Yangyang Zhao, Mingyu Chen, Yuhang Liu *et al.* IMPULP: A Hardware Approach for In-process Memory Protection via User-Level Partitioning. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY

IMPULP: A Hardware Approach for In-process Memory Protection via User-Level Partitioning

Yangyang Zhao[1,2], Mingyu Chen[1,2], Yuhang Liu[1], Zonghao Yang[2], Xiaojing Zhu[1], Zonghui Hong[2], and Yunge Guo[2]

[1] *Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, China*
[2] *University of Chinese Academy of Sciences, Beijing 100049, China*

E-mail: zhaoyangyang@ict.ac.cn; cmy@ict.ac.cn; liuyuhang@ict.ac.cn; yangzonghao@ict.ac.cn; zhuxj@ict.ac.cn; hongzonghui@ict.ac.cn; guoyunge@ict.ac.cn

Received March 20, 2019 ; revised May 8, 2019 .

Abstract In-process memory abuse brings in security concerns in recent years. With the increasing complexity of applications, a program may call third-party functions and components. Since third-party code is neither controlled by the programmer nor limited when accessing memory resources within a process, once third-party codes contain security vulnerabilities, the program will suffer information leakage and control flow hijacking. However, current solutions like Intel MPX(Memory Protection Extensions) severely degrade performance, while other approaches like Intel MPK(Memory Protection Keys) lack flexibility in dividing security domains. In this paper, we propose IMPULP, an effective and efficient hardware approach for in-process memory protection. The rationale of IMPULP is user-level partitioning that user code segments are divided into different security domains according to their instruction addresses, and accessible memory spaces are specified dynamically for each domain via a set of boundary registers. Every memory access related instruction will be checked according to its security domain and the corresponding boundaries. Illegal in-process memory access of untrusted code segments will be prevented. IMPULP can be leveraged to prevent a wide range of in-process memory abuse attacks, such as buffer overflows and memory leakages. For verification, an FPGA prototype based on RISC-V instruction set architecture has been developed. We test seven cases to verify the effectiveness of IMPULP, including five memory protection function tests, a test to defense typical buffer overflow, and a test to defense famous memory leakage attack named Heartbleed. We execute the SPEC CPU2006 benchmark to evaluate the efficiency of IMPULP. The performance overhead of IMPULP is less than 0.2% runtime on average , which is negligible. Moreover, the resource overhead is less than 5.5% for hardware modification of IMPULP.

Keywords in-process isolation, memory protection, out-of-bounds, user-level partitioning

1 Introduction

In conventional memory protections, only memory accesses requiring inter-process communications are checked and prevented if violations are detected [2]. With the increasing complexity of applications, a program inevitably calls third-party functions and components in the same address space. When the third-party codes contain security vulnerabilities, the adversary abuses the memory resources within a process to execute codes illegally in the context of the attacked process, leading to privilege escalation and sensitive data leakage. For example, in the Heartbleed [3] incident that nearly affected the entire Internet, the attacker used a vulnerability in the OpenSSL code to pass an illegal length to the `memcpy` function to obtain sensitive data, including user account and password information. Moreover, some malicious libraries in mobile apps [4] steal sensitive data using privileged APIs (Application Programming Interface). In addition, the at-

Regular Paper

tacker leverages third-party plugins [37] to bypass the protection of current program, resulting in memory-corruption vulnerabilities.

In the past, software-based memory protection like SFI (Software Fault Isolation) [29], was neither complete nor efficient due to instrumenting every memory-access instruction for run-time checks. Therefore, recent research explores more practical hardware extensions. For example, Intel MPX (Memory Protection Extensions) [28] provides new instructions of x86 instruction set architecture (ISA) to configure the bounds for a pointer to a buffer. However, Intel MPX severely reduces the performance of programs due to the idea of bounds checking for each instruction of a process. Furthermore, Intel has recently announced Control-flow Enforcement Technology [27] and Memory Protection Keys (MPK) [26]. However, these technologies either provide hardware support limited to a specific mitigation, or cause unnecessary performance overhead.

To take advantage of hardware to improve performance, we propose IMPULP for efficient in-process memory protection. Instead of instrumenting each memory access instruction like Intel MPX, IMPULP only inserts an API function to restrict the accessible address range before each cross-domain function call. Moreover, the out-of-bounds check is performed by modified CPU pipeline automatically when the program is running. Hence, run-time overhead can be greatly reduced in IMPULP compared to Intel MPX. Since the partition of security domain is achieved in user-level, the performance overhead of domain switching in IMPULP is smaller than other domain-based methods, such as Intel MPK and memory domains on ARM32[42].

We develop the IMPULP prototype on an FPGA platform integrated with RISC-V [18] [17] ISA. We add new groups of dedicated CPU registers, and modify Linux kernel to support register configuration and raise out-of-bounds exception. Experimental results show that IMPULP can effectively prevent buffer overflows and memory leakages. We also test typical applications in SPEC CPU2006 benchmark to evaluate the performance loss of IMPULP. The results show that the runtime performance overhead is less than 0.2% on average. In addition, the resource overhead is less than 5.5% for hardware modification of IMPULP.

Our contributions can be summarized as follows:

1) We propose IMPULP, a novel hardware-enforced approach for in-process memory protection. IMPULP divides user-level security domains by the range of corresponding instruction addresses. User code with different instruction addresses can access different ranges of memory address space. When untrusted user code segments access out-of-bounds memory, the modified CPU pipeline would report illegal state and generate exception. The process is halted timely.

