Put Your Memory in Order: Efficient Domain-based Memory Isolation for WASM Applications

Hanwen Lei
Key Laboratory of High-Confidence Software Technologies (MOE), School of Computer Science, Peking University
China
lei_hanwen@stu.pku.edu.cn

Ziqi Zhang
Key Laboratory of High-Confidence Software Technologies (MOE), School of Computer Science, Peking University
China
ziqi_zhang@pku.edu.cn

Shaokun Zhang
Key Laboratory of High-Confidence Software Technologies (MOE), School of Computer Science, Peking University
China
skzhang@pku.edu.cn

Peng Jiang
Key Laboratory of High-Confidence Software Technologies (MOE), School of Computer Science, Peking University
China
pengjiang_pku2020@stu.pku.edu.cn

Zhineng Zhong
Key Laboratory of High-Confidence Software Technologies (MOE), School of Computer Science, Peking University
China
zhongzhineng@pku.edu.cn

Ningyu He
Key Laboratory of High-Confidence Software Technologies (MOE), School of Computer Science, Peking University
China
ningyu.he@pku.edu.cn

Ding Li
Key Laboratory of High-Confidence Software Technologies (MOE), School of Computer Science, Peking University
China
ding_li@pku.edu.cn

Yao Guo
Key Laboratory of High-Confidence Software Technologies (MOE), School of Computer Science, Peking University
China
yaoguo@pku.edu.cn

Xiangqun Chen
Key Laboratory of High-Confidence Software Technologies (MOE), School of Computer Science, Peking University
China
cherry@sei.pku.edu.cn

ABSTRACT

Memory corruption vulnerabilities can have more serious consequences in WebAssembly than in native applications. Therefore, we present PKUWA, the first WebAssembly runtime with memory isolation. Our insight is to use MPK hardware for efficient memory protection in WebAssembly. However, MPK and WebAssembly have different memory models: MPK protects virtual memory pages, while WebAssembly uses linear memory that has no pages. Mapping MPK APIs to WebAssembly causes memory bloating and low running efficiency. To solve this, we propose Domain Isolated Linear Memory (DILM), which protects linear memory at function-level granularity. We implemented DILM into the official WebAssembly runtime to build PKUWA. Our evaluation shows that PKUWA can prevent memory corruption in real projects with a 1.77% average overhead and negligible memory cost.

CCS CONCEPTS

• **Security and privacy → Software and application security**.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CCS '23, November 26–30, 2023, Copenhagen, Denmark
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0050-7/23/11…$15.00
https://doi.org/10.1145/XXXXXX.XXXXXX

KEYWORDS

WebAssembly; Memory Isolation; MPK

1 INTRODUCTION

WebAssembly is a bytecode language with portability, speed, and memory efficiency [33]. It compiles from C/C++, Rust, Go, etc. [43], and runs on independent runtimes and browsers [70]. WebAssembly's linear memory architecture is vulnerable to attacks like modifying static variables through stack overflow [43]. Since linear memory allows arbitrary memory access, existing software-based defenses for native applications are not as effective in WebAssembly. *Stack and Heap Canaries* [20] only prevent sequential buffer overflow, which is not enough for WebAssembly [43]. *Address Space Layout Randomization (ASLR)* [63] is less effective in WebAssembly because the attacker can scan the whole memory space without causing page faults. *Metadata-based Defenses* [34, 42, 53, 59] such as shadow memory [59] and code-pointer integrity (CPI) [42] are not suitable for WebAssembly because they require storing metadata in a piece of secure memory. However, there is no such secure area in linear memory. *OOP Security features* that exist in languages (like Java, JavaScript, and Python) [7, 28, 48, 64] are also not applicable to WebAssembly because it is a low-level bytecode that supports multiple languages, including C. Introducing the OOP security features can break the portability of WebAssembly. Therefore, a stronger memory protection mechanism is needed for WebAssembly.

To strengthen the security of WebAssembly, in this paper, we aim to propose a novel memory isolation approach for WebAssembly that limits memory-related attacks to a small segment of linear memory called domain. This approach prevents arbitrary memory accesses and mitigates WebAssembly-specific memory vulnerabilities. Our approach is inspired by the MPK-based memory protection approaches for *native applications* [55, 65], which use the Intel MPK hardware feature to create protected memory regions for native applications and defend against stack and heap overflow-based attacks with high efficiency [35, 55, 65]. We leverage the same hardware feature to create isolation domains in WebAssembly, aiming for a highly efficient linear memory protection approach.

Although it sounds promising, building a memory isolation approach based on Intel MPK for WebAssembly is fundamentally challenging due to the **Memory Model Mismatch** between the linear memory of WebAssembly and the page-based memory model of MPK. MPK adds a special protection key to each entry in the page table, which means it has to protect memory on a granularity at a page level, e.g., the developer needs to first call mmap to allocate a page and then protect the page with pkey_mprotect. Unfortunately, since the *concept of the page does not exist in WebAssembly*, the developer cannot use mmap in WebAssembly. Even if we forcibly port them to the WebAssembly runtime, adopting such an incompatible isolation model (e.g., the APIs and instructions discussed in Section 2.3) will break the default assumption of WebAssembly, leading to potential memory bloating and high-performance overhead. Note that although there are several enhancements to using MPK in native applications, including libmpk [55], ERIM [65], Hodor [35], PKU-safe [39], EPK [31], and many other techniques [41, 67], they are all page-based and suffer the same problem in WebAssemly as the vanilla version of MPK APIs.

For linear memory bloating, calling mmap to obtain a page-aligned memory address will unnecessarily expand the linear memory, even if the WebAssembly application does not require that much memory. This expansion happens each time the developer calls pkey_mprotect and mmap, which can lead to quick memory bloat. This problem is particularly concerning for WebAssembly applications, as the linear memory grows continuously and cannot be reduced once it has expanded. As a result, memory leakage can occur and impact the usability of the application. According to our assessment, using mmap and pkey_mprotect can consume up to 10 times more memory than the standard version of WebAssembly. Thus, developers must manage memory manually to prevent bloating and ensure optimal.

For high runtime overhead, calling mmap and pkey_mprotect in WebAssembly is much slower than calling them in native apps. Accessing these functions requires the use of WebAssembly System Interface (WASI), which has slower APIs due to WebAssembly's sandboxed design. When accessing system resources, WASI APIs must cross two boundaries, resulting in complex and time-consuming calls. In our evaluation, directly using mmap and pkey_mprotect can slow down WebAssembly apps by 18.79% on average, with 2.82 times slower WASI calls compared to system calls.

To solve the challenge of memory model mismatch, we propose PKUWA (Protection Key in User space for WASM) to provide memory protection for WebAssembly. Our key insight is to use functions as protection units due to their clear entry and exit points, allowing for automatic security checks. Additionally, functions are independent entities, typically accessing the same memory. By using functions as the granularity for memory protection, we can reduce the need for domain switching compared to using memory objects. Our solution utilizes the Domain Isolated Linear Memory (DILM) model to divide linear memory into different domains, limiting each WebAssembly function to access memory in only one domain. With DILM, we can isolate memory and prevent memory-related vulnerabilities in WebAssembly.

More importantly, DILM is compatible with Intel MPK, which enables efficient memory isolation. By assigning different MPK protection keys to different domains in DILM, PKUWA can use the MPK hardware to quickly detect unauthorized memory access. However, implementing DILM on MPK and ensuring efficiency is not trivial. There are two main technical challenges. The first one is how to minimize the WASI calls when using PKUWA. Since WASI calls are slow in WebAssembly, it is not efficient to implement all PKUWA interfaces as WASI calls. To overcome this challenge, PKUWA avoids costly WASI calls and improves performance by adding a new instruction to WebAssembly that modifies the permission bits in PKRU. The second one is how to manage memory in different domains effectively. Since memory in different domains has different MPK protection keys, a naive approach would require frequent switching of the protection keys to access memory in all domains, which is inefficient. To overcome this challenge, we design a novel distributed linear memory management module that avoids switching MPK protection keys while keeping the memory in different domains, achieving both time and memory efficiency in linear memory usage.

We have implemented a prototype of PKUWA in Wasmtime [72]. To demonstrate the usefulness of PKUWA, we present a case study that shows how PKUWA can protect linear memory from three well-known vulnerabilities (Arbitrary Write, HeartBleed, and Overflow-Induced Heap Overwrite). To evaluate the efficiency of PKUWA, we collected ten applications from open-source projects that can run on the WebAssembly standalone runtime and ported them using PKUWA. Our experiment shows that DILM can reduce the runtime overhead by 9.89× compared to the naive approach that directly uses the page-based programming model of native applications, on average. For memory overhead, PKUWA does not introduce noticeable extra memory consumption compared to the vanilla Wasmtime. On the other hand, the memory consumption for using MPK APIs directly is 1.51× - 10.81× higher than the vanilla Wasmtime. Our evaluation confirms that directly using the MPK programming model for native applications in WebAssembly can significantly bloat the linear memory and degrade the performance of WebAssembly. It also shows that PKUWA can address these problems effectively and efficiently. Overall, our evaluation demonstrates the benefits of PKUWA.

This paper presents the following main contributions:

- We introduce a novel linear memory abstraction, PKUWA, that enables WebAssembly applications to run with strong isolation.
- We developed the Domain Isolated Linear Memory (DILM) model, which is compatible with linear memory and can be implemented with MPK.

- We implement PKUWA in the Wasmtime runtime and evaluate its performance on real open-source applications.

The source code of PKUWA is publicly available at:
https://github.com/hanwenlei/PKUWA

2 BACKGROUND

In this section, we discuss background information about the linear memory of WebAssembly and its security issues. We will also discuss the background information about MPK.

2.1 Linear Memory of WebAssembly

Linear memory is a *single continuous byte array* that the runtime maps from a reserved virtual memory space. Unlike other bytecode languages, WebAssembly does not provide any memory management or garbage collection scheme. This design allows developers or compilers to fully control the runtime performance. Linear memory has three properties [43, 49, 70]:

- **P1: Monotonicity.** Linear memory can only grow and never shrink. Once the application frees some memory, it remains in the linear memory until the application ends.
- **P2: Density.** Linear memory has no gaps, and the memory is contiguous. Unlike native platforms, there are no unmapped spaces between different sections of linear memory.
- **P3: Arbitrary Memory Access.** The application and the developers have full control over all data in the linear memory. The application can read and write to the *entire* memory.

Linear memory was designed for efficiency and portability as a compilation target in WebAssembly. It allows for sequential memory allocation without worrying about freeing memory. However, unlike native applications, it cannot to free memory. This means that linear memory can only grow and can cause memory bloat if not managed efficiently.

2.2 Security Risks of WebAssembly

Recent studies [36, 43, 49] reveal that WebAssembly suffers from **memory corruption** vulnerabilities, which may cause more severe consequences than attacking native code (e.g., C/C++) in some scenarios [43, 61]. Figure 1 illustrates the problem of linear memory by comparing the simple layout of WebAssembly memory (left) and the secure layout of native memory (right). The WebAssembly memory mainly comprises three parts: the heap section, the data section, and the stack section. The simplicity of WebAssembly memory differs from native memory in two aspects. First, the data section does not differentiate the data with different permissions (e.g., read-only data and writable data), and all the data in this section is readable (**P3**). Second, there is no unmapped memory between different sections, and the space from the stack section to the data section is contiguous (**P2**). On the other hand, the native memory is more complex. First, the data section is split into multiple segments, e.g., read-only section (`.rodata`) and writable section (`.data`). The constant strings are immutable during execution. Second, native memory has unmapped memory between different sections. Accessing such memory will trigger a `SIGSEGV` signal.

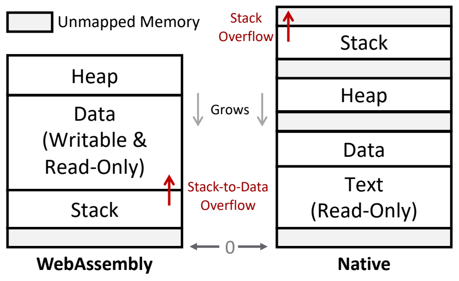

Figure 1: Example of the arbitrary file written in WebAssembly and native.

