Application Hibernation Framework with Dynamic Throttling of Resources

Sang-Hoon Choi[1], Seong-Jin Kim[1], Hanjin Park[2] and Ki-Woong Park*
[1]SysCore Lab, Sejong University, Seoul, South Korea
csh0052@gmail.com, sys.ryan0902@gmail.com
[2]The Affiliated Institute of ETRI, South Korea
hjpark001@nsr.re.kr
*Dept of Information Security, and Convergence Engineering for Intelligent Drone
Sejong University, Seoul, South Korea, woongbak@sejong.ac.kr

Abstract

Security applications that reside in system memory and run continuously are often designed to detect and defend against various attacks by running multiple processes within a system. Such security processes constantly occupy computer resources such as memory and the central processing unit while running. However, in computer systems not all types of security applications need to run continuously. The security applications that need to be run and those that do not, may change depending on the purpose and situation of the computer system. Therefore, orchestrating the complicated security applications according to the purpose and situation of the computer system would allow the system to more efficiently. To solve this problem, this study proposes a framework that dynamically adjusts the computational resources to be allocated to the process so that the process reduces the memory consumption, while preventing the starving process from being forcibly terminated.

Keywords: Resource Management, Service Orchestration, Dynamic Control, Container

1 Introduction

In a computer system, various security solutions are operated in a complex manner within a system to detect and respond to various attack strategies and techniques. Each process continuously consumes computing resources and maintains its operational state. However, in computer systems not all types of security applications need to run continuously [1, 2]. Nevertheless, even processes that are not necessary for the use of a computer system occupy computing resources and remain in the background. Therefore, a proper control mechanism is required for a large number of security applications that continuously occupy computing resources.

To efficiently manage computing resources from a process control perspective, we can inherently develop the following two scheduling algorithms: (1) process control by selectively suspending and resuming and (2) controlling processes by selectively terminating and restarting. Modern operating systems have process scheduling to use computing resources more efficiently by suspending (i.e., pausing them) processes and resuming them based on their strategies. However, the conventional process management scheme is not sufficient to manage the processes related to security applications because security processes have special characteristics, such as the self-protection function and the strong context in communication. Managing the security process is associated with the following difficulties:

The 6th International Symposium on Mobile Internet Security (MobiSec'22), December 15-17, 2022, 2021, Jeju Island, Republic of Korea
Article No. 1, pp. 1-13
*Corresponding author: Dept of Information Security, and Convergence Engineering for Intelligent Drone, Sejong Univ

- **Challenge 1: Different self-protection mechanisms for security processes**: Different self-protection mechanisms for security processes: In the Linux operating system there are various types of self-protection mechanisms to prevent processes from terminating themselves. For example, there are methods to prevent the process from being terminated by setting the flag for a particular process with the SIGNAL_UNKILLABLE flag for the process (pid 1), maintaining a keep-alive process in addition to the main application and restarting the process immediately when it terminates [3]. There may be other methods to prevent processes from terminating. The protection mechanisms for these processes are discussed in detail in Section 2.

- **Challenge 2: Time overhead for restarting the security process**: Time overhead for restarting the security process: Even if the process to be controlled does not have a self-protection mechanism, it is inefficient in terms of process execution time to terminate the process and reclaim the computing resources occupied by the process. Because the operating system must go through a series of steps to start a new process, re-executing a process after termination requires more time than using a process that is already in the system.

In this study, we propose a process hibernation scheme that can be used as a base technology for security application orchestration in a Linux environment to improve the computing resource inefficiency caused by a large number of security processes that constantly occupy resources and maintain the operational state to protect the computer system. To achieve the efficient use of computing resources while addressing the above challenges, the proposed scheme efficiently allocates computing resources to the security processes and keeps the processes running instead of pausing or terminating them.

The process hibernation scheme can efficiently use the computing resources while bypassing the activation of the security application's self-protection mechanism by allocating the minimum resources to the security processes that do not need to be fully executed at the moment and keeping them in the running state. (i.e., the proposed scheme can solve challenge 1.) Second, the proposed scheme can also save time required to restart the process after termination, as mentioned in challenge 2, because the proposed scheme maintains the security processes as running states without terminating the process. The user can notify the program running the process hibernation scheme to stop minimizing the resource footprint of a process and activate it.

The remainder of this paper is organized as follows. Section 2 provides an overview of the background knowledge of the underlying technology of the process hibernation scheme. Section 3 describes the design and implementation of the process hibernation scheme. Section 4 describes the experiments conducted to verify the operation of the process hibernation scheme. Finally, Section 5 presents the conclusions of this study.

2 Related Work

This section reviews the process control methods proposed in previous studies. Previous studies on process control were analyzed, highlighting their limitations and explaining the specifics of the proposed process hibernation scheme. Kim et al. proposed application-aware dynamic memory request throttling, which controls the memory request rate generated in low-priority application instances to a low level to strictly limit the performance degradation of high-priority applications due to resource contention [4]. They distinguished application instances into critical and normal working instances and grouped them into critical and normal control groups (cgroups). Subsequently, the memory request rate generated in the normal cgroup was controlled to a low value when needed, to avoid excessive memory contention when running critical job instances. However, such a method of process control is not compatible with

the existing commercial off-the-shelf (COTS) kernel because it requires a change in kernel code related to IPC and task creation to designate an application as a critical task.

