kernel 1`, and `Kernel 3`, and 6 out of 14 for `Kernel 2`.

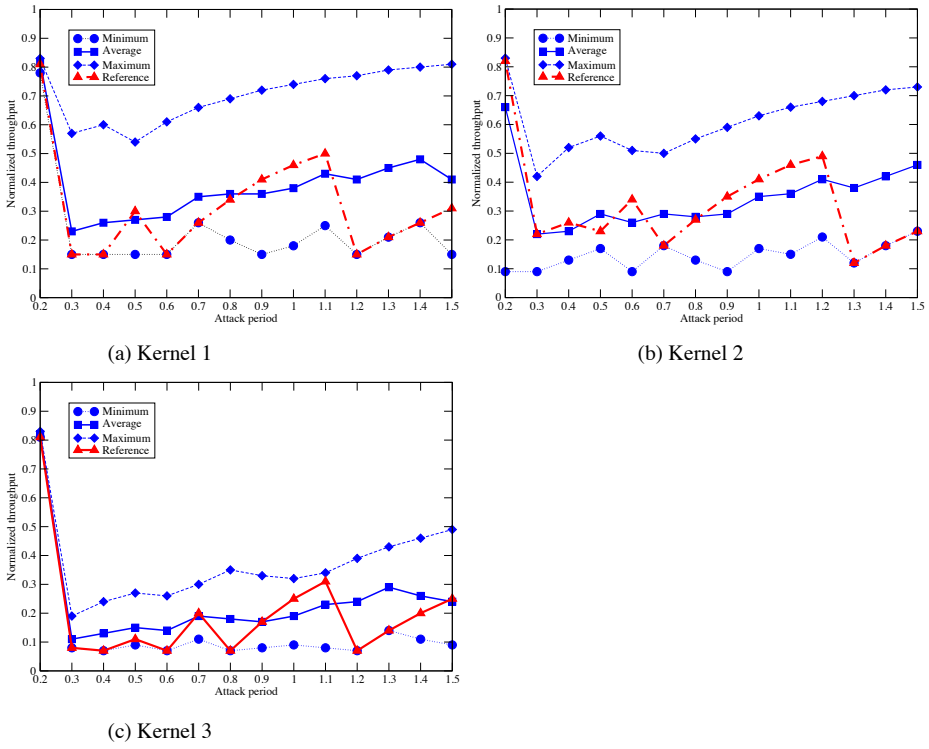
Although improvements are are modest in terms of average throughput (under most attack periods, the kernels obtain an average throughput below 0.5, but this is still better than the average using the non-randomized TCP-recommended minRTO setup), but when the maximum and minimum values are considered, a different picture emerges. Figure 5.4 compares minimum, average and maximum throughput per period, for the three kernels. For `Kernel 1` *all* maximum throughputs are above 0.5, in `Kernel 2` they are all above 0.4 and only in `Kernel 3` are they low and oscillate between 0.2 and 0.5. This means that *some of the hosts are obtaining near-normal throughputs*, which is precisely the goal of a

	Kernel 1			Kernel 2			Kernel 3		
Attack Period	**Avg.**	**Gain**	**Surv.**	**Avg.**	**Gain**	**Surv.**	**Avg.**	**Gain**	**Surv.**
0.2	**0.82**	0.01	77.78	0.66	-0.20	50.00	**0.82**	0.01	61.11
0.3	**0.23**	0.55	83.33	0.22	-0.01	27.78	**0.11**	0.43	83.33
0.4	**0.26**	0.80	83.33	0.23	-0.14	16.67	**0.13**	0.75	83.33
0.5	0.27	-0.10	22.22	**0.29**	0.23	61.11	**0.15**	0.38	61.11
0.6	**0.28**	0.87	83.33	0.26	-0.23	22.22	**0.14**	0.88	83.33
0.7	**0.35**	0.36	77.78	**0.29**	0.61	83.33	0.19	-0.04	50.00
0.8	**0.36**	0.04	27.78	**0.28**	0.02	50.00	**0.18**	1.47	83.33
0.9	0.36	-0.11	27.78	0.29	-0.17	27.78	**0.17**	0.03	50.00
1.0	0.38	-0.16	22.22	0.35	-0.14	27.78	0.19	-0.23	22.22
1.1	0.43	-0.13	22.22	0.36	-0.20	16.67	0.23	-0.26	11.11
1.2	**0.41**	1.79	83.33	0.41	-0.16	16.67	**0.24**	2.25	83.33
1.3	**0.45**	1.15	83.33	**0.38**	2.20	83.33	**0.29**	1.00	83.33
1.4	**0.48**	0.84	83.33	**0.42**	1.35	83.33	**0.26**	0.32	55.56
1.5	**0.41**	0.33	55.56	**0.46**	0.99	83.33	0.24	-0.01	33.33

Table 5.4: Throughput obtained using the nine values of minRTO randomization, for `Kernel 1`, `Kernel 2` and `Kernel 3`. Column 1 is the attack period T. For each kernel column, subcolumn 1 (Avg.) gives the network average using the distribution given by Equation B.1, subcolumn 2 (Gain) shows the percent gain with respect to the reference throughput (the network average if all hosts were using fixed minRTO= 1 s), subcolumn 3 (Surv.) shows the percentage of hosts surviving the attack (using the definition of survival given in Equation 5.11). Average throughputs which are larger than the reference (no randomization) throughput are emphasized.

diversity defense: that some members of the population survive almost unaffected.

The results for the model (range 0.1 to 2.1 s randomization) are presented in Figure 5.5. The curves confirm the tendency noted in the data for the kernels: the maximum throughput that some host can obtain under attack is much larger than the the nonrandomized network throughput (reference). Even the average network throughput exceeds the reference throughput for most of the attack periods. In conclusion, from the experimental data for the tested kernels, and the model for the standard TCP calculation, it is possible to assert that *randomization noticeably improves the average network throughput under attack*



(a) Kernel 1

(b) Kernel 2

(c) Kernel 3

Figure 5.4: Minimum, average and maximum throughput for the nine values of randomization, per attack period, for `Kernel 1`, `Kernel 2`, and `Kernel 3`. In red, the average for the reference minRTO = 1 s (unrandomized average throughput under attack). Observe how the maximum values exceed the reference throughput in all kernels at all attack periods. Even the network averages are larger than the reference in a considerable number of cases.

for most attack periods, and under all attack periods has at least one member which can continue transmitting with a high throughput.

