December 19, 2018

Fine-grained memory protection

Jose German Rivera

Fine-grained memory protection

ABSTRACT

Aside from malicious software gaining access and corrupting sections of computer memory, even legitimate software can inadvertently overwrite and damage its own memory content. Memory protection units exist that enable programmers to cordon off sensitive sections of memory. However, there is currently no mechanism to protect sensitive memory from code running in the same address space as that of the memory. Further, software that runs in privileged mode can change protection status of memory regions by modifying registers of the memory protection unit itself.

This disclosure provides a mechanism for protected memory to be accessible by code segments that are specifically authorized to do so. Per the techniques described herein, a protected memory region includes addresses of protected memory and addresses of code-space that is authorized to access such protected memory. An instruction that requests access to protected memory is tested for authorization. If such authorization succeeds, execution of the instruction continues uninterrupted, else an exception is thrown.

KEYWORDS

- Memory protection
- Memory management
- Fine-grained
- Single-address space
- MPU
- MMU

BACKGROUND

There is generally some chance that software can potentially corrupt contents of its own writable memory. This is especially true for single-address space applications written in languages such as C/C++, which are by design close to machine-level language for purposes of efficiency, and which have no built-in overwrite protections. Examples of single-address space applications include OS kernels, bare-metal embedded software running a microcontroller, etc.

For software that runs under the protection of a memory management unit (MMU), e.g., user applications, OS kernels, etc., code areas are protected from corruption, as they typically only have read permission. However, writable data sections and memory-mapped I/O areas can be modified unintentionally. For bare-metal embedded software running a microcontroller with no MMU support, even code areas are not free of the possibility of corruption. Although a memory protection unit (MPU) can be used to reduce the chances of accidental corruption, software running in privileged mode can inadvertently or deliberately corrupt registers of the MPU itself, resulting in disablement of the MPU.

To the extent that an MMU provides memory protection by partitioning memory on a per-application basis, it does so at the page level, typically 4 kilobytes or larger. An MPU provides memory protection at the region level, typically 32 bytes or larger. Although an MMU is typically used to provide data protection/isolation between multiple virtual address spaces, it can in theory be used to provide data protection within the same address space. For example, by dynamically mapping/unmapping pages of virtual memory, data structures or I/O registers encapsulated by a given code module can be made visible upon entry to the public routines of the module and hidden upon exit from the public routines of the module.

This approach requires protected data structures (or I/O registers) to be in their own

memory pages, as the MMU only provides page-level granularity of protection. A similar approach can be implemented using an MPU. However, existing MMUs/MPUs do not support restricting access to a given memory address range from code modules running in the same address space.

It is thus observed that once access is granted to a given address range, any part of the code running in the same address space can access that address range, thereby opening a security loophole or possibility of malfunction. The problem is therefore one of providing fine-grained data protection or isolation from code within the same address space.

DESCRIPTION

This disclosure provides techniques that restrict access to a given address range to specifically authorized code modules or segments. Per the techniques, a given code module with encapsulated internal data or I/O registers is authorized to access such data or I/O registers. Access from other code triggers an access violation exception. Similarly, invocation of safety-critical or security-sensitive code is restricted to an expected set of callers. Calls from other modules trigger an access violation exception, which is handled suitably by the OS.

A memory-protection unit (MPU) has a programming interface to specify regions of memory that are privileged, e.g., protected from reading and/or writing and/or execution, by applications other than privileged applications. Examples of privileged applications include kernel processes, processes owned by a root user, etc.

Per techniques of this disclosure, a memory-protection unit is augmented with registers that enable programmers to specify regions of code-space that are authorized to access a protected region of memory. Protected memory can include memory that stores data or code. If

code resides in protected memory, execution of such code is subject to the caller having authorization to do so.

