Multi-Variant Execution Research of Software Diversity

Zhenwu Liu, Zheng Zhang*, Jiexin Zhang, Hao Liu
State key Laboratory of Mathematical Engineering and Advanced Computing, PLA Strategic Support Force Information Engineering University, ZhengZhou, HeNan, 450001, China
*e-mail: ponyzhang@126.com

Abstract. As more and more software products are threatened by malicious reverse analysis, along with software products are pirated, tampered with and so on, it is of great significance to study software security protection technology in depth. As a software security protection technology, software diversification introduces uncertainty into the target program and provides probabilistic protection for the target program. Multi-Variant Execution (MVE) are fine-grained implementation of software diversification that produces functionally identical variants at the system call level. This article first introduces the related concepts of multi-variant execution. Secondly, it expounds the key technologies of multi-variant execution implementation-variant generation, variant monitor, input/output and synchronization, monitor-variant communication. Security of monitors and different variant communication technologies are analyzed and compared, their advantages and defects are pointed out respectively. Several implementation methods of multi-variant execution design are summarized. Finally, the reasons for the false negatives and false positives of attack events are analyzed, and the prospects and challenges of using multi-variant execution techniques to implement security systems are summarized.

1. Introduction

The development of software analysis tools helps software developers analyze and improve software programs. At the same time, software analysis technology can also be used for reverse analysis of software products, such as stealing the intellectual property of a program, identifying and exploiting vulnerabilities in the program. Attackers can easily get what they need from many blog sites that share software reverse engineering tools and documentation. Software programs’ reverse analysis and abstraction into higher-level representations makes it easier for attackers to identify vulnerabilities in software programs and exploit them. The attackers insert trojans, viruses, and worms by exploiting vulnerabilities discovered during the reverse analysis process, or initiate a denial of service attack, causing the software service to crash, making other legitimate users unable to request the service normally. Therefore, reverse analysis already poses a significant threat to software users and network security.

Although the copyright law covers most of the intellectual property contained in the software, these laws are difficult to be effectively implemented due to issues such as law enforcement costs in reality. In order to prevent the software reverse analysis from succeeding, encryption technology is used in security protection. The program is encrypted at compile time and decrypted at execution time as needed. Since the program needs to be decrypted after encryption, this brings about a certain performance loss and time and space overhead.
Another economical way to protect software is to use software diversification. In this way, software vendors will take the technical means rather than legal means to protect intellectual property. Programs that are distributed on a large scale are also vulnerable to attacks of the same size. An attacker only needs one exploit to attack the same replica using the same attack method or even code. The way nature solves this problem is biodiversity. A single plant may die from a pathogen, but due to genetic variation, the entire species can survive.

Software developers often use automated software diversification techniques to confuse code to a certain degree, making it difficult for attackers to reverse-analyze the confusing program, avoiding the discovery of software vulnerabilities and the leakage of program source code, ensuring software service security and intellectual property rights. Modern operating systems basically support Address Space Layout Randomization (ASLR)[1]. ASLR randomly changes the code's address in memory, thereby reducing the execution success rate of the attack code.

Software diversity is a probabilistic defensive approach. The core idea of active defense technology such as Moving Target Defense[2], Mimic Defense[3], is to apply software and hardware diversification techniques to dynamically change the attack surface and increase the workload of the attacker. These active defense technologies hope to change the "rules of the game", using uncertainty to make the offensive and defensive balances tend to defenders, fundamentally changing the asymmetry of attackers and defenders.

Software diversification technology is a powerful support for these active defense technologies, which has great research significance. After using the software diversification technology on the software program, the uncertainty of the attack is increased, and the unitary and homogeneity problems brought by the software standardization are avoided. Thereby avoiding the large-scale data leakage and service paralysis caused by the above problems once the attack is successful. At the same time, a mechanism that votes on the output can be a probabilistic defense against attacks such as memory attacks. With the rapid development of cloud computing services and the popularity of multi-core processors, the fine-grained domain of software diversification -- (Multi-Variant Execution, MVE) technology has also been rapidly developed.

