Rewind & Discard: Improving Software Resilience using Isolated Domains

Merve Gülmez
Ericsson Security Research
Kista, Sweden
imec-DistriNet, KU Leuven
Leuven, Belgium
merve.gulmez@kuleuven.be

Thomas Nyman
Ericsson Product Security
Jorvas, Finland
thomas.nyman
@ericsson.com

Christoph Baumann
Ericsson Security Research
Kista, Sweden
christoph.baumann
@ericsson.com

Jan Tobias Mühlberg
imec-DistriNet, KU Leuven
Leuven, Belgium
Université Libre de Bruxelles
Brussels, Belgium
jan.tobias.muehlberg@ulb.be

***Abstract*—Well-known defenses exist to detect and mitigate common faults and memory safety vulnerabilities in software. Yet, many of these mitigations do not address the challenge of software *resilience* and *availability*, i.e., whether a system can continue to carry out its function and remain responsive, while being under attack and subjected to malicious inputs. In this paper we propose *secure rewind and discard of isolated domains* as an efficient and secure method of improving the resilience of software that is targeted by run-time attacks. In difference to established approaches, we rely on compartmentalization instead of replication and checkpointing. We show the practicability of our methodology by realizing a software library for Secure Domain Rewind and Discard (SDRaD) and demonstrate how SDRaD can be applied to real-world software.**

***Index Terms*—software resilience, compartments, rollback**

I. Introduction

A high level of availability for software systems is hard to achieve. The problem becomes even harder when dependability guarantees for distributed systems are required and software-level attacks are in scope: software written in unsafe languages can suffer from various memory-related vulnerabilities [1] that allow run-time attacks, such as control-flow attacks and non-control-data attacks [2], to compromise program behavior and let adversaries gain access to vulnerable systems.

According to the Google “0day In the Wild” dataset over 70% of the zero-day vulnerabilities between July 2014 and June 2022 can be attributed to memory-safety issues [3]. Research into run-time attacks has, during the past 30 years, led to an ongoing arms race between increasingly sophisticated attacks and run-time defenses to mitigate such attacks [4]. Today, major operating systems (OSs) provide such mitigations by default. This includes non-executable stack and heap areas, address-space-layout randomization (ASLR) [5], toolchain hardening options such as stack canaries [6], and hardware-enforced control-flow integrity (CFI) [7]. However, virtually all currently known defenses mitigate detected attacks by terminating the victim application [4], [8]–[23]. Thus, even though applications are hardened against run-time attacks, the response can still be leveraged by attackers to create availability issues and denial-of-service (DoS) conditions while the application is restarted, or to bypass security controls by resetting volatile system state, e.g., counts of failed login attempts.

Mitigations that focus on system availability typically involve application replication or checkpoint-and-restore mechanisms [24], which have a non-negligible performance impact themselves. Service-oriented applications are at particular risk as even a temporary failure in a critical component can affect a large number of clients. An example for such an application is *Memcached*, a general-purpose distributed memory-caching system, which is commonly used to speed up database-driven applications by caching database content. In this case, for example, restarting the application after a fault can easily take several minutes, while checkpointing or replication will have to deal with the large runtime state of the application.

Contributions. To address limitations of current defenses and to improve the resilience of software that is being targeted by run-time attacks, we propose *secure rewind and discard of isolated domains*: a mechanism that allows the state of a victim application under attack to be efficiently restored to an earlier state that is known to be unaffected by the attack. This is possible by leveraging hardware-assisted software fault isolation to compartmentalize the application into distinct *domains* that limit the effects of run-time attacks to isolated memory compartments. An application can be instrumented to isolate, e.g., "high-risk" code that operates with untrusted input in a secure in-process sandbox and rewind the application state if an attack is detected against sandboxed code. Domains can be nested to allow for efficient and secure rewinding in different software architectures and use cases. In particular, service-oriented applications can be augmented with resilience against run-time attacks to limit the impact to concurrent clients.

We show the practicability of our methodology by realizing a software library for *secure domain rewind and discard* (SDRaD) for commodity 64-bit x86 processors with *protection keys for userspace* (PKU) [25], [26] and demonstrate how SDRaD can be applied to real-world software in case studies on Memcached, a popular distributed memory-cache system (§ V-A), the NGINX web server (§ V-B), and OpenSSL (§ V-C). In summary, the contributions of this paper are:

- *Secure Rewind and Discard of Isolated Domains* is a novel scheme to improve software resilience against run-time attacks by rewinding the state of a victim application (§ III).
- We explore different *design patterns for compartmentaliza-*

tion and rewinding and discuss their applicability to retrofit software with secure rewind and discard (§§ III-D to III-F).

- We provide SDRaD, a realization of secure rewinding for commodity 64-bit x86 processors with PKU (§ IV, [27]).
- We show that SDRaD can be used with limited refactoring of application code only, as we apply it in three case studies (§ V). Benchmarks on Memcached and NGINX exhibit a worst case performance overhead <7.2%, negligible overhead (2%–4%) in realistic multi-processing scenarios, and negligible memory overhead (0.4%–3%).
- We assess the security, applicability, and limitations of our approach in § VI and compare it to related work in § VII.

II. Background

Rollback recovery techniques [28] have been studied in the context of a wide variety of applications ranging from programming language constructs [29] to recovery protocols for distributed system [28]. At a high-level, such techniques described in prior work can be divided into *checkpoint-based* and *log-based* approaches. Checkpoint-based techniques, such as *checkpoint & restore* (§ II-A) rely on recording transient system state so that it can later be recovered from a previously prepared *checkpoint*. Checkpoints in prior work are predominantly based on reproducing the process's memory image in a manner that can later be restored [24], [30]–[34]. Log-based approaches [35] combine checkpointing with recording the necessary information of nondeterministic events to replay each event during the recovery process.

In this work, we avoid the pitfalls of checkpoints that reproduce process memory by leveraging hardware-assisted fault isolation to partition an application process into distinct, isolated domains. This compartmentalization facilitates secure rewinding of application state by isolating the effects of memory errors. This enables rewinding to application states that precede the point of failure in the application's call graph and are unaffected by a caught and contained error.

Pre-existing work that bases rollback on compartmentalization properties is aimed at improving OS reliability to driver failures [36], [37] or targets embedded systems [38]. In contrast, secure rewind and discard of isolated domains targets application software in commercial off-the-shelf (COTS) processors and does not require OS or hardware changes.

A. Checkpoint & Restore

Application checkpoint & restore [24] is a technique for increasing system resilience against failure. By saving the state of a running process periodically or before a critical operation, a failed process can later be restarted from the checkpoint. The cost of this depends on the amount of data needed to capture the system's state and the checkpointing interval.

Several studies have focused on optimizing system-level and application-level checkpointing [30]–[32]. Secure checkpointing schemes [33] generally leverage cryptography to protect the integrity and confidentiality of checkpoint data at rest. However, they generally do not consider attacks that tamper with checkpointing code at the application-level. Furthermore, bulk encryption of checkpoint data is too costly for latency-sensitive applications, e.g., network traffic processing or distributed caches, unless load balancing and redundancy is present.

B. Software Fault Isolation

Software fault isolation (SFI) [39] is a technique for establishing logical protection domains within a process through program transformations. SFI instruments the program to intermediate memory accesses, ensuring that they do not violate domain boundaries. Since transitioning from one domain to another stays within the same process, SFI solutions can offer better run-time efficiency compared to traditional process isolation, especially in use cases where domain transitions are frequent. SFI has been successfully deployed for sandboxing plug-ins in the Chrome browser [40], isolating OS kernel [41] and modules [10], [12], [16], as well as code accessed through foreign function interfaces in managed language runtimes [42].

SFI enforcement can be realized in different ways. The principal method to realize SFI for native code binaries is the use of an inline reference monitor through binary [12] or compiler-based rewriting [10], [16], [23], [43] of the application binary. Recent SFI approaches leverage hardware-assistance (§ II-C) to further improve enforcement efficiency [44]–[56]. Existing approaches to SFI share the drawback of memory vulnerability countermeasures as they respond to detected domain violations by terminating the offending process.

C. Memory Protection Keys

Memory protection keys (MPK) provide an access control mechanism that augments page-based memory permissions. MPK allows memory access permissions to be controlled without the overhead of kernel-level modification of page table entries (PTEs), giving it a significant performance advantage compared to completely OS-controlled memory protection facilities, e.g., `mprotect()` [19]. On 64-bit x86 processors, PKU are supported in Intel's [25] and AMD's [26] microarchitectures. Similar hardware mechanisms are also available in ARMv8-A [57], IBM Power [58], HP PA-RISC [59] and Itanium [60] processor architectures.

Each memory page is associated with a 4-bit protection key stored in the page's PTE on 64-bit x86 processors. The access rights to memory associated with each protection key are kept in a *protection key rights register* (PKRU) that allows write-disable and access-disable policies to be configured for protection keys. These policies are enforced by hardware on each memory access.

III. Secure Domain Rewind and Discard

We propose *secure rewind and discard of isolated domains*, a novel approach for improving the resilience of userspace software against run-time attacks that augments existing, widely deployed run-time defenses. First, we present our threat model, system requirements (§ III-A), and high-level idea (§ III-B). §§ III-C to III-F delve into specific aspects of the design.

A. Threat Model and Requirements

Assumptions. In this work, we assume that the attacker has arbitrary access to process memory, but is restricted by the following assumptions about the system:

A1 A W⊕X policy [4] restricts the adversary from modifying code pages and performing code-injection.

A2 The application is hardened against run-time attacks and can detect an attack in progress, but not necessarily prevent the attacker from corrupting process memory.

We limit the scope of **A2** to software-level attacks. Transient execution [61] and hardware-level attacks, e.g., fault injection [62], rowhammer [63] etc., which are generally mitigated at hardware, firmware, or kernel level, are out of scope.

Requirements. Our goal is to improve the resilience of an application against active run-time attacks that may compromise the integrity of the application's memory. We introduce a mechanism for *secure rewind* with requirements as follows:

R1 The mechanism must allow the application to continue operation after system defenses (**A2**) detect an attack.

R2 The mechanism must ensure that the integrity of memory after recovering from the detected attack is maintained.

To facilitate **R1** and **R2** the application is compartmentalized into isolated domains with the following requirements:

R3 Run-time attacks that affect one domain must not affect the integrity of memory in other domains

R4 The attacker must not be able to tamper with components responsible for isolation or transitions between domains, or data used as part of the rewinding process.