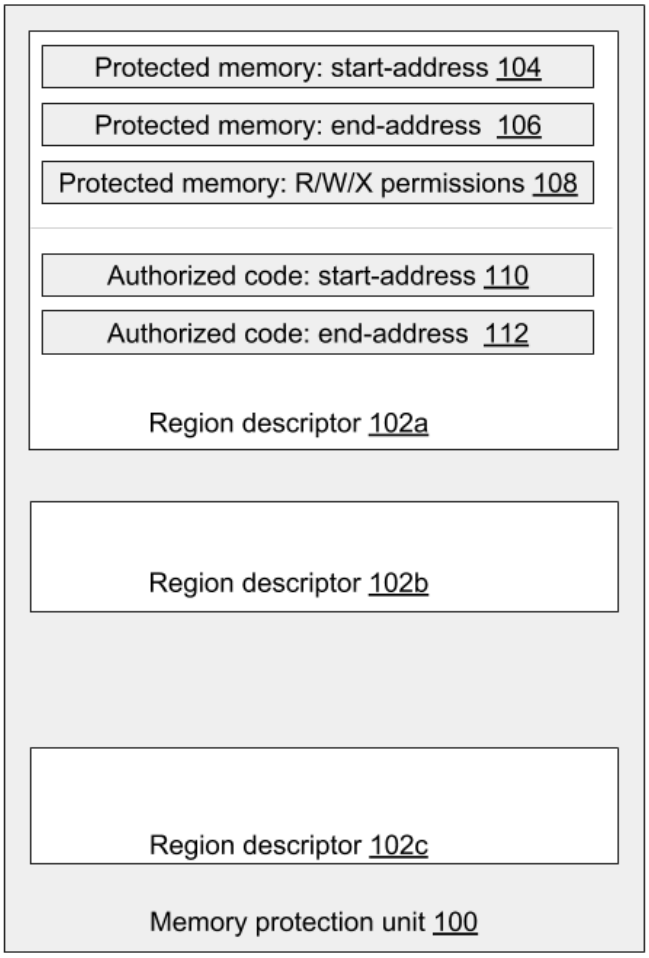

Fig. 1: Memory protection unit with use of authorized code registers

Fig. 1 illustrates an example memory protection unit with provision for specifying authorized code segments, per techniques of this disclosure. The memory protection unit (100) includes one or more region descriptor sub-units (102a-c), each pertaining to a certain region of memory. Within a region descriptor, protected memory space (also known as target memory region) is identified by registers indicating a start-address (104), an end-address (106), and the nature of permissions (108) granted, e.g., read and/or write and/or execute. Code that is authorized to access the protected region of memory is identified by a start-address register (110)

and an end-address register (112). Alternately, regions of protected memory space and authorized code-space can be specified by a start-address register and a size register.

MPU registers, e.g., as in Fig. 1, are writable by privileged code, e.g., kernel code, but not ordinarily by user applications. If a user application requires access to these registers, such application code can make system calls to do so. Direct access to MPU registers by user applications, e.g., bypassing the invocation of a system call, causes an exception.

A developer of kernel or other privileged code accesses the MPU registers using specifically designed APIs. Alternately, source code can be annotated with pragmas (e.g., compiler or preprocessor directives) to indicate protected regions of memory and corresponding authorized segments of code. Still alternately, kernel developers can leverage the compiler, with its built-in knowledge of the target architecture and memory map, to generate code that populates the registers of the MPU. Using the compiler to thus set up protected memory regions and authorize code segments during the start-up phase of a program enables the programmer to achieve superior memory protection with almost no additional programming burden.

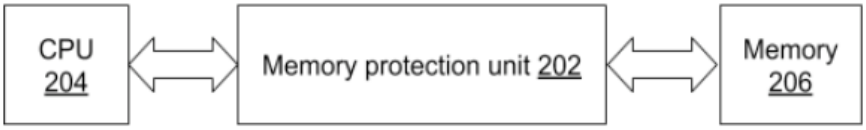

Fig. 2: Memory protection unit as a hardware module

Architecturally, the MPU can be implemented as a hardware module (202) situated between a CPU core (204) and memory (206), as shown in Fig. 2. The MPU intercepts memory accesses (e.g., memory loads/stores) from the CPU and determines if the code requesting access to a particular region of memory is authorized to do so. If a code instruction is authorized, e.g., the location of the instruction falls within the start and end addresses of the authorized code

regions (as illustrated in Fig. 1), the instruction is permitted to access the memory without interruption.

If a memory-access attempt is made by code located outside of the authorized code range, the CPU triggers an access protection fault exception. The hardware module that includes the MPU can be advantageously integrated onto the same semiconductor chip that includes the CPU, e.g., integrated with the CPU core. In the case of smaller microcontrollers, the mechanism described herein can be advantageously integrated into an existing MPU. In the case of a larger processor or ASIC, the mechanisms described herein can be advantageously integrated into an existing memory management unit (MMU).