This paper will comprehensively summarize the research on the multi-variant execution in recent years. The key technologies of monitor, input/output, synchronization and monitor-variant communication are introduced. The advantages and disadvantages of monitor security and different communication technologies are compared and analyzed. Then it analyzes the causes of false positives and false negatives. The tradeoffs between performance and security are also analyzed. Finally, the work of the full text is summarized.

2. Background

2.1. N-Version Programming
N-Version Programming (NVP) was introduced in the 1970s by Chen and Avicienis[4]. The core idea is to have multiple programmer teams develop the same software independently and then run it in parallel to improve the fault tolerance of the entire system. Both the version generation and synchronization mechanisms require manual operation. NVP requires multiple independent software development teams and may have separate toolchain sets. Even if all teams are running in parallel, the total development time is determined by the slowest team development time. In n-version programming, multiple independent solutions of the algorithm are implemented separately from the same specification, and the implementation and maintenance costs are large. Therefore, n-version programming is only used for the most critical applications regardless of cost, such as space flight software.

2.2. Multi-Variant Execution
Compared to the term "version" used in the initial NVP to describe manual and stand-alone development program implementations. The use of the term "variant" by Cox et al. to describe a copy
of a program for obtaining automatically generated security has been widely adopted[5]. Variants of MVE are automatically generated from the same source code, without the need to manually rewrite variants. It significantly reduces the development and maintenance costs of variants, usually does not have a significant impact on performance. It is suitable for users with higher security requirements, but security assurance is non-deterministic.

Multi-variant execution[5-7] is a runtime monitoring technique that distributes the same input to multiple variants, typically performing two or more diverse programs (variants) in locksteps while monitoring their behavior at the system call level. Each diversified variant receives the same program input, but responds differently to malicious input of certain attacks, detecting exploits before an attacker has a chance to compromise the system. If the monitoring agent detects a difference in behavior of variants at some synchronization point, the execution of the program will be terminated. Therefore, an attacker must reliably exploit all online program variants at the same time, send each malicious input to different variants and destroy them at a time to ensure that they behave consistently. By performing this active defense through multi-variant execution, it is more difficult for an attacker to attack successfully. Multi-variant execution can interrupt the execution and alert when the monitoring agent detects an attack, effectively mitigating the impact of the attack. Studies[8] have shown that multi-variant execution can protect against multiple type of attacks, such as code reuse, information leaks, stack buffer overflows, code injection, etc.

![Figure 1. Overview of all MVE design components.](image)

3. Key technologies for implementation

3.1. Variant Generation

The variant generator is implemented based on a compiler transformation, generating variants of code diversification; M. Chew and D. Song,etc[9] is directed to the system call interface between the process and the operating system kernel. By changing the number of the system call, the injected code executes random system calls, resulting in different behaviors and even errors. The application needs to be recompiled to use the correct system call mapping. Program variants should have the same response for the same system call. DCL[10] improves an early technique for defeating memory exploits -- Address Space Partitioning (ASP), allows protected programs to use the full virtual address space. DCL relies on executing and copying multiple run-time variants of the same application under the control of monitor and ensuring that there are no code segment overlaps in the address space of the variant due to the lack of overlapping code segments. Thus, no Return Oriented Programming (ROP) [11] attack can change the behavior of all variants in the same way. By monitoring the variant's I/O operations and stopping execution when requesting any different I/O operations, the monitor can
effectively block any ROP attacks. DCL implements full immunity against ROP attacks in user space without modifying the underlying Linux operating system. kMVX[12] using FT-Linux[13] to implement kernel-level multi-variant execution. By making specific changes to the memory layout of each variant and multiple diverse kernel variants are running simultaneously on the same machine, kMVX can detect the different behaviors when an attacker attempts to exploit during a system attack, and fully defend against information disclosure vulnerabilities in the kernel. It also retains the Linux User Space Application Binary Interface (ABI): the application binary interface, allowing existing applications to run unmodified on the system. VARAN[18] implements its own C library functions based on the Bionic C library. To support the use of Linux system calls, VARAN uses a modified version of the linux_syscall_support.h header file.

