Postprint: A Survey on False Positive Analysis and Solutions in Multi-Variant Execution

Authors: Xi Ruicheng, Zhang Zheng, Zhu Pengzhe, Liu Zijing

Date: 2022-05-18T00:00:00+00:00

Abstract

From a security perspective, Multi-variant Execution (MVX) is widely applied in cybersecurity defense; however, MVX exhibits a common problem: the false positives generated during convergence when various execution instances return content to the arbiter are difficult to resolve. Excluding objective factors such as machine environments, false positives occur because the voter initiates security judgments on inconsistent variables upon receiving converged information. In addition to inconsistent variables caused by genuine attacks, these are intermingled with inconsistent variables produced by normal system operation (e.g., memory descriptors, port numbers, random numbers, code, and thread invocation order within processes), thereby causing voter misjudgment and impacting the normal operation of multi-variant systems. Reducing the false positive rate of multi-variant execution can effectively enhance system efficiency and defense capability. This study categorizes recent types of multi-variant execution, summarizes the false positive problems and solution strategies in multi-variant execution, and analyzes the causes of arbitration false positives in multi-variant execution. Three strategies are selected for experimental analysis in specific application scenarios: the Pina algorithm for synchronization, compiler module instrumentation, and narrowing the voting boundary. The functionality and performance of each method are analyzed, and the advantages and disadvantages of each strategy are identified. Finally, unresolved challenges in existing multi-variant execution technologies and future research directions are discussed.

Full Text

Preamble

Vol. 39 No. 10
Application Research of Computers (ChinaXiv Cooperative Journal)
Accepted Paper

A Survey of Analysis and Solutions for Multi-Variant Execution False Alarm Problems

Xi Ruicheng[1],[2], Zhang Zheng[1],[2]†, Zhu Pengzhe[1],[2], Liu Zijing[1],[2]
(1. School of Cyber & Space Security, Strategic Support Force Information Engineering University, Zhengzhou 450001, China;
2. Purple Mountain Laboratories, Nanjing 211100, China)

Abstract: From a security perspective, Multi-variant Execution (MVX) is widely employed in network security defense. However, MVX suffers from a common problem: when each execution variant returns content to the arbiter, the false alarms generated by the convergence of outputs are difficult to resolve. Excluding objective factors such as machine environments, false alarms occur because the voter begins making security judgments on inconsistent variables after receiving converged information. In addition to inconsistent variables caused by real attacks, normal system operations also generate inconsistent variables (such as memory descriptors, port numbers, random numbers, code, and thread calling sequences within processes), leading to voter misjudgment and affecting the normal operation of multi-variant systems. If the false alarm rate of multi-variant execution can be reduced, system efficiency and defense capability can be effectively improved.

This paper classifies types of multi-variant execution in recent years, summarizes the false alarm problems and solution strategies, analyzes the causes of false alarms in multi-variant execution, and selects three strategies for experimental analysis in specific application scenarios: the Pina algorithm for synchronization, compiler module instrumentation, and narrowing the voting boundary. We analyze the functionality and performance of each method and identify their respective advantages and disadvantages. Finally, we discuss unresolved challenges in existing multi-variant execution technologies and future research directions.

Keywords: multi-variant execution; security judgment; inconsistent variables; systematic false alarm

---

1.1 Security Vulnerabilities in Software Development

During system-level development, limitations of programming languages and developers' lack of security knowledge introduce vulnerabilities such as integer overflow, dangling pointers, and buffer overflows into applications. While extensive use of third-party libraries and open-source code improves efficiency, it also introduces security risks. For example, process control flow hijacking attacks—the most common type of software vulnerability—involve attackers exploiting memory information leaks to bypass protection mechanisms, obtain critical address information, inject payloads into programs, and steal sensitive data. Multi-variant execution is one of the effective methods to defend against

process control flow hijacking attacks.

1.2 Introduction to Multi-Variant Execution

The multi-variant execution architecture consists of execution variants and a monitoring module. Variants can be implemented as multiple processes or threads. Multi-variant execution technology [1,2] can both protect systems from memory corruption attacks and improve system operational efficiency. Its key idea is to run multiple different variants synchronously, provide them with identical inputs, and monitor their execution to detect divergences.

From a security defense perspective [3–6], if internal differences among running variants lead to significantly different responses to malicious inputs, the monitor can detect these differences during execution and then issue alerts or terminate execution, achieving the defense effect. This technology transparently executes identical inputs by enforcing or monitoring lock-step execution of variant system calls, receives outputs from each variant after execution, and enables the monitor to detect execution differences and issue alerts. The multi-variant execution framework is shown in Figure 1 [Figure 1: see original paper].

Relevant literature on multi-variant execution architecture primarily follows research institutions and teams. Below is a brief introduction to the development of more than ten major types of multi-variant execution systems:

In 2006, Cox et al. [21] first proposed the N-Variant execution architecture for software security, which defined equivalent items and detection mechanisms for ideal MVX system security characteristics [22–25]. This architecture used memory space heterogeneity and instruction set tagging to generate variants. Each variant functioned as an independent process. However, the system could not support multi-threaded redundant program execution and, due to incomplete functionality, limited the use of system calls. In the same year, Berger proposed the DieHard multi-variant execution architecture [26], which achieved heterogeneity by transforming heap objects in each variant. Each variant received input and output results to a shared memory region, where each variant had its own dedicated buffer. A voting process compared outputs from the shared memory region at predetermined synchronization points to produce correct results. The disadvantage was that the system could not defend against stack corruption attacks [27,28].

In 2007, Cavallaro [29] proposed a multi-variant architecture based on Address Space Partitioning (ASP), which "shifted" the address space of one variant in the framework by k bytes, changing the relative distance between the address spaces of two variants to achieve diversification and thereby prevent certain address overwrite attacks. The limitation was that the system could not resist attacks such as relative addressing.

In 2009, Salamat et al. proposed Orchestra [30], a multi-variant architecture that designed two variants with stacks growing in opposite directions based on

stack structure characteristics. The monitoring module in this framework ran as a separate process, using the ptrace API to intercept and compare system calls from variants without requiring any modifications to the operating system. The disadvantage was that intercepting and comparing system calls incurred significant performance overhead and required frequent context switching.

In 2003, Stijn Volckaert's team at Ghent University designed the GHUMVEE [31] monitor. Like other security-oriented multi-variant frameworks, GHUMVEE's monitoring component relied on the ptrace API [32,33] to monitor variants, running as a separate process in user space. It monitored system calls requiring synchronization and could correctly handle multi-threaded replication and asynchronous signal delivery. The disadvantage was frequent context switching.