As shown in Listing 1, this code snippet demonstrates how an adversary may exploit memory corruption to perform arbitrary file write. The code receives user input and stores it in `buf` at line 2 and line 4. Then, the code opens a file and appends a constant string to it from line 7 to line 9. In this example, `"file.txt"`, `"a"`, and `"Append constant text."` are constant strings that should not be modified.

```c
1 // Somewhere in the binary
2 char buf[32];
3 // Stack-based buffer overflow
4 scanf("%[^\n]", buf);
5 ...
6 // Write "constant" string into "constant" file
7 FILE *f = fopen("file.txt", "a");
8 fprintf(f, "Append constant text.");
9 fclose(f);
```

Listing 1: Stack-based buffer overflow that causes an arbitrary file write.

For WebAssembly, an overflow on the local variable `buf` may allow the user to write arbitrary data in arbitrary files. Note that in Figure 1, the data section follows the stack section. If the user input is larger than the size of `buf`, `buf` may overflow to the data section and overwrite the constant strings. For Listing 1, a malicious user can modify the file write process (line 7 to line 9) by constructing a malicious input to overflow. As a consequence, the attacker is able to control the target file to write (`"file.txt"` at line 7), the file opening mode (`"a"` at line 7), and what to write in the file (`"Append constant text."` at line 8). Thus, the attacker can *write arbitrary data to arbitrary files*. Note that this type of attack does not work for native applications.

2.3 Memory Protection Keys

Intel Memory Protection Key (MPK) is a new hardware feature that allows fast and flexible modification of memory permissions in user space. MPK uses a field in the Page Table Entries (PTEs) called *protection key* and a hardware register called `PKRU`. The protection key indicates the access rights of the memory page, and `PKRU` stores the current protection keys and their corresponding permissions. The hardware checks the permission by comparing `PKRU` with the protection key of the PTE on every memory access. MPK is efficient because it enables the program to manipulate the permissions of multiple memory pages in user space without invoking the kernel.

Therefore, the overhead of the permission check is negligible and the overhead of the PKRU permission switch is minimal [62, 65, 71]. The concept of memory protection key is adopted by various architectures, not only by Intel MPK. These architectures include ARM memory domain [5], RISC-V's Donky [57], IBM's Power architecture [16], HP PA-RISC [54], and Itanium (IA-64) [13].

Listing 2 illustrates how to use MPK according to the official Linux document [38]. The developer first calls `pkey_alloc` to create a new protection key `PKEY_DISABLE_WRITE`, which means the key is not writable (line 1). Then the developer calls `mmap` to allocate a new memory page (line 2) and calls `pkey_mprotect` to assign the page to `pkey` (line 3). Note that `pkey_mprotect` only works on page-aligned memory. During the execution stage (line 4), the memory of `ptr` is secure and cannot be modified. To change the memory content, the developer first changes the permission of `pkey` to `0` (writable; line 5), then updates the memory content (line 6), and finally restores the permission of `pkey` to `PKEY_DISABLE_WRITE` (line 7).

```
1 pkey = pkey_alloc(0, PKEY_DISABLE_WRITE);
2 ptr = mmap(NULL, PAGE_SIZE, PROT_NONE,
  ↪  MAP_ANONYMOUS|MAP_PRIVATE, -1, 0);
3 ret = pkey_mprotect(ptr, PAGE_SIZE, PROT_READ|PROT_WRITE, pkey);
4  //... application runs here
5 pkey_set(pkey, 0);  // clear PKEY_DISABLE_WRITE
6 *ptr = foo;  // assign something
7 pkey_set(pkey, PKEY_DISABLE_WRITE);  // set PKEY_DISABLE_WRITE
  ↪  again
```

Listing 2: A typical example of using MPK to protect memory.

The programming model of MPK (Listing 2) is not compatible with WebAssembly's linear memory because it uses the page-based memory model and WebAssembly does not support `mmap` and page-aligned memory. Note that libmpk [55] and many other optimizations of MPK [31, 35, 39, 56, 65] also use the page-based memory model, thereby can not be directly applied to WebAssembly. By calling `mmap`, WebAssembly appends a new virtual memory page at the end of the linear memory, which would waste memory space. Furthermore, `mmap` is much slower in WASI than a system call, which would reduce the performance of WebAssembly applications. Our evaluation shows that directly using the unmodified MPK programming model can increase the linear memory size by up to 1.63 times and increase the WebAssembly application overhead by 18.79% on average even when only 5% of memory objects are protected by MPK. Different from libmpk, in PKUWA, we introduce a new memory isolation model that addresses the mismatch between the linear memory and the page-based protection model. We also effectively utilize the nature of WebAssembly, which only allows control flow to jump to the beginning of code blocks, to prevent ROP-based bypassing attacks. Moreover, we use a strategy of swapping memory between different domains to reduce the runtime cost and memory bloating. More details will be described in Section 4.

2.4 Threat Model

In this paper, we focus on the memory vulnerability. We consider a WebAssembly application containing malicious functions (attacker), trusted functions (victim), and normal functions. Malicious functions contain vulnerabilities that the attacker can use. Trusted functions contain sensitive data or control flow that the adversary tries to attack.

The malicious functions can be utilized in two ways: (1) the attacker can actively exploit the vulnerabilities in the malicious functions, or (2) the malicious function can directly run the malicious code. For example, the malicious function can be supplied by a malicious user in the third-party library. To exploit the vulnerabilities in malicious functions, the adversary can feed special trigger input or utilize function return values (e.g., pointers). We do not make any assumptions or constraints on the type of functions as long as the functions comply with WebAssembly semantics.

We assume the trusted functions do not contain vulnerabilities and do not proactively leak sensitive data from legitimate output. We consider the hardware and the OS as trusted. We consider Spectre attacks [40], side-channel attacks [12, 74], and fault attacks [52] to be out of the scope of this paper because they exploit hardware vulnerabilities and are unrelated to memory vulnerabilities.

3 LINEAR MEMORY WITH ISOLATION DOMAINS

We introduce a new memory isolation model for WebAssembly linear memory, called DILM, that addresses the mismatch between the linear memory and the page-based protection model of MPK. Unlike conventional MPK-based memory protection models on native code, which can only protect memory page-by-page [14, 65], the DILM model is function- and domain-based, which allows controlling the access rights of functions to memory domains. The function- and domain-based memory isolation enables DILM to map memory segments of arbitrary length to the domain linear memory, which is not feasible in conventional MPK-based memory protection models. The DILM model has the following three advantages:

- **Memory Isolation.** DILM provides in-process memory isolation for WebAssembly applications. Under the DILM model, a function cannot access the memory without authorization. This isolation gives WebAssembly applications a similar level of memory protection as native applications.
- **Backward Compatibility.** DILM is fully compatible with the existing linear memory model. In other words, current WebAssembly applications can run seamlessly on a runtime that supports DILM.
- **Efficiency.** DILM can be implemented on top of MPK to ensure efficiency. We will discuss the details of optimizations in Section 4.

Figure 2 illustrates the idea of the DILM model in its middle and right parts. The middle part shows four domains (Domain 0-3), each with several segments of linear memory in different colors. The right part displays five WebAssembly functions, which can be linked to various domains. The connection between a function and a domain indicates that the function has access to the memory within that domain. DILM will block a function from accessing a domain that it is not linked to. For instance, the malicious function in Figure 2 has no access to the memory in Domain 3.

Formal Definition. Formally, we define the DILM model as a bipartite graph $G = \langle F, D, d_0, E \rangle$, where each $f_i \in F$ is a WebAssembly function, each $d_j \in D$ is an isolation domain, d_0 is a special initial domain, and $E = \{\langle f_i, d_j \rangle\}$ is a set of edges that links F

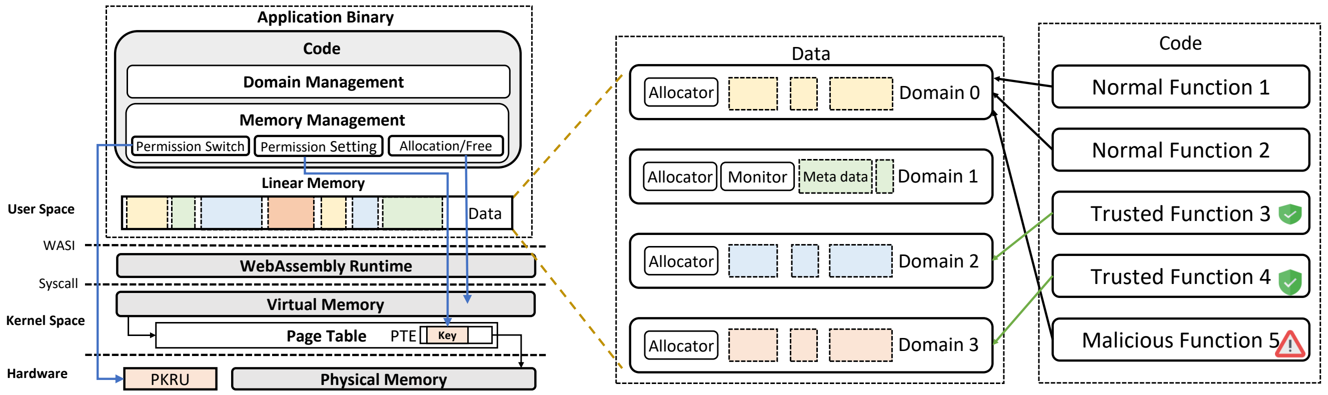

Figure 2: The Domain Isolated Linear Memory (DILM) model.

and D. An isolation domain $d_j \in D$ is a subset of the linear memory L with two properties: 1) isolation domains are disjoint (i.e., $\forall d_g, d_h \in D, d_g \cap d_h = \emptyset$), and 2) every address in L belongs to one domain (i.e., $\bigcup d_g = L$). Thus, the isolation domains partition the linear memory L. We do not assume that the addresses are contiguous in each isolation domain. A domain may consist of multiple segments of linear memory. In the DILM model, an edge $\langle f_i, d_j \rangle$ indicates that f_i can access the addresses in d_j. If f_i attempts to access the addresses in d_j but $\langle f_i, d_j \rangle \notin E$, the DILM model will REJECT such access, enforcing memory isolation. We distinguish two types of edges: *level-1* and *level-2* edges. A level-1 edge grants read-only permission to the function for a specific domain. A level-2 edge grants read and write permissions to the function for the corresponding domain. We adopt this two-level permission system inspired by process address space in operating systems [30]. We also require that each function is connected to EXACTLY ONE domain. This prevents an adversary from using a function as a bridge to access data from other domains illegitimately. Initially, we connect all functions to the special domain d_0. Developers can switch a function from d_0 to other domains later.

Operations on DILM: We define a set of operations on the DILM graph G. These operations are exposed to developers so that they can realize memory isolation in their WASM applications. We detail the operations as follows:

- *Create* creates a new isolation domain d_g in D.
- *Release* deletes an isolation domain d_g from D.
- *Add* adds a segment of linear memory to one isolation domain.
- *Swap* moves a segment of linear memory from one domain to another.
- *Switch* changes the connected domain of a function to another.

To maintain compatibility with the linear memory monotonicity of WASM, our DILM design does not include an operation for deleting a linear memory segment from a domain. Instead, it only allows swapping linear memory segments between domains(Swap). We rely on a user-level memory allocator to manage the linear memory in each domain and achieve memory efficiency, as we explain in Section 4.2. This is similar to conventional WASM.

