공학석사 학위논문

Memory Allocation for Page Level Isolation in ROS Composition

ROS Composition에서 페이지 수준 격리를 위한

메모리 할당 방법

2023년 2월

서울대학교 대학원

전기 · 정보공학부

이 경 룡

Memory Allocation for Page Level Isolation in ROS Composition

지도 교수 백 윤 흥

이 논문을 공학석사 학위논문으로 제출함
2023년 2월

서울대학교 대학원
전기·정보공학부
이 경 룡

이경룡의 공학석사 학위논문을 인준함
2023년 2월

위 원 장 문 수 묵 (인)

부위원장 백 윤 흥 (인)

위 원 이 병 영 (인)

초 록

Robot Operating System (ROS)는 로봇 애플리케이션을 만드는데 널리 쓰이는 미들웨어이다. ROS composition 방법이 등장하면서, 하나의 프로세스 내에 여러 개의 노드를 같이 실행하는 것이 가능해졌다. 그러나, 하나의 프로세스 내에 여러 개의 노드가 존재할 때는 모든 노드가 하나의 메모리 공간을 사용하므로, 각 노드에 동적으로 할당된 메모리 개체들이 페이지의 경계에 상관없이 무질서하게 배열된다. 본 논문에서는 ROS composition 프로세스에서 페이지 수준 격리를 적용하기 위해서, 각 노드에 동적으로 메모리를 할당할 때 서로 다른 영역에 메모리를 할당하도록 하는 메모리 할당 방법을 제안한다. 본 논문에서 제안하는 방법은 기존에 컴파일 된 노드 바이너리에 대해서도 적용이 가능하다. 해당 방법에서는 메모리 할당 및 해제 요청을 특수하게 설계된 메모리 할당자로 보내어 처리한다. 이 때, 어느 영역에서 해당 요청을 처리해야 하는지 결정하기 위해 필요한 정보를 추적하여 확보하며, 이를 이용하여 각 요청을 적절하게 처리한다.

주요어 : 메모리 할당, 페이지 수준 격리, ROS composition
학 번 : 2021-25362

목 차

그림 목차

제 1 장 Introduction

Recently, robots have been increasingly used for reasons such as cost reduction, and the development of artificial intelligence has made it possible to implement complex movements that were previously impossible. Various software components and libraries are provided in order to implement such complex behaviors of robots. One of the most widely used middleware of this kind is the Robot Operating System (ROS) [1].

Formerly, the application components of ROS, called nodes, were executed in a single process per each node. After some time, ROS composition methods [2] have emerged, allowing the composition of multiple nodes executed together in a single process. However, when multiple nodes are composed into a single process, multiple independent nodes share the same memory space, and therefore the attack surface becomes wider, allowing attackers to compromise several nodes by detouring through only a subset of nodes in the process. Furthermore, the C++ language, which is a well-known memory unsafe language, is currently one of the officially supported languages, which may allow the attacker to read contents of arbitrary memory regions through memory bugs such as buffer overflow.

In order to address this issue, one may try to apply memory isolation on a per-node basis. There are currently many ways to apply memory isolation. One of the simplest techniques is SoftBound [3], which verifies each memory access using base and bound information for each pointer, but this technique has a relatively high cost of 67(%) on average, and requires compiler assisted instrumentation of each memory access. In order to reduce the cost of isolation, a special hardware mechanism such as Intel MPK may be utilized. A technique that utilizes such a feature is ERIM [4], which incurs a cost of up to 4.3(%) on average on typical applications, which is quite lower than SoftBound. In order to apply memory isolation with moderate overhead using Intel MPK, and also avoid compiler assisted instrumentation on each memory access, it would be very

beneficial to apply page level isolation.

Currently, however, since the nodes share the same heap, dynamically allocated memory chunks of different nodes are intermixed, regardless of the page boundary. This paper introduces a memory allocation method to dynamically allocate memory chunks on separate regions for each node, even on precompiled node libraries. Upon the assumption that ROS nodes are provided as a shared library binary, the key challenge is to devise a way to adequately handle memory requests that are issued from the node. This issue is tackled by altering the dynamic linking process so that the memory requests are passed to a specially designed allocator that handles the memory requests considering which node the request came from. Here, context tracing and memory space tracing techniques are leveraged in order to pinpoint which node issued the memory request. Through this study, it is expected that various efficient region-based isolation techniques can be seamlessly applied in ROS composition.