In 2015, Hosek's team designed the Varan [34] multi-variant execution framework, which first introduced an asynchronous sharing mechanism similar to an in-memory record-replay framework. One variant served as the leader, directly executing system calls and writing results to a shared ring buffer. Other variants were followers, merely reading results from the ring buffer. The monitor was placed inside each variant, significantly reducing cross-process context switches. It loaded kernel modules to intercept system calls to detect variant behavior and unify variant outputs. The monitor acted as an intermediate component, reporting variant status to the kernel and performing recovery exception handling when a variant crashed or behavioral differences were detected. Varan combined selective binary rewriting [35] with high-performance event streams to provide a flexible and efficient user-space solution, but did not resolve false alarms generated by system calls between variants.

In 2016, Stijn Volckaert's team introduced the monitoring module IP-Mon [36,37]. ReMon combined the advantages of cross-process and in-process monitors. The in-process monitor in ReMon resided within program input, while the cross-process monitor enforced lock-step execution of potentially dangerous system calls. On the other hand, safe system calls proceeded without external monitoring, thereby improving system efficiency. The system placed the monitoring module inside application processes and used a "record+replay" method to provide multi-threading support. The record+replay method, also known as the leader/follower approach, initialized by establishing primary and secondary variants. The monitor captured the execution order of synchronization operations in the primary variant and enforced the same order in other variants.

In 2016, Koning et al. [38] proposed the MvArmor multi-variant architecture based on Dune, a hardware virtualization-based system model. This architecture improved the performance bottleneck of multi-variant architecture and balanced performance and security through custom security policies [39]. The architecture could dynamically generate variants through environment awareness. Unfortunately, the Dune virtualization framework was not thread-safe and has not been continuously updated.

In 2018, LUK et al. proposed the BUDDY [40] multi-variant execution architecture from a security defense perspective. BUDDY set synchronization points at I/O operations and monitored output ports, significantly improving architectural security defense capabilities and reducing the number of monitoring synchronization comparisons when attackers sent data externally. After intercepting system calls, the monitor returned results from a synchronization buffer to each variant, minimizing overall system performance overhead and resisting information leakage. The system's flaw was that it could not effectively solve false alarm problems between variants.

In 2019, SalamatB et al. proposed the Kmvx architecture [41] based on the operating system kernel, placing the monitor inside the kernel. The architecture generated two variants in kernel space by running two diversely compiled kernel variants on the same machine and constructing two completely disjoint virtual memory mappings. When executing system calls, both kernels processed simultaneously and performed synchronous checks on execution results to determine if kernel memory leaks existed. This architecture could prevent information leakage in the kernel but still had significant performance overhead, with some functions remaining to be explored.

In the same year of 2019, Alexios Voulimeneas et al. designed the DMON framework [42]. DMON distributed a set of variants across a set of heterogeneous physical machines. Variant heterogeneity included different instruction sets, character orders, calling conventions, system call interfaces, and potential hardware security feature differences. DMON designed two monitors: the L-MON monitor supervised the primary variant, while the secondary variant was supervised by its own F-MON monitor. These components interacted whenever a variant executed a system call. Whenever the primary or secondary variant attempted to enter or exit a system call, the corresponding L-MON or F-MON interrupted and suspended the variant's state, read the call number of the interrupted system call, and invoked a dedicated handler routine in the monitoring process. This routine implemented checking logic and replication logic for each variant's system call. The monitor interrupted upon entering system calls to check the handler. In F-MON, information about the variant state was collected, sent to L-MON, and waited for L-MON to confirm that the secondary variant was in a state equivalent to the primary variant. In L-MON, the checking handler waited for incoming state information from F-MON, compared this state information with the primary variant's state, and notified F-MON of the comparison result.

In 2020, Xiaoguang Wang et al. designed the MonGuard system [43]. MonGuard is a system that protects in-process monitors and libraries. It utilizes Intel MPK to efficiently update memory access permissions. MonGuard implements execute-only memory and code randomization to hide monitoring code, using it to implement a protected in-process MVX monitor. Experimental results show that MonGuard can significantly improve monitoring performance through reasonable intra-component isolation.

In 2020, based on mimic defense principles [1,2,44,45], Pan Chuanxing et al. implemented the MimicBox system [46]. When data outflow from redundant execution processes was detected in virtual memory space, the voter actively performed voting. MimicBox used ptrace system calls to implement system call interception, replacement, voting content extraction, and return value overwriting. The effectiveness of mimic execution defense was verified through the mimic pwn challenge in the 3rd Mimic Strong Network Cup Elite Challenge, but the voter experienced false alarms. The main reason was that redundant execution variants in MimicBox had inconsistent parameters such as process IDs, random numbers, and file descriptors, affecting the voter's adjudication.

A summary of multi-variant execution is shown in Table 1 .

1.3 Dynamic Heterogeneous Redundancy Concept

With the development of multi-variant execution, the concept of Dynamic Heterogeneous Redundancy (DHR) [7,8] emerged. This concept adds dynamic and feedback characteristics to multi-variant execution, as discussed in literature [9,10]. DHR consists of five main components: input module, processing module, output module, construction module, and scheduling module [11–13]. The input module divides user input into multiple copies and distributes them to each variant. The processing module sends multiple outputs from each variant's execution to the voter. The output module adjudicates the received multiple outputs, determining that if n or more results are consistent, the result is output; otherwise, the output process is truncated. The construction module builds the variant set. The scheduling module dynamically selects algorithms to choose variants from the variant set for scheduling. After variant scheduling is completed, the service variant is taken offline, cleaned, and rolled back to its original state. The dynamic heterogeneous redundancy framework is shown in Figure 2 [Figure 2: see original paper].

Each variant has a different structure but equivalent functionality. The voter determines whether an attack has occurred by comparing information differences between variants. Once the voter determines that the system is under attack, it calls a dynamic selection algorithm to select appropriate variants from backup heterogeneous components to replace abnormal variants, and performs negative feedback to the scheduling module through voting. DHR exhibits randomness, dynamism, and diversity. From an external perspective, these endogenous characteristics enable information systems to exist as a black box [14–16] when facing uncertain attacks, greatly increasing the difficulty for attackers. Theoretically, when variants in a multi-variant system have no common vulnerabilities, the system is sufficiently secure.