Given that some hosts can obtain large throughput with the randomization, it is interesting to study the distribution of throughputs. The distances between minimum and maximum shown in Figures 5.4 and 5.5 could be an indication of a favorable dispersion, but do not guarantee anything (the data could be clumping around the minimum and maximum, for example). As discussed before, $dispersion(T)$ gives a real estimate. Figure 5.6 shows the distribution of $dispersion(T)$ for the three kernels, and Figure 5.7 presents the

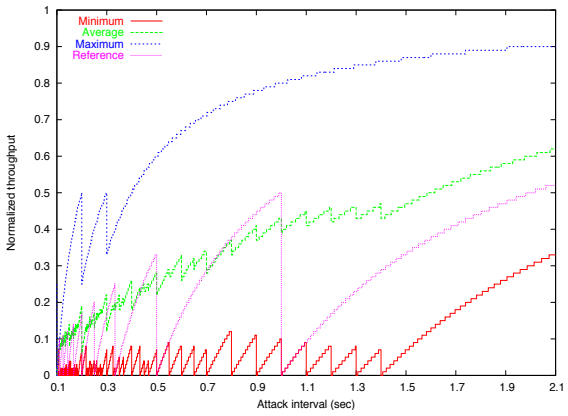


Figure 5.5: Standard TCP (theoretical) normalized throughput for minRTOs ranging from 100 to 2100 ms. Shown are minimum, average and maximum throughputs. The reference (nonrandomized) throughput of the network is plotted for comparison. Observe that the maximum throughput is consistently larger than the throughput obtained with no randomization (reference), and the average throughput is larger than the reference throughput for most of the attack periods.

dispersal for the standard RTO calculation (from the model). The dispersion is much closer to uniform in the model, which might have to do with the access to a larger number of data points for the test. The dispersion for the 9 minRTO dataset is moderate, even for periods where the minimum and maximum throughput values are well separated. It is difficult to draw conclusions from these data, except that the throughputs are clearly not clumping into a few values. More data are required for a good estimate of $dispersion(T)$. Although the available values suggest reasonable dispersion, the $dispersion(T)$ estimate is not clear. In support of the hypothesis that dispersion is adequate, a histogram of the throughputs per minRTO was calculated, and is shown in Figures 5.8 and 5.9, for `Kernel 1` and `Kernel 2` respectively [5]. In these figures we can observe that the throughputs among different minRTOs exhibit diversity for most of the attack periods. The only period where this is not true is 0.2 s, but at that attack period all minRTOs have a very large (around 0.8) throughput. Therefore, I conclude that *for most attack periods, the responses among hosts with differ-*

[5] `Kernel 3` is excluded from this analysis because its performance is extremely poor.

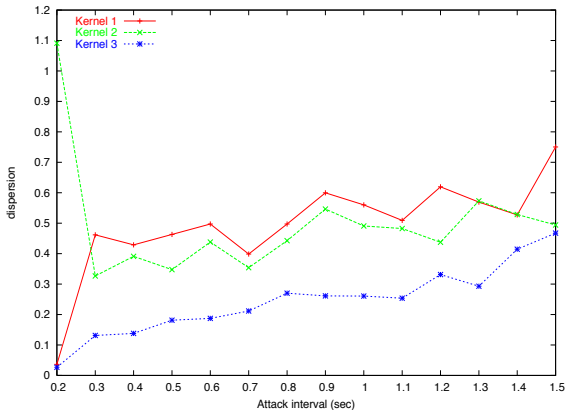


Figure 5.6: Dispersion of the throughputs per period for the three kernels.

ent minRTOs would be heterogeneous enough to make good use of the lost bandwidth.

At this point, all evidence regarding the direct benefits of the randomization as compared with a non-randomized system have been made. Two issues remain. The first has to do with the existence of an optimum minRTO for the defense, and the second concerns the evaluation of the role of the randomization of the α, β and k parameters.

For the randomization to make sense at all, there should be no 'sweet-spot': an optimal value for the defender to use. If there is such a value, it would be a good result and the final, engineered response to the exploit. When examining Figures 5.8 and 5.9 the 200 ms minRTO appears to be a good candidate (judging from the limited number of minRTOs sampled). However, it should not be used as a homogeneous minRTO value for the entire network given that it could cause flooding in times of real congestion [1]. The same consideration applies to using minRTO=25, 40 and 50 ms. After that, there are no clear candidates as the throughput patterns for larger minRTOs keep changing with the attack periods. Therefore, I conclude that *the defender cannot use a single minRTO optimum*, which supports the randomization argument.

I will now discuss briefly the suggested kernel modifications. It is clear that the stan-

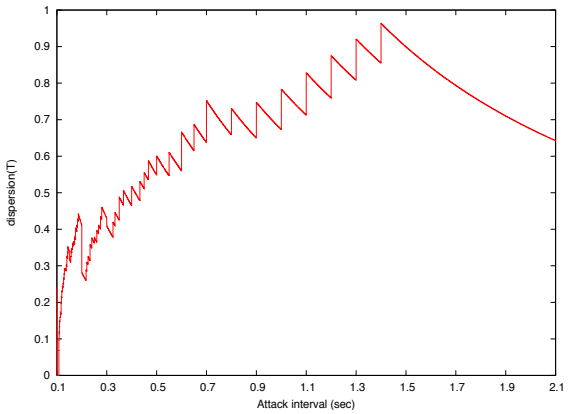


Figure 5.7: Dispersion of the throughputs per period in the standard TCP RTO calculation, from the model

dard Linux implementation gives the best overall performance with the minRTO randomization, although followed very closely by `kernel 2`. The reason for the disparity in that particular case is that the consistently larger RTOs dampened the amount of throughput delivered by the flows, as they would have to wait longer before retransmitting. There are periods, however, where they slightly outperform `kernel 1`. It is difficult to definitively choose between `Kernel 1` and `Kernel 2` on the basis of the small set of tests attempted. On the other hand, `kernel 3` (shown for parameters $\alpha = \frac{1}{2}, \beta = \frac{1}{16}$ and $k = 8$ that gave the *best* performance) performed consistently worse than either `kernel 1`, or `kernel 2`, thus disproving my hypothesis that randomizing those parameters could confer a benefit (at least against this type of attack). The most probable explanation is that the loss in both dispersion and bandwidth was due to the artificial enlargement of RTOs across the range of minRTOS. The evidence points against increasing the RTO range, be it via the 3 non-minRTO parameters using a `kernel 3` setup or simply by using an extremely large minRTO range.