3.2. The Multi-variant Monitor

3.2.1. Monitor implementation. Most monitors are implemented based on the ptrace mechanism provided by the UNIX operating system[10][14-16]. Although easy to use and no kernel modification is required, existing system call monitors based on the ptrace interface perform poorly. Since ptrace can only access four bytes at a time, it must be called multiple times to access large blocks of memory. Each call requires a context switch from the monitor to the OS kernel, which is very inefficient when a large number of read buffers are needed. The monitor process must execute several additional system calls to copy the buffer to the monitored variant. For applications with tightly I/O bindings, system performance can be reduced by up to two orders of magnitude.

3.2.2. Monitor security. Different multi-variant execution designs make different trade-offs. Choosing to run the monitor in kernel space[12][14] brings high performance but requires the addition of kernel patches or new kernel modules, and the monitor must be run in privileged mode with a large attack surface that can easily compromise the entire system once the monitor is successfully attacked[14]. Running monitors in user space[10][15-16] is divided into in-process monitoring and cross-process monitoring[10][20]. In-process monitoring places the monitor in the process of the variant. Most multi-variant execution designs use cross-process monitoring, which monitors the process independently and uses a single centralized monitor component that is outside the address space of the variant. Replicas can be isolated from the monitor with hardware forced protection boundaries.

Running the monitor in user space isolates the variants from the operating system kernel and monitors all communication between them and the kernel. The monitor is implemented as a non-privileged process, using process debugging tools to intercept system calls. The operating system kernel patch does not need to be reapplied to the updated version of the kernel. If the monitor is compromised, the attacker will be subject to user-level privilege and require a privilege escalation to gain system-level access. At the same time, the monitor is an independent process with its own address space. No other process in the system, including a variant process, can manipulate its memory space directly. So it is difficult to compromise the monitor by controlling program variants. Since the monitor does not process user input and acts only as a proxy for dispatching user input to the variant, it will be difficult to destroy it by sending malicious input to the monitor.

3.3. Input/output and synchronization

Multi-variant execution is a monitoring mechanism that controls the state of the variant being executed and verifies that the variant meets the predefined rules. The monitor is responsible for checking and ensuring that the program variants are functioning properly. Depending on the security and feasibility, the monitor will implement different granularity detection ranges, from checking only the final output of each variant to comparing the system calls for each execution. The scope of the monitored granularity detection determines how long it takes to capture the attack behavior. Coarse-grained monitoring has lower overhead than fine-grained monitoring because it reduces the number of comparison and synchronization points.
Most multi-variant execution implementations choose system calls as synchronization points and compare their behavior at the synchronization points. All variants execute system calls that have the same function and are equivalent in parameters, and monitor the behavior of each variant to prevent variants from accessing data or resources outside of their process space. In the multi-variant execution technique, since multiple variants must be executed at the same time, and the running result of each variant is monitored in real-time, the running overhead is large.

The monitor creates a child process for each variant. When the variant needs data or resources outside of its process space, it interrupts execution. Whenever a variant makes a system call, the monitor intercepts the request and suspends the variant. The monitor then attempts to synchronize the system call with other variants. All variants need to use equivalent parameters and make the exact same system call in the same time period.

Parameter equivalence does not necessarily mean that the parameter values are identical. When the argument is a pointer to a buffer, the contents of the buffer are compared. The pointer itself can be different, but the monitor expects the buffer contents to be the same. Non-pointer arguments are considered equivalent only if they are the same. The monitor determines if the variant is in the correct state according to the following rules. Assuming that p1 to pn are variants of the same program p, they are considered to be in a consistent state if the following conditions are met at each synchronization point:

1. All system calls in the system call set at the synchronization point are consistent;
2. The elements in the parameter set of all system calls at the synchronization point are consistent;

All variants execute in parallel and should arrive at the sync point almost simultaneously, up to the maximum wait time for the monitor to wait for the variant. Multi-variant execution must synchronize variant execution and remain transparent to the end user. The user observes the I/O operation only once. Therefore, while multiple variants are running in parallel, the monitor copies and distributes program input once for each variant, and compares the variant output so that the user obtains only one program output, ensuring the system's safety with consistent output results. It is almost impossible for an attacker to use the same malicious input to make certain exploits for all variants at the same time. So it is highly probable that a condition that does not satisfy one of the above conditions will occur. Then the monitor checks the consistency of the variant behavior to determine whether it exists attack. Once an inconsistent behavior causes an alarm to be triggered, the monitor will take appropriate real-time actions according to the configured scheduling policy to ensure that the service is normal, such as terminating and restarting all variants or voting between variants, then terminating and restarting the exception variant.