Muralidhara et al. proposed application-aware memory channel partitioning (MCP) to increase the efficiency of memory resources shared by applications in an environment with multiple applications running in the system [5]. MCP increases memory usage efficiency by controlling the memory channels that a process could use, so that each process can independently access separate parts of memory without causing memory contention. Nevertheless, it is difficult to introduce MCP into existing computer systems because additional hardware support is required to estimate the row-buffer hit rate of each application to achieve process control.

Yun et al. proposed BWLOCK, a memory access control framework to protect the memory performance critical sections (MPCSs) of soft real-time applications [6]. BWLOCK protects MPCS by dynamically controlling the memory bandwidth of the processes through a user-level library and kernel-level memory bandwidth control mechanism. The user-level library of BWLOCK provides a lock-like API that can declare the MPCS for real-time applications. When a real-time application enters the MPCS, the kernel-level memory bandwidth control mechanism controls the memory bandwidth corresponding to the rest of the process until the MPCS completes. While BWLOCK is a groundbreaking scheme for protecting the MPCS of real-time applications, it is subject to the constraint that the application to be controlled must be implemented with the user-level library of BWLOCK. If it is not possible to modify the code of an application running on the computer system, there is a limit to the control of the process by this mechanism.

Iyer et al. proposed a memory architecture for a chip multiprocessor architecture (CMP) platform that supports quality of service (QoS) to handle the unsupported hardware or software that could control the allocation of platform resources such as cache space and memory bandwidth to individual workloads [7]. This architecture supports high-priority applications with more cache and memory resources according to the guidelines for the operating environment. In addition, this architecture also enables dynamic resource reallocation during runtime to optimize the performance of high-priority applications while minimally affecting the performance of low-priority applications. However, the platform had difficulties in terms of compatibility with the existing kernel, as it required additional hardware support and kernel modifications.

Table 1: Requirements for the implementation of techniques proposed in previous studies and this study

Requirements Study	**Kernel code modification**	**Hardware addition/modification**	**Application code modification**
[4]	X	-	-
[5]	-	X	-
[6]	-	-	X
[7]	X	X	-
Our Research	-	-	-

Existing studies on process control methods for efficient management of computing resources can be broadly divided into studies based on kernel code modification, hardware addition or modification-based, and application code modification (with library). Table 1 lists the requirements for the process control methods proposed in previous studies. A method that requires kernel code modification or hardware addition/modification has the limitation of being difficult to apply to existing commercial computer systems because it requires modifications. In addition, a method that provides a library and requires implementation of an application using the library is limited because it is impossible to control the application provided by a third party whose code cannot be modified. Unlike the techniques proposed in previous studies, the process hibernation scheme does not require kernel code modification or hardware addition/modification.

3 Process Hibernation Scheme

This section presents the design and implementation of the process hibernation scheme for efficient orchestration of security applications in a Linux environment. The process hibernation scheme can be used as a general-purpose security application control method providing a single and consistent control of various self-protection mechanisms for the processes. Even if the hibernation scheme is applied to a process that does not have a self-protection mechanism, the process resource efficiency and the time required to prepare the process for use can be saved.

3.1 Overview of the Process Hibernation Scheme

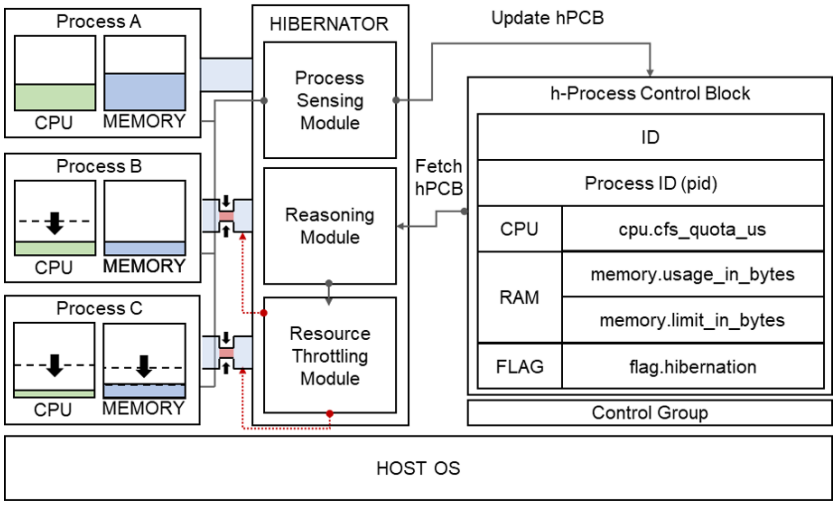

Figure 1: Structure of the Process Hibernation Scheme.