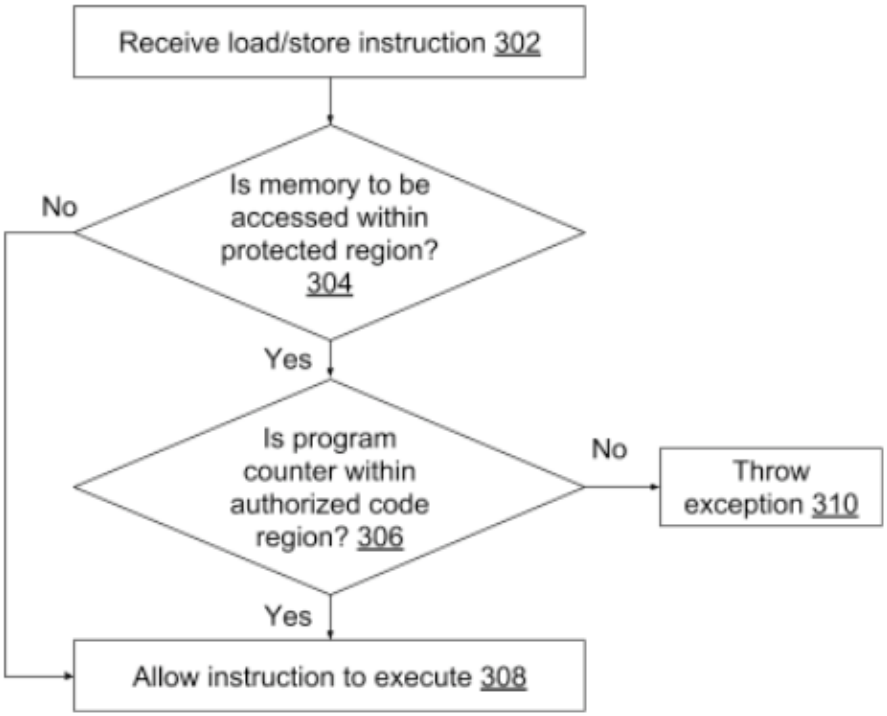

Fig. 3: Determination of whether an instruction is authorized to access a memory location

Fig. 3 illustrates an example procedure for an MPU to determine if an instruction is authorized to access a particular memory location. Upon receipt of an instruction (302), e.g., an

instruction to load from or store into a certain memory location, the MPU tests if the memory to be accessed lies within a protected region (304). If the memory location does not lie within a protected region, the instruction executes without interruption (308). If the memory location lies within a protected memory region, the program counter is checked to see if it includes a code-instruction address that lies within an authorized code region (306). If the program counter includes a code-instruction address that lies within an authorized code region, then the instruction executes without interruption (308), else an exception is thrown (310).

Example scenarios where the techniques of this disclosure apply include:

- Single-address space applications such as kernel or other privileged code.
- User applications running within a single section of a multi-address memory map.
- Relatively small microcontrollers running bare-metal operating systems on physical memories.
- Larger microprocessors utilizing heavy-weight operating systems, virtual memory, and memory management units to partition memory into mutually isolated multiple sections, each such memory section allocated to a single application.
- Device drivers, implemented such that device-driver code, and no other code, is authorized to access memory-mapped I/O registers of the corresponding peripheral.
- Safety or security-critical code, such that only authorized callers can invoke such code.

Device drivers, bare-metal/ heavy-weight operating systems (and kernels thereof), microcontrollers/ microprocessors running single or multi-address space applications, etc. are found almost ubiquitously. The techniques of this disclosure therefore apply to most consumer devices, e.g., laptops, desktops, mobile devices, IoT devices, medical equipment, security devices, etc.

CONCLUSION

Aside from malicious software gaining access and corrupting sections of computer memory, even legitimate software can inadvertently overwrite and damage its own memory content. Memory protection units exist that enable programmers to cordon off sensitive sections of memory. However, there is currently no mechanism to protect sensitive memory from code running in the same address space as that of the memory. Further, software that runs in privileged mode can change protection status of memory regions by modifying registers of the memory protection unit itself.

This disclosure provides a mechanism for protected memory to be accessible by code segments that are specifically authorized to do so. Per the techniques described herein, a protected memory region includes addresses of protected memory and addresses of code-space that is authorized to access such protected memory. An instruction that requests access to protected memory is tested for authorization. If such authorization succeeds, execution of the instruction continues uninterrupted, else an exception is thrown.