B. High-level Idea

The objective of the secure rewind mechanism is to recover the application's execution state, after a memory defect has triggered, to a prior state before the application's memory has been corrupted (**R2**). Thus, the application can resume its execution and continue to provide its services without interruption (**R1**). To facilitate this, the application is compartmentalized into separate domains that each execute in isolated memory compartments allocated from the process's memory space. Should the execution of code inside a domain fail due to a memory defect, the memory that belongs to the domain must be considered corrupted by the attacker. However, since the effects of the memory defect are isolated to the memory belonging to the failing domain (**R3**, **R4**) the application's execution can now be recovered by: 1) discarding any affected memory compartments, 2) rewinding the application's stack to a state prior to when the offending domain began its execution, and 3) performing an application-specific error handling procedure that avoids triggering the same defect again, e.g., by discarding the potentially malicious input that caused it.

C. Domain Life Cycle

The overall life cycle of a domain is depicted in Figure 1, showing the steps taken to isolate execution in a *nested domain* from its parent. We consider a call to a function or library *F* that takes one in-memory argument and returns a value. The call is instrumented by code for Secure Domain Rewind and Discard,

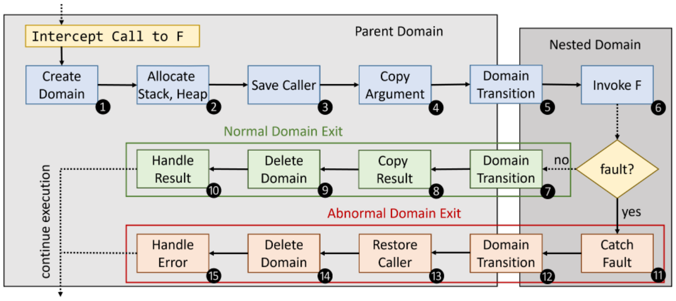

Fig. 1: Domain life cycle for calling an internal or library function *F* in a nested domain from a parent domain. Dotted arrows represent execution of user space instructions.

which first creates a new domain ❶ and then allocates separate stack and heap memory ❷. Then the caller's *execution context*, containing information such as register values, including the stack and instruction pointers, and signal mask, is saved for later use by the rewind mechanism ❸ (in a manner similar to C `setjmp()` [64]). Next, the input argument is copied onto the new heap ❹ and the domain transition step ❺:

- updates the hardware-enforced memory access policy: grant access to the new domain's memory areas and protect all other memory, global data is made read-only;
- switches execution to the stack of the new domain.

After entering the newly spawned domain, all subsequent code will run in that domain until the next transition. Here, we just call *F* ❻ which may result in one of two outcomes: A *normal domain exit* occurs when *F* runs to completion and returns back to the call site without any errors. Execution resumes in the parent domain ❼, the result is stored in the return variable ❽, and the nested domain is deleted ❾. At last, control transfers to developer-provided handler code for normal exits ❿.

An *abnormal domain exit* occurs if *F*'s code tries to access memory past the confines of the domain's memory area, or a possible run-time attack is detected by defense mechanisms ⓫. The execution of the domain is halted and privileges of the parent domain are restored ⓬. Application execution is resumed by restoring the calling environment from the information stored prior to invoking the offending domain ⓭, effectively rolling back application state to the point before the domain started executing. The failing domain is deleted and its memory is discarded ⓮. Control transfers to a custom error handler ⓯.

In general, after an abnormal domain exit, the application is expected to take an alternate action to avoid the conditions that led to the previous abnormal exit before retrying the operation. For example, a service-oriented application can close the connection to a potentially malicious client.

Rewinding the application state is limited to the state of application memory. Operations that have side-effects on an application's environment, e.g., reading from a socket, are still visible to the application after rewinding. Generally, different software architectures may require different design patterns for secure rewinding. We highlight some of these patterns below.

D. Domain Types and Patterns

As part of process initialization, all application memory, including stack, heap, and global data, are assigned to the *root domain*, which forms the initial isolated domain where an application executes. Application subroutines are compartmentalized into *nested domains* from which rewinding can be performed at any point during execution. As the names suggests, domains can be nested in the sense that several domains can be entered subsequently starting from the root domain, each with a dedicated rewind procedure (cf. § III-E). While read-only access from any nested domain to data in the parent domain may be allowed, writable access is forbidden in order to contain any memory safety violations to the nested domain. By default, nested domains have read access to the root domain to enable reading global variables. We envision two flavors of isolation with rewinding for application subroutines:

- **Protecting the application from a subroutine:** Code that may have unknown memory vulnerabilities, e.g., third-party software libraries, can be executed in a nested domain by instrumenting calls to the functionality so that they execute in their own domain and may be rewound in case memory-safety violations are detected.
- **Protecting a subroutine from its caller:** application code operating on sensitive data such as cryptographic keys can be isolated from vulnerabilities in its callers, preventing the leak and loss of such data. For example, encryption, decryption, or key derivation functions from the OpenSSL library can be isolated in their own nested domain to protect the application's cryptographic keys if a fault occurs in a calling domain. Listing 2 in § IV-A shows a concrete example of isolating OpenSSL.

To support the two application scenarios described above, we identify two design patterns for nested domains. The type of a domain determines how it continues its life cycle, in particular what happens to it upon normal domain exit.

Persistent Domains A persistent domain retains all its memory areas after the application's execution flow returns to the parent domain. Another code path may enter the persistent domain again, at which point access to memory areas that belong to the persistent domain is again allowed. Practically, in Figure 1, creation step ❶ is only needed for the first invocation and deleting the domain in step ❾ is omitted.

Modules that maintain state information across invocations should be isolated in a persistent domain so that their state is not lost after normal domain exits, e.g., some software libraries may encapsulate such state by creating "context" objects that persist across calls. Distinct contexts can be compartmentalized by ensuring each object is allocated in different persistent domains. An example of this pattern is OpenSSL which can be instantiated multiple times within an application using different contexts and associated key material. Cryptographic keys of one context remain isolated in memory from others if each concurrent context is assigned separate persistent domains.

On abnormal exits from any domain, the rewind mechanism is triggered and all state of the domain is discarded. This may have serious repercussions for an application, if the program state depends on data isolated in a persistent domain. When, for example, the abnormal exit leads to the loss of session keys for a TLS connection, the application may only be able to recover by re-initializing the affected cryptographic context and close all connections that were using the now lost keys.

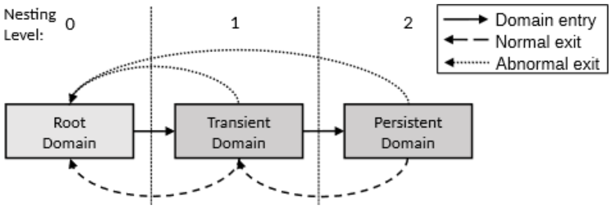

Fig. 2: Deeply nested domains. Normal exits occur in reverse domain entering order. Abnormal exits may deviate from that: both persistent and transient domain rewind to root domain.

The parent domain may or may not be given access to a persistent nested domain's memory. For instance, in the case of cryptographic libraries, access to the persistent domain's memory from any other domain should be blocked to protect sensitive data stored by the library. In such cases data cannot be directly passed between the caller and callee in distinct domains and shared data, e.g., call arguments and results need to be copied between domains via a designated shared memory area. This is similar to how data is passed between different protection domains in, e.g., Intel Software Guard Extension (SGX) enclaves [65].

Transient Domains Memory areas assigned to transient nested domains persist until the application's execution flow returns from such a domain to the parent domain at which point the stack as well as *unused* heap memory areas assigned to the transient nested domain are discarded. For this the rewind mechanism needs to keep track of memory allocations in a nested domain. Any allocated memory in a transient nested domain's heap area can be merged back to the parent domain's heap area or discarded upon a normal domain exit, depending on the specific application scenario. Generally, transient domains are most useful for functionality that is called only once during a typical program invocation.

E. Domain Nesting and Rewinding

Domain nesting enables creating new isolated domains within others. Each nested domain has exactly one parent domain, which is responsible for creating the nested domain. All domains may have zero or more nested child domains, i.e., nested domains may be created by already nested domains.

Transient and persistent-style domains can be nested with each other. One example of a domain nesting configuration is illustrated in Figure 2. Here, the first level of domains is transient and its subsequent nested domain can be persistent. This setup allows developers to simplify error handling by directing a rewind from the more deeply nested persistent domain to also return to the recovery point established for the transient domain. On the other hand, abnormal exits from the transient domain do not affect the nested persistent domain

TABLE I: SDRaD API. udi: user domain index.

API Name	Arguments	Description
① `sdrad_init()`	udi, options	Initialize Domain *udi*
② `sdrad_malloc()`	udi, size	Allocate *size* memory in domain *udi*
③ `sdrad_free()`	udi, adr	Free memory at *adr* in domain *udi*
④ `sdrad_dprotect()`	udi, tddi, PROT	Set domain *udi*'s access permissions to *PROT* on target data domain *tddi*
⑤ `sdrad_enter()`	udi	Enter Domain *udi*
⑥ `sdrad_exit()`	—	Exit Domain *udi*
⑦ `sdrad_destroy()`	udi, options	Destroy Domain *udi*
⑧ `sdrad_deinit()`	udi	Delete return context of Domain *udi*

and leave it to developer-provided error handling routines to decide if the persistent domain needs to be destroyed as well.

At an abnormal domain exit, the rewind can occur from any nesting level to a lower one as required by an application, but an abnormal root domain exit terminates the program.

F. Multithreading

The secure rewind mechanism supports POSIX threads. In most cases, threads need to communicate with each other using shared memory, hence they need to access root domain memory. Consequently, it would not be possible to isolate two threads completely from each other or from the main process.

Nevertheless, it is still possible to isolate partial code paths within the thread to separate domains, i.e., each thread can still create nested domains with stacks and heaps that are isolated from the root domain and other nested domains. Hence, each thread may recover via rewinding from errors in such nested domains. If one of the threads suffers an abnormal exit from the root domain, the rewind mechanism cannot recover other threads and the application must be terminated.

Threads have shared access to global and root domain heap memory, to per-thread stack areas and thread-local storage. A shared root domain allows a higher number of parallel threads to be supported, if the available domains provided by the underlying memory protection mechanism is limited, as only threads that instantiate nested domains consume domain slots. However, one could allow the developer to configure stricter, non-uniform access privileges between threads.

IV. Prototype Implementation

We implement the concept of secure domain rewind and discard as *SDRaD* [27] – a C-language Linux library for the 64-bit x86 architecture using PKU as the underlying isolation primitive. The library provides APIs to control the life cycle of domains. *SDRaD* does not require Linux kernel patches beyond those potentially needed by run-time defense mechanisms. The Linux kernel supports PKU from version 4.9. Further details on the implementation are provided in as a technical report [66].