This paper is organized as follows. In section 2, background knowledge regarding what is memory isolation, and how ROS process is organized is described. In section 3, design and implementation of the allocator and memory request handling mechanism is described in detail. In section 4, the experiment settings are described, and execution overhead is quantified. Lastly, the conclusion of this paper is provided in section 5.

제 2 장 Background

제 1 절 Robot Operating System (ROS)

The Robot Operating System (ROS) [1] is a set of libraries and tools for developing a robot application software. It is an open source software that is provided by Open Robotics, a nonprofit public benefit corporation. ROS is known to be currently used by various organizations and vendors.

ROS 2 is the latest version of the Robot Operating System. In ROS 2, a node is a unit component of an application that is executed in ROS. ROS also provides a library that provides various functionalities such as inter-node communication. At the core of this functionality is the ROS client library (rcl) [5] that is written in C. Based on this library, the rclcpp library [6] is provided to support C++ language, which is an official language of ROS.

For inter-node communication, ROS provides various communication models. A publish-subscribe model is supported as a functionality called a topic. In this model, a node can subscribe to a topic by its name and wait for a message to be published. After that, another node can publish a message to this topic, so that all of the subscribers to that topic can receive the published message. Another model that is supported is the server-client model, as a functionality called a service. In this model, a node can open a service and wait for any other node to send a request to that service. Upon a request to that service from another node, the node that opened the service executes a callback function to handle the request. Finally, there is a functionality called an action, that is based on a communication model that is specially designed for usage in communications between robots.

제 2 절 Structure of ROS Process

The process that the ROS application is executed in consists of several components. Figure 1 shows the typical structure of ROS applications. In ROS 2, nodes are typically executed based on an executor that is provided as part of the ROS library. In order to allow several nodes to be executed together on a single process, the executor executes predefined functions of each node when needed. ROS assumes that the functions are implemented in a non-blocking fashion, and based on that assumption, the executor schedules the execution of functions of nodes, and the procedures are executed one by one, possibly on a single thread. The users may compose the provided nodes by compiling and executing their own execution base that calls the initialization routines and creates and registers the nodes that the user wants to execute. When the node is executed, it may use several library functions that are provided by the C/C++ standard and the functions that are provided by the ROS library.

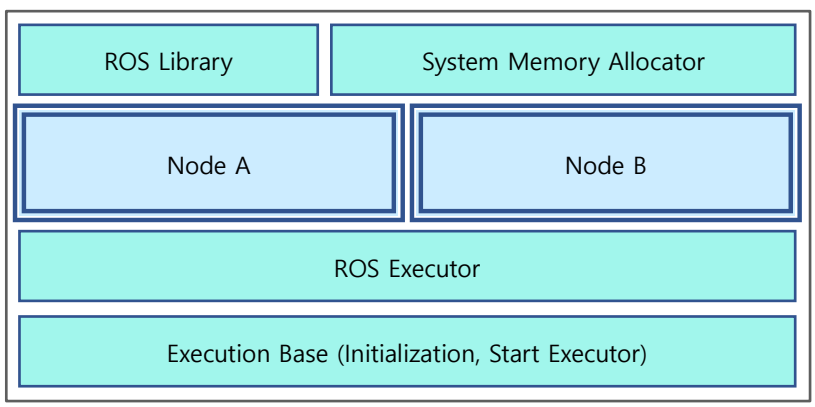

Figure 1 ROS Process Structure

Upon creation of the ROS process, the execution base will specify in advance or dynamically load needed node binaries. When the node binaries are loaded to the process, the dependent libraries such as the ROS library and the system library (e.g. glibc [7]) are also loaded and needed functions are dynamically linked to the node binaries. The loaded and dynamically linked binaries are dynamic libraries that may be patched and recompiled without any information

of the binaries that use it. It is this point that will make possible the technique specified in this paper.

제 3 절 Page Level Isolation

Isolation is a technique to control the memory access so that a specific component can access only a predefined subset of the memory space and any access to other memory regions are detected or blocked.

Isolation can be applied in various ways. One possible way is SoftBound [3] that verifies the accessed pointers of each memory access based on the base and bound of the allowed region. However, in order to use this technique, the isolated component must be instrumented to properly check the pointers on each memory access. Also, since each memory access instruction is instrumented, the cost is relatively high. In order to reduce the overhead of isolation, page level isolation techniques exist that utilize special hardware features such as Intel MPK or ARM Domains to modify memory page permissions with low overhead and check each memory access based on the modified page permissions. One technique that utilizes Intel MPK is ERIM [4], which incurs a cost of up to 4.3(%) on average on typical applications, which is quite low. However, since the memory permissions can be granted on a page-per-page basis, special care might be required to seamlessly apply isolation on a page granularity.