Since some security applications have self-protection mechanisms, it is difficult to apply the existing process control methods (suspending/resuming by SIGSTOP/SIGCONT or kill/restart of a process that is not actively in use) used in the Linux operating system, as mentioned in Challenge 1. Even if the process can be terminated because there is no self-protection mechanism, the restart is time inefficient, as mentioned in Challenge 2. To overcome these challenges, the proposed scheme monitors the resources of a process (e.g., memory), calculates the resource limit that needs to be allocated to the process, and assigns it to the process.

Therefore, we propose a new scheme called the process hibernation scheme. The process hibernation scheme consists of three modules: a process sensing module, reasoning module, and resource throttling module, as shown in Figure 1. We also defined a special data structure called the h-process control block (hPCB) that stores process-related information for these modules to efficiently control the processes. The process-sensing module measures the resource usage of the processes and manages the information by updating it to the hPCB. The reasoning module retrieves the data from the hPCB, evaluates the current resource status of the process, and quantifies it in terms of a score. The measured score is used to determine the extent of change in the resource allocation for the process. The resource-throttling module adjusts the resource limit for a process based on the score calculated by the reasoning module to minimize resource usage without terminating the controlled process. For the three modules mentioned above, the process hibernation scheme uses the control group namespace as the base technology for controlling the processes. The cgroup is a Linux kernel function that limits and isolates the resource usage (CPU, memory, etc.) of processes [8].

3.2 The h-Process Control Block Data Structure

The process hibernation scheme defines and uses hPCB, a data structure that stores and manages the information required for efficient process control. The three modules for process hibernation access hPCB to retrieve information for process orchestration or to update process monitoring information. The structure of hPCB is shown in Figure 2. The h-process Control Block consists of seven fields. The description of each field is as follows:

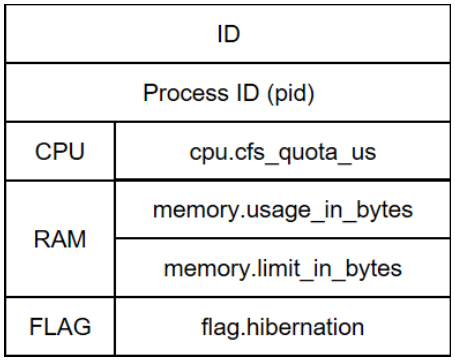

Figure 2: Structure of h-process control block (hPCB).

- `ID:` This is the identifier of hPCB.
- `Process ID:` Specifies the pid of the process included in hPCB.
- `cpu.cfs_quota_us:` Indicates the 1 cycle CPU time used by processes in the cgroup.
- `memory.usage_in_bytes:` Displays how much memory the controlled process uses.
- `memory.limit_in_bytes:` Indicates the maximum amount of memory used by the controlled process.
- `flag.hibernation:` This is the identifier of hPCB.

3.3 Process Sensing Module

The process-sensing module, highlighted in Figure 3, monitors the resource usage and maximum resource usage limit of the process specified by the user. The process sensing module updates the resource usage and limit of the monitored process in the hPCB.

First, we calculate the occupancy by proc for the CPU. When the user-specified occupancy is exceeded, the CPU usage time is limited by the cpu.cfs_quota_us control. Second, in the case of RAM, the memory.usage_in_ byte information of the control group to which the process belongs is monitored to obtain the memory usage of the current process, and the maximum allowed memory resource usage is monitored by the memory.limit_in_bytes information.

3.4 Reasoning Module

The reasoning module measures the score based on the hPCB information for process control, as shown in Figure 4. The reason for measuring the score for process control is to prevent the process from being forced to terminate due to lack of resources, and to minimize the memory usage of the process. If a value lower than the memory usage currently being used by the process is allocated to the cgroup to which the

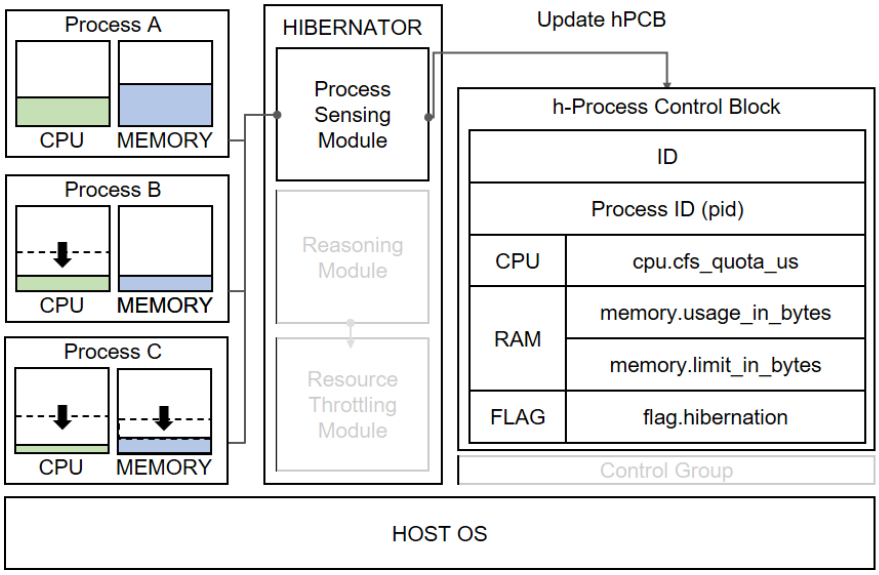

Figure 3: Process Sensing Module.