A result that is not shown but worth noting is the effect of the attacks on 'filtered' flows. As observed in [65], short-lived flows are sometimes able to use the leftover bandwidth

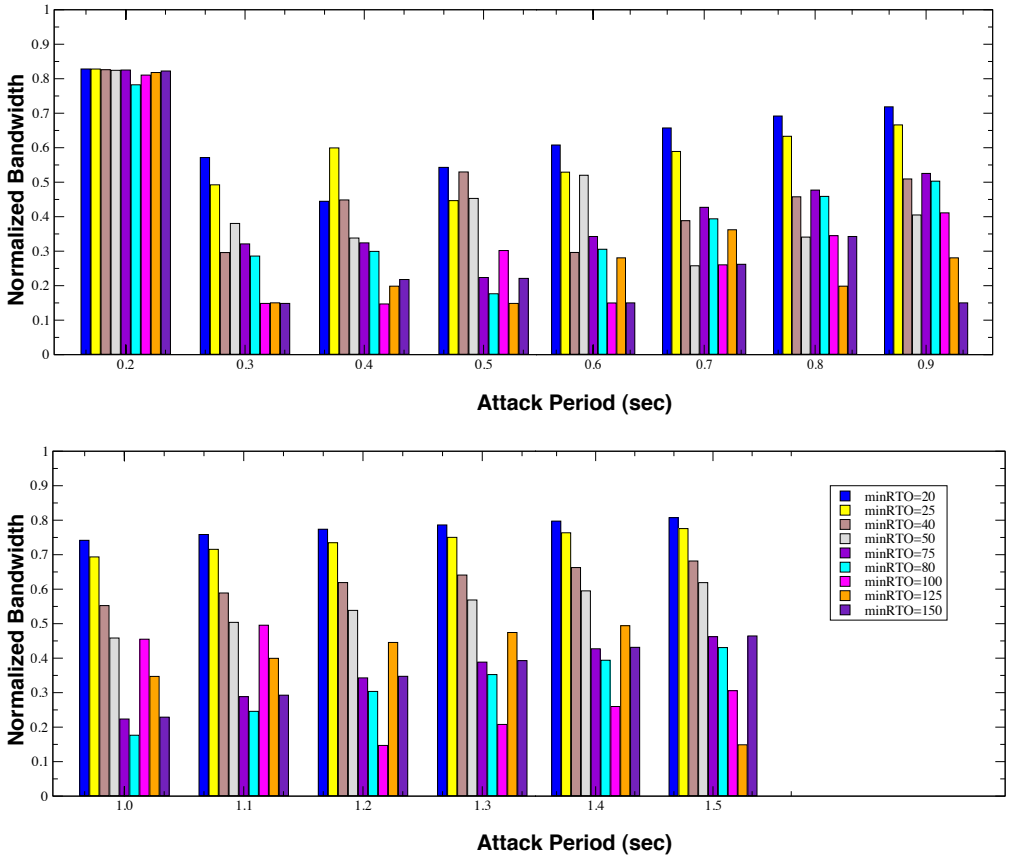


Figure 5.8: Histogram of normalized throughput per period, for minRTO $\in\{0.20, 0.25, 0.40, 0.50, 0.75, 0.80, 1.00, 1.25, 1.50\}$ s, for `kernel 1`.

because they either (a) start and finish the transmission in between attack bursts; or (b) having fewer total packets to send are able to finish before the long flows by smuggling one packet into the network once in a while. As we observed in the mixed transmissions, this translates into actual *gains* in bandwidth for small transmissions with respect to their bandwidth in the absence of attacks, when they are competing fairly with the other TCP flows. However, this gain is apparent only because they are affected by the attack as well, so they never manage to use all the available bandwidth. The random survival of these small transmissions argues for research into randomly restarting the congestion status of

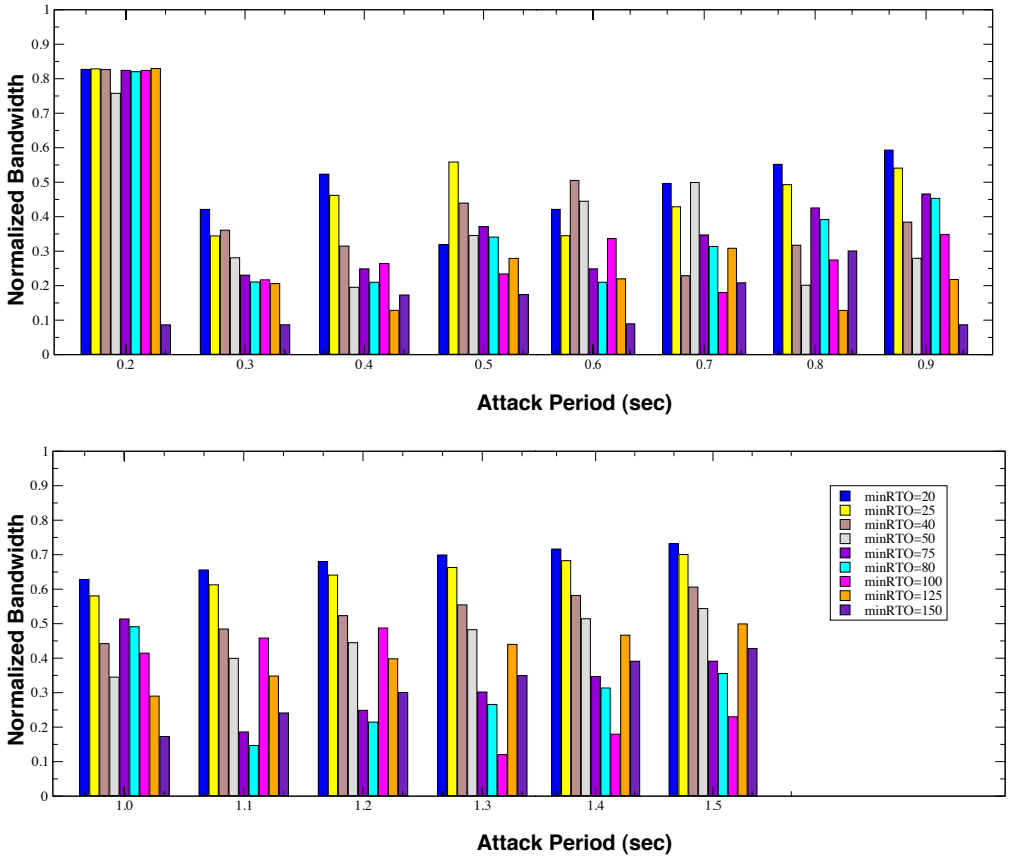


Figure 5.9: Histogram of normalized throughput per period, for minRTO $\in\{0.20, 0.25, 0.40, 0.50, 0.75, 0.80, 1.00, 1.25, 1.50\}$ s, for `kernel 2`.

longer transmissions. This is left as future work.

5.8 Background and related work

I have already discussed extensively the Kuzmanovic and Knightly paper [65] that presented the attack against which the randomization in this paper is targeted. Their paper demonstrates not only the attack's feasibility and reach, but also its relative insensitivity to

different TCP congestion control schemes and even router-assisted mechanisms. The article explores minRTO randomization as well, with the assumption that each flow chooses a different minRTO. The authors' conclusion, based on a derivation of the approximate average aggregate flow, is that randomization only minimally shifts and smoothes the attack throughput pattern while leaving the network still open to the attack. As this research demonstrates, by changing the perspective to individual hosts using different but fixed minRTOs, at any attack period, some of the hosts are able to successfully survive the attack and even occupy the bandwidth left open by affected computers. I also believe that a limitation of their analysis in [65] is that they examined the randomization over a fairly small range of minRTOs (1 to 1.2 s).