2) We present a pair of IMPULP APIs to dynamically specify the address range of memory protection. The switch of different security domain is performed with slightly modification of a program using this pair of APIs. This design reduces runtime overhead in the following two ways. IMPULP requires no traps into kernel compared to the conventional domain isolation method. Besides, IMPULP diminishes the costs of instrumentation for each instruction, since it merely sets registers when a cross-domain function call occurs.

3) We build an FPGA prototype system with an enhanced RISC-V CPU core, a modified Linux kernel as well as a compiler extension to support IMPULP. We test the prototype with both functional tests and performance benchmarks. The results show that IMPULP effectively performs in-process memory protection with low runtime overhead and hardware cost.

The paper is organized as follows. In Section 2, we

introduce the state-of-the-art in-process memory protection methods and discuss the characteristics and disadvantages of them. Section 3 provides a brief introduction of adversary model of our design. Section 4 details the design of IMPULP. In Section 5, we present the hardware and software implementation of IMPULP. Moreover, we briefly introduced the usage of IMPULP. Section 6 provides experimental results to verify that IMPULP is both effective and efficient. The hardware cost is also evaluated in this section. Section 7 concludes this paper.

2 Background

2.1 In-process memory abuse attacks and defenses

In-process memory corruption is usually caused by improper internal memory access, and triggered by many unreliable factors within a user process, such as third-party libraries or plugins, improper function parameters, buffer overflow, out-of-bounds memory access and speculation cache. The attackers take advantage of in-process memory security risks mentioned above, once they have penetrated into the context of target process, they can freely access (or abuse) memory content of victim programs. Specific attack methods include but not limited to code-injection [30], code-reuse [22] [23], and data-only attacks [24] [25].

Code-injection attackers inject malicious code segments as data into the address space of a process, and then execute the code to gain control of the process, thereby achieving attack by generating many other processes or modifying system files. Typical defenses like DEP (data execution prevention) [16] add an execute permission bit in page table. The operating system mitigates most code injection attacks by managing the bit to set code segment non-modifiable, while data segment modifiable but not executable. Specific schemes include no-execute page-protection (NX) [13] processor defined by AMD and execute disable bit (XD) [1] processor defined by Intel.

Even if code-injection attacks are eliminated, attackers can change the execution order of in-process code to rewrite sensitive data in in-process memory spaces with code-reuse attack, such as ROP (return-oriented programming) [15], COP (call-oriented programming) [39], and JOP (jump-oriented programming) [14]. The defense method includes ASLR (address space layout randomization) [38], CFI (control flow integrity) [12] and CPI (code-pointer integrity) [11]. The basic idea of ASLR is to load the codes into a random location, so that the attacker cannot find the exact target code location, increasing the difficulty of code-reuse attacks. Both CFI and CPI limit the control transfer when program runs, so that a program is always limited by the original control flow graph and code pointer.

Data-only attacks modify critical data variables such as branching conditions to change program. Data flow integrity (DFI) [9] [10] mitigating data-only attacks by restricting data access. To give developers an efficient way to protect sensitive data like cryptographic keys at source code level, IMIX [8] added a new instruction called `smov` to protect metadata.

2.2 State-of-the-art memory protection approaches

From the design point of view, existing memory protection approaches can be divided into the following categories.

The first type divides a part of memory from memory space to store sensitive information, and applies various memory protection techniques to prevent attacks to the security part of memory. This type is represented by Intel SGX (software guard extensions), which aims

Table 1. Features of state-of-the-art in-process memory isolation techniques

Hardware-enhanced mechanism	**Memory bounds check**	**VT based method**	**Domain based method**		**ISA based method**
Names	***Intel MPX***	***Dune***	***Shreds***	***MPK***	***IMIX***
characteristics	Every instruction is checked to prevent out-of-bounds memory access	Intel VT-x provides user with full access to protected hardware	ARM holds the access permissions for 16 domains, to isolate sensitive data	Intel X86 supports 16 domains of memory protection key, to divide user virtual space	adds new instruction named smov to access critical metadata
shortcoming	high performance overhead	The kernel is exposed to user, introducing security risks	offer limited security due to a support of specific hardware and recompiled libraries	permission setting register is accessible for user program, that introduce security risks	a security risk occurs when the adversary modifies code and then executes the `smov` instruction with controlled arguments

at strongly isolating sensitive code and data from the operating system, hypervisor, BIOS, and other applications [6]. However, Intel SGX protects only small parts of the application that handle sensitive data, and can be used for memory protection only at high performance costs due to overheads for entering and exiting the enclave [5].

The second type hides the memory address that stores sensitive information, making it unpredictable to an attacker for memory protection purposes. This type is represented by ASLR (address space layout randomization) [38]. ASLR is commonly applied to user-space applications and OS kernels due to its effectiveness and low overhead. Unfortunately, ASLR suffers from various side-channel attacks. A combination of a software bug and the knowledge of data structure addresses can lead to private code execution.

The third type divides a part of memory from memory space for untrusted code segments. When the program is running, the developer checks whether the memory access boundary or the access permission is broken, thus preventing possible security attacks. This type is represented by Intel MPX [28] and Intel MPK [26]. Our method of IMPULP also belongs to this type.

Another type of methods does not divide security domains for the entire memory space, but explores fine-grained partitions. For example, they tag a memory block with several bits in hardware to distinguish the security domains. This type is represented by tagged memory [36]. Tagged memory adds a tag for memory or register unit to store the semantic information of data. Permission check and memory isolation can be performed according to the corresponding tag. This method lacks real implementation due to the high hardware cost.

Many classic software countermeasures against memory abuse, such as Libsafe [35], LibsafeXP [34], and stack canaries [33], always incur high performance overhead. Therefore, we only illustrate the features of typical techniques in table 1, and list the shortcomings of these techniques. Intel MPX implements bounds checking in hardware [28]. Therefore, it provides new instructions to configure a lower and upper bound for a pointer to a buffer. When the program is running, Intel MPX leverages another instruction to quickly check whether this address points into the buffers boundaries.