1.4 Development of Multi-Variant Execution

Before the proposal of multi-variant execution, diversification ideas based on operating systems were designed, including diversified machines and system call-

level mapping randomization. There were also software-based diversified code approaches, such as program-level instruction set randomization [17–20]. With the emergence of N-Variant, increasing numbers of domestic and international teams began researching multi-variant execution. After more than a decade of development, multi-variant execution has flourished. This paper selects representative systems for analysis.

2 Causes of False Alarms in Multi-Variant Execution

The essential cause of false alarms in multi-variant execution systems is that during redundant execution of multiple variants, the system itself provides parameters with inherent uncertainty, whose attributes are unpredictable during execution, such as random numbers and timestamps. Due to the heterogeneous redundancy of systems transformed by multi-variant execution, when users initiate requests, multiple variants may produce inconsistent responses, causing voting misjudgment by the monitoring module.

False alarm problems in multi-variant frameworks are inevitable and represent inherent issues of multi-variant systems themselves. Depending on granularity, false alarm problems manifest differently. Below are several common examples illustrating false alarm problems and their hazards:

2.1 Case 1: False Alarms Caused by File Read/Write

The file read/write case is shown in Figure 3 [Figure 3: see original paper]. The essence of the Linux operating system is that everything is defined through files, making file descriptors critically important, as most system calls require file descriptors for execution. Since process descriptors are exclusive resources of processes and are not shared between different processes.

Assume a system has two processes, treated as two variants, each with two threads. When the system performs a file opening operation, since the process is the smallest unit of resource ownership and the thread is the smallest scheduling unit, and a thread is an execution entity within a process, the actual operation is completed by a thread. That is, the primitive instruction for the sysc_{open} operation is executed by a certain thread within the process, but which specific thread performs the operation and the execution order are uncertain. When a process opens a file, if no specific thread is designated for the file opening operation, the two threads within the process will compete for the sysc_{open} system call. In a multi-variant execution framework, to ensure correctness during voting, it is necessary to maintain consistent input between primary and secondary variants. Under normal operations, opening a single file does not require considering multi-threading issues. However, when opening two files, the two threads within the process compete, easily causing confusion. If thread scheduling differs between two variants, their behavior transmitted to the voter will also differ. While process PIDs are unique throughout the system, thread PIDs differ.

This false alarm belongs to the category of voter adjudication errors. During mimicry transformation, parts that do not require heterogeneity were heterogeneously processed, causing inconsistent results from normal execution of multiple variants, which the voter incorrectly adjudicated as attacks, severely affecting the availability of multi-variant execution systems. Multi-variant execution systems rely on multiple functionally equivalent but heterogeneous process variants generated through software diversification techniques. Although randomization techniques such as Address Space Layout Randomization (ASLR) [47] ensure consistent functionality when the same program runs multiple times, inconsistencies may still exist at the operating system level, mainly reflected in file information corresponding to file descriptors opened by programs and process IDs.

2.2 Case 2: False Alarms Caused by Multi-Threading

Case 2 is the everyday producer-consumer scenario (Figure 4 [Figure 4: see original paper]). In a multi-threaded redundant execution environment, assume the primary and secondary variants each have a buffer. Only after the producer generates resources and places them in the buffer can the consumer consume resources. When the buffer has no resources, the consumer cannot consume. The producer generating one resource is recorded as +1, and the consumer consuming one resource is recorded as -1. During program execution, the primary variant's buffer resources undergo +1, -1, +1, -1 operations, while the secondary variant's buffer resources undergo +1, +1, -1, -1 operations. When displaying results, both achieve the same functionality, but during implementation, they perform different modifications. In the monitoring component's voter, if the monitoring granularity reaches the system call level, inconsistent results may occur during voting.

2.3 Case 3: Random Number False Alarms

Taking the random module in OpenSSL as an example, OpenSSL [48,49] is a cryptographic secure communication protocol module that can generate highly random passwords. Both ends of data transmission use a pair of keys for encryption and decryption to protect session security [50]. For example, the public key encrypts the session key sent by the user to the server, and the private key decrypts the session key. During SSL interaction, random numbers need to be generated. OpenSSL generates random numbers by computing digests of internal system data. In a multi-variant execution framework, each variant's OpenSSL module generates independent random numbers. During voter adjudication, if the voting granularity is too coarse, misjudgment may occur, treating the random numbers sent from variants as inconsistent parameters, leading to false alarms. Random number false alarms are shown in Figure 5 [Figure 5: see original paper].

2.4 Case 4: Code Heterogeneity False Alarms (Figure 6)

In code functions, to achieve a specific functionality, different programmers may write different code with different content but identical output due to different coding habits and function characteristics [51].

Figure 6 [Figure 6: see original paper] shows a code heterogeneity case diagram. In earlier multi-variant execution architectures, most adopted coarse granularity, comparing final outputs from each variant for voting judgment. This phenomenon did not receive much attention. However, with engineering development and the introduction of multi-threading concepts, monitoring points in multi-variant execution architecture monitors have become more numerous, monitoring granularity has become finer, and voting requires comparing not only variant results but also finer granularities, such as individual system calls of each variant. If a multi-variant execution architecture adopts finer granularity, monitoring the system call layer of variants and performing synchronous voting at the system call level, false alarm problems will occur. As shown in the two code blocks in the figure, the left code outputs "hello world" in one line, requiring one sys_{write}() operation, while the right code outputs "hello world" requiring two sys_{write}() operations. In the monitor, if the voting judgment rule compares system call information, false alarms can easily occur.

3 Solutions

3.1 Pina Synchronization Algorithm

Pina et al. proposed a Domain-Specific Language (DSL) [52] based on software diversity to solve the problem of benign false alarms caused by different system call execution orders between variants. When initializing the multi-variant execution environment, the architecture uses a monitoring process to intercept all system calls issued by variants. Rule matching between two different variant versions is shown in Figure 7 [Figure 7: see original paper].

DSL solves false alarm problems caused by different system call execution orders between variants. Starting from system call trace logs, rules are manually written as needed. These logs are obtained from each version separately. The specific comparison method: one version serves as the "leader," called the recording side, directly executing system calls and writing its results to a shared ring buffer. Other versions are called followers, reading the recording side's system call order from the ring buffer to complete their own system calls. At each system call execution, the algorithm performs match comparison to see if statements from both sides are equivalent. The first stat on both sides is equivalent, matching successfully and proceeding to the next step. The left side shows open, while the right side shows dup+lseek, which does not match successfully at this point. Pre-written rules are then used to determine whether open is equivalent to dup+lseek. When equivalence is determined, a skip operation is performed to jump to the next system call comparison. The next step finds close on both sides, successfully matching.