제 4 절 Memory Allocator

Typically, a programmer can allocate a local variable in the stack or a global variable that is shared throughout the whole execution time. However, there are cases where the programmer cannot know the needed size of memory space in advance or needs a temporary buffer that will be used by several functions in a short time window. In this case, the programmer may request a dynamic memory

allocation to the memory allocator so that a memory space of the specified size is provided in a dynamically resizable memory space, called a heap.

Basically, memory allocators receive memory allocation requests to provide a memory chunk of a specific size, and memory deallocation requests that return formerly allocated memory chunks by the same memory allocator. Typical memory allocators handle requests of small sizes by dividing a piece of memory chunk from a big memory region acquired by the memory allocator in advance. Therefore, one may not assume that the memory chunks are organized in a certain way, but rather are scattered around the whole memory space, regardless of the page boundary. There are several memory allocator implementations in order to address various specific challenges for memory allocation, but typical memory allocators acquire memory regions on a granularity of a multiple of pages, aware of the common MMU hardware structure.

제 3 장 Design and Implementation

As described so far, in ROS, several nodes can be loaded into a single process and be executed together, based on a ROS executor. These nodes share the same memory space, and therefore share the same heap. On the left of Figure 2 is an example of a possible memory allocation status using the current implementation. Allocated memory chunks are handled from the same heap, by the same allocator in the standard C library. As a result, allocated memory chunks are interleaved without any order, regardless of which node requested the memory chunk. However, in order to seamlessly apply page level isolation without modification of node binaries, memory requests from each node should be handled on the caller's own heap, or at least the memory chunks from other nodes should be separated by page boundaries, as in the right of Figure 2.

To do this, a special memory allocation unit is inserted as part of the ROS library, and existing ROS library functions, including the executor, will be modified to properly use the new memory allocation management unit, as in Figure 3. Then, using this modified library, the user may specify in the execution base so that this new functionality will be activated properly. The memory allocation unit consists of 4 parts. First, by using the malloc/free hooking mechanism to alter the dynamic linking process, the memory requests are redirected to a trampoline so that the newly inserted handler is called. Second, the (de)allocation routine will be called by the trampoline with the requested size and which heap to use, and this routine is where the actual memory allocation happens. Third, in order to choose the right heap to handle the memory request, the execution context tracer constantly keeps tracking which node is currently being executed for this reason. Lastly, the memory tracer is used to track the address and owner of each heap, in order to recognize the owner of the memory chunk using only its address, so that on a memory deallocation request, the memory chunk will be returned to its exact owner.

