OPEN ACCESS Freely available online

Journal of Information Technology and Software Engineering

Research Article

Improving the Security of Time-Bounded Mission-Critical K-Variant Systems by Inserting Unreachable Code

Berk Bekiroglu*, **Bogdan Korel**

Department of Computer Science, Illinois Institute of Technology, Chicago, USA

ABSTRACT

K-variant is a multiple variant architecture designed to enhance the security of time-constrained systems, particularly against memory exploitation attacks. In a K-variant system, variants are generated through controlled source code program transformations. Inserting unreachable code is one of the program transformations used to generate new variants in K-variant systems. It is possible to generate functionally equivalent programs by inserting random unreachable code. Because critical instructions in memory are shifted by inserting unreachable code, the survivability of K-variant systems can be improved against memory exploitation attacks. The purpose of this study is to determine the effectiveness of inserting unreachable code in enhancing the security of time-bounded K-variant systems against memory exploitation attacks. The effect of inserting unreachable code on the survivability of time-bounded K-variant systems is investigated experimentally for a variety of memory attacks. The results indicate that increasing the number of variants by inserting unreachable code significantly improves the survivability of time-bounded K-variant systems against memory exploitation attacks. We conclude that introducing unreachable code into time-bounded K-variant systems significantly improves the security of time-bounded K-variant systems against memory exploitation attacks while maintaining a reasonable runtime and memory overhead.

Keywords: K-variant architecture; Source code transformation; Memory exploitation attacks; System security; Multiple variant architecture

INTRODUCTION

Mission-critical systems have stringent security requirements during operation, as compromising them can result in catastrophic losses. Numerous attacks may occur during the operation of those systems to compromise them. Failure to implement such systems can result in legal and financial consequences for organizations. As a result, numerous defense mechanisms have been developed to enhance these systems' survivability.

Memory exploitation vulnerabilities enable adversaries to take control of systems, expose sensitive data, or simply cause systems to crash and fail. By exploiting memory vulnerabilities such as a buffer overflow and a format string, attackers can write and read arbitrary memory locations. Numerous defense mechanisms have been implemented to enhance security against memory vulnerabilities. Operating systems employ a variety of defense mechanisms, including address space layout randomization, Data Execution Prevention (DEP), and stack canaries. Nonetheless, adversaries have breached these defense mechanisms. Additionally, these defense mechanisms and their variants terminate programs properly when they detect attacks. That is not a possibility for time-constrained mission-critical systems, as re-running a failed program is impossible [1-10].

Multiple variant architectures have been used to increase security by reducing successful attacks. Generally, multi-variant execution provides statistical protection to defenders. The K-variant is one of the fault-tolerant architectures used to improve the security of systems with multi-variant execution. K-variant systems enable the execution of functionally equivalent programs concurrently to accomplish a task or mission. A task or mission can be completed with at least one surviving variant from an attack.

Correspondence to: Berk Bekiroglu, Department of Computer Science, Illinois Institute of Technology, Chicago, USA, Tel: 3129122538; E-mail: bbekirog@iit.edu

Received: 01-Mar-2022, Manuscript No. JITSE-22-15849; **Editor assigned:** 03-Mar-2022, PreQC No. JITSE-22-15849 (PQ); **Reviewed:** 17-Mar-2022, QC No. JITSE-22-15849; **Revised:** 02-May-2022, Manuscript No. JITSE-22-15849 (R); **Published:** 09-May-2022, DOI: 10.35248/2165-7866.22.12.301

Citation: Bekiroglu B, Korel B (2022) Improving the Security of Time-Bounded Mission-Critical K-Variant Systems by Inserting Unreachable Code. J Inform Tech Softw Eng. 12:301

The level of diversity distinguishes the K-variant architecture from other multi-variant execution systems. Numerous multi-variant execution systems generate new variants *via* binary-level program transformations. On the other hand, the K-variant architecture generates variants through simple, secure, and inexpensive source code program transformations. Because program transformations are performed at the source code level, variant portability is significantly increased. The adaptability of transformed programs' compilation enables them to run in a variety of environments, which is impossible with operating system-dependent binary program transformations. On the other hand, if the source code is not available, the K-variant architecture is not applicable. In that case, other multi-execution systems that utilize binary-level program transformations may be preferred. In addition, source code program transformations require the compilation of programs. Thus, unlike binary program transformations, K-variant systems have a one-time compilation cost before running the system.

The survivability of K-variant systems can be significantly increased in the face of memory exploitation attacks directed at the program's data segment in memory. Inserting dummy buffers, increasing the size of existing buffers, increasing the dimension of existing buffers, and converting primitive data type variables to buffers are all examples of program transformations that move vulnerable data into memory. Nonetheless, none of these program transformations alter the memory location of the code (text) segments of programs. As a result, if adversaries compromise an address within a code segment, these program transformations do not increase the survivability of K-variant systems. The purpose of this paper is to shift vulnerable instructions in memory for each variant in order to increase security against memory exploitation attacks directed at the code segment. This paper introduces a program transformation technique for inserting unreachable code into the code segment of a program in order to shift the vulnerable instructions. Inserting code that is unreachable has no effect on the program's normal execution. When comparing the original program, the memory addresses of critical instructions are shifted by introducing unreachable code. As a result, the survivability of multiple variant systems against memory exploitation attacks may be enhanced [11-16].

This research incorporates the following contributions:

- For K-variant systems, a program transformation for inserting unreachable code is presented.
- Experiments are conducted to determine the effectiveness of program transformation on the survivability of K-variant systems under various attack types.
- Investigating and comparing the effectiveness of various attack types against an increasing number of variants and attack attempts.
- Investigation of the runtime and memory overhead associated with systems generated by the insertion of unreachable code transformations.
- The experimental results indicate that generating multiple variants by inserting unreachable code results in a significant increase in survivability at a reasonable cost.

The remainder of this paper will be structured as follows. Section 2 discusses the architecture and attack model of the K-Variant. Section 3 discusses the underlying research. Section 4 provides an illustrative example of how to transform a program by inserting unreachable code. Section 5 describes the program transformation that occurs when unreachable code is inserted. Section 6 discusses the different attack types that exist in K-variant systems. Section 7 discusses the experimental study and its findings, which demonstrate the effectiveness and cost of the program transformation. Section 8 concludes and describes future works.

MATERIALS AND METHODS

K-Variant architecture and attack model

The K-variant architecture is an alternative to the N-version architecture, which utilizes multiple versions of a program to complete a mission or task concurrently. Using the same specifications, different developer teams create variants of a system in the N-version architecture. Each version of the N-version architecture may be designed differently and even implemented in a different programming language. As a result, each version of the N-version architecture is expected to have a unique set of vulnerabilities. As a result, exploiting vulnerability in one variant may have no impact on other variants in a system.

While the N-version is an effective architecture for enhancing security, it is quite expensive. Each additional variant may cost the same as developing a new piece of software due to the fact that it requires unique design, implementation, and testing. As a result, the cost of the N-version architecture may double or triple for two- and three-version systems, respectively. To obtain the benefits of the N-version architecture at a reasonable cost, the K-variant may be used. The K-variant architecture generates variants automatically through program transformations. As a result, the cost of creating new variants has decreased significantly. Additionally, the K-variant architecture's program transformations generate functionally equivalent variants. As a result, no extensive software testing is required to ensure that each variant is correct [17-22].

The system depicted in Figure 1 is a K-version system with three variants. Variant #1 is the original program that has not been transformed. The system's other two variants are generated through program transformations. As illustrated in Figure 1, program transformations in a K-variant architecture modify or shift the memory addresses of critical instructions. The program trace contains critical instructions and changing any of them will almost certainly cause the program to crash or produce incorrect output. In the attack model depicted in Figure 1, the same memory address is attacked simultaneously in all variants. Variant #2 was compromised as a result of the attack, as the attacked address was vulnerable. Variant #1 and Variant #3, on the other hand, survived the attack. The attack failed because two variants continued to operate normally and produce correct outputs.

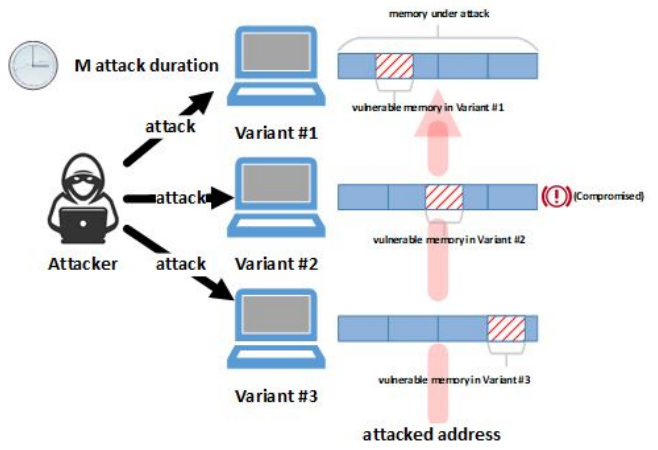

Figure 1: K-variant architecture with three variants.