3.4. MONITOR-VARIANT COMMUNICATION

In order to improve performance, Salamat B, Jackson T, Wagner G, et al[17] create a memory block for each variant that is shared by the monitor and each variant. When the monitor needs to write to the memory space of the variant, the monitor writes the data to shared memory and then forces the variant to read from the shared memory and write the data to its address space. At the same time, the monitor forces the variant to read the required memory block from its address space and write it to the shared memory, and then the monitor reads the block from shared memory. VARAN[18] performs user space monitoring through selective binary rewriting and fast shared memory ring buffers to avoid the overhead of using ptrace. MvArmor[16] utilizes Dune's[19] hardware-assisted monitoring capabilities to provide secure process monitoring. MvArmor's performance are comparable to ReMon, but due to Dune's limitations, it currently does not support multi-threaded applications. Varan's event flow design and MvArmor's variant synchronization strategy are based on a ring buffer design that communicates through the shared memory area set up at the start of the monitor. This shared memory area contains a ring buffer that is optimized to hold system call parameters and return values and can be used to compare function call parameters between variants. The key difference is that Varan's event flow design is completely asynchronous and does not support the synchronous detection strategy used by MvArmor's security design.
We have summarized several multi-variant execution technologies. The key factors that distinguish them are that the monitor runs in kernel space or user space, and running in a variant process context. GHUMVEE (2013) developed DCL (2016) and Remon (2016). MvArmour (2016) developed kMVX (2019).

Table I summarizes the typical multi-variant execution designs mentioned in the above documents. The second column represents the synchronization point of the monitor to the variant output. The third column shows the types of attacks that can be defended in the literature. The fourth column is the monitor implementation. The fifth column shows the monitor’s run permissions, and the last column shows security benchmark.

Table 1. Overview of typical MVE systems.

| Technique  | Synchronization | Defense                  | Implementation | Privilege   | Security Benchmark |
|------------|-----------------|--------------------------|----------------|-------------|-------------------|
| GHUMVEE    | program points  | control flow            | ptrace         | user space  | CVE-2013-2028     |
| DCL        | program points  | hijacking                | ptrace         | user space  | CVE-2010-4221     |
|            |                  | exploits                 |                |             | CVE-2012-4409     |
| ReMon      | syscall          | unknown attacks and low-level memory errors | ptrace | kernel space | logical argument |
| MvArmor    | syscall          | memory error             | ptrace         | user space  | CVE-2004-0488     |
|            |                  | exploits                 |                |             | CVE-2014-0160     |
| kMVX       | I/O and syscall  | kernel information leaks |               | kernel space | CVE-2016-4569     |
|            |                  |                          |                |             | CVE-2013-2237     |
| Varan      | syscall          | binary rewriting         |               | context     | CVE-2016-0728     |

4. False positive
There are a variety of behavioral inconsistencies between variants. The internal conditions and behaviors of the system and system events may lead to differences in variant behavior, leading to false positives in MVE. The monitor sets the maximum time to wait for a synchronous system call. Since there are some differences in the execution efficiency of each variant, the execution time is not the same. Some variants due to their own differences cause asynchronism of time, to some extent, will lead to system calls outside the monitor set time, resulting in false positives. Multi-variant execution must correctly handle the scheduling of child processes and threads, asynchronous signals, file descriptors, process IDs, time, and random numbers to prevent these differences from causing the monitor to raise false alarms and interrupt the execution of variants.

The multi-variant execution technology basically uses the system call as a comparison point. In order to reduce the false positive rate, the system calls are classified and different monitoring strategies are adopted. MVE specifies which system calls can be executed by the variant and which must be run by the monitor. The decision is based on the following parameters:
(1) System calls that change the state of the system are executed by the monitor and the results are copied into the variant. For example, a system call that creates a file on the system must be executed once by the monitor and no variants are allowed to run.