A. SDRaD API

Developers use the SDRaD API calls shown in Table I to flexibly enhance their application with a secure rewind mechanism, accounting for the design patterns described above.

Domains are initialized by `sdrad_init()`① where the developer chooses a unique index to reference the domain in future API calls. *Execution* and *data* domains may be created, where the latter may hold shareable data pages but cannot execute code. For execution domains we further distinguish domains that are *accessible* or *inaccessible* to their parent, and whether an abnormal domain exit should be handled in the parent or grandparent domain. A domain can only be initialized once per thread (unless it is deinitialized or destroyed before by the programmer) and the point of initialization for execution domains marks the execution context to which control flow returns in case of an abnormal domain exit.

The API call's return value fulfills two roles. When the domain is first initialized, it returns *OK* on success or an error message, e.g., if the domain was already initialized in the current thread. On abnormal domain exit, control flow returns another time from the init function and the return value signifies the index of the nested domain that failed and was configured to return to this point. This means that error handling for abnormal domain exits needs to be defined in a case split on the return value of the `sdrad_init()`① function.

After initialization, memory in an execution or data domain can be managed using `sdrad_malloc()`② and `sdrad_free()`③, e.g., to be able to pass arguments into the domain. Note that this is only allowed for child domains of the current domain that are accessible. For inaccessible domains, a shared data domain needs to be used to exchange data. Using `sdrad_dprotect()`④, access permissions to a data domain can be configured for child domains.

An execution domain initialized in the current domain can be entered and exited using `sdrad_enter()`⑤ and `sdrad_exit()`⑥. This switches the stack and heap to the selected domain and back, and changes the memory access permissions accordingly. Currently, the SDRaD does not copy local variables on such domain transitions. Such variables need to be passed via registers or heap memory.

Supporting the transient domain design pattern, child domains can be deleted using `sdrad_destroy()`⑦, with the option to either discard the domain's heap memory or, if accessible, merge it to the current domain. The persistent domain pattern is then implemented by simply not destroying the domain after exiting it, so that it can be entered again.

An important requirement is that a nested execution domain needs to be destroyed before the function which initialized that domain returns. Otherwise, the stored execution context to which to return to would become invalid as it would point to a stack frame that no longer exists. To provide more flexibility, `sdrad_deinit()`⑧ allows to just discard a child domain's execution context but leave its memory intact. Before entering the domain again, it needs to be re-initialized, setting a new return context for abnormal exits.

Usage Example. We implement the domain life cycle shown in Figure 1 for a function *F* that receives pointer `arg` to an object of size `size` as input and returns an integer-sized value. In the example, we first initialize a new accessible execution domain `udi_F` for function *F* ①. If an abnormal exit occurs in that domain, control returns here, so we save the error code in `err`. If initialization succeeded, we allocate a local variable `r` in a register to retrieve the return value later ②. We also allocate

```
int err = sdrad_init(udi_F, EXECUTION_DOMAIN |
                     ACCESSIBLE | RETURN_HERE);   ①
if (err == OK ) {
   // prepare passing return value and argument
   register int r asm ("r12");  ②
   register void *adr asm ("r13");
   adr = sdrad_malloc(udi_F, size);  ③
   if (!adr && size>0) { return MALLOC_FAILED; }
   if (size>0) { memcpy(adr, arg, size); } ④
   sdrad_enter(udi_F);  ⑤
   // invoke F on copy of argument and save return value
   r = F(adr); ⑥
   sdrad_exit(); ⑦
   if (ret) { *ret = r; } ⑧
   sdrad_free(udi_F, adr); ⑨
   sdrad_destroy(udi_F, NO_HEAP_MERGE); ⑩
}
return err;
```

Listing 1: Using API calls (orange) to call `F(arg,size)` in its own domain (pseudocode for Figure 1, error handling omitted).

memory for the input argument at `adr` in the new domain ③. Since that domain is accessible, we can copy the argument directly from the parent domain ④. Afterwards, we enter the nested domain ⑤ and invoke *F* on the copy of the argument, saving the return value ⑥. After exiting, we are back in the parent domain ⑦. We can copy the return value to the desired location (which is inaccessible to the nested domain) ⑧. We free all temporary memory ⑨ and destroy the nested domain, freeing its remaining memory ⑩. As a side note, calling `sdrad_free()` is actually redundant here; `sdrad_destroy()` would free this memory as well with the `NO_HEAP_MERGE` option. Finally, we return `OK` in case of normal domain exit, or the error code otherwise. Users of the shown isolated version of *F* can then define their own error handling depending on this return value.

OpenSSL Listing 2 shows an `EVP_EncryptUpdate()` wrapper for OpenSSL that implements the persistent domain pattern in § III-D. Here, OpenSSL allocates its data, such as the context (`ctx`) ❶ in a domain which is inaccessible from its parent. The caller can hold a pointer to `ctx`, but the object itself is inaccessible to the parent domain. Arguments are copied in via a data domain ❷. The wrapped function must read buffered input and write its output to its parent domain. There are three possible design choices for passing data between the respective domains: 1) the OpenSSL domain has read-only access to the parent, i.e., it's called from the root domain; input can be read directly, but output must be copied through the data domain used for argument passing ❸, ❺ 2) the parent domain is inaccessible to the OpenSSL and it must copy both input and output via the data domain used for argument passing ❸, ❹, ❺ 3) the parent domain is responsible for setting up a shared data domain between the respective domains and the wrapper can access the shared area directly via the argument pointers.

This persistent domain can be combined with a transient domain as shown in Figure 2 to 1) encapsulate the pointer to `ctx` within an outer domain, 2) protect the root domain from errors in the caller, e.g., an `out` buffer of insufficient size, and 3) simplify error handling for the OpenSSL domain.

B. Implementation Overview

The SDRaD implementation [27] has four components: 1) a hardware mechanism for enforcing in-process memory protection, 2) an isolated *monitor data domain* that houses control data for managing execution and data domains, 3) initialization code that is run on startup of an application that is linked to SDRaD, setting up the monitor domain and memory protection, and 4) a trusted reference monitor that realizes the SDRaD API calls with exclusive access to the monitor data domain.

Memory Protection SDRaD uses PKU (cf. § II-C) as a hardware-assisted SFI mechanism to create different isolated domains within an application governed by different memory access policies. When a domain is created, a unique protection key is assigned for it. At each domain transition, PKRU is updated to grant access to memory areas as permitted for the newly entered domain, and to prevent access to other memory areas. Our evaluation platform supports Intel PKU, hence it allows us to manage up to 15 isolated domains at a time for each process. Software abstractions for MPK, like libmpk [19], increase the number of available domains at the cost of falling back to the much slower `mprotect` system call.

SDRaD Control Data SDRaD stores global control data in the monitor data domain for keeping track of, e.g., the registered domain identifiers and the protection key usage. It also stores per-thread information for domains such as stack size, heap size, parent domain, and memory access permissions. To support abnormal domain exits, the currently executing domain and the saved execution contexts are stored, too.

Initialization An application is compiled with the SDRaD library to use the rewind mechanism. Then, a constructor function is executed before `main()` to assign all application memory to the initial isolated domain as a root domain associated with one of the PKU. It also initializes SDRaD global control data where the default stack and heap size for domains is configurable through environment variables. Furthermore, it sets the root domain as active domain, to be updated at domain transitions by the reference monitor, and finally initializes a signal handler. For the multithreading scenario, SDRaD has a thread constructor function as well, that is executed before the thread start routine function to assign a thread memory area to a domain and associate it with one of the protection keys.

Reference Monitor The reference monitor records domain information in SDRaD control data. It also performs domain initialization, domain memory management, and secure domain transitions, including updating the memory access policy, and saving and restoring the execution state of the calling domain. Only the reference monitor may update the PKRU register to gain access to the monitor data domain. The monitor code is executed using the stack of the nested domain that invoked it.

Error Detection Memory access violations are generally reported to userspace software either via 1) a `SEGFAULT` signal, e.g., when a domain tries to write past the confines of its memory area, or 2) calls to runtime functions inserted by instrumentation, e.g., GCC's stack protector calls `__stack_chk_fail()` if a stack guard check fails. Standard glibc versions of this function terminate the application, but we replaced it with our own implementation, allowing SDRaD to respond to stack guard violations. Moreover, during process initialization, SDRaD sets up its own signal handler for the

```
int __wrap_EVP_EncryptUpdate(EVP_CIPHER_CTX *ctx❶, unsigned char *out, int *outl, const unsigned char *in, int inl) {
    register evp_encrypt_update_args_t *args asm ("r12"); // holds copied function arguments and return value, is kept in
                                                         // a callee-saved register (r12) to remain accessible after
    ...                                                  // sdrad_enter(OPENSSL_UDI) changes the domain stack

❷  args = sdrad_malloc(OPENSSL_DATA_UDI, sizeof(evp_encrypt_update_args_t));
    args->ctx = ctx;                        // copy ctx from current domain to shared data domain
    args->inl = inl;                        // copy inl from current domain to shared data domain


    if (out != NULL && inl >= 0) {          // inl + cipher_block_size is upper bound for yet unknown output size
        args->out = sdrad_malloc(OPENSSL_DATA_UDI, inl + cipher_block_size);
    } else { args->out = NULL; }
                                       ❸ Set up output buffer. Replaced with args->out = out if out points to shared data domain.

    if (in != NULL && inl >= 0) {
        args->in = sdrad_malloc(OPENSSL_DATA_UDI, (size_t)inl);
        memcpy(args->in, in, inl);          // copy in from current domain to shared data domain
    } else { args->in = NULL; }
                                       ❹ Set up input buffer. Replaced with args->in = in if in points to a readable domain.
    sdrad_enter(OPENSSL_UDI);               // execute real EVP_EncryptUpdate in inaccessible domain
    args->ret = __real_EVP_EncryptUpdate(args->ctx, args->out, &(args->outl), args->in, args->inl);
    sdrad_exit();

    *outl = args->outl;                     // copy out outl value from shared data domain
    if (out != NULL) {                      // copy out encrypted data from shared data domain
        memcpy(out, args->out, (size_t)*outl);
    }
                                       ❺ Copy out. Omitted if out points to a shared data domain.
    ...
}
```

Listing 2: Wrapper function for `EVP_EncryptUpdate()` that executes OpenSSL in a persistent nested domain (excerpt). Data is passed between parent and nested domain via shared data domain `OPENSSL_DATA_UDI`. Error handling is omitted for brevity.