As with other multi-variant/version execution architectures, the K-variant architecture requires a dependable voting mechanism to obtain a single result from multiple variants. Additionally, each variant's results can be passed through an acceptance test to eliminate compromised variants prior to voting.

A metric known as the probability of an unsuccessful attack (Pu) is used to determine the survivability of K-variant systems. Pu (K, M) denotes the probability of a system with K variants being successfully attacked after M attempts. If the attack's Pu (K, M) value is 1, the attack will always fail. As a result, Pu (K, M)=1 denotes the system with the greatest survivability in a K-variant. On the other hand, when Pu (K, M) is equal to zero during an attack, no K-variant system survives. All defense strategies in K-variant architecture attempt to maximize Pu in order to increase survivability [23-31].

The attack model from the previous study is used in this paper. In that attack model, adversaries are assumed to be capable of accessing and writing to any memory address *via* vulnerability such as a buffer overflow or format string. In actual attacks, this assumption may not always be correct due to the existence of additional security mechanisms such as No executable and NO Read and execute (NORAX). As a result, the attack model used in this paper represents the defender's worst-case scenario. Thus, in actual attacks, the effectiveness of program transformation in K-variant systems is expected to be significantly greater.

Background/Related works

Buffer overflow attacks may allow attackers to write memory addresses outside the buffers' allocated memory. When a program contains buffer overflow vulnerability, attackers can manipulate the call stack's return address by providing malicious input. Additionally, the attacker may embed malicious code within the input to execute it. To prevent such attacks in modern operating systems, the execution of code within the data segment is prohibited through the use of a security mechanism known as Data Execution Prevention (DEP). Nonetheless, attackers circumvent DEP by employing more sophisticated stack smashing attacks. Rather than providing malicious code as an input, Return-oriented Programming (ROP) makes use of gadgets (pre-existing instructions in memory that end in a return instruction such as ret). In ROP, the attacker must locate useful memory gadgets in order to execute the malicious code. ASLR is a defense mechanism that is also used in modern operating systems to make it more difficult to guess the location of processes and functions in memory. Each time a program is executed, the virtual addresses of the stack, heap, code, and library segments are randomized to reduce the likelihood of a successful ROP. Nonetheless, attackers can bypass ASLR by brute-forcing the address space and exploiting libraries that are not ASLR enabled. Control Flow Integrity is another effective defense mechanism against buffer overflow attacks (CFI). A Control Flow Graph (CFG) is generated from the source and binary code of a program to ensure that control instructions are transferred to a valid address. CFG determines the destination addresses during execution. As a result, whenever a destination address is modified, the CFI detects the attack. However, attackers may circumvent CFI by exploiting vulnerabilities that go unnoticed by CFI [32-40].

Numerous defense mechanisms and technologies have been developed to enhance memory exploitation attack security. No executable, No Read and Execute, Stack Smashing Protection (SSP), Code hiding, Address-space Layout Randomization, Control Flow Integrity, and Renew Stack Smashing Projector (Renews), are some of the techniques to enhance the security against the memory exploitation attacks.

Software diversity is a concept used to improve system security. Some randomizations, such as the addition or removal of non-functional instructions, may improve system security against buffer overflow attacks. Software diversities are implemented at various levels to improve system security, including application, execution, operating system, and hardware. To improve security against memory exploitation attacks, address space, stack space, heap, and instruction set are randomized for diversity. ROP (Return-oriented programming) attacks are also mitigated by code randomization techniques. Larsen and Sadeghi conducted a condensed survey on software diversity techniques and data leaks.

Numerous program transformation techniques are used to address security vulnerabilities in the authorization, authentication, and input validation processes Certain security flaws are caused by unsafe libraries such as "stripy" in the C programming language, which does not check buffer length. Program transformations can be used to substitute safe functions for unsafe functions. Additionally, program transformations can be used to mitigate the C programming language's integer vulnerability. Also, secure programming languages such as Rust can be used to circumvent security vulnerabilities.

Multi-execution has been used to enhance security in the context of information flow control. Data integrity and confidentiality violations can be avoided by executing a program with varying levels of security. Secure multi-execution is a sound and precise procedure.

Introducing garbage instructions, such as No-Operation Instructions (NOPs) or new complex instructions, into a

program is a way to increase security against a variety of attacks, including memory corruption, code injection, and code reuse.

RESULTS

Statistical protection against memory exploitation attacks has also been achieved through automated software diversity. At the instruction level, automated software diversity techniques include garbage code insertion, random register allocation, and instruction reordering. Analytically, the process of generating multiple variants in K-variant systems by inserting dummy buffers was investigated. A theoretical simulator demonstrated that inserting dummy buffers can significantly increase the system's survivability against memory exploitation attacks. Finally, the effectiveness of the K-variant architecture against memory exploitation attacks targeting data segments in memory was investigated.

The motivating example for inserting unreachable code

This section will demonstrate the effectiveness of unreachable code transformation on code segment attacks. The example demonstrated that while other program transformations in K-variant systems are ineffective at preventing an attack that manipulates the program trace *via* stack smashing, the unreachable code transformation is.

In K-variants systems, the program transformation of inserting dummy buffers has been used to generate new variants to improve security against memory exploitation attacks. The vulnerable data shifter program transformations relocate vulnerable data in a program's memory segment. Some of the vulnerable data shifter program transformations include inserting dummy buffers, increasing the size of buffers, increasing the dimension of buffers, and converting primitive data type variables into buffers. Although vulnerable data shifter program transformations improve attacks on data segments, they are ineffective against attacks on memory code segments. Thus, the program transformation of inserting unreachable code is introduced to shift the vulnerable code in memory.

(Figure 2) illustrates a motivating example of inserting unreachable code. The following motivating example demonstrates the advantage of inserting unreachable code over vulnerable data shifter program transformations. Three print calls are required in Figure 2a to obtain the expected outputs from the original program. The attacker takes advantage of vulnerability such as a buffer overflow or a format string to execute code in memory that skips two print statements, thereby completing the mission. The source code and malicious binary code are shown in Figure 2b, which is either stored in memory or injected by attackers exploiting vulnerability. The malicious Call function in Figure 2b enables attackers to conduct a stack smashing attack, which modifies the program trace by manipulating the program stack's return address. The malicious call at line 3 in Figure 2b causes the last two print calls to be skipped, preventing the expected outputs from being generated.

One of the vulnerable data shifter program transformations is inserting dummy buffers. The transformed program is depicted in Figure 2c. The stack smashing attack is still successful after inserting dummy buffers into the attacked program in Figure 2c. Because of the malicious code, the last two print calls are still not being executed. The illustrative example shows that vulnerable data shifter program transformations provide no benefit in such attacks.

Finally, an unreachable code is inserted into the source code in Figure 2d. The unreachable chunk contains program statements that were generated at random, including integer declaration and assignment, array declaration and initialization, and a compound while condition statement. To ensure that the unreachable statement chunk is not executable, it is enclosed in an if (false) condition statement. Following the unreachable statement transformation, the final two print calls are executed, producing all outputs.

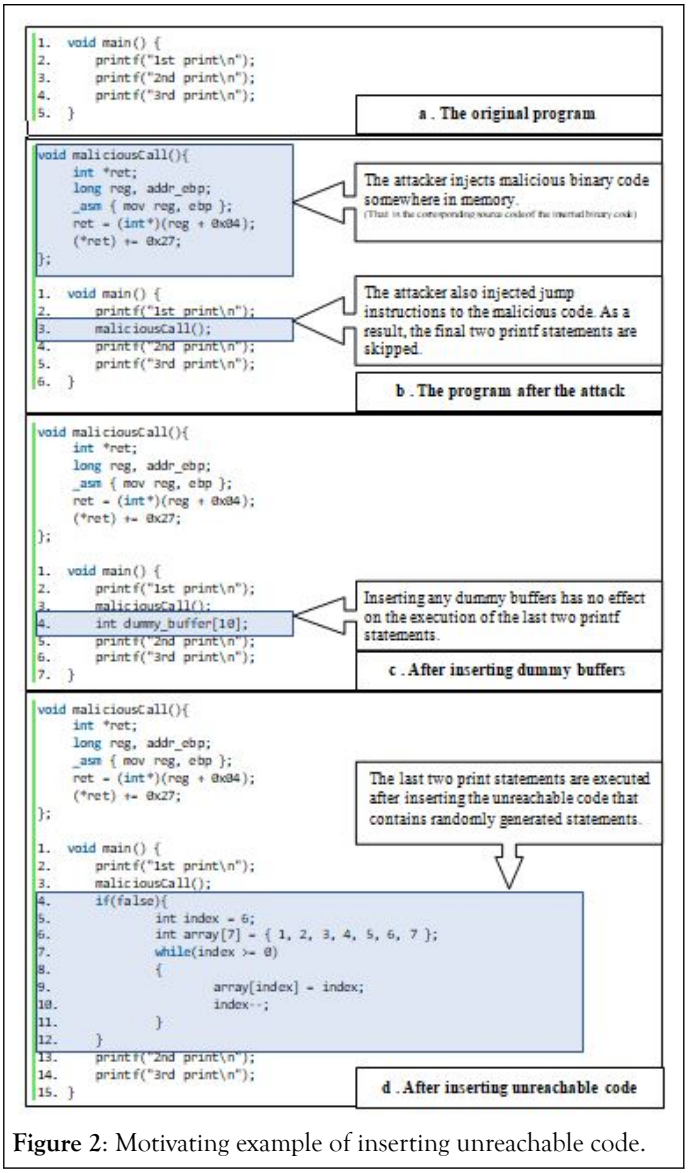

Figure 2: Motivating example of inserting unreachable code.