Domain Access Control. We use a function-based domain access control system in DILM. This system allows a function f_i to access a memory protected in an isolation domain d_j only if $\langle f_i, d_j \rangle \in E$ (i.e., f_i has access to d_j). For instance, if a function a needs to access a memory in domain d_j that it cannot access, it has to invoke a function b in domain d_j to access the data indirectly. Note that b does not return the data in domain d_j to a, but rather acts as a proxy function that computes on the data and returns the result to a. The arguments and return values are copied to the target domain (the domain of b for arguments and the domain of a for return values) before being used by the target function. This prevents the data in domain d_j from leaking to a. Moreover, DILM prohibits functions from passing (or returning) cross-domain pointers. When a caller calls a callee in another domain, the arguments and return values of the callee cannot be pointers. This is because adversaries may use cross-domain pointers to access the data in other domains illegally. Only when the caller and callee are in the same domain they can use pointers to pass arguments and return values. This ensures the data security in different domains.

4 DESIGN OF PKUWA

We propose a novel runtime for WASM applications, PKUWA (Protection Key in User space for WASM), based on DILM.

A pure software implementation of the DILM model would incur high overhead [58, 65, 69], as it would require checking the memory access permission of each function. Therefore, we leverage MPK [38] to realize DILM more efficiently in PKUWA. Figure 2 shows the general architecture of PKUWA, which extends the linear memory management module in the existing WASM runtime with DILM. PKUWA uses MPK to implement the DILM model and its five operations. Moreover, PKUWA provides modules for domain and memory management, as well as optimizations to reduce memory consumption and runtime overhead.

Special Domains. PKUWA defines two special domains in DILM: Domain 0 and Domain 1. Domain 0 is the initial domain d_0 that all functions start with. Domain 1 is a privileged domain that only the **monitor** function can access. The **monitor** function manages the meta-data of domains for PKUWA. PKUWA does not allow any user-defined function to access Domain 1.

4.1 Realizing DILM with MPK

PKUWA implements DILM based on the protection keys of MPK. Specifically, PKUWA assigns a unique MPK protection key to each isolation domain. Since Intel MPK supports 16 protection keys, PKUWA can have at most 16 isolation domains. Among the 16 domains, Domain 1 is reserved by PKUWA for domain and memory management. Domain 0 and Domains 2-15 are available to users. The information about the domains, such as the mapping between the domains and protection keys, is stored in the PKUWA meta-data in Domain 1.

We realize the domains through MPK as follows. Given an isolation domain with n segments of linear memory $d_g = \{s_1, s_2, \ldots, s_n\}$, where s_i is a segment of linear memory, PKUWA first finds the corresponding Linux virtual memory pages of these segments from the WASM runtime. Because the linear memory is mapped from a contiguous space in the virtual memory, we can use the address of each segment to identify the virtual memory page. In the runtime, we maintain the base address of the virtual memory space that is mapped to the linear memory. Given the linear memory address of the segment (consisting of a base address and an offset), we can add the offset to the virtual memory's base to get the segment's virtual address. Then we can align the virtual address to identify the virtual memory page. Assume the set of corresponding Linux virtual pages of segments of linear memory in d_g is $p_g = \{p_1, p_2 ... p_m\}$, we assign each p_i the MPK protection key of domain d_g.

Note that a segment $s_i \in d_g$ may not be page-aligned on virtual memory, meaning that it may not fully occupy the last virtual page it corresponds to. Therefore, there may be a wasted virtual memory area at the end of the last page. To address this issue, PKUWA leverages a Linear Memory Manager (LMM) to utilize the memory efficiently, which we will discuss in detail in Section 4.2.

4.1.1 Realizing the Five Operations.

PKUWA implements *Create* by generating a domain ID in the meta-data and linking it to an MPK protection key. The operation *Release* is implemented to remove the domain ID and its corresponding MPK protection key.

The main challenge for implementing *Create* and *Release* is efficiency. PKUWA has to invoke the `pku_alloc` function through WASI to allocate an MKP protection key. However, this process is slow in practice. Therefore, we propose a lazy-free strategy that reuses existing MPK protection keys.

In the lazy-free strategy, when *Release* is invoked to free a domain, PKUWA does not free the MPK protection key. Instead, it retains the MPK protection key for future use and clears the protection key in the virtual memory pages. Finally, before making the linear memory in the freed domain available to other domains, PKUWA sets it to zero to ensure data security.

The rationale behind the lazy-free strategy is to avoid `pku_free` WASI calls. After we free the protection key, we need to clear the memory in it to avoid leaking sensitive data. We can reserve this memory and defer memory cleanup until the next memory allocation. At the same time, the high overhead of WASI calls is avoided.

PKUWA implements the *Add* and *Swap* operations by keeping track of free memory lists for different domains. When the developer wants to *Add* new memory to a domain, PKUWA searches the corresponding domain's memory list and finds a memory chunk that is large enough to meet the request. If such a memory chunk does not exist, PKUWA invokes `mmap` to allocate new memory pages. When *Swap* is invoked, PKUWA moves the memory chunk from the source domain's memory list to the target domain's memory list.

PKUWA achieves *Switch* by maintaining and modifying a function table. The table records the associations between functions and domains. Each entry represents a domain, and each element in the entry represents a function. An entry contains one element if and only if the function is associated with the domain. To switch a function from a domain d_i to another domain d_j, PKUWA first removes the element from the entry of d_i and then adds it to the entry of d_j.

4.1.2 Domain Access Control.

PKUWA ensures that unauthorized functions cannot access a domain that is beyond their scope. We use the `PKRU` register to control the access permission of the domains. This register has 32 bits, and every 2 bits indicates the domain's permission on one protection key. One bit indicates the read permission, and the other bit indicates the write permission. Setting a permission bit means that programs have permission to access the virtual page.

When a function f_i attempts to access a linear memory in a specific domain d_j, PKUWA first verifies whether $\langle f_i, d_j \rangle$ is in E by checking the metadata in Domain 1. If $\langle f_i, d_j \rangle$ is not in E, PKUWA aborts the execution of the function. Otherwise, PKUWA finds the corresponding virtual pages of the linear memory. Then it resets all permission bits in the `PKRU` register and only enables the corresponding permission for the MPK protection key of the domain d_j. Finally, PKUWA runs f_i. By doing this, PKUWA only allows f_i to access the memory in domain d_j. If it tries to access memory in other domains, the MPK hardware will automatically reject the access.

The security of the domain switch is ensured by the privileged Domain 1, which stores `PKRU` states of all the domains in its metadata. Each `PKRU` state represents the domain's permission on all 16 protection keys. When a domain switch occurs, the control flow is first transferred to the monitor, a privileged function that can access Domain 1, to load the `PKRU` state of the target domain from the metadata and set the `PKRU` register accordingly. Because Domain 1 is a privileged domain, functions, except for the monitor, cannot access Domain 1's data. Thus, the metadata and `PKRU` states are secured.

To reduce the runtime overhead of access control checks for every function call in a WASM application, we designed a **Call Gate** mechanism for PKUWA. The main idea is that functions in the same domain have the same permission to access the memory in that domain. Therefore, we only need to check the access control and set the MPK permissions when a function calls another function in a different domain. For example, in Figure 2, *Normal Function 1* can call *Normal Function 2* without any checks, since they are both in the same domain. However, if *Normal Function 1* wants to call *Trusted Function 3* in another domain, it has to call $gate(f_k, argv*)$, where f_k is *Trusted Function 3* and $argv*$ is its parameter list. The Call Gate function will then verify the access control and set the `PKRU` bit for the target domain.

The Call Gate is a function *gate* that wraps cross-domain calls. If a function f_j needs to call a function f_k in a different domain, it cannot directly call f_k, otherwise MPK will reject the memory accesses in f_k. Instead, it has to call $gate(f_k, argv*)$, where $argv*$ is the parameter list of f_k. The Call Gate function will then perform the access control checks and set the permission bit in PKRU for f_k's domain.

We can think of the Call Gate as a direct function in WebAssembly. However, WebAssembly applications are vulnerable to function hijacking, which lets an adversary overwrite the function table and redirect a function call to another one [43]. If we implement the Call Gate as a function, attackers could call a function in different domains by corrupting the heap metadata [43]. Furthermore, making the call gate a WASI API can also harm the efficiency of WebAssembly applications.

Therefore, we do not use a call gate function but add a new instruction to the WebAssembly instruction set, WAWRPKRU. This instruction has a similar meaning to WRPKRU in native applications. In particular, WAWRPKRU modifies the state of the PKRU register within WebAssembly applications. To implement the Call Gate, we insert WAWRPKRU and the code to verify the permission of a function to a domain before every cross-domain call. We do this by extending the WASM compiler, which we will explain in Section 5.

The nature of WebAssembly guarantees the security of WAWRPKRU-based Call Gate. In WebAssembly, code and data are separate. It also prevents common ROP attacks [2, 60] because it does not allow jumps to arbitrary addresses. WebAssembly only allows the control flow to jump to the beginning of code blocks (e.g., functions and loops). Thus, there is no way for the attacker to jump to the WAWRPKRU instruction, and our Call Gate is safe in practice.

4.2 Linear Memory Management on DILM

Similar to the conventional WebAssembly runtime, DILM maintains the property of monotonicity in the linear memory. The memory in each domain can be managed flexibly. This design decision ensures backward compatibility but also requires an extra mechanism for efficient linear memory management. Therefore, we introduce a LMM to prevent the memory in each domain from exploding.

On a high level, the functionality of LMM is similar to the conventional linear memory management module in WASM applications without DILM. Specifically, LMM provides `malloc` and `free` functions to developers. It internally maintains two lists of free and busy linear memory segments, respectively. When a developer calls `malloc`, our LMM first checks if it can reuse some free memory segments. If not, it uses the WebAssembly instruction `memory.grow` to extend the linear memory. Otherwise, it directly returns the free memory to the developer. When the developer calls `free`, the LMM moves the freed memory from the busy list to the free list.

The main challenge in designing LMM is to make PKUWA efficient in managing multiple isolation domains that are controlled by MPK. PKUWA has to scan all the virtual memory that is protected by MPK in every domain to manage the memory in different domains. However, since LMM can only access the MPK protection key of one domain at a time, this scanning process requires frequent changes of MPK permissions, which leads to high overhead.

To address this challenge, LMM adopts a decentralized memory management model, where each domain handles all the memory with the same permission. LMM relies on the monitor to manage the permissions of different domains. Each domain has an **allocator** that handles the isolated memory in the domain. The allocator is created when the domain needs to allocate memory for the first time. When the application calls different functions, the monitor verifies the permission of the target function and switches to the corresponding domain. The memory allocator offers `malloc` and `free` primitives for functions to allocate memory objects in a fine-grained way.

The monitor maintains a list of free pages that can be distributed to different allocators. The allocators maintain a list of free memory chunks in memory pages. When a function needs to allocate new memory, the monitor selects the allocator according to the permission of the function. The allocator inspects the list of free memory chunks and allocates the memory chunk with the smallest size that can hold the requested memory. If there is no free memory chunk, the allocator requests the monitor to allocate new memory pages. Then the monitor inspects the list of memory pages and allocates the memory pages with the smallest size that can hold the requested memory from the allocator. If there is no free memory page, the monitor requests the WebAssembly runtime to allocate 16 contiguous pages via the WebAssembly instruction `memory.grow`. The monitor allocates the requested memory page to the allocator and keeps other pages in its list.

The advantage of the decentralized memory management model is that PKUWA can conveniently merge memory chunks of the same permissions into one chunk within domains. This is because each domain only needs to manage the memory of one permission. The domain allocator will never need to merge two chunks of different permissions.

Figure 2 also shows how PKUWA manages the linear memory with decentralized domain allocators. The figure shows five domains, and each domain has a domain allocator. The linear memory is split into multiple memory segments, and each domain contains multiple segments.

Flexible Memory Swap. LMM uses a strategy of swapping memory between different domains to reduce the runtime cost of memory allocation. Instead of making system calls to release and request memory, the allocator reuses the memory pages freed by one domain and assigns them to another domain. The monitor maintains a list of free memory pages that are returned by the allocator when a domain releases a contiguous block of at least 16 pages. This threshold matches how WebAssembly extends the linear memory by allocating virtual memory, but it can be tuned according to the application's memory usage pattern. When a domain needs new memory, the monitor first tries to allocate from the list, clearing the memory pages before handing them to the allocator.