In the code heterogeneity case mentioned in Section 2.4, the typical case involves open operations and dup+lseek operations. Pina et al. also proposed an algorithm for automatically extracting DSL rules from system call trace [53] pairs. In the DSL language, LSR in the text is the recording side, and RSR is the follower side. DSL expresses rules for recording and replaying operations between two system call sequences. At each step, for each sequence, DSL takes the next system call as input and outputs the operation to be performed. For each system call between the recording sequence and replay sequence (steps 1 and 4 in Figure 7), DSL advances both sequences by one position through MATCH.

Figure 8 [Figure 8: see original paper] shows the sequential flow of the matching process. Empty circles represent Nop, which is a no-operation instruction used to control time cycles. Nop is an abbreviation for no operation. Executing a Nop instruction only increments the program counter PC, thus occupying one machine cycle. The EXEC system call does not create a new process but only replaces the content of the original process context; the code segment, data segment, and stack segment are replaced by the new process.

3.2 Compiler Module Instrumentation Strategy

At the micro level of threading, even with identical input to several variants, if the thread scheduling order differs between variants, their externally visible behavior may also differ, causing unnecessary differences during voting. Such benign false alarm problems affect multi-variant multi-threaded execution, as multi-variant execution environments inherently rely on detecting differences to identify attack behavior.

[56] The before_{{{sync}}*{{op}}}* and *after*{{{sync}}*{{op}}}* functions are shown in Figure 9 [Figure 9: see original paper], where black code represents source code and red represents instrumented code. The instrumented synchronization agent is a dynamic link library loaded and linked into the program at runtime through the LD*{PRELOAD} environment variable. During loading, synchronization agents of different variants mount to the synchronization buffer (sync buffer) through inter-process communication interfaces. The primary variant's synchronization agent records the execution order of sync ops in the sync buffer, while secondary variants query the sync op sequence and control the execution order of sync ops.

In the instrumentation module, software programs are refined internally through diversified compilation technology.

3.2.1 Total Order Synchronization Method In user space, a buffer is allocated that runs serial mutex locks. When a lock occupies buffer resources, it must wait for the lock to be released. [57] Semaphores A and B are used to identify the resource occupation status of the two threads being synchronized. From time t1 to t2, thread m1 executes system calls and thread s1 uses semaphore A for identification. At the start of an execution, A is 0, and at the end, A is

1. m1 and s1 are mutually exclusive for A. After one execution ends, thread s1 begins reading the call order of m1's buffer. Similarly, m2 and s2 are mutually exclusive for B. During the synchronization phase, each variant can only execute one system call simultaneously, achieving the goal of multi-threaded redundant execution. The total order synchronization diagram is shown in Figure 10 [Figure 10: see original paper].

3.3 Narrowing the Voting Boundary Strategy

False alarm problems caused by multi-variant execution technology significantly impact the usability of the technology and hinder its development. Traditional multi-variant execution transforms ordinary systems into multi-variant systems by first conducting functional tests, then examining voter logs, analyzing errors generated during testing, selecting false alarms, and performing secondary development on the system to fill false alarm "gaps." However, multi-variant systems contain multiple heterogeneous redundant variants. Filling one "gap" requires transforming multiple variants, consuming substantial resources and not effectively solving false alarm problems long-term.

Pan Chuanxing et al. proposed the concept of redundant heterogeneous boundaries [58]. Typically, overly large redundant heterogeneous boundaries cause transformers to perform heterogeneous redundancy processing on unnecessary components, leading to voting false alarms. Shao Yuwen et al. [59] proposed methods for selecting completely redundant heterogeneous components and optimal redundant heterogeneous sets. Based on heterogeneous redundancy, they narrowed the voting boundary by pruning unnecessary redundant heterogeneous components until the system with the fewest false alarms was obtained, representing the optimal redundant heterogeneous component set. Some components may not require heterogeneous redundancy but rather compatibility transformation, thus avoiding false alarms introduced during redundant heterogeneous transformation.

Shao Yuwen et al., based on the attack surface model for dissimilar redundancy information systems proposed by Wang Liqun's team [60], used attack cost-effectiveness ratio [61] to measure system security. They defined the multi-variant system's attack surface as a tuple consisting of the system's input and output set, channel transmission set, and untrusted data item set. The formal representation is:

For this attack surface surf, its measurement result can be expressed as:

where , , and respectively represent the attack cost-effectiveness ratios of system entry and exit point components , system channel components , and untrusted data item components .

System vulnerability can be expressed as:

Figure 11 [Figure 11: see original paper] shows statistics of false alarm rates for commercial websites. As seen in Figure 11, reasonably narrowing the mimicry

boundary under the premise of not affecting security can effectively reduce or even eliminate false alarm rates.

3.4 Comparative Experiments

System security is negatively correlated with system vulnerability [62]. The greater the system vulnerability, the smaller the system security.

Let systemA represent the system before multi-variant transformation, and systemB represent the system after multi-variant transformation. E represents the set of all components in systemA. E0 represents the optimal redundant heterogeneous component set, and EH represents the completely redundant heterogeneous component set. surfA represents the attack surface of the system before multi-variant execution transformation, and surfB represents the attack surface after mimicry transformation.

a) First, calculate the security gain [58] of the transformed system.

This experiment selected CentOS 7.3 as the environment and established three virtual machines with identical hardware configurations for testing. Environment 1 adopted the DSL strategy, Environment 2 adopted the compiler module instrumentation strategy, and Environment 3 adopted the narrowing voting boundary strategy. All three environments used three execution variants and one voter. The performance test environment is shown in Table 3 .

Table 3 Environment Hardware Configuration

Item	Configuration
CPU	Intel(R) Xeon® CPU E5-2603 v4 @ 1.70 GHz 6 cores
System	CentOS 7.3

b) Assign E0 the value of EH.

c) Arbitrarily select a component p from E0, where surf0 represents the attack surface of the system after removing the component.

The security gain at this point is:

For the four false alarm scenarios mentioned in Chapter 2, tests were conducted separately. The voting logs of the voter during execution were analyzed. In each corresponding application scenario, error entries were manually analyzed, and false alarm entries matching the types were selected. For example, in the multi-threaded program execution scenario, 1000 error entries were selected from each environment's voter adjudication logs. Entries with real errors such as deadlocks were excluded. Among the remaining entries, if system functionality was not affected, these entries were considered false alarms. False alarm statistics are shown in Table 4 .