`SEGFAULT` signal, where the cause for a segmentation fault is given by a signal code (`si_code`) provided by the runtime to the signal handler [67], e.g., PKU access rule violations are reported by `SEGV_PKUERR` signal code. In Linux, the `SEGFAULT` signal is always delivered to the thread that generated it. If the SDRaD signal handler is triggered by a violation in a nested domain, it causes an abnormal domain exit. For faults occurring in the root domain or being attributed to a cause the SDRaD signal handler cannot handle, the process is still terminated. SDRaD can be extended to incorporate other run-time error detection mechanisms, such as Clang CFI [68] or heap-based overflow protections (e.g., heap red zones [21]), improving the recovery capabilities. Probabilistic and passive protections such as ASLR do not detect but hinder the exploitation of memory safety violations. Our rewind mechanism is compatible with ASLR, as domains are created at run-time.

Rewinding Secure rewinding from a domain is achieved by the reference monitor saving the execution context of the parent domain into SDRaD control data when that domain is initialized. SDRaD uses a `setjmp()`-like functionality to store the stack pointer, the instruction pointer, the values of other registers, and the signal mask for the context to which the call to `sdrad_init()` returns. Note that we cannot simply call `setjmp()` within `sdrad_init()` because that execution context would become invalid as soon as the initialization routine returns. On an abnormal domain exit, the saved parent execution state is used to restore the application's state to the initialization point prior to entering the nested domain by using `longjmp()`. This rewind lets the application continue from that last secure point of execution that is now redirected to the developer-specified error handling code. To simplify programming under these non-local goto semantics [64], we only allow to set the return point once per domain and thread. Moreover, the convention that a domain needs to be destroyed or deinitialized before the function that initialized it returns, ensures that the saved execution context is always valid.

C. *Memory Management and Isolation*

Our recovery mechanism is enabled by creating different domains within an application and ensuring that a memory defect within a domain only affects that domain's memory, not the memory of others. Using the underlying SFI mechanism based on PKU, domains are mutually isolated.

Global Variables We modify the linker script to ensure that global variables are allocated in a page-aligned memory region that can be protected by PKU. At application initialization, all global variables are assigned to the root domain, but if that one was completely isolated, globals would not be accessible to nested domains. As a compromise, we make the root domain by default read-only for all nested domains. Write access to global data may then be achieved by allocating it on the heap of a shared data domain, referenced by a global pointer. Note that this approach breaks the confidentiality of the root domain towards nested domains. As our main goal is integrity, and confidential data can still be stored and processed in separate domains, we find it a reasonable trade-off for SDRaD.

Stack Management SDRaD creates a disjoint stack for each execution domain to ensure that the code running in a nested domain cannot affect the stacks of other domains. The stack area is allocated when first initializing a domain and protected using the protection key assigned to that domain. As an optimization, we never unmap the stack area, even when the domain is destroyed, but keep it for reuse, i.e., when a new domain is initialized. At each domain entry, we change the stack pointer to the nested domain stack pointer and push the return address of the `sdrad_enter()` call, so that the API call returns to the call site using the new stack. Then we update the PKRU register according to memory access policy for that domain. A similar maneuver is performed when switching back to the parent domain's stack via `sdrad_exit()`.

Heap Management Because heap isolation in SDRaD requires memory management with strict guarantees that allocations within a domain are satisfied only from memory reserved for that domain, we opted to use an allocator that natively supports fully disjoint heap areas instead of the default glibc GNU Allocator [69]. For our implementation, we chose the Two-Level Segregated Fit (TLSF) allocator [70], [71]. TLSF is a "good-fit", constant-time allocator that allocates memory blocks from one or more pools of memory. Each SDRaD domain is assigned its own TLSF control structure and memory pool that correspond to the domain's subheap. The initial size of these pools is configurable via an environmental variable.

We interpose functions from the `malloc()` family with wrappers and place SDRaD before libc in library load order. Upon first call to memory management within a domain, its heap is initialized and the memory pool is associated with the domain's protection key. Having independent subheaps allows the developer to either discard or merge a domain's subheap with the parent's when the former domain is destroyed. Note however, that subheaps are never merged back after abnormal exits, as the data must be considered corrupted. We extended TLSF with a straight-forward implementation of subheap merging but omit a detailed description for brevity.

V. Case Studies

We evaluate the performance of SDRaD with three different real-world case studies: Memcached, NGINX, and OpenSSL. Two aspects are evaluated: 1) rewinding latency on an abnormal exit, 2) performance impact of the isolation mechanism. We run our experiments on Dell PowerEdge R540 machines with 24-core MPK-enabled Intel(R) Xeon(R) Silver 4116 CPU (2.10GHz) having 128 GB RAM and using Ubuntu 18.04, Linux Kernel 4.15.0. We compiled Memcached and NGINX with `-O2` optimizations, `-pie` (for ASLR), `-fstack-protector-strong`, and `-fcf-protection`.

A. Memcached

Memcached [72] is a general-purpose distributed memory caching system, which is used to speed up database-driven applications by caching database content. To do so efficiently, Memcached stores its state in non-persistent memory; after termination and restart, clients must start over and resend a large amount of requests to return to the situation prior to the restart. Even in real-world deployments with built-in redundancy and automatic remediation, small outages can take up to a few minutes to re-route requests to an unaffected cluster [73]. Several studies propose to use low latency persistent storage for Memcached [74], [75] but these solutions come with a non-negligible performance overhead. As availability and resilience of Memcached to unforeseen failures is of high importance, it is a worthwhile target for hardening with SDRaD. The main thread in Memcached accepts connections and dispatches them among worker threads to handle related requests. Memcached uses a hash table to map keys to an index and slab allocation to manage the in-memory database. Memcached has an event-driven architecture, handling each client request as an event. The clients can send get, set, and update commands with key and value arguments. To handle a request, command parser subroutines in Memcached classify the client request, then the key-value pairs are fetched from, inserted in, or updated in the database. If a client event contains a malicious request leading to memory corruption, the database and hash table, as well as the complete application memory area, are corrupted and Memcached must be restarted. As a result, one malicious request affects the availability of the service to all clients.

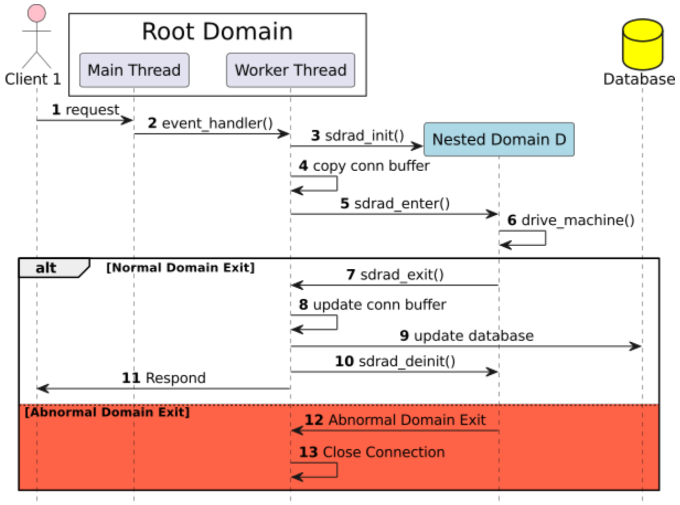

Fig. 3: Sequence diagram of client request for Memcached with SDRaD. Function `drive_machine()` is executing in nested domain *D* until normal or abnormal domain exit.

Memcached with SDRaD We propose that each client event should be handled in a nested domain. In case of memory corruption, the abnormal domain exit occurs in the nested domain, we discard the related nested domain contents and come back to the root domain securely. Memcached closes the related connection, and it can continue its execution, handling another client request without restarting. Figure 3 shows a sequence diagram of Memcached with SDRaD. We configure SDRaD with a shared root domain (see in § III-F), because the main thread needs to communicate with the worker threads. Each event is handled using the `drive_machine()` (**6**) function with a corresponding connection buffer. We isolate this function using the SDRaD API. Recall from § III-D that nested domains may only have read access to data that belongs to the parent domain. Nevertheless, certain subroutines, such as `drive_machine()`, need to update shared state residing in a parent domain, e.g., the connection buffer. As a solution, the event handler that calls `drive_machine()` initializes an accessible, nested domain *D* (**3**) and makes a *deep copy of the connection buffer* that is made available to `drive_machine()`

(**4**). It then enters D (**5**) and calls `drive_machine()` (**6**) to handle the client request, working on a copy of the connection buffer. After successfully handling the request, it exits from D (**7**), the original connection buffer in the parent domain is updated with any changes present in the shared copy (**8**). On abnormal domain exit, the copied connection buffer is discarded. Since the event handler returns after handling the request, we need to invalidate the saved execution context of D. As `drive_machine()` does not allocate persistent state in D, we could use the transient domain pattern and destroy D. However, for efficiency, we reuse the copied connection buffer used by the domain, hence `sdrad_deinit()` (**10**) is used.

The `drive_machine()` function also needs to read and write the hash table and database to perform look-ups, insertions, and updates. We allocate it in a dedicated data domain, accessible by the nested domain of each thread.

To allow inserts and updates, we wrap the `slabs_alloc()` function, that normally returns a pointer to a memory area in the database, to return a copy of that area to insert the key-value pair. Similarly, we wrap `store_item()` which stores new data and updates the hash table. Each event handler first performs its operation on a copy of the corresponding item. On normal domain exits from D, we insert the key-value pair to the database, and update the hash table (**9**). On an abnormal domain exit (**12-13**), the corrupt key-value pair is discarded along with all other domain memory. Note that this solution delays updates to the database. However, due to the atomic nature of the Memcached requests, consistency is not affected.

Our changes were limited to two source files in Memcached and 484 new lines of wrapper code. In total, the changes amounted to ~550 LoC of the 29K SLoC code base (~2%).

Memcached uses a shared mutex to synchronize worker threads. Here, our copying mechanism for shared data does not work, because it would hide concurrent accesses to the mutex and break the synchronization. We opted to create a separate data domain for the mutex that every worker can access. See § VI for a security discussion of this scheme.

Rewinding Latency We reproduced CVE-2011-4971 [76] to verify the SDRaD rewind mechanism and compiled Memcached v1.4.5 with SDRaD. This CVE crashes Memcached via a large body length value in a packet, creating a heap overflow but SDRaD ensures that this overflow is limited to current execution domain, hence it triggers the domain violation and an abnormal domain exit occurs. We measured the mean latency of abnormal domain exit starting with catching `SEGFAULT` until after we close the corresponding connection to 3.5μs (σ=0.9μs). For comparison, in our experiments the restart and loading time for 10GiB of data into Memcached was about 2 minutes. Thus, an attacker who successfully launches repeated attacks could knock out the Memcached service without rewinding (DoS). While this is clearly dominated by the loading time, even applications without such volatile state, but ultra-reliable low-latency requirements, can benefit from rewinding. For reference, we measured the mean latency to restart the Memcached container automatically at about 0.4s (400000μs, σ=19000μs).