With respect to attacks targeting unnecessarily fixed parameters in protocols, the infamous SYN Flood attack [95], which will be discussed in more detail in the next chapter, could be seen as an attack on two parameters: the connection request queue length and the timeout for its entries. In fact, many early and incomplete defenses against this attack changed only one or both of these parameters [67]. The parameters had no predefined or even recommended value in any specifications and had been inherited from original version(s). There are other considerations about this attack that make the randomization of these two parameters insufficient, but it is safe to state that if the TCP implementations in use when the attack became public had largely distributed buffer length and timeout parameters, the impact of the attack would have been greatly reduced.

A related category of attacks includes those that force a protocol into slower states. Most of these attacks do not use protocol regularities, but instead exploit the weak per-segment authentication of TCP packets. Some examples of these attacks are: (1) the TCP RST DoS, where the attacker guesses the sequence numbers of an ongoing connection and sends an RST request to close it, and (2) the TCP Sloth attack that sends zero size window advertisements to the sender, thus keeping it from restarting the connection [33]. In this last one, there could have been a subtle exploitation of the timing regularities of TCP. This

attack works by sending the advertisement after the sender requests information about the connection to the receiver, which is synchronized by using a very predictable timer, so a blind attacker could have timed the advertisements. However, as published it was a sender-side exploit with the attacker being able to read all packet going out of the sender, so the timing was not used at all. For the (hypothetical) synchronized version of this exploit, a randomization of the clock could have helped prevent the (theoretical) improved version of this attack. It is not clear that in the current version of TCP this issue still exists.

I am not aware of other current attacks that force protocols into slower states, but it is just a question of time before some other protocol is exploited in a similar manner, or a different attack is found using TCP. The more randomized the individual behavior, the less probable any future attack will have full success.

5.9 Future work

A strategy of randomly resetting the congestion control state should be explored as it might confer additional benefits to the overall minRTO randomization scheme.

Randomizing α, β and k could still be useful in the context of a `kernel 1` setup for flows with large RTTs. This must be explored further.

TCP contains many parameters that are fixed arbitrarily. However, whether this is a general characteristic of protocols remains to be proven. It is to be expected that simpler, lower layer, or non-reliable protocols are not as vulnerable to exploitation of fixed parameters. The identification of implicit, explicit but unnecessary, or dangerous parameters in others protocols must be undertaken.

5.10 Summary

In this chapter, three real and one theoretical implementation of the TCP RTO calculations were used to argue that randomizing parameters in a protocol can have protective effects against attacks targeting protocol flaws. As long as the parameters' uniform distribution average equals the recommended or required values for the parameters, a sub-network will be protected and as a whole will still behave as a good neighbor. To demonstrate this point, the *shrew* attack [65] was used. By moving the perspective from averages to individuals, and considering the differences in throughput for each of them, it is possible to build a network that is more resilient to this attack. More specifically, I presented evidence that the randomization:

a. has a negligible cost;

b. improves the *average* network throughput under attack in most cases;

c. under *all* attacks, guarantees that there is at least one host that achieves near normal throughput; and

d. allows hosts to have a response heterogeneous enough so they can achieve larger than predicted throughputs.

I also showed that there are no **usable** (larger than or equal to 1 s) minRTO optimum for the defender, so the randomization maximizes the benefits, while (on average) preserving the minRTO IETF requirement. Slowly changing the random minRTO in the hosts will allow all of them to share the benefits of the smaller minRTOs.

Finally, I disproved my hypothesis that randomizing the α, β and k parameters could have a beneficial effect, (at least in the context of 'close' networks) as the modifications to make them visible to the final RTO setup made the sender response times extremely slow.

Appendices

Appendix A

Auxiliary derivations for the theoretical models of random code execution

A.1 Encoding of the IA32 Markov Chain Model

In this appendix, we discuss the details for the construction of the Markov chain representing the state of the processor as each byte is interpreted.

If $X_t = j$ is the event of being in state j at time t (in our case, at the reading of byte t), the transition probability $P\{X_{t+1} = j | X_t = i\}$ is denoted p_{ij} and is the probability that the system will be in state j at byte $t+1$ if it is in state i for byte t.

For example, when the random sequence starts (in state *start*), there is some probability p that the first byte will correspond to an existing one-byte opcode that requires an additional byte to specify memory addressing (the Mod-Reg-R/M (MRM) byte). Consequently, we create a transition from *start* to *mrm* with some probability p: $p_{start,mrm} = p$. p is the number of instructions with one opcode that require the MRM byte, divided by the total number of possibilities for the first byte (256). In IA32 there are 41 such instructions,

so $p_{start,mrm} = \frac{41}{256}$.

If the byte corresponds to the first byte of a two-byte instruction, we transition to an intermediate state that represents the second byte of that family of instructions, and so on. There are two exit states: *crash* and *escape*. The *crash* state is reached when an illegal byte is read, or there is an attempt to use invalid memory, for an operation or a jump. The second exit state, *escape*, is reached probabilistically when a legitimate jump is executed. This is related to the escape event.

Because of the complexity of the IA32 instruction set, we simplified in some places. As far as possible, we adhered to the worst-case principle, in which we overestimated the bad outcomes when uncertainty existed (e.g., finding a legal instruction, executing a privileged instruction, or jumping). The next few paragraphs describe these simplifications.

We made two simplifications related to instructions. The IA32 has instruction modifiers called prefixes which can generate complicated behaviors when used with the rest of the instruction set. We simplified by treating all of them as independent instructions of length one byte, with no effect on the following instructions. This choice overestimates the probability of executing those instructions, as some combinations of prefixes are not allowed, others significantly restrict the kind of instructions that can follow, or make the addresses or operands smaller. In the case of regular instructions that require longer low-probability pathways, we combined them into similar patterns. Privileged instructions are assumed to fail with probability of 1.0 because we assume that the RISE-protected process is running at user level.

In the case of conditional branches, we assess the probability that the branch will be taken, using the combination of flag bits required for the particular instruction. For example, if the branch requires that two flags have a given value (0 or 1), the probability of taking the branch is set to 0.25 . A non-taken branch transitions to the *start* state as a linear instruction. All conditional branches in IA32 use *relative* (to the current Instruction

Pointer), 8 or 16-bit displacements. Given that the attack had to be in an executable area to start with, this means that it is likely that the jump will execute. Consequently, for conditional branches we transition to *escape* with probability 1. This is consistent with the observed behavior of successful jumps.

A.2 Definition of loose and strict criteria of escape

Given that the definition of escape is relative to the position of the instruction in the exploit area, it is necessary to arbitrarily decide if to classify an incomplete interpretation as an escape or as a crash. This is the origin of the loose and strict criteria.