4.3 Guardian Pages

PKUWA uses the DILM to create guardian pages between the stack, data, and heap sections of the linear memory. This mechanism prevents the arbitrary file write attack shown in Listing 1. Although developers can implement such a mechanism manually with DILM, it would be tedious. Therefore, PKUWA provides the guardian pages by default.

To implement guardian pages, PKUWA allocates two memory segments in Domain 1 and places them between the stack, heap, and data sections. Note that no functions can access the memory in

Domain 1. Hence, if an attacker tries to access the data in other sections through stack overflow, she will encounter the guardian pages and trigger a memory access error. Thus, PKUWA can stop overflow attacks that aim to corrupt application data in other sections.

4.4 Inter-Domain Calls

Transferring cross-domain arguments and return values when calling functions can be a challenge. This is because DILM prohibits functions from passing or returning pointers across domains. Therefore, in order for PKUWA to function properly, it must copy the arguments and return values to the target domain. If the arguments and return values are basic types of WebAssembly, they will be copied to the operand stack by the runtime, as the runtime is responsible for maintaining these types.

The process of passing pointers and nested structures is a bit complex. To overcome this, PKUWA utilizes a shared memory where the contents pointed by pointers and nested structures are copied. Both the target and source domains can access the protection key of the shared memory. To facilitate data transfer to the shared memory, PKUWA offers a specialized copy-API for programmers. The programmers must specify the size of arguments/return values in the copy-API. In the case of nested structures, the copy-API may need to be invoked iteratively. This copying mechanism is similar to previous works such as Donky [57] and Intel SGX Edger8r [15].

4.5 Library Calls and Implicit Calls

Another challenge is to handle library calls and implicit calls. Functions in a library may have complex dependencies. It is difficult for developers to assign functions in libraries to different domains in a fine-grained manner. To reduce the burden on developers, PKUWA assigns all library function calls to the default domain and uses the same copy mechanisms to secure the data in library calls.

Implicit functions may be automatically generated, making it difficult for PKUWA to assign them to domains. PKUWA handles them in a similar way as C++ overloading (for implicit functions). PKUWA takes the name and argument types of the implicit function to identify the function address. Then PKUWA adds domain switching code around the function call.

5 BUILDING APPLICATIONS WITH PKUWA

To facilitate the development of WASM applications with PKUWA, we provide a set of APIs for C/C++ and Rust that enable developers to leverage the isolation domains of PKUWA. Moreover, we offer a set of compilation toolchains to optimize the performance of PKUWA-based applications.

The workflow of building an application on PKUWA consists of the following steps. Developers first write the application source code with PKUWA APIs. Then, the source code is processed by the frontend compiler, which converts the APIs to WebAssembly bytecode including our new instruction (WAWRPKRU). The output of the frontend compiler is the WASM binary code (`*.wasm`), which is passed to the backend compiler of the PKUWA runtime. In this step, the user-space instructions are compiled into the machine code. For the instructions that need system resources, the backend compiler uses the WASI library to generate machine code for system calls. The WASI library includes the WASI general APIs and the APIs for domain management. After the compilation, the machine code runs on the WebAssembly runtime and interacts with the hardware.

5.1 Application Programming Interface

We provide main APIs for DILM in Table 1, including *Create*, *Release*, *Switch*, and Call Gate. We don't provide interfaces for *Add* and *Swap* to avoid high overheads. Instead, developers can use `malloc` and `free` to implicitly perform these operations. This design ensures that developers can only obtain new memory from a domain through our LMM, which will try to reuse existing linear memory first. In addition to the operations of DILM, we also provide the `pku_init` function to initialize PKUWA's metadata, such as the function table, the allocator of Domain 0, the monitor and allocator of Domain 1, and the protection key and permission of Domain 0 and Domain 1.

Table 1: Main PKUWA APIs.

API	Description
int pku_init(void)	PKUWA initialization.
int domain_create()	Create a new isolated domain
int domain_free(int domain_id)	Release an isolated domain
PKU_CALL_REGISTER(domain_id, func)	Switch *func* to a new domain
PKU_CALL(func, argv*)	The Call Gate for calling func

The high-level programming workflow for using PKUWA is as follows. First, the developer should invoke `pku_init` to set up the PKUWA environment and connect all functions to Domain 0. Then, the developer can use `domain_create` to create an isolation domain and receive a domain ID. After creating a domain, developers can use `PKU_CALL_REGISTER` to *Switch* a function from Domain 0 to a new domain. After switching, the function is considered as trusted and has access to the isolated domain memory. When there is a cross-domain function invocation, the developer *must explicitly* use `PKU_CALL` to wrap the function call. Finally, when the memory of a domain is no longer needed, the developer can use `domain_free` to clean up the domain.

5.2 Frontend Compiler

The frontend compiler takes the source code written in C/C++ or Rust with our APIs as input and produces a valid **.wasm* executable that can run on PKUWA as output. The main challenge for the frontend compiler is to translate our APIs into the corresponding WASM bytecode.

For all APIs except for *PKU_CALL*, PKUWA implements them as new API calls in WASI. Thus, the frontend compiler only has to map the corresponding API calls to WASI calls. For *PKU_CALL*, PKUWA embeds its logic, along with the new WASM instruction, `WAWRPKRU`, to bypass WASI system calls and directly modify the permission of protection keys. This is because WASI invocation is slow, and the application may require frequent domain switching. If *PKU_CALL* also used the WASI interface, the extra overhead would be high.

5.3 Backend Compiler

The backend compiler of PKUWA takes the WebAssembly binaries produced by the frontend compiler and generates machine code from them. It can handle most WebAssembly instructions using existing rules, but it has special rules for the new permission switch instruction, WAWRPKRU.

PKUWA's backend compiler is built on Cranelift [3], a fast code generator that outputs machine code. Cranelift uses two intermediate representations (IRs): Clif and Vcode. Clif is a high-level IR, while Vcode is a low-level IR. The compiler first translates the WebAssembly binary format into Clif, then lowers Clif to Vcode, and finally emits machine code from Vcode.

To compile WAWRPKRU, PKUWA extends Clif and Vcode with new rules. For Clif, PKUWA defines the basic semantics of WAWRPKRU, such as the input and output operands, input types, and output type inference. For Vcode, PKUWA imposes two register constraints due to the requirement of the new instructions. The register constraints are: 1) %ecx and %edx must be 0, and 2) the input and output of the new instructions must be stored in %eax. The first constraint can be met by explicitly assigning 0 to %ecx and %edx using the mov instruction. The second constraint is more complex. In this phase, every WebAssembly instruction (including WAWRPKRU) will be translated into a sequence of machine code instructions. PKUWA employs the GET_REGS function in the Vcode interface to ensure that the input/output of the machine code sequence (of the WebAssembly instruction) is stored in %eax at the end of the sequence [4]. For output, this ensures the second constraint. But for input, this is not sufficient because we need to store the input at the start of the sequence. To solve this problem, PKUWA explicitly adds a mov instruction to copy the input to %eax at the start of the sequence.

6 CASE STUDIES

To demonstrate the effectiveness of PKUWA in protecting WebAssembly applications, we use realistic vulnerabilities besides the attack shown in *Listing 1*, which is mitigated by our guardian pages described in Section 4.3. The studied vulnerabilities are HeartBleed, PDFAlto, and libjpeg. We select these CVEs due to two reasons. First, as these CVEs have been extensively analyzed in the literature on memory isolation or related attacks [17–19, 21, 46, 47, 55, 65], we can make PKUWA conceptually comparable with prior isolation mechanisms. Second, as there are few applications written directly in WebAssembly, we instead select CVEs in other languages and try to protect WebAssembly binaries that are compiled from these languages against the vulnerabilities. Due to the page limit, we only discuss the case of HeartBleed in this section and leave other cases in Appendix A.

6.1 HeartBleed Vulnerability

HeartBleed (CVE-2014-0160) is a severe vulnerability in the OpenSSL library [50] that exposes server data to malicious clients through SSL connections. A client can craft a request that makes the server return more data than expected in the response. This extra data may include sensitive information, such as private keys and session keys. The root cause of HeartBleed is that OpenSSL fails to validate the argument (the length of the data to be copied) for a memcpy function.

Listing 3 and Figure 3 show a code snippet and a demonstration for HeartBleed. The code is from the official SSL implementation [6], with some semantic-preserving changes for presentation. The variable bp points to the start of the buffer, which stores the response data copied from pl (Line 25 and Figure 3). The length argument (length_arg) in the request *should be (but is not checked) equal to* the actual size of the payload (payload_len in Figure 3). Therefore, a malicious client can specify a large length argument that exceeds payload_len. Line 25 will copy extra data that *follows* payload *in the server's memory space* (the sensitive data in Figure 3). These stolen data will be sent to the client in the response. As Figure 3 shows, if the client sets length_arg to payload_len+1000, the server will return sensitive data in the response.

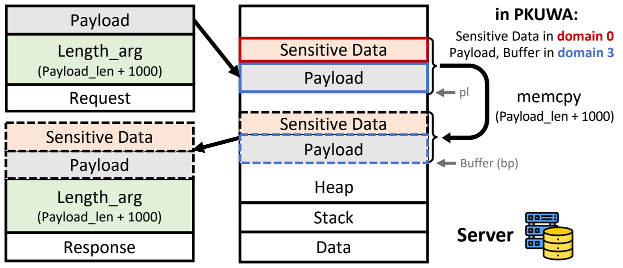

Figure 3: HeartBleed vulnerability (CVE-2014-0160) and use PKUWA to protect sensitive data from it.

```
1  int dtls1_read_bytes(SSL *s, ...) {
2      ...
3      dtls1_process_heartbeat(s);
4      ...
5  }
6  int dtls1_process_heartbeat(SSL *s) {
7      unsigned char *p = &s->s3->rrec.data[0], *pl;
8      unsigned int length_arg, padding = 16;
9      /* Read type and payload length */
10     unsigned short hbtype = *p++;
11     // Read paylod from p
12     n2s(p, length_arg);
13     pl = p;
14     ...
15     if (hbtype == TLS1_HB_REQUEST) {
16         unsigned char *buffer, *bp;
17         // Allocate memory for the response
18         buffer = OPENSSL_malloc(1+2+length_arg+padding);
19         bp = buffer;
20
21         /* Response type, length and copy payload */
22         *bp++ = TLS1_HB_RESPONSE;
23         s2n(length_arg, bp);
24         // Copy payload data to bp
25         memcpy(bp, pl, length_arg);
26         ...
27     }
28     ...
29 }
```

Listing 3: HeartBleed vulnerability in OpenSSL.

The HeartBleed vulnerability poses a greater threat in WebAssembly than in native platforms. This is because, in native platforms, the memory allocated dynamically is not always contiguous. For Listing 3, the memory space after pl may not be allocated at line 25. Copying from such an address will cause SIGSEGV. However, in WebAssembly, the objects in the linear memory are contiguous by default (**P2** in Section 2.1). The memory after pl is always allocated. Therefore, the attacker can access additional data without raising any exceptions.

```
1 int domain = domain_create(3);
2 ...
3 PKU_CALL_REGISTER(domain, dtls1_read_bytes);
4 ...
5 PKU_CALL(dtls1_read_bytes(ssl,
6         SSL3_RT_APPLICATION_DATA, buf, len, peek));
```

Listing 4: Protecting sensitive data from HeartBleed with PKUWA.

Listing 4 demonstrates how a developer can use PKUWA to isolate the overflow in `dtls1_read_bytes`. The first step is to create a domain (Domain 2) with `domain_create` (Line 1), and then use `PKU_CALL_REGISTER` to associate `dtls1_read_bytes` with the domain (line 3). This also detaches `dtls1_read_bytes` from the original Domain 0, which means that `dtls1_read_bytes` can no longer access the data in Domain 0. All the variables allocated in `dtls1_read_bytes` (`s`, `pl`, `bp`) are stored in Domain 2. As Figure 3 illustrates, the `payload` and `buffer` are separated from the sensitive data on the server. When the attacker tries to read extra data after `payload` in Figure 3, the payload exits Domain 2 while other sensitive data remains in Domain 0. Therefore, `memcpy` at line 25 in Listing 3 can only access the data within Domain 2. Thus PKUWA can prevent HeartBleed vulnerability.