As discussed in Section 1, Intel MPX introduces high performance overhead.

Another type of hardware-enforced mechanism is based on Intel VT-x technology. Dune[32] builds a process abstraction with a small virtual kernel that initializes virtual hardware and mediates interactions with the physical kernel. Users access privileged resources through the interface provided by Dune and avoid the trap into kernel. Therefore, Dune significantly improves the performance. To isolate memory, Dune maintains both regular and protected EPT (extended page table) mappings to enforce privilege checking. Since Dune exposes kernel modules directly to the user without additional security measures, it introduces serious security risks.

Memory domain based methods include Shreds of ARM platform and Intel MPK. Shreds[21] uses the memory domain of ARM for access control. A shred can be viewed as a flexibly defined segment of a thread execution. Each shred is associated with a protected memory pool with specific id. If the domain id mismatch occurs, the memory access is illegal. However, the change of memory domains needs to modify page table in kernel, which introduces high overhead. Intel MPK [26] divides the memory into different domains. Each domain has a specific value called memory protection key. For an x86 system, the key of a page is marked with four bits in page table. Once a process accesses the memory, hardware checks whether the key of the process matches the key from the memory block. A mismatch will trigger an exception. Compared with Shreds, when an application changes protection domain, MPK requires no modification of the page tables, and therefore the performance overhead is lower. However, the user program can control read and write permissions for each domain by changing the corresponding registers, which introduce security risks.

Recently, there exist new methods based on modified ISA. IMIX [8] extends the Intel x86 ISA with a new instruction `smov` to protect sensitive metadata. However, the requirement of immutable code in adversary model of IMIX is hard to achieve, especially when a process calls more and more complex third-party codes. Once the adversary injects new code or modifies existing code, and then executes the `smov` instruction with controlled arguments, a security risk generates.

3 Adversary Model

Throughout our work, we use the following adversary model and assumptions, which are consistent with prior work in this field of research [11] [19] [23].

Memory corruption. We assume the presence of a memory-corruption vulnerability, which the adversary can repeatedly exploit to read and write data according to the memory access permissions.

Kernel Security. This paper divides security domains in user-level. Therefore, the system kernel is hypothesized to be secure, which means this paper will not discuss the attacks and defenses about system kernel.

Inter-process Security. This paper focuses on in-process memory protection. Therefore, we assume that different processes are safely isolated, which means this paper will not discuss the attacks and defenses across different processes.

Security risk of user code. When applications call untrusted third-party code, a process may induce vulnerabilities and lead to information leakage and control flow hijacking. Security risks of user code include vulnerable OpenSSL code, malicious libraries in mobile apps, third-party plugins, and any untrusted code segments which may induce an attack.

4 Design of IMPULP

IMPULP divides in-process code segments into different security domains by the instruction address. The first version of IMPULP divides the code of the process into two parts, trusted code segments and untrusted code segments. IMPULP leverages the instruction address to distinguish between trusted and untrusted code segments by the following steps.

1) At *compile time*, we set the instruction address range of trusted code segments. The corresponding boundary registers are updated by kernel at load time only once. Therefore, it is impossible for untrusted code segments to modify the instruction address range of trusted code to ensure security. 2) At *run time*, the hardware obtains the current program counter (PC) value and compares it with the instruction address range of trusted code segments. If the PC is inside the range, the current instruction originates from a trusted code segment. Otherwise, it comes from an untrusted code segment.

For trusted code segments, the accessible memory address covers the entire address space of the process. For untrusted code segments, IMPULP limits the accessible memory address within parts of the address space and sets access permissions for safety by the following steps. 1) At *compile time*, we specify the accessible memory space for untrusted code segments by instrumenting three instructions within the trusted code segments. These three instructions set the lower and upper bound of accessible memory space, and the access permission of the memory space. They can be packaged as an API function. The encapsulated pair of APIs is as follows. *start_protect (addr, len, cfg, index); end_protect (index);* See the details in Section 5.2.1. We name untrusted code segment as library function. We add the API function right before the library function is invoked in the program. 2) At *run time*, the corresponding registers are updated when the instrumented instructions are executed. When the illegal command is issued by out-of-bounds memory requests from untrusted code segments, the hardware will detect out-of-bounds or illegal access when checking the corresponding registers. Then the modified CPU pipeline generates an interrupt and records the current interrupt category information.

With the address space restrictions of memory access, sensitive data is isolated from out-of-bounds memory access of in-process untrusted instructions. However, attackers may execute code-reuse attacks. For example, if untrusted code segments make use of `jump` or `branch` to reach the address where the accessible memory boundaries are stored, attackers can break through the above boundaries by modifying the values in boundary registers. Even a jump between library functions can introduce security risks. IMPULP provides strong protection against ROP attack, which means, the jump from library function to any illegal address is prohibited.

Since several new groups of boundary registers are introduced into the process, if these register groups are modified by malicious code, the security defense is corrupted. Therefore, an automatic hardware checking mechanism is added in IMPULP. These new registers, which designed for security enhance, are unwritable by untrusted code segments.