Table 4 False Alarm Statistics

Environment	Method	Multi-threaded Out-of-Order Execution	OS File Read/Write	Random Number	Code Heterogeneity
Environment 1	DSL Strategy	211 entries	12 entries	23 entries	25 entries
Environment 2	Compiler Instrumentation	45 entries	12 entries	25 entries	45 entries
Environment 3	Narrow Mimicry Boundary	45 entries	12 entries	25 entries	45 entries

d) Continuously repeat step c) until all components in E0 have been selected once. The components in E0 at this point constitute the optimal redundant heterogeneous component set.

Table 2 Specific Configuration of Proxy Server and Variants

Proxy Server	Variant 1	Variant 2	Variant 3
System Type	Windows10	Ubuntu16	Centos7.0
Server Type	Nginx	Apache	Nginx
PHP Version	PHP5.6	PHP5.5	PHP7.0
PHP Tag	PHP Tag 1	PHP Tag 2	PHP Tag 3
Database Tag	Database Tag	Database Tag 2	Database Tag 3

This method takes the transformation of commercial websites as an example, selecting PHP script components, database service components, operating system components, and web server components as component set E0. Step four is continuously executed to select the optimal redundant heterogeneous component set. First, components requiring transformation are selected, then the commercial website system undergoes redundant heterogeneous transformation. Through access log analysis, false alarms in normal user requests (5000 arbitrary requests) are counted. The comparison of false alarm rates among the three strategies is shown in Figure 12 [Figure 12: see original paper].

As seen in Figure 12, in the multi-threaded out-of-order execution scenario, the compiler instrumentation method effectively reduces false alarms generated during multi-threaded execution, synchronizing thread call sequences between variants. The DSL method obviously does not consider multi-threaded execution

scenarios, generating false alarms. Once threads execute out of order, syntax matching fails. Similarly, the narrowing voting boundary method can also avoid some multi-threaded program out-of-order false alarms.

In the OS file read/write scenario, three test environments performed 1000 file open, write, and delete operations respectively. Although inconsistent process IDs (PIDs) and other issues occurred, multiple variants forwarded outputs to the voter through interface ports. Process IDs themselves did not generate many false alarms, so none of the three methods effectively solved such false alarms, and their respective advantages were not evident.

In the random number application scenario, random numbers generated by the OpenSSL module are also a high-frequency false alarm scenario. The narrowing mimicry boundary method, by finding the optimal redundant heterogeneous set, can obviously narrow the boundary for random number seed generation and effectively reduce random number false alarms. In comparison, the DSL and compiler instrumentation methods are less effective.

In the code equivalence heterogeneity application scenario, DSL can effectively determine code equivalence and reduce false alarms from this scenario, while compiler instrumentation and narrowing voting boundary do not consider false alarms caused by code heterogeneity.

4 Analysis of False Alarm Solutions

In multi-variant execution architecture, voter adjudication false alarm problems have always been a major issue in engineering implementation. The three solutions address part of the false alarm problem from different granularities and application scenarios, but each still has certain defects. For example, the DSL strategy was proposed in 2014 and did not consider that multi-threaded program execution might generate false alarms, making it unsuitable for multi-threaded program applications.

4.1 Functional Analysis

This section evaluates functionality based on whether the solution can effectively resolve false alarm problems. The evaluation of whether the three solutions achieve functional objectives primarily depends on whether they implement complete multi-variant execution system functionality while reducing false alarm rates, and whether the method can be generalized and applied to transform and innovate other multi-variant architectures.

First, the DSL strategy implements multi-variant execution synchronization at the software level. It designs algorithms for semantic judgment of operation instructions and proposes an algorithm for automatically extracting corresponding rules from systematic system calls, providing operations required for redundant execution. Experiments prove this method can solve code equivalence heterogeneity problems between different variant versions. When two variant versions

diverge and may issue different system calls or some different non-deterministic parameters, the monitor issues warnings and stops execution or terminates divergent versions. This method can be applied to different versions of the same program and can run native versions in parallel with versions used for dynamic analysis. This method can process ambiguous statements in variants at the semantic level, effectively avoiding false alarms.

In the compiler module instrumentation method, traditional system call monitors [63,64] rely on specific system call sequences for checking and do not compare system call parameters, making them vulnerable to masquerading attacks [65]. Mimicry attacks may execute sequences of dozens of system calls to evade detection. Finding such sequences is difficult. Static analysis techniques are typically used to disable security-critical system calls [66], but this approach is costly and disables some functional functions. The compiler instrumentation synchronization method embeds monitors within applications and sets up additional monitors in independent processes, performing monitoring and I/O replication operations at the system call level. Monitoring the system call interface, by default, all I/O operations can be monitored and replicated, and all potentially dangerous operations can be stopped at this interface, providing necessary security guarantees. Monitors implemented through loadable kernel modules to intercept system calls are most effective because they do not require additional context switches. The essence is to lock shared resources of threads, blocking thread execution.

The narrowing voting boundary method constructs an attack surface set. The specific method uses permutations and combinations to construct appropriate redundant heterogeneous component sets. By controlling one component while keeping others unchanged, the attack cost-effectiveness ratio formula is used to measure system security and false alarm resolution capability. The fundamental goal of multi-variant execution is to provide reliable, trustworthy, and available functionality. This method's advantage is its obvious effect in solving random number false alarm scenarios generated by each variant, effectively avoiding the impact of random numbers on the system. The strategy has wide applicability and is suitable for most application scenarios.

4.2 Performance Overhead Analysis

Another obstacle to applying multi-variant execution systems in practice is performance considerations. First, from a performance balance perspective, the key factors affecting system performance are the granularity of variant synchronization and interception and the communication cost between variants and monitors. The more monitoring objects and the finer the monitoring synchronization point granularity, the higher the performance, but the workload also increases accordingly. Some teams adopt distributed multi-variant execution architecture to better improve monitoring capability and security performance, such as the Dmon distributed multi-variant architecture. Dmon utilizes naturally existing cross-platform diversity to implement instruction set architecture execution and

leverages different hardware security mechanisms of ARM and X86 to further enhance multi-variant execution security. However, running Dmon's replication logic requires expensive network communication and may have risks of channel attacks. Except for government units or a few institutions, few enterprises would consume double or more costs to implement distributed multi-variant execution architecture. Therefore, when considering performance, it is necessary to measure multi-variant execution architecture and system security, visualize system characteristics, and delete some redundant attributes to reduce costs. Secondly, some teams in kernel-based multi-variant architectures can greatly improve overall architecture performance by deleting redundant attribute parameters during variant initialization or embedding monitors into the kernel.