In terms of the Markov chain, the loose and strict classifications are defined as follows:

a. Loose escape: Starting from the *start* state, reach any state except *crash*, in m transitions (reading m bytes).

b. Strict escape: Reach the *escape* state in m or fewer transitions from the *start* state (in m bytes).

If T is the transition matrix representing the IA32 Markov chain, then to find the probability of escape from a sequence of m random bytes, we need to determine if the chain is in state *start* or *escape* (the strict criterion) or not in state *crash* (the loose criterion) after advancing m bytes. These probabilities are given by $T^m(start, start) + T^m(start, escape)$ and $1 - T^m(start, crash)$ respectively, where $T(i, j)$ is the probability of a transition from state i to state j.

A.3 Partition graph for the PowerPC encoding

Figure A.1 illustrates the partition of the symbols into disjoint sets using the execution model given in 4.1.

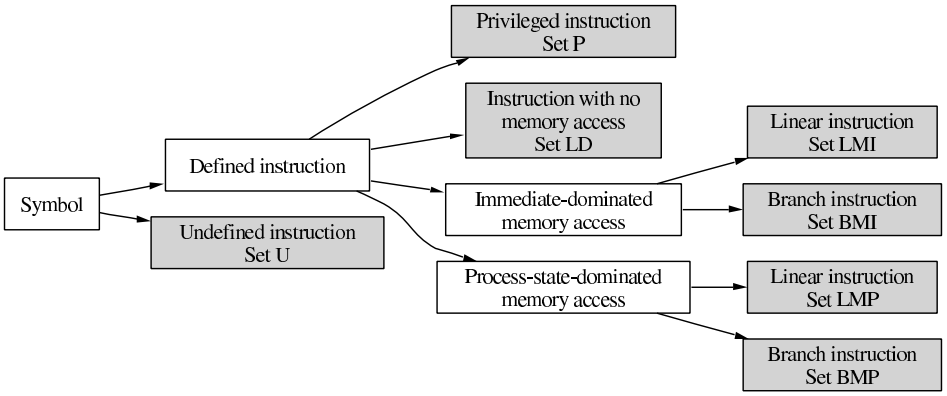


Figure A.1: Partition of symbols into disjoint sets based on the possible outcome paths of interest in the decoding and execution of a symbol. Each path defines a set. Each shaded leaf represents one (disjoint) set, with the set name noted in the box.

A.4 Relative short branches

The set of branches that are relative to the current Instruction Pointer with a small offset (defined as being less or equal than 2^{b-1}) are separated from the rest of the branches, because their likelihood of execution is very high. In the analysis we set their execution probability to 1, which is consistent with observed behavior.

A fraction of the conditional branches are artificially separated into L_{MI} and L_{MP} from their original B_{MI} and B_{MP} sets. This fraction corresponds to the probability of taking the branch, which we assume is 0.5. This is similar to the IA32 case, where we assumed that a non-branch-taking instruction could be treated as a linear instruction.

To determine the probability that a symbol falls into one of the partitions, we need

to enumerate all symbols in the instruction set. For accounting purposes, when parts of addresses and/or immediate (constant) operands are encoded inside the instruction, each possible instantiation of these data fields is counted as a different instruction. For example, if the instruction 'XYZ' has two bits specifying one of four registers, we count four different XYZ instructions, one for each register encoding.

A.5 Derivation of the probability of a successful branch (escape) out of a sequence of n random bytes.

$$\begin{aligned} P(X_n^\star) &= \sum_{i=1,\ldots,n} P(X_i) + P(L)^n \\ &= \sum_{i=1,\ldots,n} P(L)^i P(E) + P(L)^n \\ &= \left(P(E) \sum_{i=1,\ldots,n} P(L)^i \right) + P(L)^n \\ &= P(E) \frac{1 - P(L)^{n+1}}{1 - P(L)} + P(L)^n \end{aligned} \tag{A.1}$$

A.6 Derivation of the lower limit for the probability of escape.

$$\begin{aligned} \lim_{n\to\infty} P(X_n^\star) &= \lim_{n\to\infty} P(E) \frac{1 - P(L)^{n+1}}{1 - P(L)} + P(L)^n \\ &= \frac{P(E)}{1 - P(L)} \end{aligned} \tag{A.2}$$

Appendix B

Distribution for minRTOs in the sample

There is data available for nine particular minRTOs. Given that the network average must be equal or larger than 1 s, the network being studied must have picked the minRTOs according to a distribution whith that average. The distribution used for the reported results is:

$$p(minRTO = x) = \begin{cases} \frac{1}{18} & \text{if } x \in 200, 250, 400, 500, 750, 800\, ms, \\ \frac{3}{18} & \text{if } x = 1000\, ms, \\ \frac{5}{18} & \text{if } x = 1250\, ms, \\ \frac{4}{18} & \text{if } x = 1500\, ms \end{cases} \tag{B.1}$$

Appendix C

Rate derivations for GP-generated filters

C.1 Combined request interarrival time

The combined request interarrival time is found using the combined request *rate* as follows:

$$\begin{aligned} \frac{1}{\lambda_c} &= \frac{1}{\lambda_a} + \frac{1}{\lambda_l} \\ &= \frac{\lambda_a + \lambda_l}{\lambda_a \lambda_l} \end{aligned}$$

therefore,

$\lambda_c = \frac{\lambda_a \lambda_l}{\lambda_a + \lambda_l}$

C.2 Resource overflow

Little's law holds independently of the distribution of the arrival and service rates, and it states that: 'The average number of customers in a system (over some interval) is equal to their average arrival rate, multiplied by their average time in the system.' [70]. Being consistent with the notation in Chapter 6, this can be expressed as:

$$C = \frac{1}{\lambda}\mu, \tag{C.1}$$

where C is the number of customers in the system, λ is the average interarrival time of requests and μ is the average service time.

Reordering Equation C.1, we can see that $\frac{\lambda}{\mu} = C$. Therefore, if N_m the number of available resources is smaller than the load of requests C, the system is overflown. This is usually expressed through the utilization (ρ) of the system, defined as the average arrival rate divided by the average service rate has to be less than one for a stable system. That is, the system is stable if:

$$\begin{aligned} \rho &= \frac{\frac{1}{\lambda}}{N_m \frac{1}{\mu}} \\ &= \frac{\mu}{N_m \lambda} \\ &< 1 \end{aligned}$$

where $N\frac{1}{\mu}$ is the combined service rate of a system with N resources.