As demonstrated by the motivating example, inserting unreachable code can provide a variety of critical instructions in the memory code segment. If an attacker can inject malicious code into the real world, he will most likely execute more harmful operations than just skipping instructions. However, the attack model investigated in this paper neither intends to execute malicious code nor takes control of the time-bounded

mission-critical systems. However, the attack model in this paper involves skipping or manipulating instructions in the memory to prevent the system from producing expected outcomes during a time-bound mission. Injecting malicious code is one of the techniques used to implement the attack model in this paper. Buffer overflow vulnerabilities, format string bugs, and malicious programs may also perform the same type of attacks similar to the motivating example. The motivating example shows how inserting unreachable code may help improve the survivability of these types of attacks.

Program transformation of inserting unreachable code

The purpose of program transformation is to shift critical instructions in memory in order to increase the survivability of K-variant systems. Unreachable statements are inserted at random locations throughout the source code. Unreachable statements consume a small amount of memory. They do not, however, increase the runtime of a program. The following benefits accrue from incorporating unreachable code:

- It is a simple program transformation. Therefore, it can be automated easily.
- It is a safe program transformation. This means that program transformation does not introduce new bugs into the program. Therefore, no extensive software testing is required for the transformed variants.
- In theory, there is no limit to the number of unreachable statements that can be inserted. As a result, any number of unreachable statements can be inserted if sufficient memory and disk space are available.
- There are no changes to the existing source code. Thus, the existing source code is preserved.
- In contrast to binary level program transformations, the transformed source can be recompiled in other environments. This improves the portability of transformed programs.

Figure 3 depicts the program's memory layout before and after inserting unreachable code. Unreachable statement instructions are inserted in memory between two program instructions, as shown in Figure 3b. Some program instructions are shifted in memory in this manner. Some instructions, however, remain at the same memory addresses. Because the attacker can compromise more than one variant in a single attack, this may cause weakness in K-variant systems. In order to insert no executable statements at the source code level, a small number of executable statements must be inserted. For example, an unreachable code is inserted inside a false condition. Furthermore, because those condition instructions can be executed, they are vulnerable to attacks. That is because manipulating an executable condition instruction may hang the program or fail the program so that expected outputs are not produced.

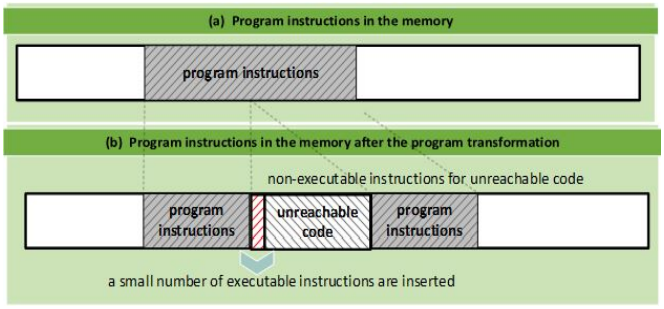

Figure 3: The memory layout of a program before and after inserting unreachable code.

DISCUSSION

The pseudo code for the program transformation is shown in Figure 4. The original source code (sc), the maximum number of unreachable statement chunks (max# Of Chunks), and the maximum number of unreachable statements within an unreachable statement chunk (max# Of Statements) are used as inputs to the program transformation. The transformed source code (sc′) is the output of the program transformation. Unreachable statements are inserted into chunks surrounded by no executable conditions. Following a random selection of unreachable statement chunks in line 2, the loop in line 4 generates random statement chunks in random locations and inserts them into the source code. Within the loop on line 4, a random location (loc) for the chunk in line 5 is chosen. In line 6, an if (false) condition chunk is inserted into the source code to generate unreachable code. Line 7 randomly selects the number of statements (# Of Statements) in each chunk. Line 8's inner loop generates random unreachable statements and inserts them into the condition chunk if (false). The program transformation is complete when the loop in line 4 terminates and the transformed source code is returned.

```
inputs
  sc: Original source code
  max#OfChunks: The maximum number of unreachable statement chunks
  max#OfStatements: The maximum number of unreachable statements in a chunk
output
  sc': transformed source code
1.  begin
2.    #OfStatementChunks = select a random number between 1 and max#OfChunks
3.    sc' = sc
4.    for i = 1 to # OfStatementChunks do
5.      loc = select a location after a random simple non-declarative statement in sc'
6.      insert "if(false)" condition chunk at loc in sc'
7.      #OfStatements = select a random number between 1 and max#OfStatements
8.      for i = 1 to #OfStatements do
9.         unreachableStatement = generate any random statement
10.        if unreachableStatement is a compound statement then
11.           simpStatements = generate a random # of simple statements
12.           insert simpStatements into the compound statement in unreachableStatement
13.        end if
14.        sc' = insert unreachableStatement inside "if(false)" condition chunk
15.    end for
16.  end for
17.  output sc'
18. end
```

Figure 4: Pseudo code of inserting unreachable code.

Unreachable statements are no executable program statements in this paper. In other words, there are no such program inputs to execute unreachable statements. Unreachable statements are encased in if (false) conditions to prevent them from being executed. Unreachable statements can be simple program statements (assignment, go to, return, call, etc.), compound statements (do-loop, for-loop, if-statement, switch statement,

while-loop, etc.), or declaration statements. For two reasons, declaring statements is preferable to producing unreachable statements. Firstly, one declaration statement corresponds to numerous assembly instructions. Second, the number of unreachable instructions associated with the assignment statement can be determined easily.

(Figure 5) illustrates an example program transformation involving the insertion of unreachable code. Figure 5a depicts the original program's partial source code prior to program transformation. Three unreachable statement chunks are inserted into random locations following the program transformation depicted in Figure 5b. Between lines 2 and 4, the first unreachable statement chunk is inserted. It only contains one assignment statement. The second chunk of unreachable statements is located between lines 8 and 12. The second chunk of unreachable statements includes a loop statement and two random calculation statements. Finally, the last chunk of unreachable statements is between lines 15 and 18. The final chunk contains random declaration and calculation statements. Each chunk of unreachable statements is surrounded by and if (false) condition. Thus, none of the unreachable statements in the source code are executable.

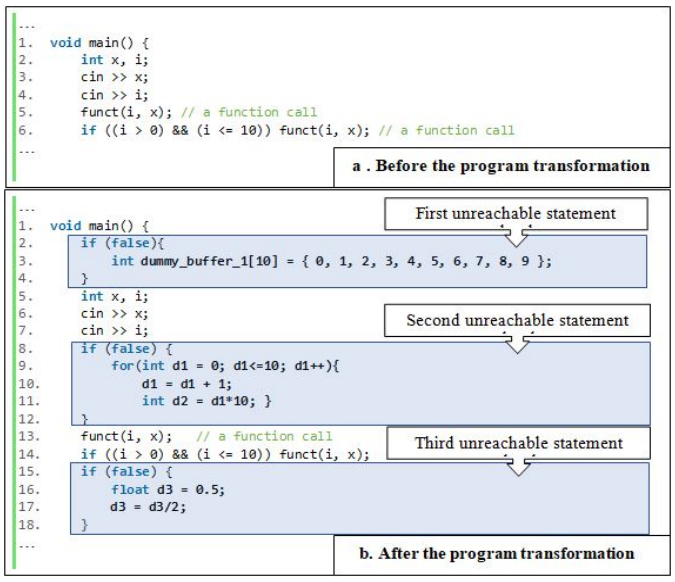

Figure 5: Inserting three unreachable statement chunks. (a) The partial source code before the program transformation. (b) The partial source code after the program transformation.