Implementing transformation at the software level, the DSL strategy itself does not require changing physical hardware, making its implementation cost the smallest among the three methods. However, setting program synchronization breakpoints is very difficult. Through language processing, a set of language processing rules is formed. DSL can easily compare differences in system call sequences issued by two executions, significantly improving code heterogeneity and single-threaded system call issues. However, this method has limitations. Proposed to solve false alarm problems, it adds its own algorithm for semantic judgment. If semantic equivalence vulnerabilities not covered by the algorithm occur during complex multi-variant execution, the entire system needs to be changed.

For the compiler instrumentation synchronization method, the essence is to lock shared resources of threads. Blocking thread execution inevitably leads to performance degradation. However, this method can effectively solve false alarm problems caused by different system call orders between variants. The limitation is that it cannot track dynamically allocated memory space because all variable and address information is statically analyzed and instrumented. Although it solves system call order problems between variants and effectively avoids false alarms, adding a large number of lock mechanisms inevitably leads to performance degradation.

In the narrowing voting boundary method, implementation cost is reflected in selecting the optimal redundant heterogeneous component set. Attack cost-effectiveness ratio measurement is a screening strategy that does not consume hardware costs and can even streamline components requiring transformation, reducing system development overhead. At the software level, this strategy adds or deletes mimicry components without needing to synchronize system call-level or thread-level behavior of each variant or change the entire system, making it easier to implement than DSL and compiler instrumentation. The disadvantage is that security measurement is difficult during component selection, and calculating security gain is challenging.

5.1 Summary of Ideas

Regarding false alarm problems in multi-variant execution, since the emergence of multi-variant architecture in 2006, it has always been a difficult problem. Uncertain and inconsistent parameters or states in variants affect voter adjudication. These problems partly come from multi-threaded programs and scheduling issues of subprocesses and threads, and partly from asynchronous signals, file descriptors, process IDs, time, and random number inconsistencies, all of which cause false alarms during voter adjudication. This paper provides examples of false alarm problems in practical applications and summarizes three different types of false alarm solutions. Through horizontal comparison of these three methods, their respective advantages and disadvantages are presented. The three methods provide great inspiration for research on reducing false alarms in multi-variant execution. This paper details cases and causes of false alarm problems in multi-variant execution, compares the advantages and disadvantages of strategies designed by domestic and international teams in recent years to solve false alarm problems, and points out functional and performance considerations that need attention during engineering implementation.

Introducing multi-threaded redundant execution scenarios into multi-variant execution has always been a research focus. There are typically two approaches to solving multi-threaded redundant execution: 1) Deterministic multi-threaded systems, which impose a deterministic order on inter-thread communication instructions by establishing a fixed schedule for each given program input. The deterministic multi-threading mode divides programs into parallel and serial phases. The serial phase: the execution phase where threads share operations is classified as a serial phase, where only one thread executes at any moment. The parallel phase: phases without shared operations allow threads to execute in parallel, with hardware counters determining the atomic start and end points of program execution. 2) Record+replay mode systems [67], which perform record+replay operations at runtime, dividing into one primary variant and multiple secondary variants. Secondary variants capture the execution order of synchronization operations in the primary variant and enforce the same order in other variants. In recent years, research around record+replay has become a key focus of multi-variant execution technology because this monitoring and interception granularity is finer and more efficient, improving overall system performance.

5.2 Unsolved Challenges in Current Technology

Although the three strategies can effectively eliminate some false alarms between variants, some situations can still cause false alarms. Actual system calls usually do not execute at the same time. For example, variants request to open files in read-only mode. If any of these files are changed by a third-party application after being read by one variant but before being read by other variants, a race condition exists, and variants will receive different data, causing differences between them. If variants attempt to directly use processor timestamp counters,

such as the RDTSC [68] instruction available on X86 processors, false alarms will be triggered. Because reading the timestamp counter is executed without any system call invocation, the monitor is not notified and cannot replace the results received by variants.

5.3 Future Outlook

Future research on false alarm problems in multi-variant execution should consider avoiding introducing unnecessary security issues, combining emerging attack methods in recent years, such as vulnerabilities in commonly used operating systems like Linux, channel attack vulnerabilities [69,70], and OpenSSL module vulnerabilities, designing from a security perspective. At the efficiency level, approaches can start from compilation-supported multi-variant execution ideas. By analyzing and extracting non-consistent parameters and addresses that generate false alarms in advance, synchronization libraries can be designed and instrumented into predecessor and successor functions. Then, according to the primary-secondary synchronization mechanism, key operations such as execution results or call orders of the primary variant are replicated to secondary variants. Another approach is from the container perspective. Containers are essentially resource-isolated processes that isolate and limit process resources through kernel mechanisms Namespace and Cgroup [71,72]. Equivalent container [72] processes each have independent file system views and will not generate false alarms due to different target files pointed to by file descriptors. Moreover, containers share the operating system kernel with the host, far surpassing traditional virtual machines in resource usage and speed. However, containers also have disadvantages. Although they can solve false alarms of system shared resources such as file descriptors, they temporarily cannot solve problems when using random numbers or random scheduling of threads in multi-threaded programs, requiring further research.

Acknowledgments

This work was supported by the National Natural Science Foundation of China (61521003) and the Purple Mountain Laboratories for Network Communication and Security.

References

[1] Yao Dong, Zhang Zheng, Zhang Gaofei, et al. Summary of Research on Multi-variant Execution Security Defense Technology [J]. Journal of Cyber Security, 2020, 5 (5): 77-94.

[2] Salamat B. “Multivariant program execution: Using multi-core systems to defuse buffer-overflow vulnerabilities,” Proc. -CISIS 2008 2nd Int. Conf.

Complex, Intell. Softw. Intensive Syst. pp. 843–848, 2008, doi: 10.1109/CISIS.2008.136.

[3] Salamat B. "Reverse stack execution in a multi-variant execution environment" Work. Compil. Archit. Tech. Appl. Reliab. Secure, pp. 1-7.

[4] Chen Ping. Research on the Attack and Defense Techniques of Code Reuse [D]. Nanjing: Nanjing University, 2012.

[5] Liu Tong, Shi Gang, Meng Dan. A Survey of Code Reuse Attack and Defense Mechanisms [J]. Journal of Cyber Security, 2016, 1 (2): 15-27.

[6] Tran M, Etheridge M, Bletsch T, et al. On the expressiveness of return-into-libc attacks [C]// International Workshop on Recent Advances in Intrusion Detection. Springer, Berlin, Heidelberg, 2011: 121-141.