process belongs as memory.limit_in_bytes, the process is forcibly terminated due to insufficient memory. If a process is forcibly terminated unintentionally, it can be recovered by the self-protection mechanism of the process. Even for a process without a self-protection mechanism, time inefficiency occurs because it takes a long time to resume or restart the process when it is needed. Because the process hibernation scheme minimizes the resources used by the process without terminating it, an appropriate resource limit is set to prevent the process from terminating due to insufficient memory, while inducing the process to reduce resource usage. The details of the process control scoring algorithm are presented in Section 3.6.

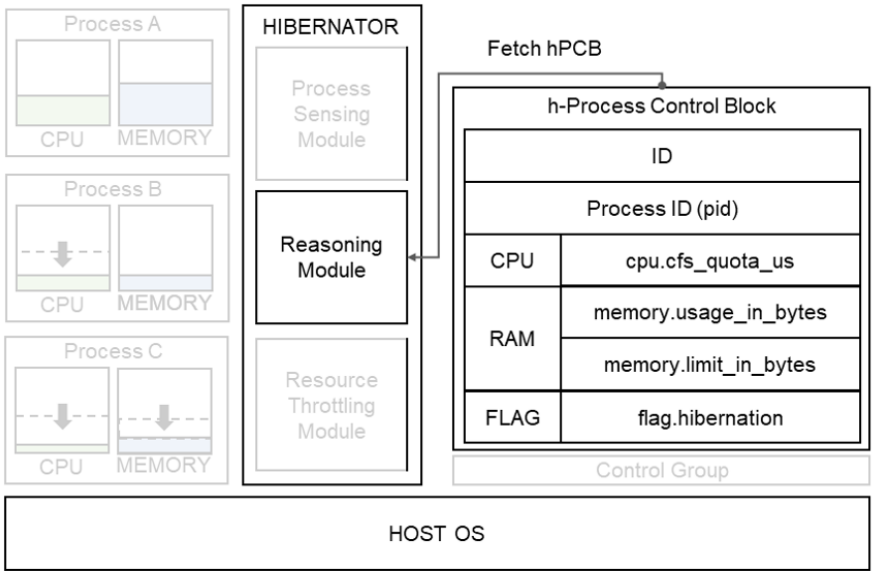

Figure 4: Reasoning Module.

3.5 Resource Throttling Module

The resource-throttling module dynamically adjusts the resource limit for a process based on the score calculated by the reasoning module to minimize resource usage without terminating the controlled process. The resource-throttling module is shown in Figure 5. The resource throttling module allocates a separate space to the control group for the processes to be controlled by the user and accesses the control group to dynamically control the maximum resource usage. There are two ways a user can register a process to be controlled.

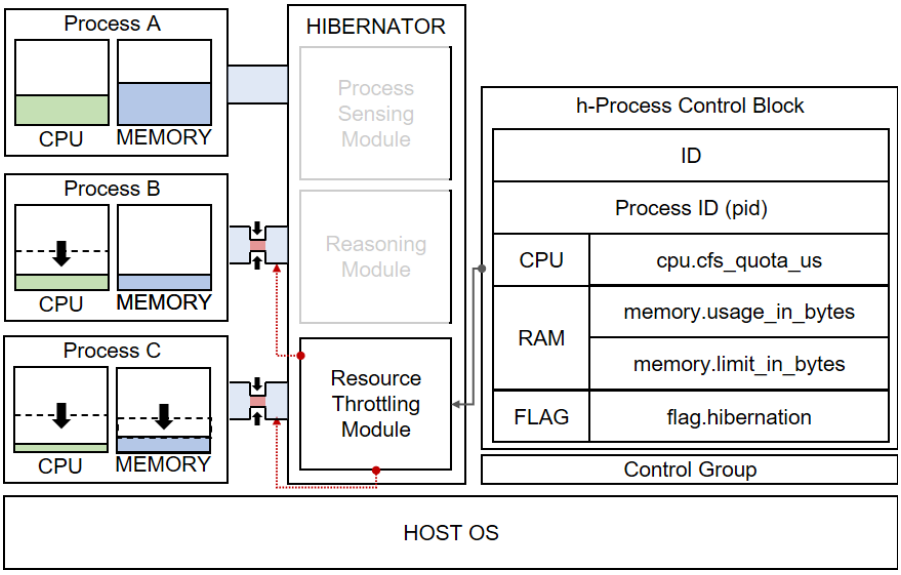

Figure 5: Resource Throttling Module.

A. When starting a new process

A new process is started by a command provided by the process hibernation system. By specifying the process start command to execute as an argument to the command to add a controlled process to the hibernation system, the hibernation system executes the process internally, returns the process pid, and registers it as a process controlled by the system.

B. When a running process is registered as a controlled process

A running process can be added to the list of processes controlled by the process hibernation system by specifying the process' pid as an argument to the command, so that the system can include the process as a process controlled by the system.