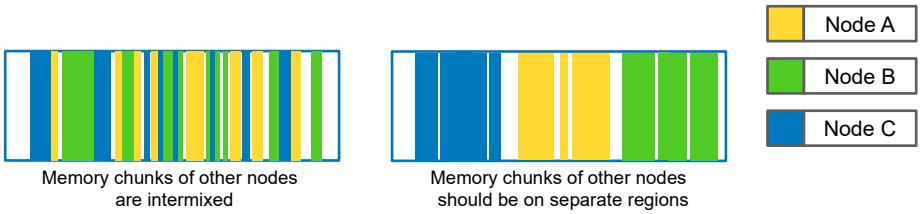

Figure 2 Memory allocation on each node should be handled on a separate region

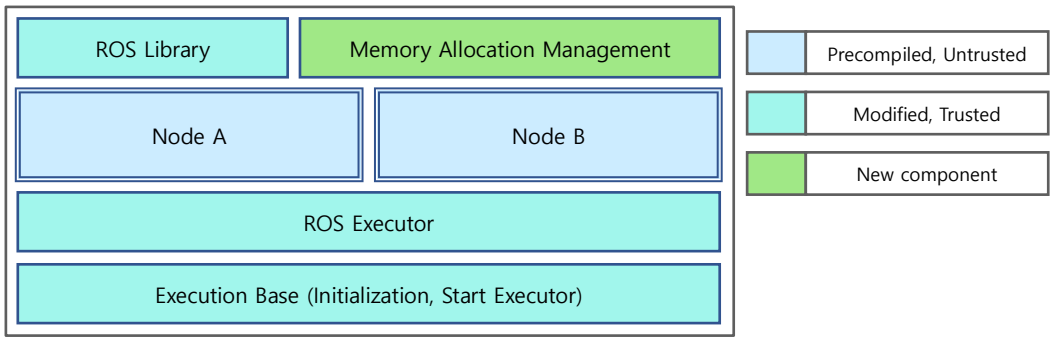

Figure 3 Modified structure of ROS process to apply page level isolation

제 1 절 Memory Request Trampoline

Most of the binaries compiled from C/C++, including ROS node libraries, assume that the system allocation library that abides by the C/C++ standard is provided on the executed system. Therefore, these binaries use the memory allocator provided by the language standard library. The memory request trampolines, specifically the memory allocation routine and the memory deallocation routine, are defined in a new ROS library, with the same name and parameters as the original memory request routines. Here, the memory allocation trampoline only takes the allocation size as the parameter, and the memory deallocation routine only takes the returned memory chunk pointer as the parameter. However, the memory requests should be handled separately, depending on which heap to handle the request. The essence of the rest of the components that are described in later sections is to provide a way to handle the memory requests with the provided parameter which would be insufficient to completely handle memory requests if additional information were not available.

The memory allocation trampoline receives only the size of the requested memory chunk, so it should choose the right heap to handle the memory allocation request. Figure 4 illustrates the behavior of the memory allocation trampoline. To do so, it first looks up the current execution context in the Context Tracer component, which is described in section 3-3. The Context Tracer tracks and stores the current execution context, specifically whether which node was being executed, or if the library function was executed, just before the memory request was issued. The execution context includes a memory allocator object that is described in section 3-2. The trampoline then passes the memory allocation request to the memory allocator object that is included in the execution context, and the memory allocator routine handles the memory request using the metadata stored in the passed memory allocator object.

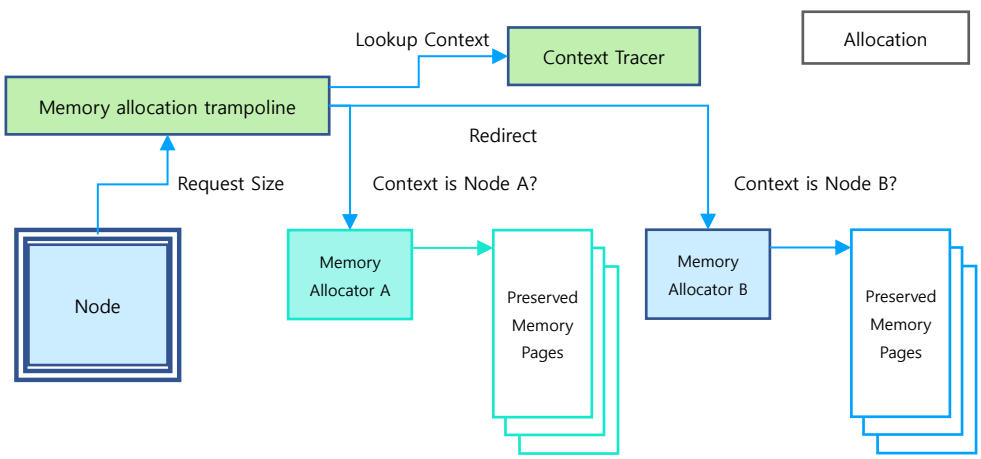

Figure 4 Memory allocation trampoline behavior

The memory deallocation trampoline behaves very similarly to the memory allocation trampoline in that it should also choose the right heap to handle the memory deallocation request, except that it only receives the pointer of the returned memory chunk. Figure 5 illustrates the behavior of the memory deallocation trampoline. It first looks up in the Memory Tracer which heap this returned memory chunk is part of. The Memory Tracer tracks the location and owner of each memory region that was fetched by each memory allocator.

Using this information, the memory deallocation trampoline can pass this returned memory chunk to the memory allocator that exactly owns this memory chunk.