Another possible form of security risk occurs when the library function invokes any system call by using the `syscall` function. A process running in kernel mode can execute any instruction in the instruction set and access any memory location in the system. Since our security model considers the kernel to be secure and does not perform security checks on memory access boundaries, a successful `syscall` can break through the IM-

PULP defense. To eliminate such security risk, we make some changes in kernel to handle `syscall`. The kernel adds a `syscall` check procedure by the following steps. 1) When a `syscall` occurs, it is determined whether the `syscall` is from trusted code segments or library function. If it comes from trusted code segments, the `syscall` will run normally. If it comes from the library function, the step 2 will be executed. 2) Compare the parameters passed in of a `syscall` with the upper and lower bounds of the library function registers. If the parameters exceed the boundaries, error report is triggered.

From the above discussion, the user-level partitioning for trusted and untrusted code segments needs modifications of the kernel. The preset of accessible memory space for untrusted code segments requires GCC extension and changes in linker script. The hardware achieving runtime check needs to modify the CPU Pipeline and add new types of interrupt. In addition, IMPULP adds new groups of registers, and new instructions in ISA are added to protect these metadata. See Section 5 for details of the changes.

5 Implementation of IMPULP

In this section, we will describe the hardware and software modifications of IMPULP in details. For the convenience of description, we name the trusted code segment in a process as primary function, and the untrusted code segment as library function.

5.1 Hardware Modification

From Section 4, we know that the hardware mainly modifies three parts. IMPULP adds groups of registers to store boundaries corresponding to user code segments in different security domains. The hardware needs to perform boundary checking while the program is running, and therefore IMPULP modifies the structure of CPU pipeline. In addition, in order to protect the boundary register groups from being illegally accessed, the hardware extends ISA instructions.

5.1.1 Adding Register Groups

IMPULP adds three types of registers in the CPU core. There is one group of primary instruction address range (PIAR) registers, consist of the starting and ending instruction address of the primary function. If the program counter (PC) value of current instruction exceeds the range, then it is considered that current instruction comes from the library function.

We provide sixteen groups of library memory address range (LMAR) registers. Each group includes one pair of starting and ending memory address accessible by library function. Therefore, we can protect sixteen objects of one library function at most, to limit the memory boundaries of the object. Whenever CPU executes load/store instruction of an object, the CPU pipeline looks up the corresponding group of registers, confirming whether the memory access of load/store instruction is out-of-bounds. Once the library function returns to the primary function, all LMAR registers are reset, which can be used by the next library function. Therefore, the configuration and reset of the LMAR registers are dynamic. The reset is also achieved by instrumenting instructions in trusted code segments.

To avoid ROP-like attack, IMPULP adds an return address register (RAR) in the CPU core, recording the return address of the library function. When the library function is called, the return address will be stored into RAR. When the library function returns to the primary function, the target address stored in program counter will be compared with the return address stored in RAR, and the mismatch will trigger return address error exception. With the help of RAR, we can assure the control flow integrity of IMPULP.

In order to ensure the security of IMPULP, access to registers is subject to the following rules:

Rule 1. The primary function could access local variables of the program and all the global variables, while the library function could only access ranges the LMAR registers indicated. This rule prevents the library functions accessing sensitive data kept by the primary function.

Rule 2. Only kernel can modify PIAR. Both kernel and primary function could modify LMAR registers.

5.1.2 Modification of CPU pipeline

To support the design of IMPULP described in Section 4, the hardware modification of CPU pipeline includes four parts, which are distributed in various stages. Figure 1 shows the changes in a typical five-stage CPU pipeline architecture. The following paragraphs will introduce the details of each part.

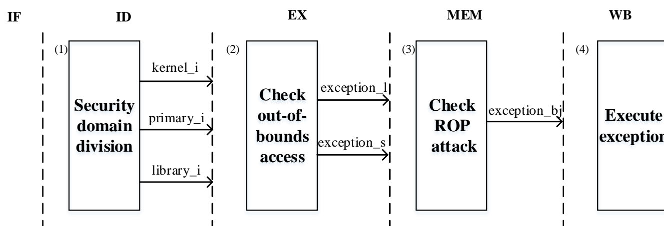

Fig.1. The processing flow modification of five-stage pipeline.

IF (Instruction fetch). An instruction is fetched from the memory by program counter. We do not make changes in this unit.

ID (Instruction decode). We added a module to divide security domain of instructions. This module contains three parts. First, IMPULP added a unit to check the type field of current instruction. If the instruction is judged as `load`, `store`, `branch` or `jump`, IMPULP will trigger the next unit, which is used to check the privilege level of the instruction. Both other types of instructions and instructions from kernel follow the conventional processing flow. Then if the privilege level is user mode, IMPULP will compare the program counter with the values stored in PIAR registers. With this unit, the instructions of the same process are divided into different security domains effectively. With the modification of ID stage, CPU obtains the information about which region the instruction code belongs to, which is kernel, primary function or library function. As shown in Figure 5.1.2, when the indicator `primary_i` is valid, the instruction originates from the primary function of user code. The function of other indicators is the same.

EX (Execution). The module to check out-of-bounds access includes two parts. According to the results of the ID stage, if the instruction is `load/store` and it belongs to library function, IMPULP will check whether the address is within legal memory bounds set in LMAR (library memory address range) registers. Out-of-bounds access will trigger an exception to CPU. The indicator `exception_l` in Figure 1 corresponds to illegal load instruction, while `exception_l` is valid when illegal memory store occurs.

MEM (Memory Access). The module added in this stage will check ROP-like attack and report illegal branch or jump of control flow. For `branch` and `jump` instructions, IMPULP divides the destination address of the same process into two parts, primary function address and library function address. If the instruction jumps from primary to library function, the return address will be stored into RAR (Return Address Register). When any `jump` or `branch` instruction occurs in a library function, IMPULP compares target address with return address stored in the RAR. If they are the same, this operation is legal. Otherwise IMPULP will trigger a return address error exception. The corresponding indicator is `exception_bj` in Figure 1.