3.6 Implementation of the Process Hibernation Scheme

This section describes the implementation details of each module that make up the process hibernation scheme proposed in this study.

A. Implementation of h-Process Control Block

The process hibernation scheme defines and uses hPCB, a data structure that stores and manages the information necessary to efficiently control the processes. The hPCB consists of six fields. Table 2 lists

the data types of each field used to implement the hPCB, and each data type was expressed using Python.

Table 2: Data types for the fields of the h-Process Control Block

Field	Data type
ID	Str
Process ID	Int
cpu.cfs_quota_us	Int
memory.usage_in_bytes	Int
memory.limit_in_bytes	Int
flag.hibernation	Bool

B. Implementation of the Process Sensing Module

The process-sensing module monitors the resource usage and maximum allowed resource usage of the process specified by the user and updates them in the h-Process Control Block. Table 3 lists the data path of the process-resource-related information that the process-sensing module accesses to monitor process information. Table 3 lists the paths for which the hPCB ID is hPCB_id.

Table 3: Data path of the data monitored by the process sensing module

Process Sensing Module Monitored Information	Path
cpu.cfs_quota_us	/sys/fs/cgroup/cpu/hPCB_id/ cpu.cfs_quota_us
memory.usage_in_bytes	/sys/fs/cgroup/memory/hPCB_id/ memory.usage_in_bytes
memory.limit_in_bytes	/sys/fs/cgroup/memory/hPCB_id/ memory.limit_in_bytes

C. Implementation of the Reasoning Module

The reasoning module measures the process control score based on hPCB information to prevent the process from being forced to terminate due to a lack of resources. Based on the calculated score, the reasoning module calculates the resource limit (e.g., the memory limit) to keep the process resource usage at a low level.

Algorithm 1 Process Scoring Algorithm

Input: Usage(x),Limit(y)

function SCORE_CALCULATION(x, y)
    $score \leftarrow (x_n/y)$ + ((1 - (x_{n-1}/x_n)) * 2)
    $size \leftarrow$ abs((x_n - x_{n-1}) + ((y - x_n) $*0.5$))
    if $Score > 1.0$ **then**
        Limit Scale Up [$size$]
    end if
    if $Score < 0.5$ **then**
        Limit Scale Down [$size$]
    end if
end function
for $iteration = 1, 2, \ldots$ **do**
    $Score\_Calculation(Resource\,usage, Resource\,limit)$
end for

The score is calculated by the weighted sum of the following two factors: the ratio between the

current memory usage and the limit, and the ratio between the increased or decreased memory usage compared to the previous usage and the memory usage of the current iteration. If the calculated score exceeds 1.0, the reasoning module determines that the resource limit of the process should be increased by the amount specified in the algorithm, thereby preventing the process from accidentally terminating due to insufficient memory. If the score is less than 0.5, it is determined that the resource allocation should be reduced by the size, and unnecessary memory allocation for the process being controlled is released to minimize the resource usage of the process. The size value is determined by the absolute value of the weighted sum of the amount increased or decreased compared to the previous value and the amount of remaining resources, as shown in Algorithm 1. Our algorithm calculates the memory limit using the real-time memory usage of the process as input. In the algorithm, x represents the memory usage and y represents the currently set memory limit.

D. Implementation of the Resource Throttling Module

The resource throttling module dynamically adjusts the memory usage limit for a process based on the process control score measured by the reasoning module to minimize the memory requirements of the process while preventing the process from terminating unexpectedly due to insufficient memory.

The resource throttling module controls the memory usage of the process by assigning an appropriate resource usage limit to the control group to which the process belongs, based on the score calculated by the reasoning module. The memory limit determined by the reasoning module algorithm, , is set to /sys/fs/cgroup/memory/hPCB_id/memory. limit_in_bytes, as shown in Algorithm 1, where the hPCB ID of the process is the hPCB_id. For CPU usage, /sys/fs/cgroup/cpu/hPCB_id/cpu. cfs_quota_us is set as a user rule to maintain the relative share of CPU time available for the processes in the cgroup.

4 Experiment and Performance Evaluation

This section describes a variety of experiments conducted to verify that the proposed hibernation scheme works well in practical environments and to evaluate its performance from different perspectives.

4.1 Experiment and Performance Evaluation

The environment used for the experiments and performance evaluation of the process hibernation scheme proposed in this study. A Linux Ubuntu 18.04.5 LTS environment with an Intel (R) Core (TM) i7-8700 CPU and 8 GB of memory was used for the experiments and performance evaluation. The experiment was conducted using the Bitwarden, ASTx, and nProtect applications.

Bitwarden is a free and open-source password management service that encrypts and manages sensitive information such as website credentials. The Bitwarden platform offers a variety of client applications, including web interfaces, desktop applications, browser extensions, mobile applications, and command-line interfaces.AhnLab Safe Transaction (ASTx) is an online integrated security product optimized for a non-ActiveX-based web standard environment that provides both security and convenience. It is a solution that can avert damage that users may incur while using web services by providing a keyboard security feature to prevent leakage and fabrication of important content such as user transaction and personal information, as well as comprehensive security functions such as malware detection, network, vulnerability, phishing/pharming detection, and the latest memory hacking protection. nProtect is a security application that protects the information assets of users with built in online security services, including keyboard security, malware detection, and network protection. nProtect supports various operating systems and web browser environments.