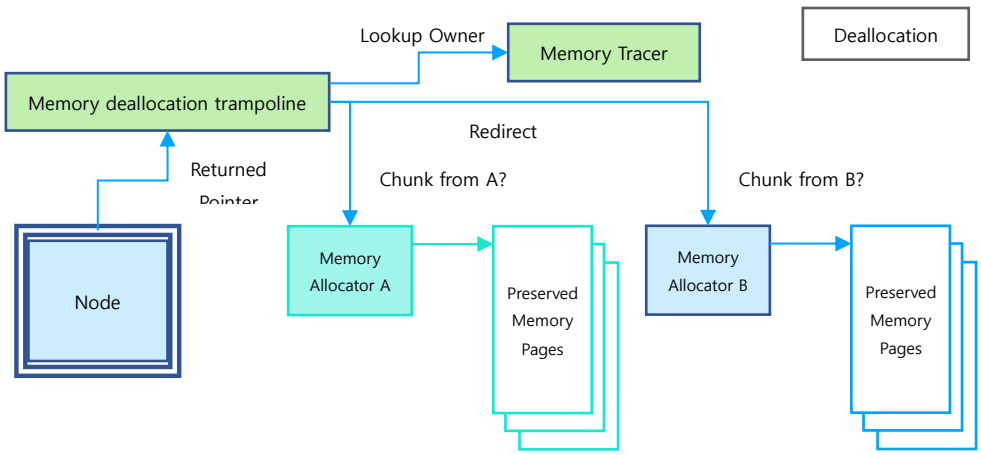

Figure 5 Memory deallocation trampoline behavior

제 2 절 Porting a Memory Allocator

Common memory allocation routines fetch memory regions in advance, so that memory requests of small sizes below a certain threshold could be handled by allocating a small chunk from this region. For large requests, they are handled by directly fetching another memory region of the adequate size. The specific implementation may vary from each memory allocator, but since most of the CPU architectures use a memory management unit on a granularity of pages, most memory allocator implementations also fetch memory regions on a granularity of pages. The metadata about the memory regions fetched from the OS is typically maintained as a global data inside the memory allocator, since most of the memory allocator implementations assume that the whole process will use this memory allocator. Therefore, by taking the approach of encapsulating the metadata as an object and modifying the routines to use the metadata on the object, it is possible to duplicate several memory allocator instances on a single memory allocator routine and make the instances operate independently, since each instance does not

even know the existence of other instances. Here, it is the responsibility of the operating system to ensure that the fetched memory regions do not overlap. Another modification to mention in advance is that it is required by the memory tracer for the memory allocator to hand information about the memory regions fetched from and returned to the OS.

제 3 절 Execution Context Tracer

The Execution Context Tracer, or shortly Context Tracer, stores information about which context, or which node, is currently being executed on this thread. The execution context is stored by each thread on a thread local storage, so multiple threads can be traced without any synchronization issues. The Context Tracer leverages the fact that the behavior of node execution resembles the behavior of process execution, in the point that the entrance to and exit from the executed component is only connected to certain management units, which are the ROS library components on the left and the OS kernel on the right in Figure 6. Here, we assume that calling library functions other than the ones that appear in Figure 6 is part of the node execution context, because the mechanisms of heap memory management proposed in this paper should also be analogously applied in these cases. Therefore, patching the possible entries to the node functions from the ROS executor and the possible exits to the ROS library to inform the Execution Context Tracer about context switches allows to up to date tracking of the current execution context, whether the ROS library function is being executed, or whose node function is being executed, so that the memory request trampoline is able to look up the execution context whenever the memory allocation routine is called.

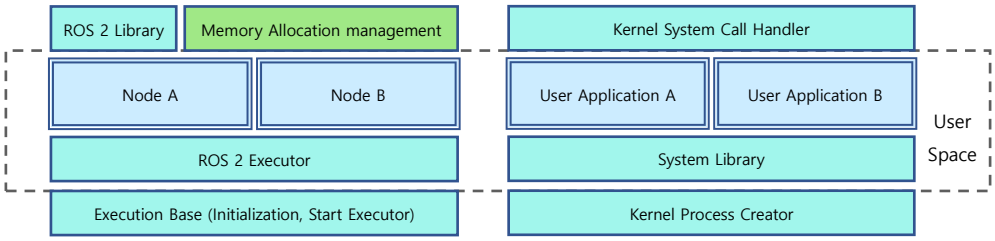

Figure 6 The behavior of node execution resembles process execution.