The above steps provide strong protection against ROP attack, since any other target address from a library function is prohibited except the right return ad-

dress to primary function. This implementation will certainly affect the application of IMPULP to complex scenarios, but for the first version of IMPULP, we do not consider function nesting.

WB (Write back). During this stage, the data loaded from memory or calculated by the ALU would be written to the register file for normal state. IMPULP added a group of registers to record the current PC value, memory access address and exception reason. When exception occurs in EX or MEM stage, CPU generates the exception with the registers.

5.1.3 *Adding IMPULP-exclusive instructions*

In order to configure IMPULP-exclusive registers, we need to extend dedicated configuration instructions on the RISC-V instruction set.

The processor core of our experimental platform supports the *RV64MAFDC* instruction set. We take one of the operation codes reserved in the *RV64MAFDC* instruction set, then the new instruction can be extended. For the convenience of addressing and extending PIAR and LMAR registers, we have designed four IMPULP-exclusive instructions namely `impulpprw`, `impulpprs`, `impulplrw`, and `impulplrs`. The first six characters represent our method, `p` or `l` means the instruction is used for setting PIAR or LMAR registers, `rw` or `rs` corresponds to *read and write* or *read and set* separately. With the extended instructions, IMPULP ensures the security of configuring registers.

5.2 Software Supplement

As shown in Figure 2, compiler supports the new IMPULP-exclusive instructions. Then according to the symbol table which is generated by compiler, the kernel delimits the primary function and configures the registers. Primary function can call API to set LMAR registers. Finally, the CPU executes the program under specific memory access limitation.

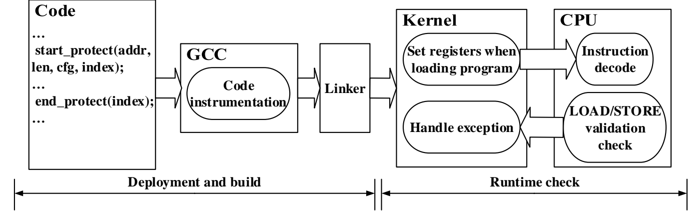

Fig.2. Developers can instrument instructions via APIs; Compiler instrument programs with APIs; Kernel configures the registers; CPU limit access ranges of data memory access.

5.2.1 *IMPULP API*

The encapsulated pair of APIs is as follows.

start_protect (addr, len, cfg, index); end_protect (index);

There exists a group of LMAR registers. `start_protect` writes the `addr` into the lower one, and the sum of `addr` and `len` into the upper one. The former refers to the lower bound of accessible memory access address, while the latter refers to the upper bound. The `cfg` defines the permission of the address set in LMAR of the library function.

When running programs, IMPULP will inquire the LMAR registers to see if the following operations are within the specific memory. In this way, malicious access to memory space would never succeed as they will be prevented by IMPULP, and raise exception. The `index` is needed to indicate specific registers used by the current API. After invocation of library function, the `end_protect` API is invoked to clear a LMAR group specified by `index`, so that IMPULP will not check the memory access range anymore.

To prevent library functions modifying LMAR registers, special configuration instructions in `start_protect` and `end_protect` could only be executed in primary function.

5.2.2 Compiler modification

RISC-V provides the companion compilation tool GCC. The compiler computes input parameter's effective data length, figures out which part of the program is primary function, and guides the kernel to fill the corresponding registers.

We added the definition of the new instruction to GCC, as well as the operation code and instruction mask of the new instruction. The modified GCC generates a corresponding binary code when processes a new instruction. When the program is executed, the hardware can set and read the value of the corresponding registers with the binary code.

We added some code in linker script to modify the memory layout of a primary function. Before the program runs, the instruction address boundaries of the primary function have been written to the corresponding PIAR registers.

5.2.3 Kernel modification

We modify the load function of a program in the kernel. When kernel loads the binary file (`ELF` file), it obtains the symbol table of the `ELF` file, which is already generated by compiler and linker script. Then `_start` (the *entry point* of `ELF`) and `__libc_csu_fini` (the end of initialization) in this symbol table are regarded as start and end address of primary function. Finally, the kernel fills the start and end address of the primary function into registers.

We modify the code of the process context switch in kernel, add the operations of saving and restoring IMPULP-exclusive registers to ensure the correctness of register information.

We modify the code of the exception execution function in the kernel, add the processing code of the IMPULP exception, and output the corresponding exception information according to the type of the exception when the IMPULP exception is generated.

We add a procedure of `syscall` check in kernel. The primary function uses a separate `syscall` entrance, while the library function uses the entrance of `libc` to execute `syscall`. After the `syscall` handler saves the context, the `syscall` check procedure is started. According to the value of `epc`(exception program counter), IMPULP judges whether the `syscall` is from primary function or library function. The `syscall` from primary function continues executing, while the `syscall` from library function performs the following check step. If the parameters passed in of a `syscall` exceed the upper or lower boundaries recorded in LMAR registers, the kernel raises an interrupt. Else, the `syscall` from library function continues executing.

5.3 Usage of IMPULP

Users apply IMPULP with several simple steps. When the user requires calling an untrusted library function, IMPULP is set to protect in-process memory by the following steps. First, the user adds a pair of APIs' code in the place where the library function is called in the program. The parameters of `start_protect` are the upper and lower bound of available memory space for untrusted function, and the permissions of this space. The `index` of `end_protect` equals to the same value in `start_protect` to reset the same group of LMAR registers. Then the user opens the compile options that support new IMPULP instructions when GCC compiles the code. After generating the executable file, the user runs the file with the attribute `-impulp`. Then the user code is running under the protection of IMPULP. Once the untrusted code accesses out-of-bounds memory, the CPU will halt the process and generates an exception.