7 EVALUATION

We presented the design and implementation of PKUWA, a system that protects the linear memory of WebAssembly, in Section 6. In this section, we evaluate the performance of PKUWA by answering the following research questions:

RQ1: How does the DILM reduce the runtime and memory overhead compared to applying MPK isolation models of native applications directly to WebAssembly?
RQ2: What is the overhead of domain switching caused by PKUWA? How do other factors, such as the number of domain switches, the number of isolated domains, and the size of allocated memory, affect the overhead?
RQ3: How effective are the optimizations of lazy-domain free and memory swaps in reducing overhead?

7.1 Implementation

We developed a PKUWA prototype based on Wasmtime [72], the official WebAssembly runtime. We also created a static library for both C/C++ and Rust to offer user-level APIs of Section 5.1. For Wasmtime, we modified its instruction parser, instantiator, Cranelift compiler, and WASI interfaces to support `RDPKRU` and `WRPKRU`, and implemented the guard pages in Section 5.3.

To implement the DILM model, we added three WASI calls in the WASI library to manage the protection keys. The memory protection keys must be managed by the OS, but WebAssembly applications cannot access OS resources. The three WASI calls are `pkey_allocate`, `pkey_free`, and `pkey_protect`. `pkey_allocate` assigns a new memory protection key to domains. `pkey_free` releases a memory protection key. `pkey_protect` protects a given memory region with a memory protection key.

7.2 Experimental Setup

We conducted the experiments in a Proxmox virtual machine with 8-core vCPU, 16GB memory, and Ubuntu 18.04 LTS with Linux kernel 4.15. We evaluated our system using a microbenchmark and a macrobenchmark. For the microbenchmark, we designed multiple small programs to measure the efficiency of domain switches precisely. For the macrobenchmark, we used ten open-source WebAssembly projects with their official test suites. The projects include bzip2 [10] (a popular data compression tool on modern Unix/Linux systems, 5164 LOC), eSpeak [23] (an open source speech synthesizer, 13119 LOC), Face-detection [44] (a face detection application based on deep learning, 7242 LOC), GNU-Chess [11] (an intelligent chess game, 7974 LOC), WhiteDB [75] (a memory database, 15837 LOC), tsf [37] (a typed stream format in WebAssembly, 23459 LOC), cjpeg [32] (compress an image file, 8948 LOC), djpeg [32] (decompress a JPEG file, 10225 LOC), coreutils [27] (utilities of the GNU operating system, 136610 LOC), and ripgrep [25] (a line-oriented search tool, 38327 LOC). Our macrobenchmark covered both computational and I/O-intensive applications from embedding and standalone scenarios.
Comparison Baselines. We compare PKUWA with two baselines: **No-Switch** and **Naive-Isolation**. No-Switch represents existing WebAssembly applications that do not use memory isolation. This baseline has no performance overhead (*performance upper-bound*) and no defenses (*security lower-bound*). Naive-Isolation represents a simple implementation of memory isolation using the `mmap`, `mprotect` and `pkey_mprotect` system calls. These are the only system calls that can manage memory permission on native platforms. We implement Naive-Isolation in WebAssembly by adding new WASI interfaces. These interfaces invoke the corresponding system calls. Note that `mprotect` and `pkey_mprotect` require the input pointer to be page-aligned (see Section 2.3). Therefore, when the developer wants to protect a memory object with Naive-Isolation, she must call `mmap`, which will forcibly extend the linear memory by at least one page. This approach increases linear memory size.

There are several approaches, such as libmpk [55] and similar approaches [31, 35, 39, 56, 65], that are similar in concept to the Naive-Isolation baseline in the page-based protection model. The main runtime and memory overhead of using MPK directly in WebAssembly are primarily due to applying the page-based model in linear memory. Consequently, libmpk and its equivalents yield similar outcomes to our Naive-Isolation baseline.

For each research question, we repeat the experiments ten times and report the average performance. We then run the Wilcoxon test to ensure the statistical significance of the results. All our results for comparison achieve $p < 0.05$.

7.3 Effectiveness of DILM

We compare DILM with Naive-Isolation and No-Switch by testing their performance and memory usage on official test suites of our benchmark. We record execution time and peak memory consumption and report the relative overhead of PKUWA and Naive-Isolation compared to No-Switch. We note that the peak memory consumption occurs at the end of execution since WebAssembly linear memory only grows monotonically. We report the relative overhead of PKUWA and Naive-Isolation with respect to No-Switch.

For PKUWA, we use a function-based protection model and randomly select 5%, 25%, 50%, 75%, and 100% of the functions to simulate the domain switch. We modify the WebAssembly runtime to implement this simulation. When the runtime executes a function call, it randomly puts the callee into a different domain and performs the call gate logic described in Section 4.1.2 with a probability of 5%, 25%, 50%, 75%, and 100%. These levels represent different workloads and assumptions for PKUWA and Naive-Isolation.

For Naive-Isolation, we use a page-based protection model and randomly select 5%, 25%, 50%, 75%, and 100% of the memory objects allocated by `malloc` to apply MPK protection. We dynamically hook the `malloc` API in the WebAssembly runtime to achieve this. For each `malloc`, we randomly decide if this memory object needs to be protected with a probability of 5%, 25%, 50%, 75%, and 100%. If so, we use `mmap` to allocate a new page and `pkey_mprotect` to set the permission with a protection key.

One challenge for evaluating Naive-Isolation is permission switching. To achieve isolation, we should use different MPK keys with minimal permissions for protected memory, which is impossible since it requires a lot of manual work. So we simulate the switches in our experiments. For Naive-Isolation, we do not change the `PKRU` register when we call `pkey_mprotect`, favoring it by ignoring switch costs. However, the main bottleneck for Naive-Isolation is on **`mmap`** and **`pkey_mprotect`**, so our protocol doesn't affect the conclusion.

Runtime Overhead. Figure 4 compares the latency overhead of PKUWA and the Naive-Isolation baseline for different percentages of protected memory. The y-axis represents the overhead percentage and the x-axis represents the application name. For PKUWA, the average overhead percentage for 5%, 25%, 50%, and 75% are 1.77%, 2.86%, 4.04%, and 5.04%, respectively. The maximum overhead for 5%, 25%, 50%, and 75% are 4.13%, 6.94%, 7.54%, and 8.32%, respectively. For the worst-case scenario (100%), the average overhead is 6.72% and the maximum overhead is 9.53%. For all cases, the overhead of PKUWA is less than 10%, which is acceptable for most applications.

For the Naive-Isolation baseline, the average overhead percentage for 5%, 25%, 50%, and 75% are 18.79%, 34.48%, 46.20%, and 54.59%, respectively. The average overhead for the worst case (100%) is 64.01% and the maximum overhead is 85.29%. On average, the overhead of PKUWA is 9.89× lower than that of Naive-Isolation baseline.

Addressing Linear Memory Bloating. We evaluate PKUWA, No-Switch, and Naive-Isolation on ten open-source WASM apps to compare their memory usage and see how well PKUWA avoids memory bloating. Table 2 shows the results. PKUWA and No-Switch use the same memory for all apps, so we only report PKUWA's memory usage with 100% protection level and omit No-Switch's results. For Naive-Isolation, we only report 5% and 100% protection levels due to space limit. Other configurations are between these levels. We see that under 5% protection levels, Naive-Isolation uses 1.63× more memory than PKUWA and No-Switch on average. This is because MPK protects memory at page level, so Naive-Isolation has to allocate a new page with `mmap` for each object. Usually, the object does not fill the page, causing memory waste at the end of pages. PKUWA manages memory in a fine-grained way. It can use free chunks within domains to protect objects, avoiding the memory blotaing problem. Due to space limitations, we have included the results for the number of inter-domain calls and copy operations in our GitHub repository.

Table 2: The maximum memory usage (in KB) of various applications is shown in the following table.

Application	PKUWA(100%)	Naive(5%)	Naive(100%)
bzip2	7360	11,412 (1.55×)	39,360 (5.35×)
espeak	320	496 (1.55×)	1,964 (6.14×)
facedetection	1088	1,788 (1.64×)	7,248 (6.66×)
gnuchess	64	102 (1.59×)	476 (7.44×)
whitedb	257,408	441,472 (1.72×)	1,953,152 (7.59×)
tsf	640	1,340 (2.09×)	6,916 (10.81×)
cjpeg	896	1,352 (1.51×)	7,056 (7.86×)
djpeg	128	196 (1.53×)	848 (6.63×)
coreutils	384	580 (1.51×)	3,184 (8.29×)
ripgrep	1,280	2,060 (1.61×)	11,280 (8.81×)

Answer to RQ1: For all the ten applications in our macrobenchmark, PKUWA achieves low overheads of 1.77%, 2.86%, 4.04%, and 5.04% for isolating 5%, 25%, 50%, and 75% functions, respectively. Compared with the Naive-Isolation baseline, PKUWA reduces the overhead by 9.89×. PKUWA does not incur any memory overhead compared with No-Switch, while the Naive-Isolation may increase the linear memory by more than 10× in the worst case.

7.4 Domain Switch Overhead

We conducted a controlled experiment to measure the overhead of domain switches. We compared the overhead of one domain switch to No-Switch and Naive-Isolation baselines. We also studied how switching overhead is affected by other factors such as the number of `PKU_CALL`, isolated domains, and allocated memory size. We constructed two programs: a target application with one `PKU_CALL` and two domain switches, and a No-Switch baseline application. Specifically, for the target application, we design a function `foo` that only contains one *add* operation. Then we put `foo` in Domain 2 and call `foo` from another function in Domain 0. Note that Domain 1 is a privileged domain and cannot be used by the application. The No-Switch application is an original WebAssembly application that only contains `foo`. We compared the time to call `foo` between the target application and the No-Switch application using direct and indirect call methods.

For reference, we also evaluate the average time of system calls and WASI calls. We use a representative system call, `open`, to measure the average time for different cases. `open` is a simple system call that opens a file. We choose `open` because it has an official implementation in both system calls and WASI calls, thus we can use it to compare PKUWA between system calls and WASI calls.

Table 3 shows the average time and error range of calling a function in nanoseconds. We can see that PKUWA is slower than the No-Switch baseline but much faster than WASI call. PKUWA adds 241ns and 276ns to direct call (comparing ① and ③) and indirect call (comparing ② and ④), respectively. This overhead is 7.54×

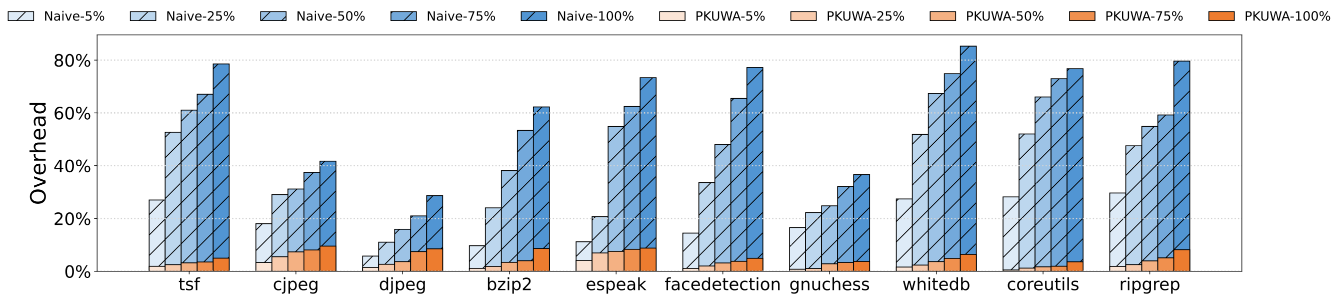

Figure 4: Comparison of latency overhead on the marcobenchmark between PKUWA (upper) and Naive-Isolation (lower). For both techniques, we report the ratio of additional latency over the No-Switch baseline.