제 4 절 Memory Tracer

The Memory Tracer stores information about which heap is owned by which memory allocator object. Figure 7 illustrates which information the Memory Tracer stores. The Memory Tracer stores the start and end address of each heap, and which context owns the heap. In order to maintain this information, the memory allocator is patched to inform the Memory Tracer on every acquisition from or release to the OS, the start and end address of the memory region that the memory allocator acquired or released, along with which memory allocator object it is operating on. Then, the Memory Tracer is available to maintain up-to-date information about which interval of memory space belongs to which memory allocator object, as in the table of Figure 7. Here, the Memory Tracer can maintain this interval data efficiently by using a balanced binary tree data structure. However, this information is stored as global data, so it does require some synchronization. Furthermore, since modern memory allocator implementations are quite efficient, traversing a balanced binary tree would impose a significant performance overhead. Therefore, a shadow memory of the acquired memory space is also maintained to support an efficient, lock free lookup. The shadow memory stores a byte per each page. The byte stored is an identifier, or specifically an index to a table that stores a pointer to each registered context. Here, the shadow memory overhead is one byte per page size, but this scheme should also work even if it uses more bytes per page size, if the user decides to bear the memory overhead. Since the

shadow memory is updated on memory acquisition or removal, it should not be modified when any memory chunk is allocated on the page. Therefore, only a simple synchronization on these page updates is enough for this scheme to work correctly, and since lookups only need to read the shadow memory, they can be done in a lock free manner.

An issue that occurs when using this shadow memory scheme is that the addresses of memory pages other than the ones managed by the registered memory allocators are unknown, and therefore the shadow pages of these unknown memory pages cannot be mapped in advance. Therefore, assuming that any address of the shadow region can be accessed, this issue is handled by installing a handler of an unmapped memory access exception, so that if the shadow memory region is accessed, the region is mapped with a page filled with null bytes.

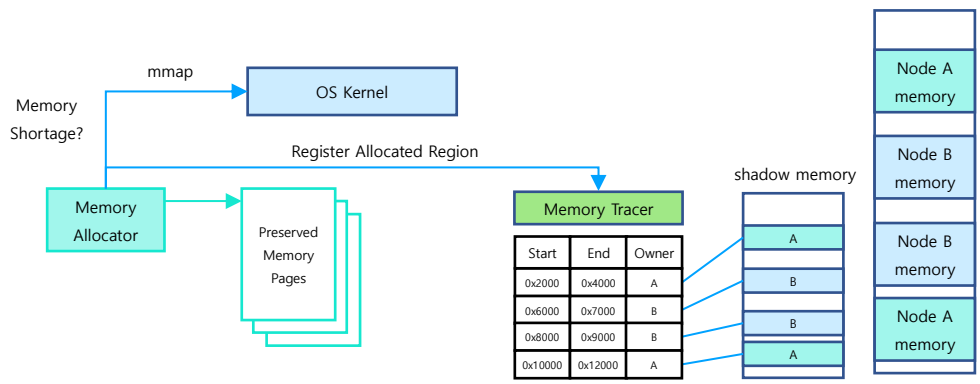

Figure 7 Memory Tracer stores data about the location and owner of each heap

제 4 장 Evaluation

In this section, the settings and results of several experiments regarding this allocator are provided. The experiment was done on an Intel i9-10900K CPU with an Ubuntu Linux 18.04 operating system. ROS dashing rcl library [5] and rclcpp library [6] was modified and fully compiled with the source code on the machine. Also, the memory allocator from glibc version 2.27 [8] was ported, which is the same implementation as the default system allocator of the operating system, for a reasonable comparison. Unfortunately, the thread caching technique was disabled since the original implementation used a thread local variable whose location and size were fixed at compile time, but this is not trivially available because the total number of memory allocators may vary depending on the node composition setting. In future works, it might be viable to apply a variably sizable thread caches to support a variable number of memory allocators.

제 1 절 Proof of Concept Experiment

In order to show that this also works on precompiled nodes, a composition experiment regarding four components is done. Figure 8 illustrates the components of this experiment. The executed nodes are leaker, and peeker. The leaker component puts an address of a dynamically allocated memory chunk on a global data space, to simulate a situation where an address to data of some node is accidentally leaked. The peeker component is a corresponding component to the leaker component, which tries to collect leaked information by acquiring this address and reading the data using this leaked address.

In order to show that it is possible to apply page level isolation using this technique, a simple page level isolation using Intel MPK is applied. This point is illustrated in Figure 8 by the double line borders

wrapping each component on the right. In this case, all of the allocator objects are assigned a dedicated pkey, which is a tag related to Intel MPK. Then, Intel MPK can be used to easily mask read or write permission of memory pages of a selected pkey. This way, page level isolation can be simply done by masking memory permissions of data from all components other than the component that is being executed.