6 Evaluation and Results

RISC-V [18] [17] is a free and open source instruction set architecture (ISA) based on modern design techniques and decades of computer architecture research. We performed all our evaluations using a modified RISC-V CPU with 1GB RAM running Linux kernel 4.6.2 and the basic frequency of CPU is 62.5MHz.

We use the Virtex-7 FPGA VC707 Evaluation Kit as our testing platform[41]. We take open source project 'sifive/freedom' as a baseline project[40]. The steps to generate the baseline project are as follows. 1) Download the project. 2) Compile the file of 'Makefile.vc707-u500devkit' in the project. 3) Create a `vivado` project and generate a `bit` file. 4) Burn the `bit` file to the test platform and run the Linux kernel.

The steps to generate IMPULP project is similar. 1) Change the codes in the directory '/freedom/rocket-chip/src/main/scala/rocket' of sources files (written in `scala` language) to implement IMPULP. 2) Re-compile the file of 'Makefile.vc707-u500devkit' in the project. 3) Replace the recompiled `verilog` file in the `vivado` project. 4) Generate new `bit` file and test the IMPULP logic.

This section contains the following contents. First, we verify the effectiveness of IMPULP. 1) We completely tested memory protection functions of IMPULP with five cases. 2) We performed a test to defense typical buffer overflow. 3) We achieved a test to defense famous memory leakage attack named Heartbleed. Then, we selected the popular SPEC CPU2006 benchmark to test the runtime overhead to prove the efficiency of IMPULP. Finally, we present the hardware overhead of IMPULP.

6.1 Effectiveness

6.1.1 Functional Test Cases

According to the description of Section 5, IMPULP can prevent library functions from reading, writing or executing out-of-bounds memory in address space, and prohibit library functions from modifying the boundary register groups associated with the security defense. In addition, when library function calls `syscall`, the memory is also protected by the procedure of `syscall` check. Therefore, we wrote five programs to test the above five functions. Functional descriptions and test results are shown in Table 2. The exception numbers of 0xC, 0xD and 0xE are IMPULP-exclusive, while the exception number of 0x8 is general.

Table 2. Five Functional Test Cases.

Attack type	**test descriptions**	**test results**
out-of-bounds read	The attack occurs when the load instruction of a library function accesses illegal memory address.	IMPULP generates an exception, the exception number is 0xC.
out-of-bounds write	The attack occurs when the store instruction of a library function accesses illegal memory address.	IMPULP generates an exception, the exception number is 0xD.
control-flow hijack	The attack occurs when a library function jumps to a memory address which exceeds the limited boundaries.	IMPULP generates an exception, the exception number is 0xE.
access security-related registers	The attack occurs when a library function tries to modify the registers of PIAR, LMAR and RAR.	The write enable signal in hardware is invalid. The security-related registers cannot be modified by any library function.
syscall abuse	The attack occurs when a library function tries to access out-of-bounds memory with a system call.	IMPULP generates an exception, the exception number is 0x8.

6.1.2 Defense Case of buffer overflow

Typical buffer overflow includes stack overflow and heap overflow. The defense case of stack overflow is shown in Table 2. Line 3 defines a stack. Line 4 and Line 6 inserts IMPULP APIs to limit the memory access boundaries of the library function (`strcpy` in Line

5). When the length of copied string is less than 10, the program runs normally. Once the length of copied string exceeds 10, IMPULP raise an exception with number 0xD, which means the load instruction of library function `strcpy` accesses the illegal memory address. The test of heap overflow is almost the same. The only different line is Line 3. Line 3 defines a heap by 'char *pass;'. And the test result is the same with stack overflow test. Therefore, IMPULP effectively defends buffer overflow attacks by setting accessible memory boundaries for vulnerable parameters with APIs.

Table 2. Defense Case of stack overflow.

1	int main(int argc, char *argv[])
2	{
3	char pass[10];
4	start_protect(pass, sizeof(pass), CFG, 0);
5	strcpy(pass, argv[1]);
6	end_protect(0);
7	printf("strcpy done");
8	return 0;
9	}

6.1.3 Defense Case of memory leakage

Heartbleed is a security bug in the OpenSSL cryptography library. It results from improper input validation (due to a missing bounds check) in the implementation of the TLS heartbeat extension. The vulnerability allowed attackers to remotely read protected memory from an estimated 24%-55% of popular HTTPS sites.

We implemented a socket server with the API provided by OpenSSL-1.0.1e in C, which initializes the socket, SSL library, and waits for the connection of client using `SSL_accept`. We wrote a socket client program to communicate with socket server. The client sends *hello request* of TLSv1.1 to server, and the connection between server and client will be established. To replay Heartbleed, we sent a malformed Heartbeat request in client. The test results show that, after the socket connection is established, the server will respond with a *hello packet* sent by the client. The length of the response packet returned is 685.

If IMPULP is not used, after receiving the Heartbeats request packet sent by the client, the server will call `memcpy` to copy the excess data (61455, up to 65535) to the returned packet and return it to the client, causing leakage of data within the program, which may have sensitive information such as the user's account number and password.

After deploying IMPULP in OpenSSL source code, that is, protecting the `memcpy` invocation in `tls1_process_heartbeat`, the unexpected copy operation raised an exception and the process is terminated. Finally, no secret data is leaked out.

6.2 Efficiency: SPEC CPU2006 benchmarks

In SPEC CPU2006 benchmarks, we added `start_protect` API function before the library function called and `end_protect` API function after the library function returned. We regard `GLIBC` as unreliable library functions, for instance, the functions like `printf`, `memset`, `strcpy`, `memcpy` are not trusted.