For the simulation trace to overflow the resource buffer, it is therefore necessary that:

$$\begin{aligned}
\frac{\frac{1}{\lambda_c}}{\frac{1}{\mu_l N_m}} &\geq 1 \\
\frac{\mu_l}{N_m \lambda_c} &\geq 1 \\
\mu_l &\geq N_m \lambda_c
\end{aligned}$$

For the legitimate requests in the simulation trace *not* to overflow the buffer it must hold that:

$$\begin{aligned}
\frac{\frac{1}{\lambda_l}}{\frac{1}{\mu_l N_m}} &< 1 \\
\frac{\mu_l}{N_m \lambda_l} &< 1 \\
\mu_l &< N_m \lambda_l
\end{aligned}$$

References

[1] Mark Allman and Vern Paxson. On estimating end-to-end network path properties. *ACM SIGCOMM Computer Communication Review*, 29(4):263–274, October 1999.

[2] Mark Allman, Vernor Paxson, and W Stevens. Tcp congestion control. Technical Report RFC 2581, April 1999.

[3] Ross Anderson. 'Trusted Computing' and competition policy - issues for computing professionals. *Upgrade*, IV(3):35–41, June 2003.

[4] William A. Arbaugh. Improving the TCPA specification. *IEEE Computer*, 35(8):77–79, August 2002.

[5] Kumar Avijit, Prateek Gupta, and Deepak Gupta. Tied, libsafeplus: Tools for dynamic buffer overflow protection. In *Proceeding of the 13th USENIX Security Symposium*, San Diego, California, U.S.A., August 9-13 2004.

[6] Algirdas Avizienis. The Methodology of N-Version Programming. In Michael Lyu, editor, *Software Fault Tolerance*, pages 23–46. John Wiley & Sons Ltd., 1995.

[7] Algirdas Avizienis and L. Chen. On the implementation of N-Version programming for software fault tolerance during execution. In *Proceedings of IEEE COMPSAC 77*, pages 149–155, November 1977.

[8] Charles Babbage. On the mathematical powers of the calculating engine. In B.Randell, editor, *The origins of Digital Computers: Selected Papers*, pages 17–52, New York, 1974. Springer. From an unpublished manuscript by C. Babbage, written in December 1837. Current location: Museum of the History of Science, Oxford.

[9] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: a transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN '00 conference on Programming language design and implementation*, pages 1–12, Vancouver, British Columbia, Canada, 2000. ACM Press.

[10] Arash Baratloo, Navjot Singh, and Timothy Tsai. Transparent run-time defense against stack smashing attacks. In *Proceedings of the 2000 USENIX annual technical conference (USENIX-00)*, pages 251–262, Berkeley, California, U.S.A., June 2000.

[11] E. Gabriela Barrantes, David Ackley, Stephanie Forrest, and DArko Stephanovic. Randomized instruction set emulation. *TISSEC*, In press.

[12] Cory Beard. Preemptive and delay-based mechanisms to provide preference to emergency traffic. *Accepted for publication by Computer Networks Journal, July 2004*.

[13] Daniel Bernstein. SYN Cookies. In *http://cr.yp.to/syncookies.html*, January 31 2002.

[14] R. M. Best. Microprocessor for executing enciphered programs, U.S. Patent No. 4 168 396, September 18 1979.

[15] R. M. Best. Preventing software piracy with crypto-microprocessors. In *Proceedings of the IEEE Spring COMPCON '80*, pages 466–469, San Francisco, California, February 1980.

[16] Sandeep Bhatkar, Daniel DuVarney, and Ravi Sekar. Address obfuscation: An approach to combat buffer overflows, format-string attacks and more. In *Proceedings of the 12th USENIX Security Symposium*, pages 105–120, Washington, D.C., U.S.A., August 4-8 2003.

[17] Stephen W. Boyd and Angelos D. Keromytis. SQLrand: Preventing SQL injection attacks. In *Proceedings of the 2nd Applied Cryptography and Network Security (ACNS) Conference*, pages 292 – 302, Yellow Mountain, China, June 2004.

[18] Derek Bruening, Saman Amarasinghe, and Evelyn Duesterwald. Design and implementation of a dynamic optimization framework for Windows. In *4th ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4)*, December 2001.

[19] Timothy R. Butler. Bochs. In *http://bochs.sourceforge.net/*, Last accessed on September 2004.

[20] CERT Coordination Center. Overview of attack trends, 2002.

[21] Monica Chew and Dawn Song. Mitigating Buffer Overflows by Operating System Randomization. Technical Report CMU-CS-02-197, Department of Computer Science, Carnegie Mellon University, December 2002.

[22] Tzi-cker Chiueh and Fu-Hau Hsu. Rad: A compile-time solution to buffer overflow attacks. In *Proceedings of the 21st International Conference on Distributed Computing Systems (ICDCS)*, pages 409–420, Phoenix, Arizona, USA, April 16-19 2001.

[23] Fred Cohen. Operating System Protection through Program Evolution. *Computers and Security*, 12(6):565–584, October 1993.

[24] Fred Cohen. A Note on the Role of Deception in Information Protection. *Computers and Security*, 17:483–506, 1998.

[25] Fred Cohen. The Deception Toolkit. In *http://all.net/dtk/*. Fred Cohen and Associates, 2002.

[26] Fred Cohen, Dave Lambert, Charles Preston, Nina Berry, Corbin Stewart, and Eric Thomas. A framework for deception. Technical report, Fred Cohen and Associates, July 13 2001.

[27] Internet Software Consortium.

[28] CORE Security. CORE Security Technologies. In *http://www1.corest.com/home/home.php*, Last accessed on September 2004.

[29] Crispin Cowan, Matt Barringer, Steve Beattie, and Greg Kroah-Hartman. Format guard: Automatic protection from `printf` format string vulnerabilities. In *Proceedings of the 10th USENIX Security Symposium*, pages 191–199, Washington, D.C., U.S.A., August 13-17 2001.

[30] Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. Pointguard: Protecting pointers from buffer overflow vulnerabilities. In *Proceedings of the 12th USENIX Security Symposium*, pages 91–104, Washington, D.C., U.S.A., August 4-8 2003.

[31] Crispin Cowan, Heather Hinton, Calton Pu, and Jonathan Walpole. A Cracker Patch Choice: An Analysis of Post Hoc Security Techniques. In *National Information Systems Security Conference (NISSC)*, Baltimore MD, October 16-19 2000.

[32] Crispin Cowan, Calton Pu, David Maier, Heather Hinton, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. Automatic detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Symposium*, San Antonio, Texas, January 1998.

[33] daemon9. Project Hades: TCP weaknesses. *Phrack*, 49(7), November 1996.

[34] Dallas Semiconductor. DS5002FP secure microprocessor chip, 1999. http://pdfserv.maxim-ic.com/en/ds/DS5002FP.pdf.

[35] Nurit Dor, Michael Rodeh, and Mooly Sagiv. CSSV: Towards a realistic tool for statically detecting all buffer overflows in c. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 155–167, June 2003.

[36] D. Eckhard and L. Lee. A theoretical basis for the analysis of multiversion software subject to coincident errors. *IEEE Transactions on Software Engineering*, SE-11(12):1511–1517, 1985.