The experiment is done as follows. First, in order to illustrate that this works on precompiled nodes, the four nodes to be used are compiled inside a docker container, where the ROS environment is identical to the host, and no modifications were made to the ROS library. Then, the node binaries in shared library format are collected to the host machine. On the other hand, the modified ROS libraries are compiled on the host environment. Lastly, the execution base is crafted to load the collected node binaries along with the modified ROS library. As a result, the nodes are successfully executed, and the abnormal memory access by the peeker component is successfully detected and a warning is issued. This illustrates that memory access to data of other components violates the effective memory permission when the memory access is done, and therefore page level isolation of dynamically allocated data is successfully enforced.

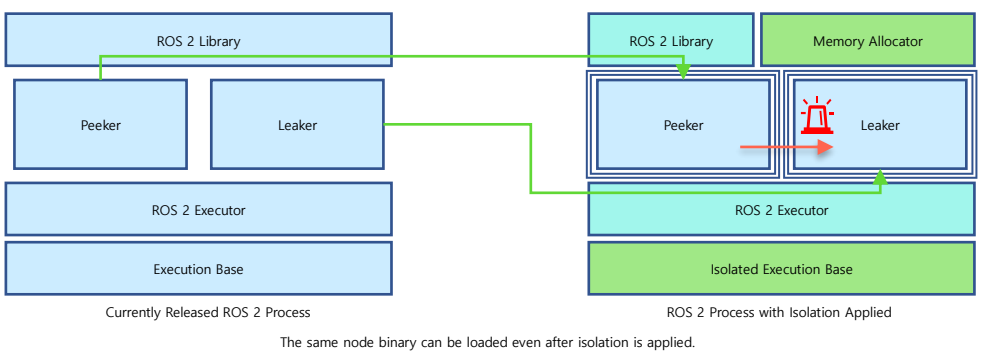

Figure 8 The setting of Proof of Concept Experiment

제 2 절 Performance Evaluation

To measure the performance benefit of using the proposed method with ROS composition, the execution time of a simple composed process with 1 publisher and 1 subscriber with various publishing periods is measured using the linux perf utility [9]. In this setting, a publisher publishes a simple message of 16 (bytes) periodically, and a subscriber receives the message. The period of publishing messages used in the experiment varies from 0.25 (ms) up to 5 (ms). This time length was chosen because a former study [10] has shown that the message transmission latency is about ~ 1 (ms). Thus, for a period as long as 5 (ms), the nodes wait for sending and receiving a message for a major portion of time. On the other hand, for a period as short as 0.25 (ms), the nodes constantly keep sending and receiving messages, which indicates a highly loaded situation. The execution time of two settings is measured. First, the execution time of a process comprising one publisher and one subscriber is measured, when a simple page level isolation with the proposed memory allocation method is applied. Also, to compare the former setting with an equivalent in terms of isolation, the execution time of two processes, each containing one publisher and one subscriber respectively, is measured. In terms of isolation, this setting is equivalent to the composed process with page level isolation, since the memory space of other processes are naturally separated, and therefore each node in their own process cannot access each other.

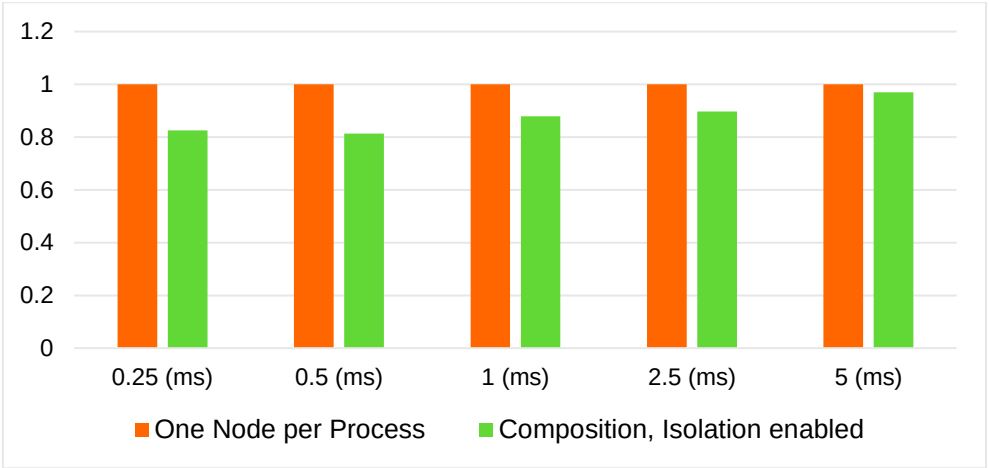

Figure 9 Execution time of one publisher and one subscriber on different settings