(2) A non-status change system call that returns a variable result must also be executed by the monitor, and the variant must receive the same system call result. For example, when a variant calls a system call that reads system time, the monitor only calls the same system call once and sends the result to all variants. This is necessary to keep the variants in compliance and prevent false positives during execution.

(3) The variant can execute a non-state change system call that produces an immutable result.

In GHUMVEE[15], system calls are divided into I/O related system calls, memory management system calls, system calls that return variable results, and process self-aware calls. I/O related system calls are only allowed to be executed once, such as read and write operations. Memory management system calls create, modify, or delete the kernel structure of a bonded process. Most system calls that return variable results are time-related system calls. Process self-aware calls include sys_getpid and sys_open(/proc/self/...) etc.

Remon[13] subdivides the system calls of read and write operations. By selecting the level of system calls for read and write operations, you can maintain high security, reduce runtime overhead, and improve system performance. Making the appropriate trade-offs between performance and security.

VARAN[18] does not distinguish between sensitive and insensitive system calls. It is in the same address space as the application. Return-oriented programming (ROP) attacks can bypass the VARAN tracking mechanism and evade detection.

5. False negatives

When using a majority voting strategy, it should be noted that the behavior of most variants does not necessarily indicate that the behavior is correct. If most variants are vulnerable to a particular type of attack, the system may incorrectly terminate a legitimate minority variant and continue to use the corrupted variant. Therefore, when a majority vote is used to judge an attack, diverse mechanisms and the number of variants play a crucial role in the judgment of the correctness of the system behavior.

MVE assumes that program variants have the same behavior under normal execution conditions and behave differently under abnormal conditions. So choosing contents of diversification will choose which categories of attacks can be stopped and which vulnerabilities can still be exploited. When all variants are not protected against a specific type of vulnerability, it is possible that the monitor is unable to determine the attack. For example, some attacks may damage the system without arousing system calls, in which case MVE can't defend against such attacks. Because MVE is not designed to prevent certain vulnerabilities, there is no inherent protection against these vulnerabilities. Such specific attacks initiated against a MVE environment will successfully destroy all variants, and any system calls in the injected code will be successfully executed by the variant.

However, it is more difficult for an attacker to build and launch a specific attack against MVE environment and variants. This requires knowing a lot about the defense mechanisms, such as how many variants are being used, how to create variants, and all the parameters used. The attacker must manually exploit each variant. During the attack, each successful variant must maintain the same behavior as the other normal variants. The detection is evaded by causing the variant to perform an equivalent system call when the attacker exploits. For each variant, the exploit process must be repeated separately in order for the attacker to successfully exploit all variants, instructing the variant to execute the system call and the injected code. In order to reduce the false negative rate, a variety of diverse techniques can be used to provide the attacker with the smallest attack surface. As the number of diverse technologies and the number of variants used in a multi-variant execution environment increase, the difficulty level of successful attack also increases significantly.
6. Summary
Multi-variant execution is performed in parallel through different variants, making it difficult for an attacker to simultaneously perform attacks on multiple variants, thereby triggering threat alerts and achieving a certain defense effect. Although multi-variant execution greatly improves the security of the software, it relies on the implementation of multi-variant execution environment. The scope of application is relatively insufficient and it does not have universality. In the actual process, due to the limitation of the defense mechanism itself, under certain circumstances, there will be cases of false positives or false negatives. In the implementation process, performance and security trade-offs are also needed to ensure the applicability of multi-variant execution.

Acknowledgments
The authors thank Jiexin Zhang for his assistance and encouragement in this research. And this research is supported by National Key Research and Development Program of China (2018YFB0804003).