Table 3: The average running time for different call types.

Call Type	Cost (ns)
① Direct Call (No-Switch)	39 ± 4.05
② Indirect Call (No-Switch)	46 ± 3.56
③ Direct Call (PKUWA)	280 ± 9.18
④ Indirect Call (PKUWA)	322 ± 7.74
⑤ open Syscall	861 ± 34.53
⑥ open WASI	2429 ± 70.54

smaller than the open WASI call (⑥) and 2.67× smaller than the open system call (⑥).

We investigate the effect of three factors on the domain switch overhead: the number of `PKU_CALL` instructions (#Switches), the number of isolated domains (#Domains), and the size of allocated memory. We vary #Switches from 0 to 200K, #Domains from one to fifteen, and the allocated memory size from 0 to 8MB. We find that these factors do not influence the switching cost. The switching cost of PKUWA is determined by the `WRPKRU` instruction, which has a constant execution time.

Answer to RQ2: The switch overhead is on the nanosecond level and is not affected by the total number of domain switches or the number of isolated domains.

7.5 Ablation Study on Optimizations

We evaluate the performance of the optimizations in Sections 4.1.1 and 4.2: lazy domain free and flexible memory swap. Figure 5 compares the two optimizations.

We test the impact of lazy-free by creating and freeing a new isolated domain repeatedly. We use a simple function `bar` that creates a domain with `domain_create` and frees it with `domain_free`. We vary the number of times we run `bar` from 1 to 1000. The left part of Figure 5 shows the average latency. The figure shows that without lazy-free, the application spends a lot of time calling WASI interfaces. With lazy free, PKUWA only needs to clear the memory of the freed domain and avoid slow WASI calls. According to Figure 5, lazy-free can reduce the latency by 3.43×.

We evaluated flexible memory swap's effectiveness by measuring a WebAssembly application's memory usage. We allocated 4MB of memory in one domain, then freed it. In another domain, we allocated new memory ranging from 64KB to 4MB. The memory

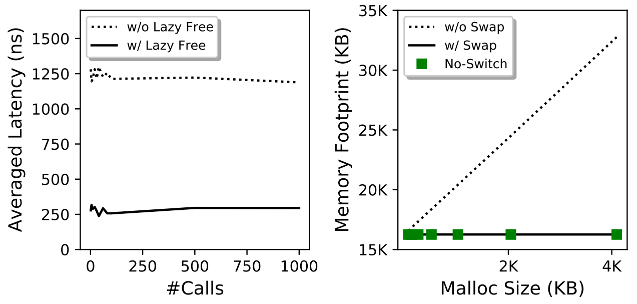

Figure 5: Optimization effect of domain lazy free and memory swap.

footprint is shown in Figure 5. Without swap, memory usage increased linearly with allocated memory size, but with swap, it remained consistent. Freed memory is reused by other domains, eliminating the need for new memory allocation.

Answer to RQ3: The two optimizations, lazy domain free and flexible memory swap, can reduce the overall latency and memory footprint.

8 DISCUSSION

More Protection Keys. PKUWA is based on MPK and can use 16 protection keys. `libmpk` can virtualize more protection keys than MPK's hardware support. For WebAssembly applications that require more than 16 protection keys, developers can combine `libmpk` with PKUWA for better scalability.

Alternatives to Intel MPK. Our prototype of PKUWA uses Intel MPK, but the DILM model is not hardware-specific. It works with any memory protection scheme similar to MPK, such as those in ARM, RISC-V, PowerPC, and Itanium CPUs [5, 13, 16, 57].

Granularity. Enclosure [26] defines Object Granularity Isolation (OGI) which places each object in a separate isolation domain. However, OGI can cause significant domain switch overhead as each object requires two domain switches (one to start and one to end the access). Hence, we have chosen to use function-level isolation as it is easier to manage and better suited to page-based hardware isolation mechanisms. Functions group objects together in the same domain, which allows them to be stored on the same page. Accessing these objects only requires two domain switches before and after the function call.

9 RELATED WORK

Software Based Isolation. There are several approaches that leverage software techniques, such as program analysis techniques [1, 26, 69], runtimes of dynamic languages [9, 22, 29, 77], and OS-level features [8, 45]. However, these techniques are orthogonal to PKUWA and cannot be directly utilized in WASM.

MPK-based Memory Isolation. MPK-based techniques [31, 35, 39, 55, 56, 65] can reduce the overhead of context switches between sandbox and native binaries. `libmpk` provides a secure software abstraction to use MPK [55]. ERIM provides a more secure mechanism for to use of MPK [65]. PKU-safe [39] uses MPK to protect heaps. EPK [31] enhances the efficiency of MPK in native applications. Some other techniques are proposed to improve the security of native applications [35, 56, 57, 68]. All these techniques adopt the page-based isolation model, which does not fit the linear memory of WebAssembly.

Other Hardware-Enforced Memory Isolation. Researchers have developed various hardware extensions to ensure memory isolation on main processors, such as Mondrian Memory Protection (MMP) [76] and CODOMS [66]. Intel X86 ISA has instructions for safe memory access [24, 51] and RISC instruction (CHERI) allows for fine-grained separation and memory isolation [73]. The DILM can be applied to other hardware implementations, along with being designed for MPK.

10 CONCLUSION

When using MPK in WebAssembly, it can be challenging due to potential memory bloating and reduced efficiency when trying to directly use the MPK APIs. This is because of differences between the linear memory of WebAssembly and the page-based protection model of MPK. To overcome this issue, we propose the DILM model to create isolation domains at the function level in WebAssembly. Our evaluation shows that PKUWA provides strong isolation guarantees while only having an average runtime overhead of 1.77% and negligible memory overhead.

ACKNOWLEDGMENTS

We sincerely thank our Shepherd and all the anonymous reviewers for their valuable comments. This work was partly supported by the National Key Research and Development Program (No. 2022YFB4501802) and the National Natural Science Foundation of China (No. 62172009, No. 62141208).

REFERENCES

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2005. Control-Flow Integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security* (Alexandria, VA, USA) *(CCS '05)*. Association for Computing Machinery, New York, NY, USA, 340–353. https://doi.org/10.1145/1102120.1102165
[2] Bytecode Alliance. 2016. WebAsssembly Design - Security - Memory Safety. Retrieved April 12, 2023 from https://github.com/WebAssembly/design/blob/master/Security.md#memory-safety
[3] Bytecode Alliance. 2023. Cranelift. https://github.com/bytecodealliance/wasmtime/tree/main/cranelift
[4] Bytecode Alliance. 2023. Cranelift Document. https://github.com/bytecodealliance/wasmtime/blob/main/cranelift/docs/index.md
[5] ARM. 2001. ARM Developer Suite Developer Guide. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0056d/BABBJAED.html
[6] OpenSSL Project Authors. 2021. OpenSSL. https://www.openssl.org/
[7] Nataliia Bielova. 2013. Survey on JavaScript security policies and their enforcement mechanisms in a web browser. *The Journal of Logic and Algebraic Programming* 82, 8 (2013), 243–262.
[8] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. 2008. Wedge: Splitting Applications into Reduced-Privilege Compartments. In *5th USENIX Symposium on Networked Systems Design and Implementation (NSDI 08)*. USENIX Association, San Francisco, CA, 309–322. https://www.usenix.org/conference/nsdi-08/wedge-splitting-applications-reduced-privilege-compartments
[9] Zack Bloom. 2020. Cloud Computing without Containers. https://blog.cloudflare.com/cloud-computing-without-containers/
[10] bzip2 and libbzip2. 2022. https://www.sourceware.org/bzip2
[11] GNU Chess. 2022. https://www.gnu.org/software/chess
[12] Bart Coppens, Ingrid Verbauwhede, Koen De Bosschere, and Bjorn De Sutter. 2009. Practical Mitigations for Timing-Based Side-Channel Attacks on Modern X86 Processors. In *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy (SP '09)*. IEEE Computer Society, USA, 45–60. https://doi.org/10.1109/SP.2009.19
[13] Intel Corporation. 2000. Intel IA-64 architecture software developer's manual, revision 1.1.
[14] Intel Corporation. 2016. Intel(R) 64 and IA-32 Architectures Software Developer's Manual. https://software.intel.com/en-us/articles/intel-sdm
[15] Intel Corporation. 2017. Intel Software Guard Extensions (Intel SGX) SDK. https://software.intel.com/sgx-sdk
[16] IBM Corporation. 2017. Power ISA version 3.0b.
[17] The MITRE Corporation. 2018. CVE-2018-14498. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-14498
[18] The MITRE Corporation. 2018. CVE-2018-19664. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-19664
[19] The MITRE Corporation. 2021. CVE-2021-46822. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-46822
[20] Crispin Cowan, Calton Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. 1998. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Proceedings of the 7th Conference on USENIX Security Symposium - Volume 7* (San Antonio, Texas) *(SSYM'98)*. USENIX Association, USA, 5.
[21] NATIONAL VULNERABILITY DATABASE. 2022. CVE-2022-32324. https://nvd.nist.gov/vuln/detail/CVE-2022-32324
[22] Úlfar Erlingsson, Silicon Valley, Martín Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. 2006. XFI: Software Guards for System Address Spaces. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7* (Seattle, WA) *(OSDI '06)*. USENIX Association, USA, 6.
[23] eSpeak text to speech. 2022. http://espeak.sourceforge.net
[24] Tommaso Frassetto, Patrick Jauernig, Christopher Liebchen, and Ahmad-Reza Sadeghi. 2018. IMIX: In-Process Memory Isolation Extension. In *Proceedings of the 27th USENIX Conference on Security Symposium* (Baltimore, MD, USA) *(SEC'18)*. USENIX Association, USA, 83–97.
[25] Andrew Gallant. 2023. ripgrep. https://github.com/BurntSushi/ripgrep
[26] Adrien Ghosn, Marios Kogias, Mathias Payer, James R. Larus, and Edouard Bugnion. 2021. Enclosure: Language-Based Restriction of Untrusted Libraries. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 255–267. https://doi.org/10.1145/3445814.3446728
[27] GNU. 2016. Coreutils. https://www.gnu.org/software/coreutils/coreutils.html
[28] Li Gong. 2009. Java security: a ten year retrospective. In *2009 Annual Computer Security Applications Conference*. IEEE, 395–405.
[29] Google. 2020. Chromium V8 isolates. https://chromium.googlesource.com/chromium/src/+/master/third_party/blink/renderer/bindings/core/v8/V8BindingDesign.md#Isolate
[30] Mel Gorman. 2023. Process Address Space. https://www.kernel.org/doc/gorman/html/understand/understand007.html
[31] Jinyu Gu, Hao Li, Wentai Li, Yubin Xia, and Haibo Chen. 2022. EPK: Scalable and Efficient Memory Protection Keys. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA, 609–624. https://www.usenix.org/conference/atc22/presentation/gu-jinyu
[32] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. 2001. MiBench: A Free, Commercially Representative Embedded Benchmark Suite. In *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop (WWC '01)*. IEEE Computer Society, USA, 3–14.
[33] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 2017. Bringing the Web up to Speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 185–200. https://doi.org/10.1145/3062341.3062363
[34] Niranjan Hasabnis, Ashish Misra, and R. Sekar. 2012. Light-Weight Bounds Checking. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization* (San Jose, California) *(CGO '12)*. Association for Computing Machinery, New York, NY, USA, 135–144. https://doi.org/10.1145/2259016.2259034

[35] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L. Scott, Kai Shen, and Mike Marty. 2019. Hodor: Intra-Process Isolation for High-Throughput Data Plane Libraries. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference* (Renton, WA, USA) *(USENIX ATC '19)*. USENIX Association, USA, 489–503.