4.2 Experiment Details

4.2.1 Experiment to Measure the Changes in Memory Usage/Limit for the Controlled Process

Because the process hibernation scheme minimizes the resources occupied by the process without terminating it, the process should be induced to reduce its resource usage while preventing the process from terminating due to insufficient memory. An experiment was performed to verify the operation of the process hibernation scheme, which prevents the process from being forcibly terminated because the memory limit for the process was set too low while gradually minimizing the resource usage of the process. The experiment was performed using the security application nProtect.

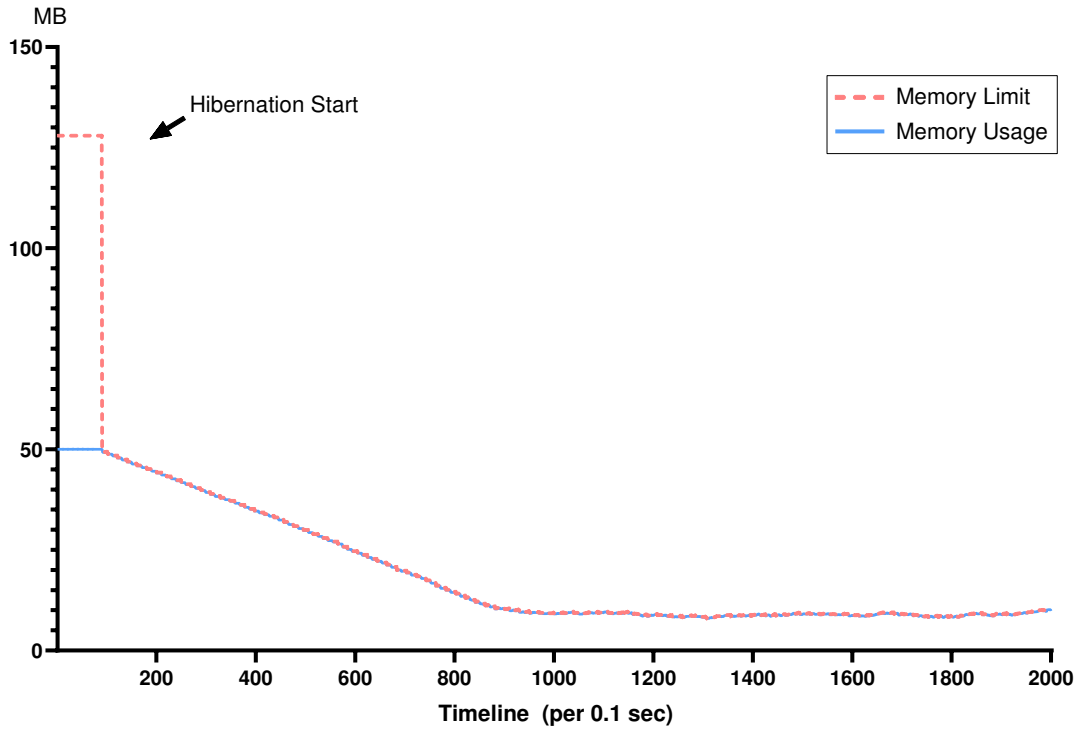

Figure 6: Changes in memory usage and memory limit when controlling a process by the process hibernation scheme.

Figure 6 shows the changes in memory usage and memory limit of nProtect, a security application, over time when the process hibernation scheme is applied to nProtect. Before applying the process hibernation scheme, nProtect occupied 50–55 MB of memory. The process hibernation scheme was applied from the point marked as ”Hibernation Start” in Figure 6, and the memory limit and memory usage of nProtect gradually decreased after the Hibernation Start. After the memory usage of a process decreased to a certain level, it did not decrease again, but increased or decreased slightly. This phenomenon can be seen in Figure 6 after the timeline from 1000. Figure 7 shows the enlarged portion of the timeline from 1400 to 1700 in Figure 6.

This phenomenon is observed in Figure 6 after the timeline from 1000. Figure 7 shows the magnified portion of the timeline from 1400 to 1700 in Figure 6. When the memory usage of the process was lowered to a certain level (i.e., hibernation state), the memory usage of the application continued to increase and slightly decrease, as shown in Figure 7. Thus, the process hibernation scheme adaptively follows the ups and downs of memory usage by adjusting the memory limit to save memory while