[37] Hiroaki Etoh and Kunikazu Yoda. Protecting from stack-smashing attacks. Web publishing, IBM Research Division, Tokyo Research Laboratory, http://www.trl.ibm.com/projects/security/ssp/main.html, June 19 2000.

[38] Hiroaki Etoh and Kunikazu Yoda. Propolice: Improved stack smashing attack detection. *IPSJ SIGNotes Computer Security (CSEC)*, (14), October 26 2001.

[39] Pierre-Alain Fayolle and Vincent Glaume. A buffer overflow study, attacks & defenses. Web publishing, ENSEIRB, http://www.wntrmute.com/docs/bufferoverflow/report.html, 2002.

[40] Robert Feldt. Generating Diverse Software Versions with Genetic Programming: an Experimental Study. *IEE Proceedings - Software Software*, 145(6):228–236, December 1998.

[41] Stephanie Forrest, Anil Somayaji, and David Ackley. Building Diverse Computer Systems. In *Proceedings of the Sixth Workshop on Hot Topics in Operating Systems*, pages 67–72, 1997.

[42] Mike Frantzen and Mike Shuey. Stackghost: Hardware facilitated stack protection.

In *Proceedings of the 10th USENIX Security Symposium*, Washington, D.C., U.S.A., August 2001.

[43] Aman Garg. Policy based end server resource regulation. Technical Report TAMU-ECE-2001-07, May 2001.

[44] D. Geer, R. Bace, P. Butmann, P. Metzger, C. Pfleeger, J. S. Quarterman, and B. Schneier. Cyber insecurity: The cost of monopoly, 2003.

[45] GNU Free Software Foundation. Gsl - gnu scientific library, last updated January 2005.

[46] Mitchell Harper. SQL injection attacks - are you safe? In *Sitepoint, http://www.sitepoint.com/article/794*, June 17 2002.

[47] Matti Hiltunen, Richard Schlichting, and Carlos Ugarte. Enhancing Survivability of Security Services using Redundancy. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN 2001)*, Gothenburg, Sweden, July 2001.

[48] Matti Hiltunen, Richard Schlichting, Carlos Ugarte, and Gary Wong. Survivability through Customization and Adaptability: The Cactus Approach. In *DARPA Information Survivability Conference and Exposition (DISCEX 2000)*, pages 294–307, January 2000.

[49] Steve Hofmeyr and Stephanie Forrest. Architecture for an artificial immune system. *Evolutionary Computation*, 7(1):1289–1296, 2000.

[50] Alefiya Hussain, John Heidemannn, and Christos Papadopoulos. Distinguishing between single and multi-source attacks using signal processing. *Computer Networks*, 46:479 – 503, June 2004.

[51] IBM. *PowerPC Microprocessor Family: Programming Environments Manual for 64 and 32-Bit Microprocessors. Version 2.0.* Number Order Nos. 253665, 253666, 253667, 253668. June 10, 2003.

[52] Kosuke Imamura, Robert B. Heckendorn, Terence Soule, and James A. Foster. *N*-version genetic programming via fault masking. In James A. Foster, Evelyne Lutton, Julian Miller, Conor Ryan, and Andrea G. B. Tettamanzi, editors, *Genetic Programming, Proceedings of the 5th European Conference, EuroGP 2002*, volume 2278 of *LNCS*, pages 172–181, Kinsale, Ireland, 3-5 April 2002. Springer-Verlag.

[53] SANS Institute. Naptha: A new type of denial of service attack. In *http://rr.sans.org/threats/naptha2.php*, December 2000.

[54] Intel Corporation. The IA-32 Intel Architecture Software Developer's Manual. Number Order Nos. 253665, 253666, 253667, 253668, 2004.

[55] Hao Jiang and Constantinos Dovrolis. Passive estimation of tcp round-trip times. *ACM SIGCOMM Computer Communication Review*, 32(3):75–88, July 2002.

[56] Trevor Jim, Greg Morrisett, Dan Grossman, Michael Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of c. In *Proceedings of the USENIX Annual Technical Conference*, pages 275–288, Monterey, California, U.S.A., June 2002.

[57] Robert W. M. Jones and Paul H.J. Kelly. Backwards-compatible bounds checking for arrays and pointers in c programs. In *Third International Workshop on Automated Debugging*, pages 13–26, may 1997.

[58] Gaurav S. Kc, Angelos D. Keromytis, and Vassilis Prevelakis. Countering Code-Injection Attacks With Instruction-Set Randomization. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, pages 272–280, Washington, D.C., U.S.A., October 27-31 2003. ACM Press.

[59] Vladimir Kiriansky, Derek Bruening, and Saman Amarasinghe. Secure execution via program sheperding. In *Proceeding of the 11th USENIX Security Symposium*, San Francisco, California, U.S.A., August 5-9 2002.

[60] John Knight and Nancy Leveson. An Experimental Evaluation of the Assumption of Independence in Multiversion Programming. *IEEE Transactions on Software Engineering*, SE-12(1):96–109, January 1986.

[61] Donald E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*, chapter 3.2-3.3. Addison-Wesley, Reading, Massachusetts, second edition, 10 January 1981. This is a full BOOK entry.

[62] John Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. The MIT Press, Cambridge, Massachussetts, 1992.

[63] John R. Koza. Scalable learning in genetic programming using automatic function definition. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 6, pages 99–117. MIT Press, Cambridge, MA, USA, 1994.

[64] Markus Kuhn. The TrustNo 1 cryptoprocessor concept. Technical Report CS555 Report, Purdue University, April 04 1997.

[65] Aleksandar Kuzmanovic and Edward W Knightly. Low-rate tcp-targeted denial of service attacks: the shrew vs. the mice and elephants. In *ACM SIGCOMM Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, Karlsruhe, Germany, August 25-29 2003.

[66] David Larochelle and David Evans. Statically detecting likely buffer overflow vulnerabilities. In *Proceedings of the 10th USENIX Security Symposium*, pages 177–190, Washington, D.C., U.S.A., August 13-17 2001.

[67] Jonathan Lemon. Resisting syn flood dos attacks with a syn cache. In *Proceedings of the BSDCon 2002*, San Francisco, California, February 11-14 2002.

[68] Hanoch Levy, Tsippy Mendelson, and Gilad Goren. Dynamic allocation of resources to virtual path agents. *IEEE/ACM Trans. Netw.*, 12(4):746–758, 2004.

[69] Kyung suk Lhee and Steve J. Chapin. Type-assisted dynamic buffer overflow detection. In *Proceeding of the 11th USENIX Security Symposium*, pages 81–88, San Francisco, California, U.S.A., August 5-9 2002.

[70] John Little, 1961.

[71] Bev Littlewood, Peter Popov, and Lorenzo Strigini. Modeling software design diversity: a review. *ACM Computing Surveys (CSUR)*, 33(2):177–208, 2001.