We choose `bzip2`, `gcc`, `mcf`, `gobmk`, `hmmer`, `sjeng`, `libquantum`, `omnetpp` and `astar` as our test benchmarks. We ran train workload three times and used average run time for each benchmark. Results show that the IMPULP configuration overhead is almost negligible, the total overhead of the IMPULP is 0.166%(249/150353) of the total execution runtime(150353s). The reason is that IMPULP is implemented on the basis of hardware check at runtime. Furthermore, boundary setting operations only occur at the entrance of a subroutine, and no instrumentation is needed within a subroutine. Therefore, the performance overhead is negligible.

Table 3 shows the execution time comparison between baseline and IMPULP. Their execution times are

almost the same. The time is recorded in seconds.

Table 3. The execution time of SPEC CPU2006 benchmark.

	Baseline (s)	**IMPULP (s)**	**Runtime cost**
401.bzip2	6167	6170	0.05%
403.gcc	167.26	167.32	0.03%
429.mcf	2060.54	2059.95	-0.03%
445.gobmk	22270	22355	0.43%
456.hmmer	14935	14981	0.31%
458.sjeng	21729	21784	0.25%
462.libquantum	14565	14637	0.49%
471.omnetpp	58322	58292	-0.05%
473.astar	10137	10156	0.19%
total	150352.80	150602.27	0.166%

6.3 Hardware Cost

First we check the utilization of FPGA resources and total on-chip power in the vivado project. As shown in Table 4, the overhead of different resources of IMPULP is less than 5.5% while the power overhead is 4.3%.

Table 4. FPGA resource usages of RISC-V Baseline and IMPULP

	Baseline	**IMPULP**	**Overhead**
Power(W)	3.747	3.911	4.3%
LUT	48519	51138	5.4%
LUTRAM	3546	3550	0.1%
FF	35833	37771	5.4%
BRAM	30	30	0

Moreover, preparing for future chip design, we also use the DC (Design Compiler) tool to compare the modified core with the original core in terms of area, cells, and power. Considering the practicality of a chip, we added L2 cache resources and tested the overall results. Table 5 gives the hardware comparison results of baseline and IMPULP. The area of IMPULP is bigger than baseline. The area overhead of the IMPULP is 0.7%(16598/2375314) while the power overhead is 0.5% (0.982/192.119). Both are negligible.

Table 5. Hardware comparison results of RISC-V Baseline and IMPULP

	Baseline	**IMPULP**	**Overhead**
Area(um^2)	2375314	2391912	0.7%
Cells	814561	822024	0.9%
Power(W)	192.119	193.101	0.5%

7 Conclusions and Future Work

The security vulnerability is inevitable without effective in-process memory protection methods. Current solutions either severely degrade performance or offer only very limited security. In this study, we propose IMPULP, a novel hardware-enforced approach for effective and efficient in-process memory protection. IMPULP classifies user process into trusted primary functions and untrusted library functions, endowing different privileges to the two parts to realize in-process data isolation. We extend the Linux kernel and the compiler to provide interfaces to IMPULP. Experimental results show that IMPULP can effectively protect in-process memory with negligible runtime overhead.

Our IMPULP is designed for scenarios where a program calls third-party untrusted code segments. The out-of-bounds memory access of untrusted code, the hijacking of control flow, and the modification of security-related registers are prohibited by IMPULP, which fully guarantees the security of user-level partitioning. Moreover, IMPULP only instruments a few instructions to modify the boundary registers before an untrusted function is called. The judgment of illegal access is automatically executed by the CPU pipeline when the program is running. Therefore, the runtime overhead is low.

The limitations of IMPULP includes that the memory protection belongs to domain protection. Therefore, the approach is implement with continuous memory address space, thus lacks of flexibility. Fine-grained protection extensions need future work. Moreover, the first version of IMPULP implementation only supports two levels of user code division, namely, primary func-

tion and library function. The protection for more levels requires more discussion.

References

[1] Intel. Intel 64 and IA-32 architectures software developer manual. Retrieved March 19, 2019 from http://www-ssl.intel.com/content/www/us/en/processors/architecturessoftware-developer-manuals.html.

[2] C. Jacomme and S. Kremer and G. Scerri. Symbolic Models for Isolated Execution Environments. In *IEEE European Symposium on Security and Privacy*, 2017, pp.530-545.

[3] Limer Eric. How Heartbleed Works: The Code Behind the Internet's Security Nightmare. Retrieved March 19, 2019 from https://gizmodo.com/how-heartbleed-works-the-code-behind-the-internets-se-1561341209.

[4] Chen Y and Reymondjohnson S and Sun Z and et al. Shreds: Fine-Grained Execution Units with Private Memory. In *IEEE Symposium on Security and Privacy*, 2016, pp.56-71.

[5] V. Costan and S. Devadas. Intel sgx explained. In *IACR Cryptology ePrint Archive*, 2016.

[6] Intel Software Guard Extensions (Intel SGX). Retrieved March 19, 2019 from https://software.intel.com/en-us/sgx.

[7] Paul Kocher and Jann Horn and et al. Spectre Attacks: Exploiting Speculative Execution. In *40th IEEE Symposium on Security and Privacy, SP*, 2019.

[8] Tommaso Frassetto and Patrick Jauernig and et al. IMIX: In-Process Memory Isolation EXtension. In *27th USENIX Security Symposium, USENIX Sec*, 2018.

[9] P. Akritidis and C. Cadar and et al. Preventing memory error exploits with WIT. In *29th IEEE Symposium on Security and Privacy, SP*, 2008.