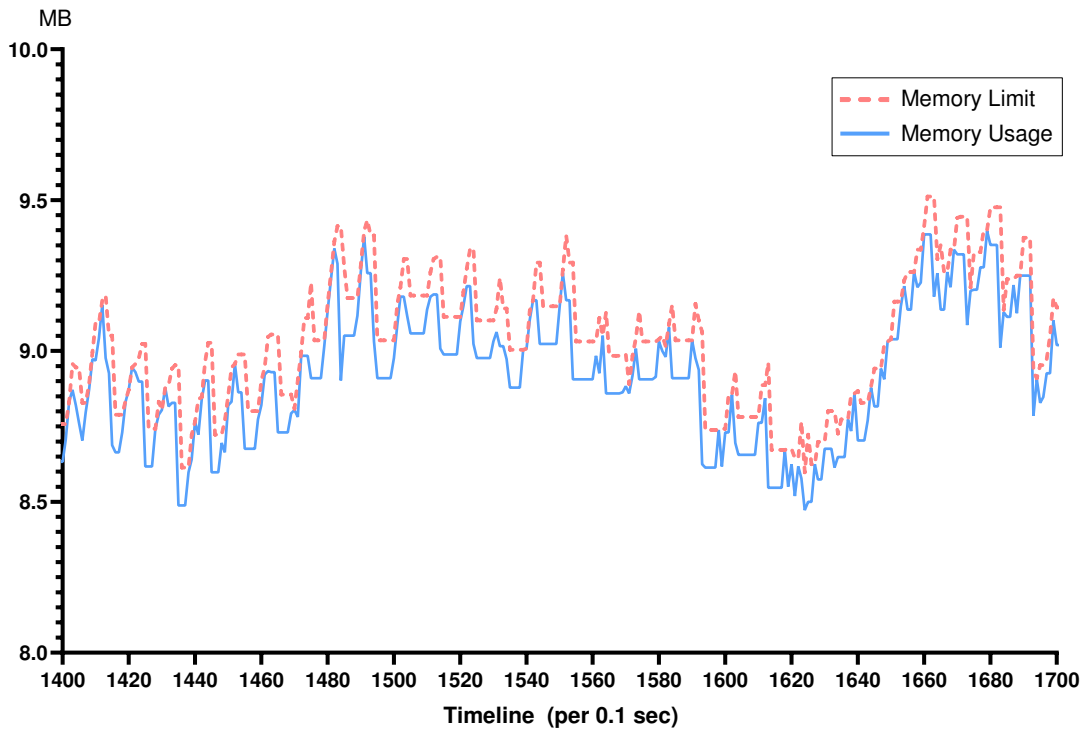

Figure 7: Memory limit/usage while the process hibernation scheme keeps process memory usage low.

preventing the process from terminating due to lack of memory. This adjustment is based on the scores calculated in the reasoning module for the controlled process. When the memory usage of the process increases, the memory limit also increases. If the memory usage of the process decreases, the memory limit for the process is also reduced.

4.2.2 Experimenting with Resource Orchestration based on Security Application Execution Scenarios

We demonstrated security application orchestration by properly controlling security applications in situations where certain scenarios in computer systems require the operation of random security applications via a process hibernation scheme.

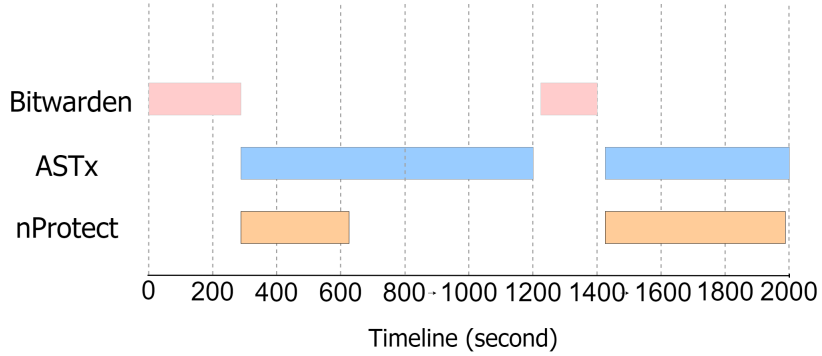

Figure 8: Security application operation scenario.

Figure 8 illustrates the security application operation scenario. An experiment was conducted to verify whether memory was actively used when the security application was required to operate according to the schedule specified in the scenario and whether the memory usage was kept low by the process hibernation scheme when the security application was not in use according to the schedule. In the experiment, the memory usage of each process that was controlled according to the schedule was measured in units of 1 s.