The assembly code for the first unreachable statement chunk in Figure 5 is depicted in (Figure 6). Microsoft Visual Studio is used to generate the assembly code. As illustrated in (Figure 6), a single statement can be interpreted as a collection of assembly instructions. The if condition statement is equivalent to two assembly instructions located at the addresses 00357E3C and 00357E3E. These two instructions are executable and require a small amount of CPU time. The assignment statement in Figure 5 corresponds to ten assembly instructions located at addresses 00357E40 to 00357E7F. Due to the fact that these assembly instructions are enclosed in the condition, which is always false; they do not consume additional CPU time during program execution.

Compiler optimizations are widely used and useful to improve the performance of programs. Most modern compilers optimize source code in such a way that no binary code for unreachable code is generated. Compiler optimization for dead code (unreachable code) elimination should be disabled to perform the program transformations depicted in (Figure 5). Otherwise, unreachable statements are not converted into binary code. Compilers usually have an option to disable the optimization of dead code elimination. After disabling that optimization, compilers generate binaries for unreachable code, as illustrated in (Figure 6).

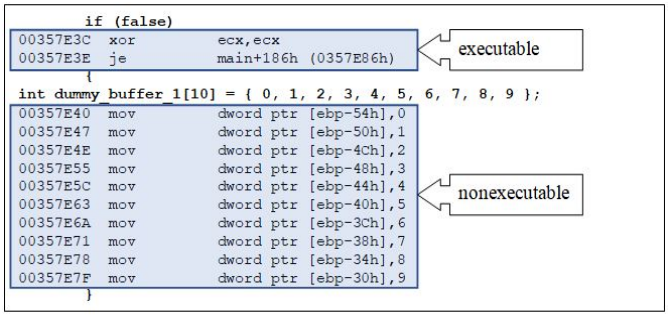

Figure 6: Assembly code for the first unreachable statement chunk in (Figure 5), after compiling the program.

To summarize, inserting unreachable code is a simple and safe program transformation. Various techniques exist for generating unreachable code. To enclose unreachable statements, a small number of executable statements need to be inserted into the source code. By inserting unreachable code, vulnerable instructions can be sifted in memory.

Attack types

Buffer overflow vulnerability and programming bugs like format string may allow attackers to write to arbitrary memory addresses. For the defenders, this is the worst-case scenario. Because of other defense mechanisms, writing arbitrary memory addresses by exploiting vulnerability may not be possible in actual attacks. As a result, the proposed program transformation may offer higher security in actual attacks.

When performing memory exploitation attacks, a variety of strategies can be used to select target addresses. The previous study examined four attack types in K-variant systems. There are four different types of attacks: uniform, normal distribution, binary, and stepwise. Random attacks use uniform and normal distributions. Binary and stepwise attacks, on the other hand, are deterministic attacks in which all target addresses are predetermined. Unlike random attacks, deterministic attacks require additional information, such as the memory size targeted and the maximum number of attack attempts.

The attacker chooses a memory address based on the attack type in this paper's attack model. If address x is chosen to be manipulated, all variants of instructions at address x are manipulated in a single attack. For a successful attack, the attacker must compromise all variants within the allowed number of attack attempts. Because a greater number of variants must be compromised for a successful attack, increasing the

number of variants is likely to increase the survivability of K-variant systems.

N denotes the size of the memory under attack in this section. Additionally, n indicates the size of the vulnerable memory, which must be equal to or less than N. The value n/N indicates the percentage of vulnerable memory. Besides that, M denotes the number of attack attempts and is a good proxy for the attack duration. As a result, M refers to the duration of the attack in the experimental study. The remainder of this section describes four attack types that were used in the experimental study. These attack types were also presented.

Uniform attack: When little or no information about the target system is available, attacked addresses can be chosen using a uniform distribution. Thus, target addresses are chosen randomly, and the probability of selecting each address is the same, which is 1/N. For K-variant systems, the uniform attack was first analyzed.

Figure 7 depicts five uniform attacks in memory. Five addresses were chosen at random from the program's first and last available addresses in memory. The first four attacks compromise non-vulnerable instructions, allowing the program to continue to run normally. The fifth attack, on the other hand, manipulates the vulnerable memory, causing the program to produce incorrect outputs or to crash.

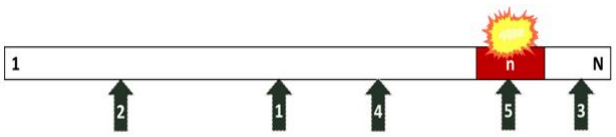

Figure 7: Uniform Attack.

Normal distribution attack: A normal distribution can be used to choose target addresses. As is the case with the uniform attack, the first attack is chosen at random using a uniform distribution. However, the second and subsequent attacks are chosen randomly from a normal distribution with a mean in the first attacked address. This attack enables adversaries to choose addresses in the vicinity of the initial attack. In contrast to the uniform attack, the normal distribution attack targets a single region of memory. It is improbable that memory addresses far from the initial attack will be chosen. Addresses located near the initial attack, on the other hand, are more likely to become targets.

The range parameter specifies the attack's perimeter. If the range is large, the attacked addresses are chosen from a larger pool of addresses. If the range is narrow, on the other hand, the target addresses are chosen from a small address set. Figure 8 illustrates five attempts of the normal distribution attack. The initial attack is chosen at random using a uniform distribution. The remaining four attacks are generated at random using a specified range of a normal distribution. Outside the range, the probability of selecting a target address is nil. Due to the vulnerability of the fifth selected address, the system is compromised in Figure 8.

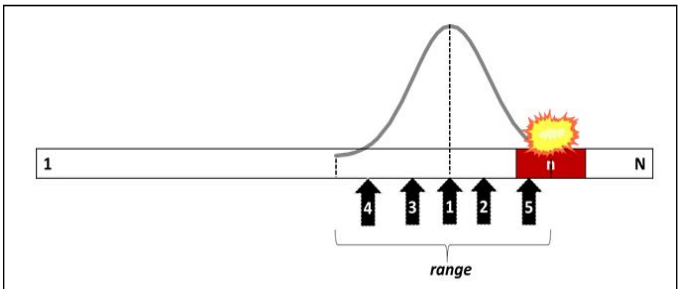

Figure 8: Normal distribution attack.

Binary attack: Target addresses do not have to be chosen at random all of the time. To select target addresses, the attacker may use a systematic approach. To select target addresses, a binary tree's breadth-first traversal is used. The first attacked address in this attack type is ½ N, which is the middle address of the attacked address space. Addresses ¼ N, ¾ N, 1/8 N, and 3/8 N follow this attack. An obvious assumption in this type of attack is that the attacker knows the size of the memory (N). Random attacks do not have to have this information.

Figure 9 depicts the binary attack on memory for seven attempts. When advancing to the next level of a binary tree, the gap between attacked addresses shrinks. If the attack lasts long enough, there is a greater chance of compromising a vulnerable address. Unlike random attacks, deterministic attacks guarantee a successful attack after a certain number of attempts, known as a survivability breaking point.

Because the same addresses are used in each repetition, the binary attack may produce the same Pu. To perform the binary attack with different results, the memory size under attack (N) can be shrunk by a random small size (r) so that different addresses are selected in each iteration. As a result, the attacked addresses in Figure 9 will differ slightly depending on the r in each repetition of the attack.

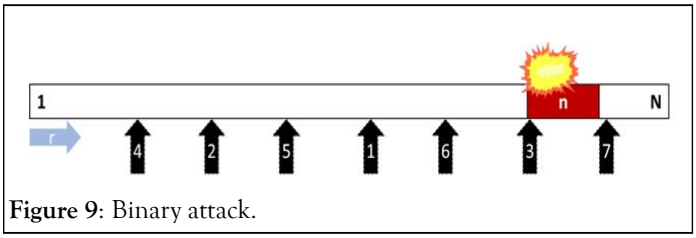

Figure 9: Binary attack.

Stepwise attack: Similar to the binary attack, the stepwise is a deterministic attack in which attacked addresses are compromised systematically. The first address attacked is the first available address, and the last address attacked is the last available address. Beginning with the first available address, addresses are compromised linearly by constant delta steps (Δ). Formula 1 computes the delta step.

Figure 10 illustrates the memory addresses that were attacked during the stepwise attack. Like the binary attack, the stepwise attack produces identical results with each repetition. To avoid repeating the same addresses, a small random starting address (r) is chosen. As illustrated in Figure 10, the first attack targets the address following r. additionally; the final attack compromises the program's last address. When the number of attacks is large enough, the stepwise method also ensures a successful attack.

$$\Delta = \left\lceil \frac{N-1}{M-1} \right\rceil \qquad (1)$$

When the delta step is less than the size of the vulnerable memory, the system will be compromised by the stepwise attack.