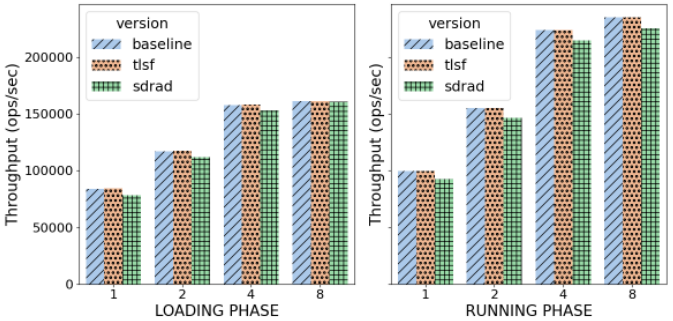

Fig. 4: Throughput of different Memcached instrumentations for different numbers of threads.

Performance Impact We used the Yahoo! Cloud Service Benchmark (YCSB) [77] to test the impact of SDRaD on Memcached performance. YCSB has two phases: a loading phase that populates the database with key-value pairs, and a running phase which performs read and update operations on this data. We used workloads with sizes of 1KiB, with a read/write distributions of 95/5. For our measurements, we stored 1×10^7 key-value pairs (1KiB each) and performed 1×10^8 operations on those pairs. Operations were performed with a Zipfian distribution over the keys. We compiled Memcached v1.6.13 and evaluated the performance with the TLSF allocator and with SDRaD as described in § V-A. We compare the results against YCSB on unmodified Memcached. Figure 4 shows the load and running phase throughput (operations/second) of the three versions for 1, 2, 4, and 8 workers over 5 benchmark runs. Each thread was pinned to separate CPU cores. We used 32 YCSB clients with 16 threads pinned to separate cores for each test. We fully saturated Memcached cores for 1, 2, and 4 threads but were unable to reach saturation for 8 threads. We concluded that TLSF has negligible impact on throughput in all our tests (< 1%). For Memcached augmented with SDRaD the load and running phase overhead is 2.9% / 4.1%, respectively, for 4 threads, and 4.5% / 5.5% for 2 threads. SDRaD introduced a worst-case overhead of 7.0% / 7.1% for a single thread. We measured a performance degradation of < 4.1% for 8 threads but lack confidence in the soundness of that result as the CPU was not saturated. We measured the memory overhead of SDRaD by comparing the maximum resident set size (RSS) after the YCSB load phase and of Memcached with SDRaD to the baseline. The mean RSS increase is 0.4% (σ=171KiB).

B. *NGINX*

NGINX [78] is an open-source web server implemented as a multiprocessing application with a master process and one or more worker processes. The master process is responsible for maintaining the worker processes that handle client HTTP requests for several connections at a time. If a malicious client request leads to memory corruption, the worker process may crash and the master process restarts it, however all active connections of that worker are lost. Due to its complexity and exposure to untrusted inputs, the HTTP parser is a vulnerable component of NGINX. Similar to [79], we propose that each client HTTP request is parsed in a nested domain. Thus, if a memory corruption is detected in the parser, an abnormal domain exit occurs and we discard the related nested domain content to come back to the root domain securely without restarting the worker process. The related connection is closed, but all other connections are unaffected.

We sandboxed the HTTP parser (ngx_http_parser.c) to execute in an accessible permanent nested domain, by instrumenting all NGINX parser functions using the SDRaD API. NGINX creates a temporary memory pool for each client request to hold a *request buffer*. We direct the allocation of these pools to a separate data domain that is accessible by the nested domain. The request buffer data structure links back to header data and URI data in the connection buffers. To protect this root domain data and make it accessible to the parser, it is copied into the nested domain and the results are copied back on domain exit. The NGINX Parser executes multiple phases, e.g., parsing request lines or headers. Thus, domain transitions occur repeatedly in one request. On an abnormal domain exit, it is not important which parser phase has corrupted the memory: we always close the corresponding connection. Hence we save as execution context the first entry point of the NGINX parser to come back to at an abnormal domain exit. Our changes were limited to one file in NGINX and 195 new lines of wrapper code. In total, the changes amount to ~220 LoC of the 150K SLoC code base (0.15%).

Rewinding Latency We reproduced CVE-2009-2629 [80] to verify the rewind mechanism and compiled NGINX v.0.6.39 with SDRaD. The CVE causes a buffer underflow in the linked connection buffer data. By having the parser operate on copies of that data in the nested domain, the underflow triggers a domain violation and thus an abnormal domain exit. We measured the latency of the abnormal domain exit starting from catching SEGFAULT to accepting a new connection. The mean latency is $3.4\mu s$ (σ=$0.67\mu s$). We compared it with restarting the worker process by the master process for reference. The mean latency is $996\mu s$ (σ=$44\mu s$). It should be noted however, that avoiding service disruption for other clients by preserving all connections handled by the worker process is arguably the more important benefit of SDRaD here.

Performance Impact We measured the SDRaD overhead to connect to NGINX remotely over keep-alive HTTP connections using ApacheBench tool. Each test has 75 concurrent connections and all clients request the same file size ranging from 0KiB to 128KiB. Figure 4 shows mean throughput (requests/second) of the three versions of NGINX with one worker process for different file sizes over 5 benchmark runs. We compiled NGINX v.1.23.1 and compared it to NGINX with TLSF allocator and SDRaD. The latter introduced overheads between 1.6% (128KiB) and 6.5% (1KiB). We scaled the number of workers for NGINX with SDRaD and observed that the overhead is independent of that number, as expected. We measured the memory overhead of SDRaD from the maximum RSS after benchmarking the 128-KiB file size with four worker processes, and comparing the RSS of NGINX with SDRaD to the baseline. The mean RSS increase is 3.06% (σ=50KiB). Profiling domain switching, we observed that 30% – 50% of the cost comes from writing the PKRU register, which flushes the processor pipeline [19], [47].

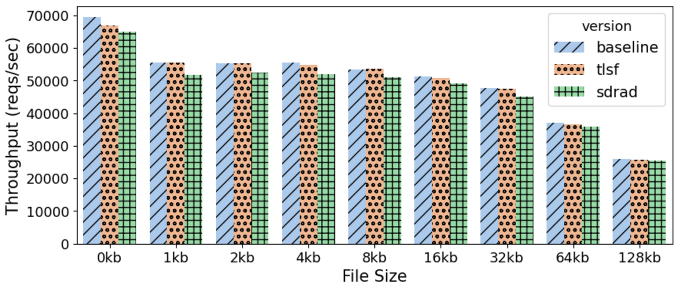

Fig. 5: Throughput of different NGINX instrumentations with one worker for different file sizes.

C. OpenSSL

SDRaD allows for isolating a library without changing it, enabling later integration into applications. As discussed in § III-D, we may either protect the application from the library or the library from the application. To demonstrate the first case, we reproduced CVE-2022-3786 in OpenSSL 3.0.6 [81]. When OpenSSL processes X.509 client certificates, the CVE causes a buffer overflow that puts an arbitrary number on the stack and may cause denial of service by crashing the application. It can be detected by stack canaries, so we isolated the vulnerable X.509 certification verification API of OpenSSL and compiled NGINX with that library and SDRaD. We verified that the CVE triggers a rewind and NGINX closes the related connection and reinitializes the OpenSSL domain before continuing execution. We added ~14 LoC in NGINX code base, ~18 LoC in OpenSSL code, as well as ~140 new lines of wrapper code.

To demonstrate protecting OpenSSL from the rest of the program, we instrumented OpenSSL 1.1.0 according to all three design choices explained in § IV-A. We evaluated the performance impact by adapting the built-in OpenSSL speed benchmark and running the *aes-256-gcm* cipher via the `EVP_EncryptUpdate` function (cf. Listing 2) for 3s, measuring the number of encryptions. As expected, `memcpy` operations cause notable performance overhead and the third option, a parent-managed shared domain, performed best. Even without copy operations, SDRaD substantially degraded the performance of cryptographic operations for small input sizes (4% to 80%). For more realistic input sizes ≥32KiB we did not measure any statistically significant overhead (< 2%).

VI. Discussion

Security Evaluation Our primary security requirement is

R1 The mechanism must allow the application to continue operation after system defenses (**A2**) detect an attack.

Our proposal satisfies this requirement by compartmentalizing applications into isolated domains where an attack against a child domain can be detected, which leads to the termination of that domain, while the parent domain is informed and can continue operation. With respect to attack detection, we assume that an attack or fault will exhibit an illegal memory access that triggers a SEGFAULT signal. We then use signal handlers to detect and handle the failure, leading to a termination of the crashed child domain and a rewinding of the parent domain to a well-defined state. Compartmentalization is achieved by implementing the following two requirements:

R2 The mechanism must ensure that the integrity of memory after recovering from the detected attack is maintained.

R3 Run-time attacks that affect one domain must not affect the integrity of memory in other domains.

In our implementation of SDRaD, we use PKU as a mechanism to enforce in-process isolation while facilitating efficient domain switches. The security of this mechanism critically relies on protecting potential gadgets in the SDRaD implementation that allow an attacker to manipulate the PKRU register.

To guarantee the security of SDRaD, the following orthogonal defenses need to be in place: 1) PKU crucially relies on untrusted domains to not contain unsafe WRPKRU or XRSTOR instructions that manipulate the PKRU register [82]. This can be guaranteed through W⊕X and binary inspection [47]. Alternatively, hardware designs for PKU-like security features restrict access to userspace configuration registers [51]. 2) Since the SDRaD necessarily contains WRPKRU instructions, we must employ a CFI mechanism to protect the API implementation of Reference Monitor and statically ensure that its API does not contain abusable WRPKRU or XRSTOR gadgets. 3) An alternative way to modify PKRU is by utilizing sigreturn is described in [82], which can be mitigated by ASLR [83] and requires kernel-level authentication of sigframe data [15]. 4) Existing PKU sandboxes do not sufficiently safeguard the syscall interface [82], [84]. Several researchers propose efficient syscall filtering to prevent untrusted domains from invoking unsafe syscalls [53], [84]. With the above security mechanisms in place, SDRaD satisfies our fourth requirement:

R4 The attacker must not be able to tamper with components responsible for isolation or transitions between domains, or data used as part of the rewinding process.