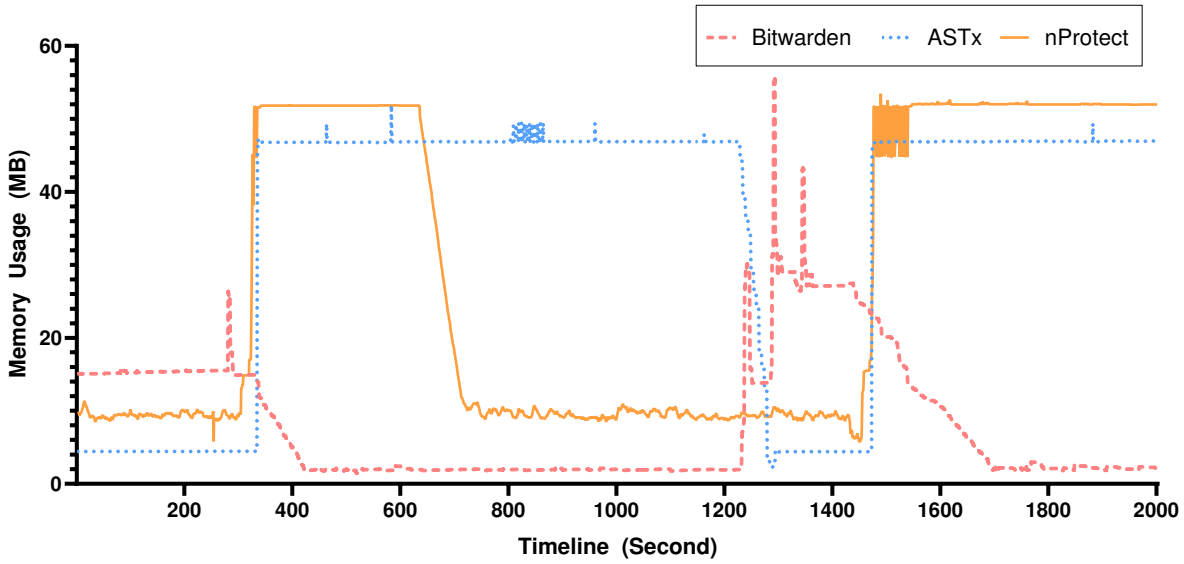

Figure 9: Changes in memory usage of security processes according to scenarios.

Figure 9 shows the change in memory usage of security applications depending on the operating scenarios of the safety applications. From the results of the experiment, it can be seen that the resource usage of each process was controlled appropriately according to the operating scenario of the security application.

5 Conclusion

In this study, we propose a process hibernation scheme that minimizes the resource usage of a security process by dynamically adjusting the resource limits of the process. The proposed scheme presents a consistent control method for a variety of self-protection security application mechanisms to overcome the difficulty of designing a control method for each security process self-protection mechanism. To design and implement the process hibernation scheme, we defined the hPCB data structure and three modules: process sensing, reasoning, and resource throttling. The process-sensing module detects the resource status and updates the process information in the hPCB. The reasoning module retrieves the resource status information from the hPCB and calculates a score for process control based on this information. The resource throttling module reduces resource usage by isolating the controlled process into a control group and setting an appropriate computational resource limit for the control group, based on the score obtained from the reasoning module. It has been proved through experiments that the process hibernation scheme can achieve the above goals.

Acknowledgments

This work was supported by the Institute of Information & Communications Technology Planning& Evaluation (IITP) , South Korea(Project No. RS-2022-00165794, Development of a Multi-Faceted Collection-Analysis-Response Platform for Proactive Response to Ransomware Incidents,40%, and Project No. 2022-0-007010001003, 30%), and a National Research Foundation of Korea (NRF), South Korea grant funded by the Korean government (Project No. NRF-2020R1A2C4002737, 20%), and the ICT R&D Program of MSIT/IITP, South Korea (Project No. 2021-0-01816, A Research on Core Technology of Autonomous Twins for Metaverse, South Korea, 10%).

References

[1] Wei Yan and Nirwan Ansari. Why anti-virus products slow down your machine? In *2009 Proceedings of 18th International Conference on Computer Communications and Networks*, pages 1–6. IEEE, 2009.

[2] Vincent W Freeh, Xiaosong Ma, Sudharshan S Vazhkudai, and Jonathan W Strickland. Controlling impact while aggressively scavenging idle resources. Technical report, North Carolina State University. Dept. of Computer Science, 2006.

[3] Zhongshu Gu, Zhui Deng, Dongyan Xu, and Xuxian Jiang. Process implanting: A new active introspection framework for virtualization. In *2011 IEEE 30th International Symposium on Reliable Distributed Systems*, pages 147–156. IEEE, 2011.

[4] Jungho Kim, Philkyue Shin, Soonhyun Noh, Daesik Ham, and Seongsoo Hong. Reducing memory interference latency of safety-critical applications via memory request throttling and linux cgroup. In *2018 31st IEEE International System-on-Chip Conference (SOCC)*, pages 215–220. IEEE, 2018.

[5] Sai Prashanth Muralidhara, Lavanya Subramanian, Onur Mutlu, Mahmut Kandemir, and Thomas Moscibroda. Reducing memory interference in multicore systems via application-aware memory channel partitioning. In *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 374–385. IEEE, 2011.

[6] Heechul Yun, Waqar Ali, Santosh Gondi, and Siddhartha Biswas. Bwlock: A dynamic memory access control framework for soft real-time applications on multicore platforms. *IEEE Transactions on Computers*, 66(7):1247–1252, 2016.

[7] Ravi Iyer, Li Zhao, Fei Guo, Ramesh Illikkal, Srihari Makineni, Don Newell, Yan Solihin, Lisa Hsu, and Steve Reinhardt. Qos policies and architecture for cache/memory in cmp platforms. *ACM SIGMETRICS Performance Evaluation Review*, 35(1):25–36, 2007.

[8] Rami Rosen. Resource management: Linux kernel namespaces and cgroups. *Haifux, May*, 186:70, 2013.