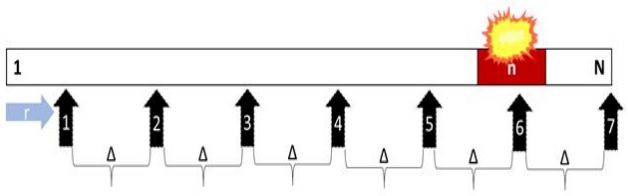

Figure 10: Binary attack.

Experimental study: The purpose of this experiment is to determine the effect of introducing unreachable code on the survivability of K-variant systems under four different attack types: uniform random, normal distribution, binary, and stepwise.

The experimental study involves:

- investigating the effect of increasing the number of variants (K)
- investigating the effect of increasing the number of attack attempts (M) on the survivability of K-variant systems for programs that are generated by inserting unreachable code
- investigating and comparing the effectiveness of four attack types
- investigate runtime and memory overhead of transformed programs

A prototype of a C++ tool for automating the transformation of unreachable code has been developed. By applying the inserting unreachable code transformation, the tool automatically generates variants from a program. The tool randomly transforms the program, resulting in a unique variant being generated each time.

The remainder of this section describes the steps taken by the tool to calculate the probability of an unsuccessful attack (Pu) and the experimental results.

The algorithm for estimating Pu is depicted in Figure 11. The tool accepts the following inputs: the number of variants (K), the number of attack attempts (M), the type of attack (A), the source code (S), and the test suite (TS). The tool's output is the probability of an unsuccessful attack (Pu). Additionally, the tool includes two parameters, RF and RA, which affect the accuracy of Pu estimations. RF represents the repetition factor of K-variant generation. In other words, RF denotes the number of distinct K-variant systems generated by inserting unreachable code. The other tool parameter is RA, which represents the number of times M attacks are repeated in the repetition factor. RA denotes the repetition of the entire attack process for each K-variant system generated. Increases in RF and RA result in more accurate estimations of Pu. However, it necessitates a greater number of executions.

The loop in line 3 of the algorithm in Figure 11 generates RF different K-variant systems. Line 3 of the loop generates K-1 source codes from the original source code by inserting unreachable code, as shown in Figure 4. In line 4, the source codes for K-1 variants and the original program are compiled. The attack is then run on the generated K-variant system. Line 6's loop repeats the entire attack simulation RA times. In line 7, M memory addresses are chosen for each repetition based on attack type A. In line 8, binaries are then manipulated from the selected memory addresses. Inside the loop in line 9, manipulated variants or binaries are executed with test cases from test suite TS. Each test case in test suite TS is associated with a single mission. As a result, only one test case is executed in each run of the experiment. If at least one of the variants passes a test case, the # of Unsuccessful attacks is increased. In line 3, at the end of the loop, Pu is obtained by dividing # of unsuccessful attacks by the total number of executed test cases, which is the product of RF, RA, and the number of test cases in TS.

```
Inputs
 K: The number of variants
 M: The number of attack attempts
 A: Attack type
 S: Source code of a program
 TS: Test Suite
Output
 Pu: Probability of an unsuccessful attack
Tool parameters
 RF: Repetition factor of K-variant generation
 RA: The number of repetitions of M-attacks in repetition factor
1.  Begin
2.  #ofUnsuccesfulAttacks = 0
3.  for i = 1 to RF do
4.     Generate K-1 source codes from source code S by inserting unreachable code
5.     Compile K source codes to generate K binaries
6.     for i = 1 to RA do
7.        Select M memory addresses by using attack A
8.        Modify K binaries based on selected M addresses
9.        for each TestCase in TS do
10.          Run K binaries with TestCase
11.          if at least one binary passed TestCase successfully then
12.              #ofUnsuccesfulAttacks++
13.          end if
14.       end for
15.    end for
16.  end for
17.  Pu = (#ofUnsuccesfulAttacks)/(RF*RA*sizeOf(TS))
18.  return Pu
19. end
```

Figure 11: Pseudo code of the tool to estimate Pu.

RF (Repetition Factor of K-variant generation) and RA (Number of repetitions of M-attack in repetition factor) are both ten in this experimental study. In addition, for each attacked variant, ten test cases with statement coverage ranging from 50% to 100% are executed. Therefore, the number of executions, RF·RA· size of (TS), in the estimation of Pu in Figure 11 is 1000. When generating each variant, the program transformation inserts at most five unreachable statement chunks into the source code. Furthermore, each unreachable statement chunk can only contain five statements. Unreachable statements are also made up of random buffer declarations and initializations. The data types and sizes of buffers in unreachable statements are also chosen at random. Furthermore, the size of each buffer is at

most 500. As a result, at most 500 elements are initialized in each buffer with randomly generated data of the selected data type.

The attacker is assumed to be capable of writing to any memory address in the attack model used in the experimental study. This may be impossible in real-world attacks. As a result, the experimental study's attack model represents the defender's worst-case scenario. Thus, the effectiveness of the program transformations can be much higher in actual attacks.

The number of attack attempts accurately represents the duration of the attack. The number of attack attempts increases as the attack duration increases. Thus, the attack duration refers to the attack durations in this paper or vice versa. In the emulation, one byte of an executable file is manipulated in each attack attempt. The attacked byte is filled with "0". In this experiment, five, ten, and twenty bytes are manipulated in short, medium, and long attacks, respectively. Twenty is the maximum number of attack attempts in the experimental study because a program has a slim chance of surviving more than twenty attacks.

The scope of this experimental study is as follows:

- The experimental study employs three programs ranging in size from 500 LOC to 5 KLOC. Table 1 contains basic information about selected programs.
- Variants are run for ten test cases with three attack durations (or attack attempts): short, medium, and long attacks.
- The analysis is performed on up to ten-variant systems.
- When selecting programs for the experimental study, the following criteria were used.
- C or C++ programs are popular and not memory-safe. As a result, programmers are accountable for memory-related security concerns.
- Open-source and accessible through public code repositories such as Gather and Code project.
- Because the tools used in the experimental study were written in. NET, the programs must be compiled and run in a Windows environment (Table 1).

Table 1: Basic information about programs in the experimental study.

Program #	Source code size	Binary size	Description
Program #1	1000 LOC	203 KB	The optimum routing path is determined using a linked state protocol.
Program #2	500 LOC	102 KB	Encrypts plain texts
Program #3	5000 LOC	313 KB	Provides services for storing, retrieving, and deleting user data.

The attacks are directed against the entire program, including all of its parts (data, instruction/text, and metadata). As a result, the attack range is 100%. The following sections present the results of the experimental study for three programs. Finally, this section compares attack types to the average Pu of all programs and discusses deductions for three programs and three attack durations.

Figure 12 depicts the effectiveness of inserting unreachable code into Program #1. (Figures 12a-c) show three line charts that correspond to different attack durations. Figure 12 shows the Probability of an unsuccessful attack (Pu) versus the number of variants for four different attack types. Looking at Figure 12a, it is clear that introducing a couple of variants in a K-variant system improves Pu significantly in a short attack. Even after introducing the second variant in binary and stepwise attacks, Pu can reach one. Random attacks, including uniform and normal distribution attacks are more likely to compromise systems than deterministic attacks with two to seven variants. Furthermore, there is no need to add eight or more variants because Pu has already achieved nearly one in all attack types.

When the average Pu of all variants is considered in Figure 12a, the uniform attack has the lowest Pu (0.902) of the four attack types. Pu fell dramatically when the number of attacks was doubled, as shown in Figure 12b. What stands out in this line chart is that increasing the number of variants does not result in a noticeable improvement in Pu for the binary attack. Except for the single variant system, the binary attack still produces the lowest Pu. Furthermore, Pu grew significantly in stepwise and uniform attacks. Despite the fact that increasing the number of variants increases Pu significantly for the normal distribution attack, it maintains the lowest Pu for all variants among all attack types. As shown in Figure 12c, Pu falls precipitously in all attack types when the attack duration is long. The data in Figure 12c is interesting because it shows that binary and stepwise attacks compromise systems regardless of the number of variants in the system in twenty attacks. Both deterministic attacks nearly reached their survivability breaking points in twenty attacks against Program #1. As a result, increasing the number of variants has no effect on Pu. Furthermore, the normal distribution attack has the highest survivability because it only targets specific memory parts. Increasing the number of variants improves Pu in both random and deterministic attacks. Pu is expected to increase after the tenth variant in the long attack in uniform and normal distribution attacks.