[10] M. Castro and M. Costa and et al. Securing software by enforcing data-flow integrity. In *7th USENIX Symposium on Operating Systems Design and Implementation, OSDI*, 2006.

[11] V. Kuznetsov and L. Szekeres and et al. Code-pointer integrity. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI*, 2014.

[12] M. Abadi and M. Budiu and et al. Control-flow integrity. In *ACM SIGSAC Conference on Computer and Communications Security, CCS*, 2005.

[13] NX bit. In *Wikipedia*, Retrieved March 19, 2019 from https://en.wikipedia.org/wiki/NX_bit.

[14] Tyler B and et al. Jump Oriented Programming: A New Class of Code-Reuse Attack. In *Proceeding of the 6th ACM Symposium on Information, Computer and Communications Security, ASIACCS*, 2011, pp.30-40.

[15] R. Roemer and et al. Return-Oriented Programming System,Languages, and Applications. In *ACM Transactions on Information and System Security*, 2012, pp.1-34.

[16] Data Execution Prevention (DEP). Retrieved March 19, 2019 from http://support.microsoft.com/kb/875352/EN-US/.

[17] Sifive. The RISC-V Instruction Set Manual Volume II: Privileged Architecture. Retrieved March 19, 2019 from https://riscv.org/specifications/privileged-isa/.

[18] Sifive. The RISC-V Instruction Set Manual Volume I: User-Level ISA. Retrieved March 19, 2019 from https://riscv.org/specifications/.

[19] H. Shacham and M. Pages and et al. On the effectiveness of address-space randomization. In *ACM SIGSAC Conference on Computer and Communications Security, CCS*, 2004.

[20] R. Gawlik and B. Kollenda and et al. Enabling client-side crash-resistance to overcome diversification and information hiding. In *NDSS*, 2016.

[21] Chen Yaohui and et al. Shreds: Fine-Grained Execution Units with Private Memory. In *Security and Privacy, SP*, 2016, pp.56-71.

[22] F. Schuster and T. Tendyck and C. Liebchen and L. Davi and A.-R. Sadeghi and T. Holz. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications. In *36th IEEE Symposium on Security and Privacy*, 2015.

[23] K. Z. Snow and F. Monrose and L. Davi and A. Dmitrienko and C. Liebchen and A. Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *34th IEEE Symposium on Security and Privacy*, 2013.

[24] X. Ge and H. Vijayakumar and and T. Jaeger. SPROBES: Enforcing kernel code integrity on the trustzone architecture. In *Mobile Security Technologies, MoST*, 2014.

[25] H. Hu and S. Shinde and A. Sendroiu and Z. L. Chua and P. Saxena and Z. Liang. Data-oriented programming: On the expressiveness of non-control data attacks. In *37th IEEE Symposium on Security and Privacy*, 2016.

[26] D. Hansen. [rfc] x86: Memory protection keys. Retrieved March 19, 2019 from https://lwn.net/Articles/643617/.

[27] Intel. Control-flow Enforcement Technology Preview. 2017.

[28] Otterstad, C. W. A brief evaluation of Intel MPX. In *Systems Conference IEEE*, 2015, pp.1-7.

[29] D. Sehr and R. Muth and et al. Adapting software fault isolation to contemporary cpu architectures. In *18th USENIX Security Symposium, USENIX Sec*, 2010.

[30] Aleph One. Smashing the stack for fun and profit. In *Phrack Magazine*, 2000.

[31] SEO J. and LEE B.and et al. SGX-Shield: Enabling address space layout randomization for sgx programs. In *Proceedings of the 2017 Network and Distributed System Security Symposium, NDSS*, 2017.

[32] Belay, Adam and et al. Dune: safe user-level access to privileged CPU features. In *Usenix Conference on Operating Systems Design and Implementation USENIX Association*, 2012, pp.335-348.

[33] Dang T. H. Y. and Maniatis P. and et al. The Performance Cost of Shadow Stacks and Stack Canarie. In *ACM Symposium on Information, Computer and Communications Security*, 2015, pp.555-566.

[34] Lin Zhiqiang and B. Mao and L. Xie. LibsafeXP: A Practical and Transparent Tool for Run-time Buffer Overflow Preventions. In *Information Assurance Workshop IEEE Xplore*, 2006, pp.332-339.

[35] Tsai Timothy K. and N. Singh. Libsafe: Transparent System-wide Protection Against Buffer Overflow Attacks. In *International Conference on Dependable Systems and Networks IEEE Computer Society*, 2002, pp.541.

[36] E. A. Feustel. On the advantages of tagged architecture. In *Transactions on Computers*, July 1973, pp.644-656.

[37] K.Naoki and T. Yamauchi and T. H. Austin. Access Control for Plugins in Cordova-Based Hybrid Applications. In *IEEE International Conference on Advanced Information Networking & Applications*, 2017.

[38] Lu K and Song C and Lee B and et al. ASLR-Guard: Stopping Address Space Leakage for Code Reuse Attacks. In *Acm Sigsac Conference on Computer & Communications Security*, 2015, pp.280-291.

[39] Sadeghi A and et al. Pure-Call Oriented Programming (PCOP): chaining the gadgets using call instructions. In *Journal of Computer Virology & Hacking Techniques*, 2017, pp.1-18.

[40] Retrieved March 19, 2019 from https://github.com/sifive/freedom.

[41] Retrieved March 19, 2019 fromhttps://www.xilinx.com/products/boards-and-kits/ek-v7-vc707-g.html

[42] ARM. ARM architecture reference manual. Retrieved March 19, 2019 from http://silver.arm.com /download/ARM_and_AMBA_Architecture/AR150-DA-70000-r0p0-00bet9 /DDI0487A_h_armv8_arm.pdf, 2015.