References
[1] Shacham, H., Page, M., Pfaff, B., Goh, E. J., Modadugu, N., & Boneh, D. (2004). On the effectiveness of address-space randomization. In: Proceedings of the 11th ACM conference on Computer and communications security. pp. 298-307.
[2] Jajodia, S., Ghosh, A. K., Swarup, V., Wang, C., & Wang, X. S. (Eds.). (2011). Moving target defense: creating asymmetric uncertainty for cyber threats. Springer Science & Business Media. Vol. 54.
[3] Wu, J. (2017). Introduction to Cyberspace Mimic Defense. Telecommunication Science, 12.
[4] Chen, L., & Avizienis, A. (1995). N-version programming: A fault-tolerance approach to reliability of software operation. In: Twenty-Fifth International Symposium on Fault-Tolerant Computing. p. 113.
[5] Cox, B., Evans, D., Filipi, A., Rowanhill, J., Hu, W., Davidson, J., ... & Hiser, J. (2006). N-Variant Systems: A Secretless Framework for Security through Diversity. In: USENIX Security Symposium. pp. 105-120.
[6] Berger, E. D., & Zorn, B. G. (2006). Probabilistic memory safety for unsafe languages. In: Acm sigplan 2006 conference on programming language design and implementation (pldi 2006). Ottawa, Canada.
[7] Salamat, B., Gal, A., Jackson, T., Manivannan, K., Wagner, G., & Franz, M. (2008). Multi-variant program execution: Using multi-core systems to defuse buffer-overflow vulnerabilities. In: 2008 International Conference on Complex, Intelligent and Software Intensive Systems pp. 843-848.
[8] Coppens, B., De Sutter, B., & Volckaert, S. (2018). Multi-variant execution environments. In The Continuing Arms Race. Association for Computing Machinery and Morgan & Claypool. pp. 211-258.
[9] Chew, M., & Song, D. (2002). Mitigating buffer overflows by operating system randomization.
[10] Volckaert, S., Coppens, B., & De Sutter, B. (2015). Cloning your gadgets: Complete ROP attack immunity with multi-variant execution. IEEE Transactions on Dependable and Secure Computing, 13(4): 437-450.
[11] Roemer, R., Buchanan, E., Shacham, H., & Savage, S. (2012). Return-oriented programming: Systems, languages, and applications. ACM Transactions on Information and System Security (TISSEC), 15(1): 2.
[12] Österlund, S., Koning, K., Olivier, P., Barbalace, A., Bos, H., & Giuffrida, C. (2019). KMVX: Detecting Kernel Information Leaks with Multi-variant Execution. In: Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 559-572.
[13] Losa, G., Barbalace, A., Wen, Y., Chuang, H. R., & Ravindran, B. (2017). Transparent Fault-Tolerance using Intra-Machine Full-Software-Stack Replication on Commodity Multicore Hardware. In: 2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS). pp. 1521-1531.

[14] Volckaert, S., Coppens, B., Voulimeanas, A., Homescu, A., Larsen, P., De Sutter, B., & Franz, M. (2016). Secure and efficient application monitoring and replication. In: 2016 {USENIX} Annual Technical Conference ({USENIX} {ATC} 16). pp. 167-179.

[15] Volckaert, S., De Sutter, B., De Baets, T., & De Bosschere, K. (2012). GHUMVEE: efficient, effective, and flexible replication. In: International Symposium on Foundations and Practice of Security. Springer, Berlin, Heidelberg. pp. 261-277.

[16] Koning, K., Bos, H., & Giuffrida, C. (2016). Secure and efficient multi-variant execution using hardware-assisted process virtualization. In: 2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). pp. 431-442.

[17] Salamat, B., Jackson, T., Wagner, G., Wimmer, C., & Franz, M. (2011). Runtime defense against code injection attacks using replicated execution. IEEE Transactions on Dependable and Secure Computing, 8(4): 588-601.

[18] Hosek, P., & Cadar, C. (2015, March). Varan the unbelievable: An efficient n-version execution framework. In ACM SIGPLAN Notices. Vol. 50, No. 4, pp. 339-353.

[19] Belay, A., Bittau, A., Mashitizadeh, A., Terei, D., Mazieres, D., & Kozyrakis, C. (2012). Dune: Safe User-level Access to Privileged {CPU} Features. In: Presented as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ( {OSDI} 12). pp. 335-348.

[20] Salamat, B., Jackson, T., Gal, A., & Franz, M. (2009). Orchestra: intrusion detection using parallel execution and monitoring of program variants in user-space. In: Proceedings of the 4th ACM European conference on Computer systems. pp. 33-46.