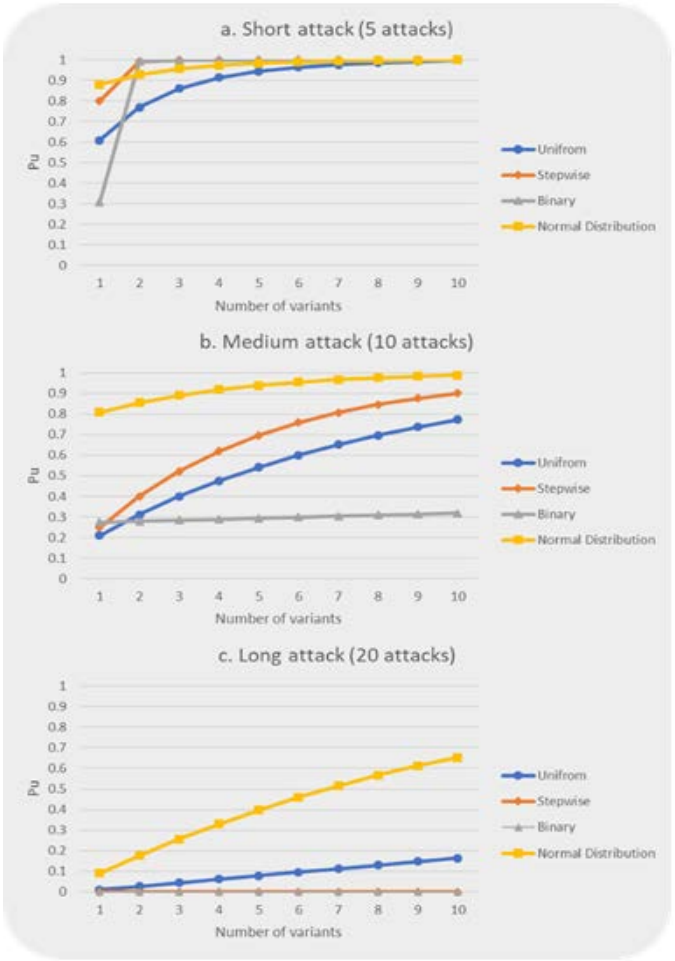

Figure 12: The effectiveness of inserting unreachable code for Program #1.

Figure 13 represents the effectiveness of inserting unreachable code into Program #2. Figure 13a shows that uniform and normal distribution attacks produce nearly identical results for a short attack. The small program size and the number of attacks (five) are two reasons for the identical results of both random attacks. Although the binary attack has a high probability of compromising a single variant system, increasing the number of variants improves Pu significantly even after the second variant. After the sixth variant, all attack types produce nearly identical results because the system has already achieved the maximum survivability for these attack types in the short attack. All attack types except the binary attack yield a similar average Pu (0.89) in five attacks when all variants are considered. From the standpoint of the attacker, the binary is the least effective of the four investigated attack types. In the binary attack, the average Pu for all variants is only 0.94. When the attack duration was doubled, as shown in Figure 13b, Pu fell in all attack types as expected. The binary attack has evolved into the most effective attack type. The stepwise attack, like the short attack, provides the lowest Pu in a single variant system. However, increasing the number of variants improves Pu significantly. Furthermore, the stepwise attack produces the lowest average Pu (0.76) of all variants. In contrast to the short attack, there is a discernible difference between uniform and normal distribution attacks. Because it targets a smaller address space, the uniform attack always has a lower Pu than the normal distribution attack. Nonetheless, when all K-variants are considered, the normal distribution achieves a lower average Pu than the stepwise attack. Figure 13c shows that Pu decreased dramatically for all attack types during the long attack. What stands out in this figure is the increase in the probability of a successful attack in deterministic attacks, which are stepwise and binary. Because deterministic attacks use a systematic approach to compromise K-variant systems, they become more efficient as the attack duration increases. Pu of binary and stepwise attacks is 0.2 and 0.25, respectively, in terms of average Pu for all K-variant systems. Pu in the uniform and normal distribution attacks, on the other hand, is only 0.27 and 0.35, respectively. Figure 13c) also clearly shows the disadvantage of the normal distribution attack. Because target addresses are chosen from a limited address space, the normal distribution attack has a lower Pu than other attack types. According to the results of the experiment study, Pu is expected to increase after ten variants for all attack types.

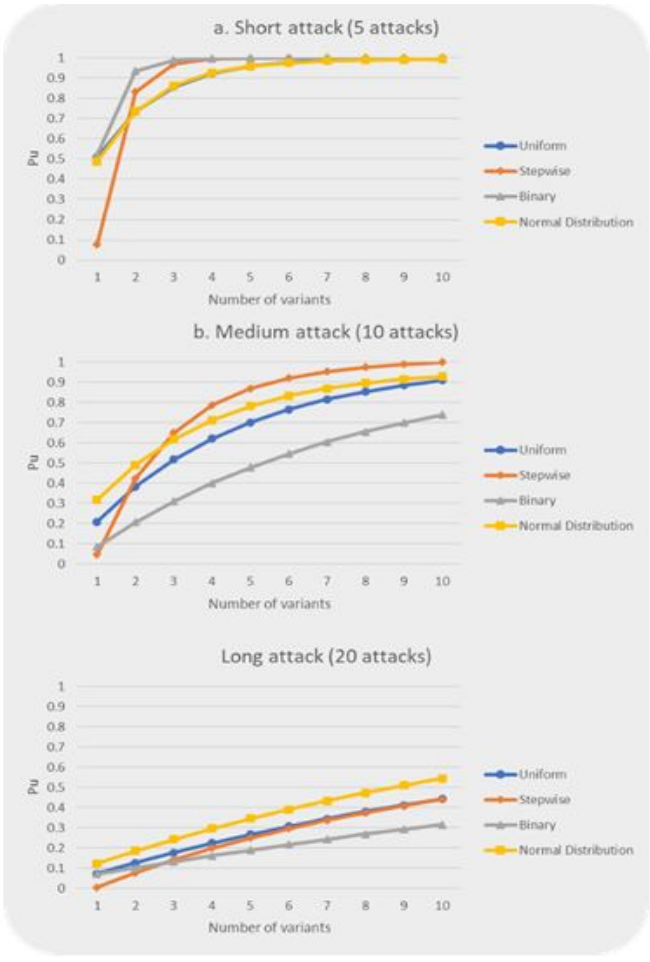

Figure 13: The effectiveness of inserting unreachable code for Program #2.

Figure 14 depicts the effectiveness of inserting unreachable code for Program #3. Only second variants, as shown in Figure 14a, provide a noticeable improvement in Pu for all attack types in short attacks. Adding more than two variants has no effect on Pu. The most intriguing aspect of the line chart in Figure 14a is the superiority of random attacks over deterministic attacks.

When the program is large, and the attack duration is short, the data in Figure 14a suggests that random attacks have a greater chance of compromising K-variant systems than deterministic attacks. Moreover, stepwise and binary attacks produce nearly identical Pu for all K-variant systems. Furthermore, from the attacker's perspective, the uniform attack is the most successful attack type in short attacks. Unlike the other programs in the experimental study, doubling the attack duration results in a minor decrease in Pu for all attack types in Figure 14b. The binary attack causes the most noticeable change in Pu. Also, the binary attack provides the lowest Pu (0.87) on average for all K-variant systems. Stepwise, on the other hand, has the highest Pu. Pu attains one following the introduction of the second variant. Although the normal distribution attack yields slightly lower Pu than the stepwise attack on average Pu for all K-variant systems, it is not as effective as the uniform attack in compromising the system. When the number of attack attempts is doubled, Pu decreases dramatically from single to four variant systems in the long attack, as shown in Figure 14c. The stepwise exploits the long attack's advantage by decreasing the delta-step value, which is the gap between attacked addresses. As a result, it becomes the most effective type of attack to compromise the system. After the third variant, the uniform and stepwise attacks produce the same Pu.