[36] Aaron Hilbig, Daniel Lehmann, and Michael Pradel. 2021. An Empirical Study of Real-World WebAssembly Binaries: Security, Languages, Use Cases. In *Proceedings of the Web Conference 2021* (Ljubljana, Slovenia) *(WWW '21)*. Association for Computing Machinery, New York, NY, USA, 2696–2708. https://doi.org/10.1145/3442381.3450138

[37] JetStream2. 2022. https://browserbench.org/JetStream

[38] The kernel development community. 2023. Memory Protection Keys. https://www.kernel.org/doc/html/latest/core-api/protection-keys.html

[39] Paul Kirth, Mitchel Dickerson, Stephen Crane, Per Larsen, Adrian Dabrowski, David Gens, Yeoul Na, Stijn Volckaert, and Michael Franz. 2022. PKRU-Safe: Automatically Locking down the Heap between Safe and Unsafe Languages. In *Proceedings of the Seventeenth European Conference on Computer Systems* (Rennes, France) *(EuroSys '22)*. Association for Computing Machinery, New York, NY, USA, 132–148. https://doi.org/10.1145/3492321.3519582

[40] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *2019 IEEE Symposium on Security and Privacy (SP)*. 1–19. https://doi.org/10.1109/SP.2019.00002

[41] Koen Koning, Xi Chen, Herbert Bos, Cristiano Giuffrida, and Elias Athanasopoulos. 2017. No Need to Hide: Protecting Safe Regions on Commodity Hardware. In *Proceedings of the Twelfth European Conference on Computer Systems* (Belgrade, Serbia) *(EuroSys '17)*. Association for Computing Machinery, New York, NY, USA, 437–452. https://doi.org/10.1145/3064176.3064217

[42] Volodymyr Kuznetsov, Laszlo Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. 2014. Code-Pointer Integrity. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Broomfield, CO, 147–163. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/kuznetsov

[43] Daniel Lehmann, Johannes Kinder, and Michael Pradel. 2020. Everything Old is New Again: Binary Security of WebAssembly. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, USA, 217–234. https://www.usenix.org/conference/usenixsecurity20/presentation/lehmann

[44] libfacedetection. 2022. https://github.com/ShiqiYu/libfacedetection

[45] James Litton, Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, Bobby Bhattacharjee, and Peter Druschel. 2016. Light-Weight Contexts: An OS Abstraction for Safety and Performance. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 49–64. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/litton

[46] Yutao Liu, Tianyu Zhou, Kexin Chen, Haibo Chen, and Yubin Xia. 2015. Thwarting Memory Disclosure with Efficient Hypervisor-Enforced Intra-Domain Isolation. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (Denver, Colorado, USA) *(CCS '15)*. Association for Computing Machinery, New York, NY, USA, 1607–1619. https://doi.org/10.1145/2810103.2813690

[47] Patrice Lopez. 2022. Pdfalto. https://github.com/kermitt2/pdfalto/issues/144

[48] mend.io. 2017. WHAT ARE THE MOST SECURE PROGRAMMING LANGUAGES? https://www.mend.io/most-secure-programming-languages/

[49] Alexandra E. Michael, Anitha Gollamudi, Jay Bosamiya, Evan Johnson, Aidan Denlinger, Craig Disselkoen, Conrad Watt, Bryan Parno, Marco Patrignani, Marco Vassena, and Deian Stefan. 2023. MSWasm: Soundly Enforcing Memory-Safe Execution of Unsafe Code. *Proc. ACM Program. Lang.* 7, POPL, Article 15 (jan 2023), 30 pages. https://doi.org/10.1145/3571208

[50] MITRE. 2014. CVE-2014-0160. https://nvd.nist.gov/vuln/detail/CVE-2014-0160

[51] Lucian Mogosanu, Ashay Rane, and Nathan Dautenhahn. 2018. Microstache: A lightweight execution context for in-process safe region isolation. In *Research in Attacks, Intrusions, and Defenses: 21st International Symposium, RAID 2018, Heraklion, Crete, Greece, September 10-12, 2018, Proceedings 21*. Springer, 359–379.

[52] Kit Murdock, David Oswald, Flavio D. Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. 2020. Plundervolt: Software-based Fault Injection Attacks against Intel SGX. In *2020 IEEE Symposium on Security and Privacy (SP)*. 1466–1482. https://doi.org/10.1109/SP40000.2020.00057

[53] Nicholas Nethercote and Julian Seward. 2007. How to Shadow Every Byte of Memory Used by a Program. In *Proceedings of the 3rd International Conference on Virtual Execution Environments* (San Diego, California, USA) *(VEE '07)*. Association for Computing Machinery, New York, NY, USA, 65–74. https://doi.org/10.1145/1254810.1254820

[54] Hewlett Packard. 1994. PA-RISC 1.1 architecture and instruction set reference manual, third edition.

[55] Soyeon Park, Sangho Lee, Wen Xu, Hyungon Moon, and Taesoo Kim. 2019. Libmpk: Software Abstraction for Intel Memory Protection Keys (Intel MPK). In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference* (Renton, WA, USA) *(USENIX ATC '19)*. USENIX Association, USA, 241–254.

[56] David Schrammel, Samuel Weiser, Richard Sadek, and Stefan Mangard. 2022. Jenny: Securing Syscalls for PKU-based Memory Isolation Systems. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 936–952. https://www.usenix.org/conference/usenixsecurity22/presentation/schrammel

[57] David Schrammel, Samuel Weiser, Stefan Steinegger, Martin Schwarzl, Michael Schwarz, Stefan Mangard, and Daniel Gruss. 2020. Donky: Domain Keys - Efficient in-Process Isolation for RISC-V and X86. In *Proceedings of the 29th USENIX Conference on Security Symposium (SEC'20)*. USENIX Association, USA, Article 95, 18 pages.

[58] David Sehr, Robert Muth, Cliff Biffle, Victor Khimenko, Egor Pasko, Karl Schimpf, Bennet Yee, and Brad Chen. 2010. Adapting Software Fault Isolation to Contemporary CPU Architectures. In *Proceedings of the 19th USENIX Conference on Security* (Washington, DC) *(USENIX Security'10)*. USENIX Association, USA, 1.

[59] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference* (Boston, MA) *(USENIX ATC'12)*. USENIX Association, USA, 28.

[60] Hovav Shacham. 2007. The Geometry of Innocent Flesh on the Bone: Return-into-Libc without Function Calls (on the X86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security* (Alexandria, Virginia, USA) *(CCS '07)*. Association for Computing Machinery, New York, NY, USA, 552–561. https://doi.org/10.1145/1315245.1315313

[61] Quentin Stiévenart, Coen De Roover, and Mohammad Ghafari. 2022. Security Risks of Porting C Programs to WebAssembly. In *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing* (Virtual Event) *(SAC '22)*. Association for Computing Machinery, New York, NY, USA, 1713–1722. https://doi.org/10.1145/3477314.3507308

[62] Mincheol Sung, Pierre Olivier, Stefan Lankes, and Binoy Ravindran. 2020. Intra-Unikernel Isolation with Intel Memory Protection Keys. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (Lausanne, Switzerland) *(VEE '20)*. Association for Computing Machinery, New York, NY, USA, 143–156. https://doi.org/10.1145/3381052.3381326

[63] PaX Team. 2002. PaX Address Space Layout Randomization (ASLR). https://pax.grsecurity.net/docs/aslr.txt

[64] Stephen Turner. 2014. Security vulnerabilities of the top ten programming languages: C, Java, C++, Objective-C, C#, PHP, Visual Basic, Python, Perl, and Ruby. *Journal of Technology Research* 5 (2014), 1.

[65] 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 *Proceedings of the 28th USENIX Conference on Security Symposium* (Santa Clara, CA, USA) *(SEC'19)*. USENIX Association, USA, 1221–1238.

[66] Lluïs Vilanova, Muli Ben-Yehuda, Nacho Navarro, Yoav Etsion, and Mateo Valero. 2014. CODOMs: Protecting Software with Code-Centric Memory Domains. In *Proceeding of the 41st Annual International Symposium on Computer Architecture* (Minneapolis, Minnesota, USA) *(ISCA '14)*. IEEE Press, 469–480.

[67] Alexios Voulimeneas, Jonas Vinck, Ruben Mechelinck, and Stijn Volckaert. 2022. You Shall Not (by)Pass! Practical, Secure, and Fast PKU-Based Sandboxing. In *Proceedings of the Seventeenth European Conference on Computer Systems* (Rennes, France) *(EuroSys '22)*. Association for Computing Machinery, New York, NY, USA, 266–282. https://doi.org/10.1145/3492321.3519560

[68] Alexios Voulimeneas, Jonas Vinck, Ruben Mechelinck, and Stijn Volckaert. 2022. You Shall Not (by)Pass! Practical, Secure, and Fast PKU-Based Sandboxing. In *Proceedings of the Seventeenth European Conference on Computer Systems* (Rennes, France) *(EuroSys '22)*. Association for Computing Machinery, New York, NY, USA, 266–282. https://doi.org/10.1145/3492321.3519560

[69] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. 1993. Efficient Software-Based Fault Isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles* (Asheville, North Carolina, USA) *(SOSP '93)*. Association for Computing Machinery, New York, NY, USA, 203–216. https://doi.org/10.1145/168619.168635

[70] Wenwen Wang. 2022. How Far We've Come - A Characterization Study of Standalone WebAssembly Runtimes. In *2022 IEEE International Symposium on Workload Characterization (IISWC)*. USA, 228–241. https://doi.org/10.1109/IISWC55918.2022.00028

[71] Xiaoguang Wang, SengMing Yeoh, Pierre Olivier, and Binoy Ravindran. 2020. Secure and Efficient In-Process Monitor (and Library) Protection with Intel MPK. In *Proceedings of the 13th European Workshop on Systems Security* (Heraklion, Greece) *(EuroSec '20)*. Association for Computing Machinery, New York, NY, USA, 7–12. https://doi.org/10.1145/3380786.3391398

[72] Wasmtime. 2020. A small and efficient runtime for WebAssembly & WASI. https://wasmtime.dev/

[73] Robert N. M. Watson, Jonathan Woodruff, Peter G. Neumann, Simon W. Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, Steven J. Murdoch, Robert Norton, Michael Roe, Stacey Son, and Munraj Vadera. 2015. CHERI: A Hybrid Capability-System Architecture for Scalable

Software Compartmentalization. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy (SP '15)*. IEEE Computer Society, USA, 20–37. https://doi.org/10.1109/SP.2015.9

[74] Mario Werner, Thomas Unterluggauer, Lukas Giner, Michael Schwarz, Daniel Gruss, and Stefan Mangard. 2019. ScatterCache: Thwarting Cache Attacks via Cache Set Randomization. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 675–692. https://www.usenix.org/conference/usenixsecurity19/presentation/werner

[75] WhiteDB. 2022. http://whitedb.org

[76] Emmett Witchel, Josh Cates, and Krste Asanović. 2002. Mondrian Memory Protection. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, California) *(ASPLOS X)*. Association for Computing Machinery, New York, NY, USA, 304–316. https://doi.org/10.1145/605397.605429

[77] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. 2009. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *2009 30th IEEE Symposium on Security and Privacy*. 79–93. https://doi.org/10.1109/SP.2009.25

A MORE CASE STUDIES BESIDES § 6

In this section, we discuss more case studies about how PKUWA can prevent memory vulnerabilities.

A.1 PDFAlto

Our second case is the heap overwrite vulnerability caused by overflow in the latest stable version of PDFAlto v0.4 (CVE-2022-32324). Listing 5 shows the code [1]. The code constructs a file path (`thePath`) and its directory length (`dirname_length`) from line 4 to line 9. Then it copies the directory path to `dirname` from line 11 to line 14. The function `strncat(char *dest, const char *src, size_t n)` appends `n` characters of `src` to `dest`. The overflow happens at line 13. This line assumes that `dirname` is `'\0'`, so it uses `strncat` to fill the space. However, at line 12, `malloc` does not initialize the memory, so `dirname` may not be `'\0'`. The `strncat` may start from the middle of `dirname`, overflow the buffer, and overwrite nearby heap objects.

```
1  int initPath()
2  {
3      ...
4      char* thePath;
5      thePath = (char*)malloc(theLength + 1);
6      int dirname_length;
7      wai_getExecutablePath(thePath, theLength,
8                            &dirname_length);
9      thePath[theLength] = '\0';
10
11     char *dirname;
12     dirname = (char*)malloc(dirname_length + 1);
13     strncat(dirname, thePath, dirname_length);
14     dirname[dirname_length] = '\0';
15     ...
16 }
```

Listing 5: The heap overflow vulnerability in PDFAlto v0.4.