It is possible to implement secure rewind and discard of isolated domains on top of other isolation mechanisms e.g., within Intel SGX to equip enclaves with rewinding or by using capability-based enforcement of isolation, e.g., CHERI. Such uses will incur different low-level security requirements and exhibit different performance characteristics. Furthermore, SDRaD is not limited to rely on SEGFAULT handling but could employ different attack oracles that, e.g., trigger when a domain invokes an unexpected system call.

Applicability As highlighted earlier, secure rewind and discard of isolated domains is particularly suited for service-oriented applications that need strong availability guarantees and may hold volatile state like client sessions, TLS connections, or object caches. Redundancy and load balancing can minimize the impact of DoS attacks, but loss of volatile state can still degrade service quality for clients, which our approach mitigates.

As demonstrated by our case study on Memcached (§ V-A), applications such as caching (proxy) servers and web accelerators which exhibit the *cold start problem*, i.e., the system does not reach its normal operation capacity until some time after initial start or restart, can benefit greatly from SDRaD.

A prime target for our mechanism are subroutines and libraries that handle sequences of external, untrusted, unsanitized data, e.g., functions that perform input validation, JavaScript engines in web browsers, database front-ends, or video, image, and document renderers, due to a heightened degree of exposure towards potential attacks. Isolating such components in their own domain allows recovering from potential memory corruptions. Furthermore, rewindings in long-running services may be reported as incidents to a Security Information and Event Management system, serving as early warning signals of an attack campaign. Blocking malicious clients via firewall rules as a response may then shield the overall system from repeated attacks.

In practice, the specific setup of domains and protections as well as the acceptable trade-off between the performance impact and provided benefit in resiliency will depend highly on application architecture and use case. As such, our design incorporates different options for compartmentalization, aiding adoption of the rewind mechanism in new and existing developments. We see retrofitting applications written in unsafe programming languages as a compelling use case in-lieu of a complete re-write in a memory-safe language.

Applications written in memory-safe languages are inherently protected against the run-time attacks we consider in this work. Nevertheless, even such applications may depend on unsafe libraries called via foreign function interfaces (FFIs). Adapting SDRaD for use with FFI from other programming languages is, however, outside the scope of this work.

Limitations We presented several examples where our approach substantially improved the dependability of relevant applications and where the refactoring required one to three person weeks of a developer not previously familiar with the code base. The effort is comparable to retrofitting applications to make use of Trusted Execution Enviroments (TEEs) such as Intel SGX and the process may be supported by automated tools, e.g., [85]. However, it is clear that not all applications can be easily compartmentalized and refactored to make use of SDRaD. For example, applications that rely on global mutexes may suffer from availability issues when a child domain holding a lock crashes and the lock is not released prior to continuation of the parent domain. Options for resolving this are, e.g., an SDRaD-aware locking mechanism as part of our library, or to employ local locks with well-defined scope instead of global locks. This also aids serializing access, e.g., when domains operate on copies of protected objects. Generally, isolating routines that work on shared state is tricky but possible using the deferred update method demonstrated for Memcached, as long as the isolated routine updates the shared state atomically and at most once.

Another potential issue comes with complex data structures used by target applications. Similar to other strong isolation mechanisms such as Intel SGX, data needs to be copied into the address space of the protection domain [86], which is done by entry wrappers. Both, manual as well as automated generation of these wrappers can be error prone and may hamper security [87]. Generally, domain transitions and domain termination bear subtle risks. Currently, confidentiality of child domain data is not guaranteed after destroying it and we leave it to the developer to realize such requirements, e.g., scrub sensitive allocations from memory before leaving the domain.

Using SDRaD to increase the availability of long-running services may open up a side-channel attack surface as observable effects of a rewind (e.g., delayed execution) can give an attacker insights into an application's execution. Coupled with the absence of re-randomization of the application's memory layout, an attacker could potentially use this to break probabilistic defenses such as ASLR. A potential protection against such attacks is to force an application restart after a configurable number of rewindings, similarly to probabilistic defenses for pre-forking applications [14].

Moreover, the defense mechanisms used in our approach are not perfect, i.e., memory or control flow may still be compromised by an undetected attack. While our approach is orthogonal to the mechanisms used, the compartmentalization of a program into domains may still help discover subsequent exploitation of such compromise more quickly as malicious memory accesses may potentially cause domain violations.

Ultimately, the security of SDRaD depends on the correctness of our library implementation and further exploration of the attack surface as well as potentially formal verification of our code are envisaged to harden our approach.

VII. Related Work

Hardware-assisted compartmentalization The idea of using MPK for compartmentalizing applications is not new; PKU in 64-bit x86 is used to augment SFI approaches that generally suffer from high enforcement overheads [23], [88]. Such work falls into two categories: 1) in-process isolation [44]–[47], [51]–[56], and 2) isolation for unikernels and library OSs [48]–[50]. Lack of PKRU access control has been scrutinized for leaving PKU-based schemes vulnerable to bypass of established isolation domains [53], [82], [84]. Proposed countermeasures include code rewriting [45], binary inspection [47], system call filtering [53], [84] and variations on the PKU hardware design [51], [89], [90]. Secure multi-threading has also been considered [54], [91], [92]. However, in-process SFI does not consider how to recover from attacks. This work addresses this gap by introducing capabilities for secure rewinding. Light-Weight Contexts [34] introduce MMU-based in-process isolation with a notion of rewinding. In difference to our work, [34] requires OS extensions and has unclear performance characteristics. Capability schemes, such as CHERI [93], [94] also enables compartmentalized fault isolation. CompartOS [38] provides recovery capabilities, but unlike SDRaD, it is geared toward safety-critical embedded systems, not COTS processors.
Checkpoint & restore Existing approaches to checkpoint & restore, such as CRIU [95] provide support for process snapshots that can enable rollback-like functionality. However, checkpoint & restore approaches generally suffer from high overheads due to relying on reproducing process memory, do not consider in-memory attacks in their threat models [30]–[33], [96], or target special computing paradigms, such as functions-as-a-service [97], [98]. SDRaD avoids these drawbacks by utilizing in-process isolation to limit the scope of attacks and to ensure the integrity of memory after rewinding.
N-variant Execution N-variant Execution (NVX) [99] provides resilience by introducing redundancy through running multiple, artificially diversified variants of the same application in tandem and terminating instances that show divergent behavior. While SDRaD shares the goal of improving software resilience, our work targets use cases for which the high cost of replicating computations and I/O across each instance is impractical.
SDRaD In comparison with related work, SDRaD aims at improving dependability of applications on COTS processors, e.g., in cloud settings, allowing recovery from attacks that compromise heap or stack memory. SDRaD requires no OS changes, incurs comparatively small runtime overheads and enables very fast application recovery, at the expense of a non-trivial but feasible engineering effort to adapt the application. This combination of features makes our approach unique in the area of dependable software.

VIII. Conclusion & Future Work

We presented the novel concept of *secure rewind and discard of isolated domains*, which complements protection mechanisms against memory safety vulnerabilities. It provides a hardening mechanism to recover from detected violations, thus improving the availability and resilience of software applications. At its core, secure rewind uses hardware-assisted in-process memory isolation to sandbox exposed functionality in separate domains: a compromise in one domain cannot spread to other parts of a program's memory. When a compromise is detected by selected defense mechanisms, a rewinding to a previously defined consistent state of the application occurs, enabling error handling and efficiently resuming the application.

We introduced *SDRaD*, our prototype library implementation of secure domain rewind and discard. We demonstrated its applicability to real software by adding it to the multi-threaded Memcached system, multiprocessing NGINX web server, and the OpenSSL library, exhibiting marginal performance overhead. Applications profit not only from faster recovery times due to the rewind mechanism, but also from avoiding possible service disruption for clients after an application crash and subsequent connection loss. In practice, no compartmentalization strategy will likely be able to recover from each and every fault or attack. Still, providing an amenable, secure, and efficient implementation of the secure rewind mechanism will fill an important gap in the current software security architecture.

A technical report with extended insights into the workings of SDRaD, our prototypic implementation, and further elaboration on the cases studies is available at [66]. Our implementation artifacts are available under a BSD license on GitHub [27]: https://secure-rewind-and-discard.github.io/

Acknowledgments

We thank Ilhan Gürel, Michael Liljestam, Eddy Truyen, Stijn Volckaert and Anjo Vahldiek-Oberwagner for their helpful feedback. This work has received funding under EU H2020 MSCA-ITN action 5GhOSTS, grant agreement no. 814035, by the Research Fund KU Leuven, by the Flemish Research Programme Cybersecurity, and by the CyberExcellence programme of the Walloon Region, Belgium.

References

[1] Ú. Erlingsson, Y. Younan, and F. Piessens, "Low-level software security by example," in *Handbook of Information and Communication Security*. Springer, 2010, pp. 633–658. [Online]. Available: https://doi.org/10.1007/978-3-642-04117-4_30

[2] S. Chen, J. Xu, and E. C. Sezer, "Non-control-data attacks are realistic threats," in *14th USENIX Security Symposium (USENIX Security 05)*. Baltimore, MD: USENIX Association, Jul. 2005. [Online]. Available: https://www.usenix.org/conference/14th-usenix-security-symposium/non-control-data-attacks-are-realistic-threats

[3] Google Project Zero, "0day "in the wild" dataset," June 2022, retrieved February 26, 2023 from https://web.archive.org/web/20230226000529/https://googleprojectzero.blogspot.com/p/0day.html.

[4] L. Szekeres, M. Payer, T. Wei, and D. Song, "SoK: Eternal war in memory," in *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, ser. SP '13. Washington, DC, USA: IEEE, 2013, pp. 48–62. [Online]. Available: http://doi.org/10.1109/SP.2013.13

[5] K. Lu, C. Song, B. Lee, S. P. Chung, T. Kim, and W. Lee, "Aslr-guard: Stopping address space leakage for code reuse attacks," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 280–291. [Online]. Available: https://doi.org/10.1145/2810103.2813694

[6] P. Wagle and C. Cowan, "StackGuard: Simple stack smash protection for gcc," in *Proceedings of the GCC Developers Summit*, 01 2003.