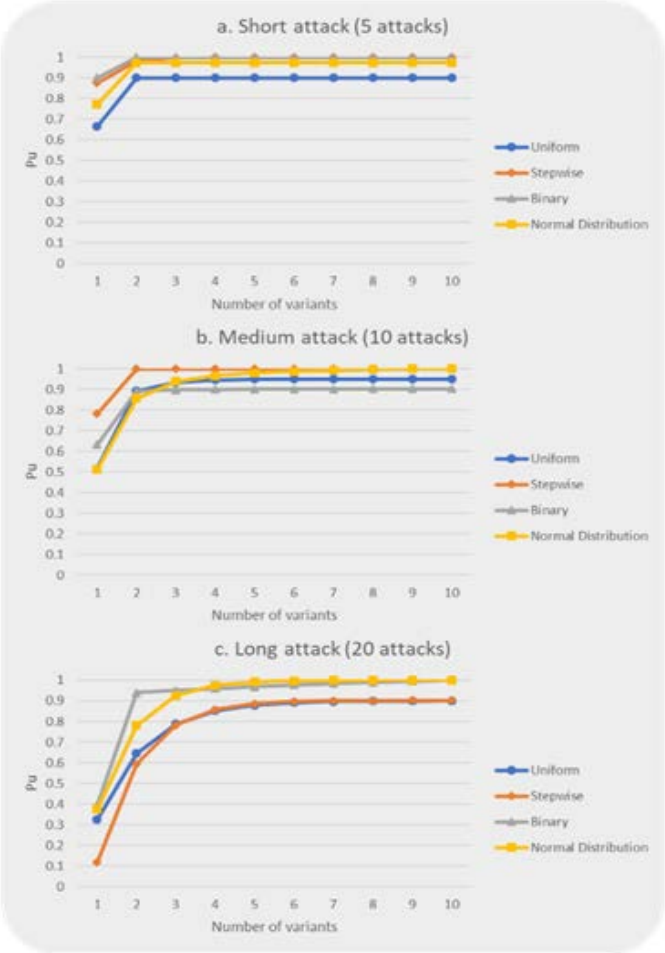

Figure 14: The effectiveness of inserting unreachable code for Program #3.

Comparisons based on the average Pu of all programs

The previous section examined the survivability of K-variant systems for three distinct programs. This section compares four different attack types using the average Pu of three different programs. The average Pu of three programs is shown in Figure 15 in relation to the increasing number of variants for four attack types over three attack durations. The comparisons in Figure 15a, Figure 15b, and Figure 15c are for short, medium, and long attacks, respectively.

Figure 15a shows that for five attack attempts, random attacks (uniform and normal distribution attacks) produce a smaller average Pu than deterministic attacks (stepwise and binary attacks). Among the four attack types, the uniform attack has the best chance of compromising K-variant systems. On the other hand, K-variant architecture significantly improves the average Pu for the binary attack in the short attack. After introducing the third variant in the binary attack, the average Pu reaches nearly one (0.996). Moreover, adding more than six variants provides only a slight improvement in average Pu for all attack types. Furthermore, no significant improvement in average Pu is observed for all attack types after the fourth variant.

When the number of attacks is increased from five to ten, the difference in the average Pu between attack types becomes obvious in Figure 15b. Due to deterministic attacks' systematic approach, the binary attack has emerged as the most effective attack type for compromising K-variant systems. However, the binary attack produces a greater average Pu than the uniform attack. Additionally, on average, the normal distribution attack has the lowest probability of compromising K-variant systems.

Figure 15c depicts the advantage of deterministic attacks for long attacks. In twenty attacks, the stepwise and binary attacks provide the lowest average Pu. Because Δ (delta step) becomes smaller in the long attack, it becomes significantly more efficient than the short and medium attack durations. Although the average Pu in the uniform and binary attacks is similar, the uniform attack yields a slightly higher average Pu than the binary attack. The long attack highlights the difference between the normal distribution attack and other attacks. As a result, the normal distribution attack is the least preferable of the four attack types in long attack durations for attackers.

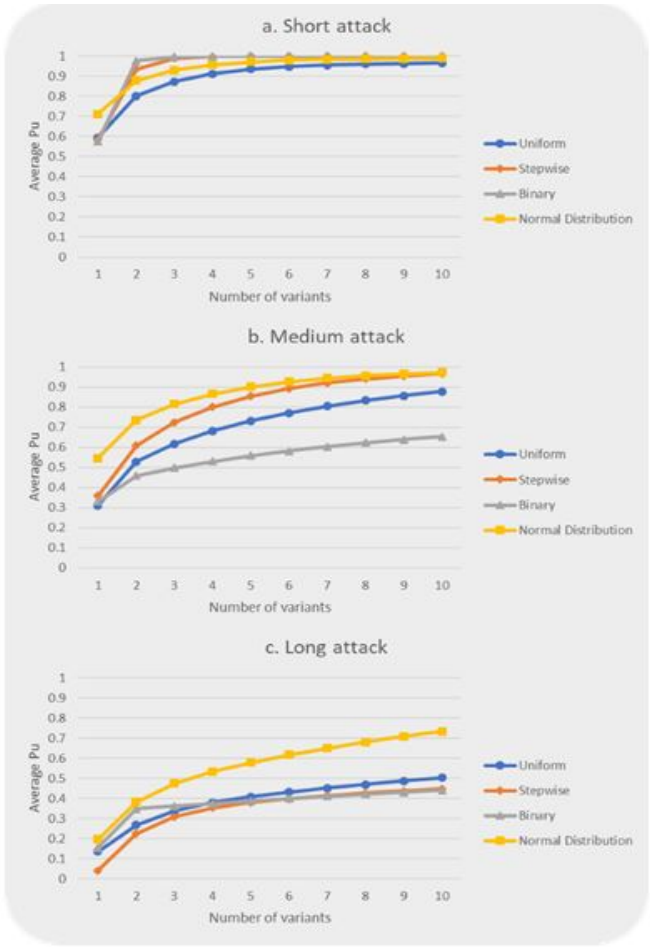

Figure 15: Comparison of attack types for tree attack durations based on the average Pu of three programs.

To summarize, random attacks may have a higher probability of compromising K-variant systems than deterministic attacks in short attacks. Deterministic attacks, on the other hand, become more efficient as the number of attack attempts increases. Furthermore, introducing the second variant always results in the most noticeable improvement in Pu for all attack types. The level of improvement in Pu is likely to diminish with each additional variant for all attack types.

Overhead of the program transformation: As is the case with all multi-variant/version execution architectures, K-variant incurs a cost associated with executing multiple variants. Each variant/version necessitates the utilization of additional CPU and memory resources. This is an overhead due to the multi-variant/version architecture's nature. This section, on the other hand, discusses the overhead associated with the program transformation caused by the addition of unreachable code.

Inserting unreachable code increases the program's size and memory usage. Unreachable statement chunks are enclosed by never-executed conditions. They do, however, consume memory. Additionally, inserted conditions remain executable and consume a negligible amount of CPU time. Five unreachable statement chunks are inserted into each variant during the experimental study. As a result, the source code contains five additional executable conditions. Due to the small number of executed instructions, CPU overhead is negligible. In the experimental study, the average memory overhead for Program #1, Program #2, and Program #3 is 2.6%, 6.4%, and 4.6%, respectively.

In an experimental study, the program transformation of inserting unreachable code significantly improves the survivability of systems against memory exploitation attacks while incurring minimal CPU and memory overhead. On the other hand, introducing a large number of unreachable statement chunks may result in a degradation of system performance. Additionally, increasing the number of statements in each unreachable statement chunk increases the variants' memory overhead.

Threats to validity of experiments: The following threats to experiment validity are identified and addressed in this experimental study.

- The number and size of programs: Three programs were used in the experimental study. Furthermore, program sizes range between 500, 1 K, and 5 K LOC. The small number of programs could be a threat to validity. These three programs may not be representative of the general behavior of program transformations and attack types. Furthermore, the three programs are relatively small in size. For large programs, different outcomes may be obtained.
- Insufficient statistical power: Because the experiment has a small number of iterations, the results may vary. Each experiment is iterated 1000 times in order to eliminate that threat. When the number of iterations is increased from 1000 to 10,000, there is no discernible difference.
- Implementation of the tool: The tool's bugs could lead to incorrect results. To counteract this threat, the tool includes multiple versions of program transformation and attack simulations. The outcomes of these variants are compared.

CONCLUSION

The K-variant is an alternative architecture to the N-variant that is intended to strengthen systems' defenses against memory exploitation attacks. In contrast to the N-version architecture, the K-variant architecture generates variants through program transformations. By implementing a safe and simple program transformation into the source code, functionally equivalent source codes are generated. Additionally, variants in the K-variant architecture can be generated *via* automated program transformation, significantly reducing the cost of variant generation.

Inserting unreachable code is a program transformation that can be used to generate variants in the K-variant architecture to improve system resistance to memory exploitation attacks. A random number of unreachable statement chunks are inserted into a random location in the source code during this program transformation. Unreachable statements are surrounded by conditions that are always false, preventing them from being executed. They do, however, consume memory addresses in memory. Inserting unreachable code is a secure program transformation. It means that the program transformation does not introduce new bugs. As a result, no extensive software

testing is required to validate variants. Furthermore, the program transformation can be easily automated, lowering the cost of generating variants significantly.

The effectiveness of the program transformation of inserting unreachable code is investigated experimentally on K-variant systems with four different attack types in this paper. The experimental study examines four attack types: uniform, normal distribution, binary, and stepwise. Three programs of varying sizes are examined less than three different attack durations. The experimental study's attack model, which allows attackers to write arbitrary memory addresses, is the worst-case scenario for defenders. Because the experimental study's assumptions may not be possible in actual attacks, the effectiveness of the program transformation is expected to be greater in the wild.