Similar to HeartBleed, PKUWA can help developers isolate the function that is vulnerable to a heap overflow and prevent it from accessing sensitive data. For example, developers can create a new isolated domain that only contains the data of the function `initPath`. This ensures that `strncat` can only overwrite the data within the domain, and not the data outside of it.

Listing 6 shows a demo code. The developer creates an isolated domain (line 1) and registers the vulnerable code with it (line 3).

```
1 int domain = domain_create(0);
2 ...
3 PKU_CALL_REGISTER(domain, initPath);
4 ...
5 PKU_CALL(initPath());
```

Listing 6: Protecting sensitive data from heap buffer overflow with PKUWA.

As a result, `thePath` and `dirname` are stored in the isolated memory space, and `dirname` is not adjacent to other objects. When heap overflow occurs at line 13 of Listing 5, no sensitive data is overwritten, and the application security is preserved.

A.2 Libjpeg

CVE-2021-46822 is a heap-based buffer overflow vulnerability in the libjpeg library. In this CVE, the PPM reader incorrectly loads a 16-bit binary PPM file into a grayscale buffer. The vulnerability is caused by a heap-based buffer overflow in the `get_word_rgb_row` function. Listing 7 shows the function. At line 27 and line 32, the code reads the raw-format PPM file (`rescale`) to the buffer (`ptr`). Both `rescale` and `ptr` variables are stored in the `source` variable. However, the size of `image_width` may exceed the size of the buffer (`ptr`) and cause a buffer overflow, which may overwrite nearby heap objects.

```
1  int process_data() {
2      ...
3      cjpeg_source_ptr source = (ppm_source_ptr)
4      (*cinfo->mem->alloc_small)((j_common_ptr)cinfo, JPOOL_IMAGE,
5                                 SIZEOF(ppm_source_struct));
6      ...
7      source->pub.get_pixel_rows = get_word_rgb_row;
8      ...
9      JDIMENSION num_scanlines = (*source->get_pixel_rows)
10                                (&cinfo, source);
11     ...
12 }
13 METHODDEF(JDIMENSION)
14 get_word_rgb_row (j_compress_ptr cinfo, cjpeg_source_ptr sinfo) {
15     ppm_source_ptr source = (ppm_source_ptr) sinfo;
16     ...
17     register JSAMPROW ptr = source->pub.buffer[0];
18     register U_CHAR * bufferptr = source->iobuffer;
19     register JSAMPLE *rescale = source->rescale;
20     ...
21     for (JDIMENSION col = cinfo->image_width; col > 0; col--) {
22         register unsigned int temp;
23         temp  = UCH(*bufferptr++) << 8;
24         ...
25         if (temp > maxval)
26             ERREXIT(cinfo, JERR_PPM_OUTOFRANGE);
27         *ptr++ = rescale[temp];
28         temp  = UCH(*bufferptr++) << 8;
29         ...
30         if (temp > maxval)
31             ERREXIT(cinfo, JERR_PPM_OUTOFRANGE);
32         *ptr++ = rescale[temp];
33         ...
34     }
35     ...
36 }
```

Listing 7: The heap-based buffer overflow in libjpeg.

To address this vulnerability, PKUWA can isolate the vulnerable functions from the sensitive data as shown in Listing 8. In this solution, an isolated domain is created (line 1), and the vulnerable code is placed within it (line 3). All the variables that are related to the overflow code (including `source` and `ptr`) are stored in the

[1] We modified the code for clarity.

isolated memory space. By doing so, PKUWA ensures that when a heap-based buffer overflow occurs (line 27 and line 32 of Listing 7), it does not overwrite any data outside the domain, thereby preserving the application's security. The results for other vulnerabilities are similar to this vulnerability, but due to space limitations, we have included them in our GitHub repository.

```
1 int domain = domain_create(0);
2 ...
3 PKU_CALL_REGISTER(domain, process_data);
4 ...
5 PKU_CALL(process_data());
```

Listing 8: Protecting libjpeg with PKUWA.

A.2.1 CVE-2018-19664. The fourth case pertains to the heap-based buffer over-read vulnerability in the libjpeg (CVE-2018-19664). The vulnerability arises when attempting to decompress a specially crafted malformed JPEG image to a 256-color BMP using djpeg. The code snippet [2] in Listing 9 illustrates this scenario, where the code writes 24-bit pixels from `inptr` at line 19 and line 20. The over-read also happens in these lines. As the output_width of cinfo may exceed the buffer size, it can lead to over-reading of adjacent heap objects.

```
1  int process_data() {
2      ...
3      bmp_dest_ptr dest = (bmp_dest_ptr)
4      (*cinfo->mem->alloc_small)((j_common_ptr)cinfo, JPOOL_IMAGE,
5                                     SIZEOF(bmp_dest_struct));
6      ...
7      dest->pub.put_pixel_rows = put_pixel_rows;
8      ...
9      (*dest->put_pixel_rows)(&cinfo, dest, num_scanlines);
10     ...
11 }
12 METHODDEF(void)
13 put_pixel_rows (j_decompress_ptr cinfo, djpeg_dest_ptr dinfo,
14                 JDIMENSION rows_supplied) {
15     bmp_dest_ptr dest = (bmp_dest_ptr)dinfo;
16     register JSAMPROW inptr = dest->pub.buffer[0];
17     ...
18     for (JDIMENSION col = cinfo->output_width; col > 0; col--) {
19       JSAMPLE c = *inptr++, m = *inptr++,
20               y = *inptr++, k = *inptr++;
21       cmyk_to_rgb(c, m, y, k, outptr + 2, outptr + 1, outptr);
22       ...
23     }
24     ...
25 }
```

Listing 9: The heap-based buffer overflow vulnerability in libjpeg.

PKUWA solves this problem by isolating the function that suffers from heap-based buffer over-read, thereby preventing the theft of sensitive data. Listing 10 shows the code snippet of how PKUWA accomplishes the isolation of the over-read in `process_data`. In this approach, the domain_create function creates an isolated domain (line 1) and places the vulnerable code in it (line 3). The `dest` and the corresponding buffer are stored in the isolated memory space, as depicted in line 3 to line 5 of Listing 9. Consequently, when a heap-based buffer over-read occurs at line 19 and line 20 of Listing 9, no sensitive data is accessed beyond its boundaries, thereby preserving the application's security.

[2]We modified the code for clarity.

```
1 int domain = domain_create(0);
2 ...
3 PKU_CALL_REGISTER(domain, process_data);
4 ...
5 PKU_CALL(process_data());
```

Listing 10: Protecting libjpeg from heap-based buffer over-read with PKUWA.

A.2.2 CVE-2018-14498. The fifth case is the heap-based out-of-bounds read vulnerability in the libjpeg (CVE-2018-14498). Out-of-bounds read in cjpeg occurs when attempting to compress a specially crafted malformed color-index (8-bit-per-sample) BMP file in which some of the samples (color indices) exceeded the bounds of the BMP file's color table. Listing 11 shows the code [3]. The code reads the color table (`colormap`) to buffer (`outptr`) from line 22 to line 24. The out-of-bounds read also happens at this point. A crafted 8-bit BMP in which one or more of the color indices (`t`) is out of range for the number of palette entries (`colormap`). Since the size of image_width may exceed the size of the `colormap`, it can result in the out-of-bounds read to nearby heap objects.

```
1  JDIMENSION process_data() {
2      ...
3      bmp_source_ptr source = (bmp_source_ptr)
4      (*cinfo->mem->alloc_small)((j_common_ptr)cinfo, JPOOL_IMAGE,
5                                     SIZEOF(bmp_source_ptr));
6      ...
7      source->pub.get_pixel_rows = get_8bit_row;
8      ...
9      return (*source->pub.get_pixel_rows) (cinfo, source);
10 }
11 METHODDEF(JDIMENSION)
12 get_8bit_row (j_compress_ptr cinfo, cjpeg_source_ptr sinfo) {
13     bmp_source_ptr source = (bmp_source_ptr) sinfo;
14     register JSAMPARRAY colormap = source->colormap;
15     JSAMPARRAY image_ptr = (*cinfo->mem->access_virt_sarray)
16       ((j_common_ptr) cinfo, source->whole_image,
17       source->source_row, (JDIMENSION) 1, FALSE);
18     register JSAMPROW inptr = image_ptr[0];
19     register JSAMPROW outptr = source->pub.buffer[0];
20     for (JDIMENSION col = cinfo->image_width; col > 0; col--) {
21       register int t = GETJSAMPLE(*inptr++);
22       *outptr++ = colormap[0][t];
23       *outptr++ = colormap[1][t];
24       *outptr++ = colormap[2][t];
25     }
26 }
```

Listing 11: The heap-based out-of-bounds read vulnerability in libjpeg.

To address this vulnerability, PKUWA offers a solution that isolates the function that suffers heap-based out-of-bounds read, preventing it from accessing sensitive data. The code snippet in Listing 12 demonstrates how PKUWA achieves isolation and prevents out-of-bounds read in the `process_data`. In this solution, an isolated domain is created (line 1), and the vulnerable code is placed within it (line 3). Both the `source` and the corresponding buffer are stored in the isolated memory space from line 3 to line 5 of Listing 11. By doing so, PKUWA ensures that when a heap-based out-of-bounds read occurs, particularly at line 22 and line 24 mentioned in Listing 11, it does not overread any sensitive data. Thus, the application security is preserved.

[3]We modified the code for clarity.

```
int domain = domain_create(0);
...
PKU_CALL_REGISTER(domain, process_data);
...
PKU_CALL(process_data());
```

Listing 12: Protecting libjpeg from heap-based out-of-bounds read with PKUWA.

B MORE RESULTS FOR § 7.3

The Number of Inter-Domain Calls (#IDC) and Copy Operations (#CO). We conducted a study to determine how often PKUWA performs inter-domain calls and inter-domain copying. We measured the number of these operations across different portions of protected functions. Table 4 shows the results for five configurations. The number of inter-domain calls (IDC) varied between 487 and 216 million. Since each IDC involves two copy operations for arguments and return values, the number of copy operations is twice as high as the number of IDCs, and they are positively correlated.

Table 4: The number of inter-domain calls and copy operations.

Application	5%		25%		50%		75%		100%	
	#IDC	#CO	#IDC	#CO	#IDC	#CO	#IDC	#CO	#IDC	#CO
bzip2	621	1,242	3,205	6,410	6,410	12,820	9,315	18,630	13,421	26,842
espeak	764,757	1,529,514	3,182,612	6,365,224	7,669,435	15,338,870	9,547,836	19,095,672	15,495,155	30,990,310
facedetection	30,104	60,208	151,552	303,104	390,172	780,344	452,656	905,312	782,092	1,564,184
gnuchess	68,430	136,860	331,506	663,012	689,457	1,378,914	994,518	1,989,036	1,368,614	2,737,228
whitedb	9,810,889	19,621,778	49,422,761	85,885,522	103,635,578	207,271,156	148,268,283	296,536,566	216,217,791	432,435,582
tsf	21,002	42,004	121,010	242,020	250,531	501,062	361,079	722,158	521,599	1,043,198
cjpeg	6,144	12,288	30,480	60,960	69,364	138,728	91,443	182,886	150,480	300,960
djpeg	487	974	2,512	5,024	6,144	12,288	8,536	17,072	12,531	25,062
coreutils	790	1,580	3,922	7,844	7,221	14,442	11,766	23,532	15,802	31,604
ripgrep	25,338	50,676	141,119	282,238	269,918	539,836	423,357	846,714	506,775	1,013,550