[7] R. de Clercq and I. Verbauwhede, "A survey of hardware-based control flow integrity (cfi)," `arXiv:1706.07257 [cs.CR]`, 2017. [Online]. Available: https://arxiv.org/ftp/arxiv/papers/1706/1706.07257.pdf

[8] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," *ACM Trans. Inf. Syst. Secur.*, vol. 13, no. 1, pp. 4:1–4:40, Nov. 2009. [Online]. Available: https://doi.org/10.1145/1609956.1609960

[9] N. Burow, S. A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler, and M. Payer, "Control-flow integrity: Precision, security, and performance," *ACM Comput. Surv.*, vol. 50, no. 1, pp. 16:1–16:33, Apr. 2017. [Online]. Available: http://doi.org/10.1145/3054924

[10] M. Castro, M. Costa, J.-P. Martin, M. Peinado, P. Akritidis, A. Donnelly, P. Barham, and R. Black, "Fast byte-granularity software fault isolation," in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, ser. SOSP '09. New York, NY, USA: ACM, 2009, pp. 45–58. [Online]. Available: http://doi.org/10.1145/1629575.1629581

[11] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton, "StackGuard: Automatic adaptive detection and prevention of Buffer-Overflow attacks," in *7th USENIX Security Symposium (USENIX Security 98)*. San Antonio, TX: USENIX Association, Jan. 1998. [Online]. Available: https://www.usenix.org/legacy/publications/library/proceedings/sec98/full_papers/cowan/cowan.pdf

[12] U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula, "XFI: Software guards for system address spaces," in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, ser. OSDI '06. Berkeley, CA, USA: USENIX Association, 2006, pp. 75–88. [Online]. Available: https://www.usenix.org/legacy/event/osdi06/tech/full_papers/erlingsson/erlingsson.pdf

[13] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz, "SoK: Automated software diversity," in *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, ser. SP '14. Washington, DC, USA: IEEE, 2014, pp. 276–291. [Online]. Available: http://doi.org/10.1109/SP.2014.25

[14] H. Liljestrand, T. Nyman, K. Wang, C. C. Perez, J.-E. Ekberg, and N. Asokan, "PAC it up: Towards pointer integrity using ARM pointer authentication," in *28th USENIX Security Symposium*, ser. USENIX Security '19. Berkeley, CA, USA: USENIX Association, Aug. 2019, pp. 177–194. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/liljestrand

[15] H. Liljestrand, T. Nyman, L. J. Gunn, J.-E. Ekberg, and N. Asokan, "PACStack: an authenticated call stack," in *30th USENIX Security Symposium (USENIX Security 21)*. Berkeley, CA, USA: USENIX Association, Aug. 2021, pp. 357–374. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/liljestrand

[16] Y. Mao, H. Chen, D. Zhou, X. Wang, N. Zeldovich, and M. F. Kaashoek, "Software fault isolation with api integrity and multi-principal modules," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, ser. SOSP '11. New York, NY, USA: ACM, 2011, pp. 115–128. [Online]. Available: http://doi.org/10.1145/2043556.2043568

[17] B. Niu and G. Tan, "RockJIT: Securing just-in-time compilation using modular control-flow integrity," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '14. New York, NY, USA: ACM, 2014, p. 1317–1328. [Online]. Available: https://doi.org/10.1145/2660267.2660281

[18] ——, "Per-input control-flow integrity," in *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15. New York, NY, USA: ACM, 2015, pp. 914–926. [Online]. Available: http://doi.org/10.1145/2810103.2813644

[19] S. Park, S. Lee, W. Xu, H. Moon, and T. Kim, "Libmpk: Software abstraction for Intel Memory Protection Keys (Intel MPK)," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA, USA: USENIX Association, Jul. 2019, p. 241–254. [Online]. Available: https://www.usenix.org/system/files/atc19-park-soyeon.pdf

[20] E. J. Schwartz, T. Avgerinos, and D. Brumley, "Q: Exploit hardening made easy," in *20th USENIX Security Symposium (USENIX Security 11)*. San Francisco, CA: USENIX Association, Aug. 2011. [Online]. Available: https://www.usenix.org/conference/usenix-security-11/q-exploit-hardening-made-easy

[21] K. Serebryany, "ARM Memory Tagging Extension and how it improves C/C++ memory safety," *;login: The USENIX Magazine*, vol. 48, no. 2, pp. 12–16, 2019. [Online]. Available: https://www.usenix.org/system/files/login/articles/login_summer19_03_serebryany.pdf

[22] V. van der Veen, D. Andriesse, E. Göktaş, B. Gras, L. Sambuc, A. Slowinska, H. Bos, and C. Giuffrida, "Practical context-sensitive CFI," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15. New York, NY, USA: ACM, 2015, p. 927–940. [Online]. Available: https://doi.org/10.1145/2810103.2813673

[23] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, "Efficient software-based fault isolation," in *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '93. New York, NY, USA: ACM, 1993, pp. 203–216. [Online]. Available: http://doi.org/10.1145/168619.168635

[24] T. Herault and Y. Robert, *Fault-Tolerance Techniques for High-Performance Computing*, 1st ed. Springer Publishing Company, Incorporated, 2015. [Online]. Available: https://doi.org/10.1007/978-3-319-20943-2

[25] *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide*, Intel Corporation, 2007, order Number: 325462-076US https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

[26] *AMD64 Architecture Programmer's Manual Volume 2: System Programming. Revision 3.38*, AMD, 2021, publication No. 24593 https://www.amd.com/system/files/TechDocs/24593.pdf.

[27] M. Gülmez, online, 2023. [Online]. Available: https://secure-rewind-and-discard.github.io/

[28] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson, "A survey of rollback-recovery protocols in message-passing systems," *ACM Comput. Surv.*, vol. 34, no. 3, p. 375–408, sep 2002. [Online]. Available: https://doi.org/10.1145/568522.568525

[29] B. Randell, "System structure for software fault tolerance," *SIGPLAN Not.*, vol. 10, no. 6, p. 437–449, apr 1975. [Online]. Available: https://doi.org/10.1145/390016.808467

[30] I. Zhang, T. Denniston, Y. Baskakov, and A. Garthwaite, "Optimizing VM checkpointing for restore performance in VMware ESXi," in *2013 USENIX Annual Technical Conference (USENIX ATC 13)*. San Jose, CA: USENIX Association, Jun. 2013, pp. 1–12. [Online].

Available: https://www.usenix.org/conference/atc13/technical-sessions/presentation/zhang

[31] J. W. Young, "A first order approximation to the optimum checkpoint interval," *Commun. ACM*, vol. 17, pp. 530–531, 1974. [Online]. Available: https://doi.org/10.1145/361147.361115

[32] Y. Liu, R. Nassar, C. Leangsuksun, N. Naksinehaboon, M. Paun, and S. L. Scott, "An optimal checkpoint/restart model for a large scale high performance computing system," in *2008 IEEE International Symposium on Parallel and Distributed Processing*, 2008, pp. 1–9. [Online]. Available: https://doi.org/10.1109/IPDPS.2008.4536279

[33] H. Nam, J. Kim, S. J. Hong, and S. Lee, "Secure checkpointing," *Journal of Systems Architecture*, vol. 48, no. 8, pp. 237–254, 2003. [Online]. Available: https://doi.org/10.1016/S1383-7621(02)00137-6

[34] J. Litton, A. Vahldiek-Oberwagner, E. Elnikety, D. Garg, B. Bhattacharjee, and P. Druschel, "Light-weight contexts: An os abstraction for safety and performance," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'16. USA: USENIX Association, 2016, p. 49–64.

[35] A. Lenharth, V. S. Adve, and S. T. King, "Recovery domains: An organizing principle for recoverable operating systems," *SIGARCH Comput. Archit. News*, vol. 37, no. 1, p. 49–60, mar 2009. [Online]. Available: https://doi.org/10.1145/2528521.1508251

[36] M. M. Swift, B. N. Bershad, and H. M. Levy, "Improving the reliability of commodity operating systems," in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '03. New York, NY, USA: Association for Computing Machinery, 2003, p. 207–222. [Online]. Available: https://doi.org/10.1145/945445.945466

[37] M. M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy, "Recovering device drivers," *ACM Trans. Comput. Syst.*, vol. 24, no. 4, p. 333–360, nov 2006. [Online]. Available: https://doi.org/10.1145/1189256.1189257

[38] H. Almatary, M. Dodson, J. Clarke, P. Rugg, I. Gomes, M. Podhradsky, P. G. Neumann, S. W. Moore, and R. N. M. Watson, "CompartOS: CHERI compartmentalization for embedded systems," `arXiv:2206.02852 [cs.CR]`, 2022. [Online]. Available: https://arxiv.org/abs/2206.02852

[39] G. Tan, *Principles and Implementation Techniques of Software-Based Fault Isolation*. Hanover, MA, USA: Now Publishers Inc., 2017. [Online]. Available: http://dx.doi.org/10.1561/3300000013

[40] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, "Native client: A sandbox for portable, untrusted x86 native code," in *2009 30th IEEE Symposium on Security and Privacy*, 2009, pp. 79–93. [Online]. Available: http://doi.acm.org/10.1145/1629175.1629203

[41] N. Roessler, L. Atayde, I. Palmer, D. McKee, J. Pandey, V. P. Kemerlis, M. Payer, A. Bates, A. DeHon, J. M. Smith *et al.*, "μscope: A methodology for analyzing least-privilege compartmentalization in large software artifacts," 2021. [Online]. Available: https://doi.org/10.1145/3471621.3471839

[42] M. Sun and G. Tan, "JVM-portable sandboxing of java's native libraries," in *Computer Security – ESORICS 2012*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 842–858. [Online]. Available: https://doi.org/10.1007/978-3-642-33167-1_48

[43] S. Liu, G. Tan, and T. Jaeger, "PtrSplit: Supporting general pointers in automatic program partitioning," in *Proceedings of the 24th ACM Conference on Computer and Communications Security (ACM CCS)*, 2017. [Online]. Available: https://doi.org/10.1145/3133956.3134066

[44] E. E. Rivera, "Preserving memory safety in safe rust during interactions with unsafe languages," Master's thesis, Department of Electrical Engineering and Computer Science, 2016. [Online]. Available: https://dspace.mit.edu/bitstream/handle/1721.1/139052/Rivera-eerivera-meng-eecs-2021-thesis.pdf?sequence=1&isAllowed=y

[45] K. Koning, X. Chen, H. Bos, C. Giuffrida, and E. Athanasopoulos, "No need to hide: Protecting safe regions on commodity hardware," in *Proceedings of the Twelfth European Conference on Computer Systems*, ser. EuroSys '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 437–452. [Online]. Available: https://doi.org/10.1145/3064176.3064217

[46] M. Hedayati, S. Gravani, E. Johnson, J. Criswell, M. L. Scott, K. Shen, and M. Marty, "Hodor: Intra-process isolation for high-throughput data plane libraries," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, Jul. 2019, pp. 489–504. [Online]. Available: https://www.usenix.org/conference/atc19/presentation/hedayati-hodor