The experimental results indicate that generating variants by inserting unreachable code is a valuable strategy for improving the survivability of K-variant systems against memory exploitation attacks at a reasonable cost. Increasing the number of variants may improve survivability significantly, especially during long attacks. In the experimental study, the effectiveness of random attacks in short attack durations and the efficiency of deterministic attacks in long attack durations were observed.

We will investigate the effectiveness of program transformation in larger programs in future work (which are significantly larger than 5 KLOC). Additionally, the performance of large programs will be examined following the addition of unreachable code. Moreover, the effectiveness of the program transformation for inserting unreachable code will be compared to the effectiveness of other K-variant architecture program transformations.

ACKNOWLEDGMENTS

Not applicable.

AUTHORS' CONTRIBUTIONS

Both authors formulated the idea of the study and wrote the paper; both authors reviewed the results and improved the final manuscript. All authors examined and approved the final manuscript.

FUNDING

Not applicable.

AVAILABILITY OF DATA AND MATERIALS

Not applicable.

DECLARATIONS

Competing interests

The authors declare that they have no competing interests.

REFERENCES

1. Avizienis A. The N-version approach to fault-tolerant software. IEEE Trans Softw Eng. 1985;11:1491–1501.
2. Bekiroglu B, Korel B. Survivability analysis of K-variant architecture for different memory attacks and defense strategies. IEEE Trans Dependable Secure Comput. 2021;18:1868–1881.
3. Bekiroglu B, Korel B. Source code transformations for improving security of time-bounded k-variant systems. Inf Softw Technol 2021;137:106601.
4. Bhatkar S. Defeating memory error exploits using automated software diversity. Phd, State University of New York at Stony Brook. 2007.
5. Bhatkar S, Sekar R, Du Varney DC (2005) Efficient techniques for comprehensive protection from memory error exploits. In: Proceedings of the 14th conference on USENIX Security Symposiu. USENIX Association, Jul 31-Aug 5, Baltimore, MD,USA,14-17.
6. Bletsch T (2011) Code-reuse attacks: new frontiers and defenses. Raleigh, North Carolina State University, United States. 95.
7. Burow N, Carr SA, Nash J, Larsen P, Franz M, Brunthaler S, et al. Control-Flow Integrity: Precision, Security, and Performance. ACM Comput Surv. 2017;50:1–33.
8. Carlini N, Barresi A, Payer M, Wagner D, Gross TR. Control-Flow Bending: On the Effectiveness of Control-Flow Integrity. 2015;161–176.
9. Chen Y, Zhang D, Wang R, Qiao R, Azab AM, Lu L, et al. NORAX: Enabling Execute-Only Memory for COTS Binaries on AArch 64. IEEE Secur Priv. 2017;304–319.
10. Coker Z, Hafiz M (2013) Program transformations to fix C integers. In: Proceedings of the 2013 International Conference on Software Engineering. IEEE. May 18-26, San Francisco, CA, USA,792–801.
11. Coker ZF (2012) Security-oriented program transformations to cure integer overflow vulnerabilities. In: Proceedings of the 3rd annual conference on systems, programming, and applications: software for humanity. Oct, Arizona, USA,103–104.
12. Cowan C, Pu C, Maier D, Hintony H, Walpole J, Bakke P, et al (1998) StackGuard: automatic adaptive detection and prevention of buffer-overflow attacks. In: Proceedings of the 7th conference on USENIX Security Symposium. Jan 26-29, San Antonio, Texas,7-5.
13. Crane S, Liebchen C, Homescu A, Davi L, Larsen P, Sadeghi AR, et al. Readactor: practical code randomization resilient to memory disclosure. IEEE Secur Priv. 2015;763–780.
14. Deswarte Y, Kanoun K, Laprie JC (1998) Diversity against accidental and deliberate faults. In: Proceedings Computer Security, Dependability, and Assurance: From Needs to Solutions. July 7-09, York, UK & Williamsburg, VA, USA,171–181.
15. Devriese D, Piessens F. Noninterference through Secure Multi-execution. IEEE Secur Priv. 2010;109–124.
16. Farvardin N, Modestino J. On overflow and underflow problems in buffer-instrumented variable-length coding of fixed- rate memoryless sources (Corresp.). IEEE Trans Inf Theory 1986;32:839–845.
17. Forrest S, Somayaji A, Ackley DH (1997) Building diverse computer systems. In: Proceedings. The Sixth Workshop on Hot Topics in Operating Systems (Cat. No.97TB 100133). Los Alamitos, CA,67–72.
18. Gionta J, Enck W, Ning P (2015) HideM: Protecting the Contents of Userspace Memory in the Face of Disclosure Vulnerabilities. In: Proceedings of the 5th ACM Conference on Data and Application Security and Privacy. March 2-4, San Antonio, Texas, USA,325–336.
19. Gisbert HM, Ripoll I. On the Effectiveness of NX, SSP, RenewSSP, and ASLR against Stack Buffer Overflows. In: 2014 IEEE 13th

International Symposium on Network Computing and Applications. 2014;145–152.

20. Hafiz M. Security oriented program transformations (or how to add security on demand). In: Companion to the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications. Association for Computing Machinery, Nashville, TN, USA. 2008;927–928.
21. Hafiz M, Johnson R. A security oriented program transformation to "add on" policies to prevent injection attacks. In: Proceedings of the 2nd Workshop on Refactoring Tools. Association for Computing Machinery, Nashville, Tennessee.2008;1–4.
22. Hafiz M, Johnson RE. Security-oriented program transformations. In: Proceedings of the 5th Annual Workshop on Cyber Security and Information Intelligence Research: Cyber Security and Information Intelligence Challenges and Strategies. Association for Computing Machinery, Oak Ridge, Tennessee, USA. 2009;1–4.
23. Hafiz M, Johnson RE. Improving perimeter security with security-oriented program transformations. In: 2009 ICSE Workshop on Software Engineering for Secure Systems. 2009;61–67.
24. Kc GS, Keromytis AD, Prevelakis V. Countering code-injection attacks with instruction-set randomization. In: Proceedings of the 10th ACM conference on Computer and communications security. Association for Computing Machinery, Washington D.C., USA. 2003;272–280.
25. Korel B, Ren S, Kwiat K, Auguste A, Vignaux A. Improving operation time bounded mission critical systems' attack-survivability through controlled source-code transformation. In: Proceedings of the 4th international conference on security of information and networks. Association for Computing Machinery, Sydney, Australia. 2011;183–190.
26. Larsen P, Homescu A, Brunthaler S, Franz M. SoK: Automated Software Diversity. In: 2014 IEEE Symposium on Security and Privacy 2014;276–291.
27. Larsen P, Sadeghi AR. The Continuing Arms Race: Code-Reuse Attacks and Defenses. Association for Computing Machinery and Morgan & Claypool. 2018.
28. Marco-Gisbert H, Ripoll I. Preventing Brute Force Attacks Against Stack Canary Protection on Networking Servers. In: 2013 IEEE 12th International Symposium on Network Computing and Applications. 2013;243–250.
29. Mashtizadeh AJ, Bittau A, Boneh D, Mazières D. CCFI: Cryptographically Enforced Control Flow Integrity. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. Association for Computing Machinery, Denver, Colorado, USA. 2015;941–951.
30. Molnar I. "Exec Shield", new Linux security feature. In: "Exec Shield", new Linux security feature. 2003.
31. Newsham T. (2000) Format String Attacks. Guardent, 1-8.
32. Paulson LD. New chips stop buffer overflow attacks. Computer 2004;37:28.
33. Roemer R, Buchanan E, Shacham H, Savage S. Return-Oriented Programming: Systems, Languages, and Applications. ACM Trans Inf Syst Secur. 2012;15:2:1–2:34 .
34. Russinovich ME, Solomon DA, Ionescu A. Windows Internals, Part 2, 6 edition. Microsoft Press. 2012.
35. Schmitz T, Algehed M, Flanagan C, Russo A. Faceted Secure Multi Execution. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. Association for Computing Machinery, New York, NY, USA. 2018;1617–1634.
36. Shaw A, Doggett D, Hafiz M. Automatically Fixing C Buffer Overflows Using Program Transformations. In: 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks. 2014;124–135.
37. Tech GB. Mission-critical systems, and why you need them managed. In: GB Tech. 2020.
38. Zhiyuan A, Haiyan L. Realization of Buffer Overflow. In: 2010 International Forum on Information Technology and Applications. 2010;347–349.
39. Xu J, Kalbarczyk Z, Iyer R. Transparent runtime randomization for security. 260–269.
40. Abatchy's blog Exploit Dev 101: Bypassing ASLR on Windows. In: Exploit Dev. 101: Bypassing ASLR on Windows. 2017.