[7] Wu Jiang Xin. Research on Cyber Mimic Defense [J]. Journal of Cyber Security, 2016 (4): 1-10.

[8] Yao Dong, Zhang Zheng, Zhang Gaofei, et al. MVX-CFI: a practical active defense framework for software security [J]. Journal of Cyber Security, 2020, 5 (4): 44-54.

[9] Wu Jiang Xin. Principles of mimic defense in cyberspace: generalized robust control and endogenous security (Volume 1) [M]. Beijing: Science Press, 2018.

[10] Wu Jiang Xin. Principles of mimic defense in cyberspace: generalized robust control and endogenous security (Volume 2) [M]. Beijing: Science Press, 2018.

[11] Ma Hai Long, Yi Peng, Jiang Yi Min. Router mimic defense architecture based on dynamic heterogeneous redundancy mechanism [J]. Journal of Cyber Security, 2017, 2 (01): 29-42.

[12] Wu Ting. Defense enhanced dynamic heterogeneous redundant architecture based on executive body division [J]. Journal of Communications, 2021, 42 (3): 122-134.

[13] Tong Qing, Zhang Zheng, Zhang Weihua, et al. Design and implementation of mimic defense Web server [J]. Ruan Jian Xue Bao/Journal of Software, 2017, 28 (4): 883-897.

[14] Zhang MingWu, Shen Hua, Mu Yi. Program confusion for virtual black box security: models, progress and challenges [J]. Journal of Computers, 2017, 40 (12): 2700-2718.

[15] Kuppa A, Le-Khac N A. Black box attacks on explainable artificial intelligence (XAI) methods in cyber security [C]// 2020 International Joint Conference on Neural Networks (IJCNN). IEEE, 2020: 1-8.

[16] Kalin J, Ciolino M, Noever D, et al. Black Box to White Box: Discover Model Characteristics Based on Strategic Probing [C]// 2020 Third International Conference on Artificial Intelligence for Industries (AI4I). IEEE, 2020: 60-63.

[17] Shacham H. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86) [C]// Proceedings of the 14th ACM conference on Computer and communications security. 2007: 552-561.

[18] Barrantes E G, Ackley D H, Forrest S, et al. Randomized instruction set emulation to disrupt binary code injection attacks [C]// Proceedings of the 10th ACM conference on Computer and communications security. 2003: 281-289.

[19] Sinha K, Kemerlis V P, Sethumadhavan S. Reviving instruction set randomization [C]// 2017 IEEE International Symposium on Hardware Oriented Security and Trust (HOST). IEEE, 2017: 21-28.

[20] Guanciale R. Protecting instruction set randomization from code reuse attacks [C]// Nordic Conference on Secure IT Systems. Springer, Cham, 2018: 421-436.

[21] Cox B, Evans D, Filipi A, et al. N-Variant Systems: A Secretless Framework for Security through Diversity [C]// USENIX Security Symposium. 2006: 105-120.

[22] Bala V, Duesterwald E, Banerjia S. Dynamo: A transparent dynamic optimization system [C]// Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation. 2000: 1-12.

[23] Salamat B. Multi-Variant Execution: Run-Time Defense against Malicious Code Injection Attacks DISSERTATION [D]. Ph.D. Dissertation. University of California, Irvine, 2009.

[24] Holland D A, Lim A T, Seltzer M I. An architecture a day keeps the hacker away [J]. ACM SIGARCH Computer Architecture News, 2005, 33 (1): 34-41.

[25] Bruschi D, Cavallaro L, Lanzi A. Diversified process replicæ for defeating memory error exploits [C]// 2007 IEEE International Performance, Computing, and Communications Conference. IEEE, 2007: 473-478.

[26] Berger E D, Zorn B G. DieHard: Probabilistic memory safety for unsafe languages [J]. Acm sigplan notices, 2006, 41 (6): 158-168.

[27] Saito T, Watanabe R, Kondo S, et al. A survey of prevention/mitigation against memory corruption attacks [C]// 2016 19th International Conference on Network-Based Information Systems (NBiS). IEEE, 2016: 500-505.

[28] Conti M, Crane S, Davi L, et al. Losing control: On the effectiveness of control-flow integrity under stack attacks [C]// Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. 2015: 952-963.

[29] Cavallaro L. Comprehensive memory error protection via diversity and taint-tracking [D]. PhD thesis, PhD dissertation, Universita Degli Studi Di Milano, 2007.

[30] Salamat B, Jackson T, Gal A, et al. Orchestra: intrusion detection using parallel execution and monitoring of program variants in user-space [C]// Proceedings of the 4th ACM European conference on Computer systems. 2009: 33-46.

[31] Volckaert S, De Sutter B, De Baets T, et al. GHUMVEE: efficient, effective, and flexible replication [C]// International Symposium on Foundations and Practice of Security. Springer, Berlin, Heidelberg, 2012: 49-64.

[32] Maurer M, Brumley D. TACHYON: Tandem execution for efficient live patch testing [C]// 21st{USENIX}Security Symposium ({USENIX}Security 12). 2012: 617-630.

[33] Hosek P, Cadar C. Safe software updates via multi-version execution [C]// 2013 35th International Conference on Software Engineering (ICSE). IEEE, 2013: 612-621.

[34] Hosek P, Cadar C. Varan the unbelievable: An efficient n-version execution framework [J]. ACM SIGARCH Computer Architecture News, 2015, 43 (1): 339-353.

[35] Hu H, Shinde S, Adrian S, et al. Data-oriented programming: On the expressiveness of non-control data attacks [C]// 2016 IEEE Symposium on Security and Privacy (SP). IEEE, 2016: 969-986.

[36] Volckaert S, Coppens B, Voulimeneas A, et al. Secure and efficient application monitoring and replication [C]// 2016{USENIX}Annual Technical Conference ({USENIX}{ATC}16). 2016: 167-179.

[37] Volckaert S, Coppens B, De Sutter B. Cloning your gadgets: Complete ROP attack immunity with multi-variant execution [J]. IEEE Transactions on Dependable and Secure Computing, 2015, 13 (4): 437-450.

[38] Belay A, Bittau A, Mashtizadeh A, et al. Dune: Safe user-level access to privileged{CPU}features [C]// 10th{USENIX}Symposium on Operating Systems Design and Implementation ({OSDI}12). 2012: 335-348.

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

[40] Lu K, Xu M, Song C, et al. Stopping memory disclosures via diversification and replicated execution [J]. IEEE Transactions on Dependable and Secure Computing, 2018.