Figure 9 illustrates the results. The results show that execution time is reduced by from 3.0 (%) up to 17.6 (%) when comparing the proposed design with isolation enabled, to when each node is executed on separate processes, which is an equivalent in terms of isolating dynamically allocated memory. Specifically, when the publish period is as long as 5 (ms), which indicates a low workload circumstance, the execution time is reduced by 3.0 (%). On the other hand, as the publish period decreases, the execution time gap becomes bigger, up to 17.6 (%) when the publish period is as short as 0.25 (ms). This indicates that there is definitely a benefit in applying page level isolation on a composed process, rather than just executing each node on separate processes, especially when the nodes are handling heavy workloads.

제 5 장 Conclusion

In ROS composition, currently all nodes in the process use the same system allocator, and therefore allocated memory chunks are scattered regardless of which node uses it. In order to apply page level isolation, a specially designed memory allocation unit is introduced. Using this memory allocation unit, memory allocation can be controlled so that each page only contains memory chunks used by only a single node, and therefore it becomes possible to seamlessly apply page level isolation. In this design, memory requests can be redirected to an allocation/deallocation trampoline. Even if only the request size or returned address is passed as the parameter, the memory requests can be handled properly. Here, this memory allocation unit fills in the gaps of insufficient information by properly tracing the required data. Experiment result shows that the proposed design allows to seamlessly apply page level isolation on dynamically allocated memory regions of a composed process running several nodes together, and therefore allows a benefit of up to 17.6 (%) execution time reduction compared to naively running the same nodes on separate processes, which is an equivalent in terms of isolating dynamically allocated memory.

Reference

[1] Open Robotics. 2022. ROS. https://www.ros.org/.
[2] Open Robotics. 2022. ROS Composition.
https://docs.ros.org/en/dashing/Tutorials/Composition.html.
[3] Nagarakatte, Santosh, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. 2009. SoftBound: Highly compatible and complete spatial memory safety for C. In Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation. 245–258.
[4] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. 2019. {ERIM}: Secure, Efficient In-process Isolation with Protection Keys ({{{{{MPK}}}}}). In 28th USENIX Security Symposium (USENIX Security 19). 1221–1238.
[5] Open Robotics. 2022. ROS 2 rcl library.
https://github.com/ros2/rcl.
[6] Open Robotics. 2022. ROS 2 rclcpp library.
https://github.com/ros2/rclcpp.
[7] GNU. 2022. The GNU C library.
https://www.gnu.org/software/libc/.
[8] Andreas K. Hüttel. 2022. Glibc Wiki, Release/2.27.
https://sourceware.org/glibc/wiki/Release/2.27.
[9] Linux. 2022. Perf Wiki.
https://perf.wiki.kernel.org/index.php/Main_Page.
[10] Sugata Y., Ohkawa T., Ootsu K., Yokota T. 2017. Acceleration of publish/subscribe messaging in ROS-compliant FPGA component. In Proceedings of the 8th International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies. 1-6.

Abstract

Memory Allocation for Page level Isolation in ROS Composition

Kyeong Ryong Lee
Electrical and Computer Engineering
The Graduate School
Seoul National University

One of the most widely used middleware in robot application development is the Robot Operating System (ROS). As ROS composition methods have emerged, it has become possible to execute multiple nodes concurrently in a single process. However, when multiple nodes are composed into a single process, multiple independent nodes share the same memory space, and therefore dynamically allocated memory chunks of different nodes are intermixed, regardless of the page boundary. In order to apply page level isolation, this paper introduces a memory allocation method to dynamically allocate memory chunks on separate regions for each node, even on precompiled node binaries. In this method, the memory requests are redirected to a specially designed allocator, and upon a memory request, on which region the memory request must be handled is determined using additional information obtained through tracing the required information.

Keywords : Memory Allocation, Page Level Isolation, ROS Composition, Trampoline, Tracing
Student Number : 2021-25362