[72] Michael Lyu, editor. *Software Fault Tolerance*. John Wiley & Sons, Ltd., 1995.

[73] Makoto Matsumoto and Takuji Nishimura. Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator. *ACM Transactions on Modeling and Computer Simulation*, 8(1):3–30, January 1998.

[74] Milena Milenković, Aleksandar Milenčović, and Emil Jovanov. A framework for trusted instruction execution via basic block signature verification. In *Proceedings of the 42nd annual Southeast regional conference (ACM SE'04)*, pages 191–196, Huntsville, Alabama, April 2 - 3 2004. ACM Press.

[75] Jelena Mirkovic, Gregory Prier, and Peter L. Reiher. Attacking ddos at the source. In *ICNP*, pages 312–321, 2002.

[76] Jelena Mirkovic and Peter Reiher. A taxonomy of ddos attack and ddos defense mechanisms. *SIGCOMM Comput. Commun. Rev.*, 34(2):39–53, 2004.

[77] Erich M. Nahum. Deconstructing specweb99. In *Proceedings of 7th International Workshop on Web Content Caching and Distribution*, Boulder, Colorado, U.S.A., August 2002.

[78] D. Nebenzahl and A. Wool. Install-time vaccination of Windows executables to defend against stack smashing attacks. In *Proceedings of the 19th IFIP International Information Security Conference*, pages 225–240, Toulouse, France, August 2004. Kluwer.

[79] George C. Necula, Scott McPeak, and Westley Weimer. Ccured: Type-safe retrofitting of legacy code. In *Proceedings of the Symposium on Principles of Programming Languages*, pages 128–139, January 2002.

[80] Nergal. The advanced return-into-lib(c) exploits. *Phrack*, 58(4), December 2001.

[81] Nicholas Nethercote and Julian Seward. Valgrind: A program supervision framework. In Oleg Sokolsky and Mahesh Viswanathan, editors, *Electronic Notes in Theoretical Computer Science*, volume 89. Elsevier, 2003.

[82] PaX Team. Documentation for the PaX project. In *Homepage of The PaX Team*, http://pax.grsecurity.net/docs/index.html, Last consulted: September, 2004. Last modified: December 2003.

[83] Vernor Paxson. Computing tcp's retransmission timer. RFC 2988, Network Working Group, November 2000.

[84] Manish Prasad and Tzi-cker Chiueh. A binary rewriting defense against stack based overflow attacks. In *Proceedings of the USENIX 2003 annual technical conference*, San Antonio, Texas, U.S.A., June 9-14 2003.

[85] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, second, (1994 reprint with corrections) edition, 1992.

[86] Calton Pu, Andrew Black, Crispin Cowan, and Jonathan Walpole. A specialization toolkit to increase the diversity of operating systems. In *Proceedings of the 1996 ICMAS Workshop on Immunity-Based Systems*, Nara, Japan, December 1996.

[87] Brian Randell. System Structure for Software Fault Tolerance. *IEEE Transactions in Software Engineering*, 1(2):220–232, 1975.

[88] Olatunji Ruwase and Monica S. Lam. A practical dynamic buffer overflow detector. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium*, February 2004.

[89] Christoph L. Schuba, Ivan V. Krsul, Markus G. Kuhn, Eugene H. spafford, Aurobindo Sundaram, and Diego Zamboni. Analysis of a denial of service attack on tcp. In *SP '97: Proceedings of the 1997 IEEE Symposium on Security and Privacy*, page 208. IEEE Computer Society, 1997.

[90] Julian Seward and Nicholas Nethercote. Valgrind, an open-source memory debugger for x86-GNU/Linux. In *http://valgrind.kde.org/*, 2004.

[91] Istvan Simon. A comparative analysis of methods of defense against buffer overflow attacks. Web publishing, California State University, Hayward, http://www.mcs.csuhayward.edu/ simon/security/boflo.html, January 31 2001.

[92] Daniel D. Sleator and Robert E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, 1985.

[93] SPEC Inc. Specweb99. Technical Report SPECweb99_Design_062999.html, SPEC Inc., June 29 1999.

[94] Andrew S. Tanenbaum. *Computer Networks*. Prentice Hall PTR, fourth edition, August 2002.

[95] TCP SYN Flooding and IP Spoofing Attacks. Number CERT Advisory CA-1996-21, November 29 2000.

[96] TCPA Trusted Computing Platform Alliance. In *http://www.trustedcomputing.org/home*, Last consulted: September 2004 2004.

[97] The HoneyNet Project. In *http://project.honeynet.org/*. The Honeynet Project, 2002.

[98] Tool Interface Standards Committee. *Executable and Linking Format (ELF)*, May 1995.

[99] Timothy Tsai and Navjot Singh. Libsafe 2.0: Detection of format string vulnerability exploits. White Paper Version 3-21-01, Avaya Labs, Avaya Inc., February 6 2001.

[100] Theodore Tso. random.c A strong random number generator. In *http://www.linuxsecurity.com/feature_stories/random.c*, Last modified: April 26, 1998 1998.

[101] Vendicator. StackShield: A stack smashing technique protection tool for Linux. In *http://angelfire.com/sk/stackshield*, Last consulted: September 4, 2004. Last site update: January 8 2000.

[102] David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. A First Step towards Automated Detection of Buffer Overrun Vulnerabilities. In *Network and Distributed System Security Symposium*, pages 3–17, San Diego, CA, February 2000.

[103] John Wilander and Mariam Kamkar. A comparison of publicly available tools for dynamic buffer overflow prevention. In *Proceedings of the 10th Network and Distributed System Security Symposium*, pages 149–162, San Diego, California, February 2003.

[104] Jie Xu, Brian Randell, C. Rubira, and R. Strod. Toward an Object-Oriented Approach to Software Fault Tolerance. In D. Avreski, editor, *Fault-Tolerant Parallel and Distributed Systems*. IEEE CS Press, 1995.

[105] Jun Xu, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. Transparent runtime randomization for security. In *Proceeding of the 22nd international symposium on*

reliable distributed systems (SRDS'03), pages 26–272, Florence, Italy, October 06-08 2003.

[106] Jun Xu, Zbigniew Kalbarczyk, Sanjay Patel, and Ravishankar K. Iyer. Architecture support for defending against buffer overflow attacks. In *Second Workshop on Evaluating and Architecting System dependabilitY (EASY), http://www.crhc.uiuc.edu/EASY/*, San Jose, California, October 2002.

[107] Yongguang Zhang, Harrick Vin, Lorenzo Alvisi, Wenke Lee, and Son K. Dao. Heterogeneous Networking: A New Survivability Paradigm. In *New Security Paradigms Workshop*, Cloudcroft, New Mexico, September 2001.

[108] Douglas Zongker and Bill Punch. lil-gp Genetic Programming System. In *http://garage.cps.msu.edu/software/lil-gp/*, Last accessed January 2005.