[41] Österlund S, Koning K, Olivier P, et al. kMVX: Detecting kernel information leaks with multi-variant execution [C]// Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems. 2019: 559-572.

[42] Voulimeneas A, Song D, Parzefall F, et al. DMON: A Distributed Heterogeneous N-Variant System [J]. arXiv preprint arXiv:1903.03643, 2019.

[43] Wang X, Yeoh S M, Olivier P, et al. Secure and efficient in-process monitor (and library) protection with intel mpk [C]// Proceedings of the 13th European workshop on Systems Security. 2020: 7-12.

[44] Zhang Yu Jia, Pang Jian Ming, Zhang Zheng, et al. A mimic security defense strategy based on software diversity [J]. Computer Science, 2018, 45 (2): 215-221.

[45] Pang Jian Ming, Zhang Yu Jia, Wu Jiang Xing, et al. Applying a combination of mimic defense and software diversity in the software security industry [J]. Strategic Study of Chinese Academy of Engineering, 2016, 18 (6): 74-78.

[46] Pan Chuanxin, Zhengzhen, Mabolin, et al. Method against process control-flow hijacking based on mimic defense [J]. Journal on Communications, 2021, 42 (1): 37-47.

[47] Gras B, Razavi K, Bosman E, et al. ASLR on the Line: Practical Cache Attacks on the MMU [C]// NDSS. 2017, 17: 26.

[48] Viega J, Messier M, Chandra P. Network security with openSSL: cryptography for secure communications [M]. "O'Reilly Media, Inc.", 2002.

[49] Jimenez M, Papadakis M, Le Traon Y. An empirical analysis of vulnerabilities in openssl and the linux kernel [C]// 2016 23rd Asia-Pacific Software Engineering Conference (APSEC). IEEE, 2016: 105-112.

[50] Cao Zhi Bo. Heartbleed Vulnerability in OpenSSL [J]. Electronic Technology and Software Engineering. 2017 (13): 262-263.

[51] Zhang Yu Jia, Zhang Xiao Chuan, Pang Jian Ming. Survey on code obfuscation research [J]. Journal of Information Engineering University, 2017, 18 (5): 635-640.

[52] Pina L, Grumberg D, Andronidis A, et al. A{DSL}Approach to Reconcile Equivalent Divergent Program Executions [C]// 2017{USENIX}Annual Technical Conference ({USENIX}{ATC}17). 2017: 417-429.

[53] Harvan M, Pretschner A. State-based usage control enforcement with data flow tracking using system call interposition [C]// 2009 Third International Conference on Network and System Security. IEEE, 2009: 49-56.

[54] Volckaert S, Coppens B, De Sutter B, et al. Taming parallelism in a multi-variant execution environment [C]// Proceedings of the Twelfth European Conference on Computer Systems. 2017: 270-285.

[55] Ho W W, Olsson R A. An approach to genuine dynamic linking [J]. Software: Practice and Experience, 1991, 21 (4): 375-390.

[56] Roemer R, Buchanan E, Shacham H, et al. Return-oriented programming: Systems, languages, and applications [J]. ACM Transactions on Information and

System Security (TISSEC), 2012, 15 (1): 1-34.

[57] Lamport L. Time, clocks, and the ordering of events in a distributed system [M]// Concurrency: the Works of Leslie Lamport. 2019: 179-196.

[58] Yao yuan, pan Chuanxing, Zhang Zheng, et al. Quantitative evaluation method of diversified software systems [J]. Acta Telecom Sinica, 2020, 41 (3): 120-125.

[59] Yuwen S, Zheng Z, Bingzheng L, et al. A Multi-Variant Voting Algorithm Based on Dynamic Feedback [C]// 2021 2nd International Conference on Computer Communication and Network Security (CCNS). IEEE, 2021: 1-7.

[60] Zhang Zheng, Wang Li Qun, Li Wei Cheng. Research on formal model for an information system's attack surface with dissimilar redundant architecture. Journal Communications [J], 2018, 39 (S2): 227-234.

[61] Wu Jiangxing. Endogenous security: redefining the security attribute of new infrastructure [J]. China Science and technology industry, 2020 (5): 7-9.

[62] Li Wei Chao, Zheng Zheng, Wang Li Qun. Redundancy adjudication modeling and risk analysis based on mimic defense architecture [J]. Journal of Cyber Security, 2018, 3 (5): 64-74.

[63] Sato M, Taniguchi H, Yamauchi T. Design and implementation of hiding method for file manipulation of essential services by system call proxy using virtual machine monitor [J]. International Journal of Space-Based and Situated Computing, 2019, 9 (1): 1-10.

[64] Garfinkel T, Pfaff B, Rosenblum M. Ostia: A Delegating Architecture for Secure System Call Interposition [C]// NDSS. 2004.

[65] Parampalli C, Sekar R, Johnson R. A practical mimicry attack against powerful system-call monitors [C]// Proceedings of the 2008 ACM symposium on Information, computer and communications security. 2008: 156-167.

[66] Ghavamnia S, Palit T, Mishra S, et al. Temporal system call specialization for attack surface reduction [C]// 29th{USENIX}Security Symposium ({USENIX}Security 20). 2020: 1749-1766.

[67] Ronsse M, De Bosschere K. RecPlay: A fully integrated practical record/replay system [J]. ACM Transactions on Computer Systems (TOCS), 1999, 17 (2): 133-152.

[68] Song C, Yongqing W. Detection and research of RTX timer actual computing workload for ONC system based on RDTSC [J]. The International Journal of Advanced Manufacturing Technology, 2011, 57 (1-4): 257.

[69] Evtyushkin D, Riley R, Abu-Ghazaleh N C S E E C E, et al. Branchscope: A new side-channel attack on directional branch predictor [J]. ACM SIGPLAN Notices, 2018, 53 (2): 693-707.

[70] Yarom Y, Benger N. Recovering OpenSSL ECDSA Nonces Using the FLUSH+RELOAD Cache Side-channel Attack [J]. IACR Cryptol. ePrint Arch., 2014, 2014: 140.

[71] Rosen R. Resource management: Linux kernel namespaces and cgroups [J]. Haifux, May, 2013, 186: 70.

[72] Xing F, Zhang Z, Ma B, et al. Design and implementation of endogenous security container based on union file system [C]// Journal of Physics: Conference Series. IOP Publishing, 2021, 2078 (1): 012080.

Note: Figure translations are in progress. See original paper for figures.