[47] A. Vahldiek-Oberwagner, E. Elnikety, N. O. Duarte, M. Sammler, P. Druschel, and D. Garg, "ERIM: Secure, efficient in-process isolation with protection keys (MPK)," in *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1221–1238. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/vahldiek-oberwagner

[48] M. S. Melara, M. J. Freedman, and M. Bowman, "Enclavedom: Privilege separation for large-tcb applications in trusted execution environments," `arXiv:1907.13245 [cs.CR]`, 2019. [Online]. Available: http://arxiv.org/abs/1907.13245

[49] M. Sung, P. Olivier, S. Lankes, and B. Ravindran, "Intra-unikernel isolation with intel memory protection keys," in *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 143–156. [Online]. Available: https://doi.org/10.1145/3381052.3381326

[50] H. Lefeuvre, V.-A. Bădoiu, c. Teodorescu, P. Olivier, T. Mosnoi, R. Deaconescu, F. Huici, and C. Raiciu, "Flexos: Making os isolation flexible," in *Proceedings of the Workshop on Hot Topics in Operating Systems*, ser. HotOS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 79–87. [Online]. Available: https://doi.org/10.1145/3458336.3465292

[51] D. Schrammel, S. Weiser, S. Steinegger, M. Schwarzl, M. Schwarz, S. Mangard, and D. Gruss, "Donky: Domain keys – efficient in-process isolation for risc-v and x86," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 1677–1694. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/schrammel

[52] X. Wang, S. Yeoh, P. Olivier, and B. Ravindran, "Secure and efficient in-process monitor (and library) protection with Intel MPK," in *Proceedings of the 13th European Workshop on Systems Security*, ser. EuroSec '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 7–12. [Online]. Available: https://doi.org/10.1145/3380786.3391398

[53] A. Voulimeneas, J. Vinck, R. Mechelinck, and S. Volckaert, "You shall not (by)pass! practical, secure, and fast pku-based sandboxing," in *Proceedings of the Seventeenth European Conference on Computer Systems*, ser. EuroSys '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 266–282. [Online]. Available: https://doi.org/10.1145/3492321.3519560

[54] P. Kirth, M. Dickerson, S. Crane, P. Larsen, A. Dabrowski, D. Gens, Y. Na, S. Volckaert, and M. Franz, "PKRU-Safe: Automatically locking down the heap between safe and unsafe languages," in *Proceedings of the Seventeenth European Conference on Computer Systems*, ser. EuroSys '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 132–148. [Online]. Available: https://doi.org/10.1145/3492321.3519582

[55] X. Jin, X. Xiao, S. Jia, W. Gao, D. Gu, H. Zhang, S. Ma, Z. Qian, and J. Li, "Annotating, tracking, and protecting cryptographic secrets with cryptompk," in *2022 IEEE Symposium on Security and Privacy (SP)*, 2022, pp. 650–665. [Online]. Available: https://ieeexplore.ieee.org/document/9833650

[56] Y. Chen, J. Li, G. Xu, Y. Zhou, Z. Wang, C. Wang, and K. Ren, "SGXLock: Towards efficiently establishing mutual distrust between host application and enclave for SGX," in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/chen-yuan

[57] *ARMv8-A Architecture Reference Manual, Version E.a*, Arm Ltd., 2019, https://static.docs.arm.com/ddi0487/ea/DDI0487E_a_armv8_arm.pdf.

[58] *Programming for AIX 7.3: Storage Protect Keys*, IBM, 2022, retrieved May 9, 2022 from http://web.archive.org/web/20220509005203/https://www.ibm.com/docs/en/aix/7.3?topic=concepts-storage-protect-keys.

[59] *PA-RISC 1.1 Architecture and Instruction Set Reference mMnual, Third Edition*, Hewlett Packard, 1994, hP Part Number: 09740-90039 https://parisc.wiki.kernel.org/images-parisc/6/68/Pa11_acd.pdf.

[60] *Intel IA-64 Architecture Software Developer's Manual Volume 1: IA-64 Application Architecture. Revision 1.1*, Intel Corporation, 2000, Document Number: 245317-002 http://refspecs.linux-foundation.org/IA64-softdevman-vol1.pdf.

[61] W. Xiong and J. Szefer, "Survey of transient execution attacks and their mitigations," *ACM Comput. Surv.*, vol. 54, no. 3, may 2021. [Online]. Available: https://doi.org/10.1145/3442479

[62] C. Shepherd, K. Markantonakis, N. van Heijningen, D. Aboulkassimi, C. Gaine, T. Heckmann, and D. Naccache, "Physical fault injection and side-channel attacks on mobile devices: A comprehensive analysis,"

Computers & Security, vol. 111, p. 102471, 2021. [Online]. Available: https://doi.org/10.1016/j.cose.2021.102471

[63] O. Mutlu and J. S. Kim, "Rowhammer: A retrospective," *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, vol. 39, no. 8, p. 1555–1571, aug 2020. [Online]. Available: https://doi.org/10.1109/TCAD.2019.2915318

[64] Linux Manual page, "setjmp(3)," August 2021, retrieved December 9, 2022 from http://web.archive.org/web/20221209081924/https://www.man7.org/linux/man-pages/man3/setjmp.3.html.

[65] *Intel Software Guard Extensions SDK for Linux OS*, Intel Corporation, 2016, https://01.org/sites/default/files/documentation/intel_sgx_sdk_developer_reference_for_linux_os_pdf.pdf.

[66] M. Gülmez, T. Nyman, C. Baumann, and J. T. Mühlberg, "Unlimited lives: Secure in-process rollback with isolated domains," arXiv:2205.03205 [cs.CR], 2022. [Online]. Available: https://arxiv.org/abs/2205.03205

[67] Linux Manual page, "sigaction(2)," August 2021, retrieved February 16, 2023 from http://web.archive.org/web/20230216011223/https://www.man7.org/linux/man-pages/man2/sigaction.2.html.

[68] Clang 15.0.0git documentation, "Control flow integrity," February 2022, retrieved February 18, 2023 from https://web.archive.org/web/20230226000332/https://clang.llvm.org/docs/ControlFlowIntegrity.html.

[69] glibc wiki, "Overview of malloc," May 2019, retrieved December 18, 2022 from https://web.archive.org/web/20221218030157/https://sourceware.org/glibc/wiki/MallocInternals.

[70] M. Conte, "Github - mattconte/tlsf: Two-level segregated fit memory allocator implementation," April 2016, retrieved December 7, 2022 from https://web.archive.org/web/20221207145055/http://github.com/mattconte/tlsf.

[71] M. Masmano, I. Ripoll, A. Crespo, and J. Real, "TLSF: a new dynamic memory allocator for real-time systems," in *Proceedings. 16th Euromicro Conference on Real-Time Systems, 2004. ECRTS 2004.*, 2004, pp. 79–88. [Online]. Available: https://doi.org/10.1109/EMRTS.2004.1311009

[72] Memcached, March 2022, retrieved March 1, 2023 from https://web.archive.org/web/20230301062721/https://memcached.org/.

[73] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani, "Scaling memcache at Facebook," in *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, ser. nsdi'13. USA: USENIX Association, 2013, p. 385–398. [Online]. Available: https://www.usenix.org/system/files/conference/nsdi13/nsdi13-final170_update.pdf

[74] V. J. Marathe, M. Seltzer, S. Byan, and T. Harris, "Persistent memcached: Bringing legacy code to Byte-Addressable persistent memory," in *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*. Santa Clara, CA: USENIX Association, Jul. 2017. [Online]. Available: https://www.usenix.org/conference/hotstorage17/program/presentation/marathe

[75] Y. Zhang and S. Swanson, "A study of application performance with non-volatile main memory," in *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*, 2015, pp. 1–10. [Online]. Available: https://ieeexplore.ieee.org/document/7208275

[76] CVE-2011-4971, December 2021, retrieved March 02, 2023 from https://nvd.nist.gov/vuln/detail/CVE-2011-4971.

[77] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proceedings of the 1st ACM Symposium on Cloud Computing*, ser. SoCC '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 143–154. [Online]. Available: https://doi.org/10.1145/1807128.1807152

[78] NGINX, July 2022, retrieved March 01, 2023 from https://nginx.org/.

[79] B. Im, F. Yang, C. Tsai, M. LeMay, A. Vahldiek-Oberwagner, and N. Dautenhahn, "The Endokernel: Fast, secure, and programmable subprocess virtualization," arXiv:2108.03705 [cs.CR], 2021. [Online]. Available: https://arxiv.org/abs/2108.03705

[80] CVE-2009-2629, July 2009, retrieved March 02, 2023 from https://nvd.nist.gov/vuln/detail/CVE-2009-2629.

[81] CVE-2022-3786, November 2022, retrieved March 02, 2023 from https://nvd.nist.gov/vuln/detail/CVE-2022-3786.

[82] R. J. Connor, T. McDaniel, J. M. Smith, and M. Schuchard, "PKU pitfalls: Attacks on PKU-based memory isolation systems," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 1409–1426. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/connor

[83] J. Corbet, "Sigreturn-oriented programming and its mitigation," February 2016, retrieved November 17, 2022 from https://web.archive.org/web/20221117234843/https://lwn.net/Articles/676803/.

[84] D. Schrammel, S. Weiser, R. Sadek, and S. Mangard, "Jenny: Securing syscalls for PKU-based memory isolation systems," in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/schrammel

[85] J. Lind, C. Priebe, D. Muthukumaran, D. O'Keeffe, P. Aublin, F. Kelbert, T. Reiher, D. Goltzsche, D. Eyers, R. Kapitza *et al.*, "Glamdring: Automatic application partitioning for Intel SGX." USENIX, 2017.

[86] V. Costan and S. Devadas, "Intel SGX explained," *Cryptology ePrint Archive*, 2016. [Online]. Available: https://eprint.iacr.org/2016/086.pdf

[87] J. Van Bulck, D. Oswald, E. Marin, A. Aldoseri, F. D. Garcia, and F. Piessens, "A tale of two worlds: Assessing the vulnerability of enclave shielding runtimes," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 1741–1758. [Online]. Available: https://doi.org/10.1145/3319535.3363206

[88] D. Sehr, R. Muth, C. Biffle, V. Khimenko, E. Pasko, K. Schimpf, B. Yee, and B. Chen, "Adapting software fault isolation to contemporary CPU architectures," in *19th USENIX Security Symposium (USENIX Security 10)*. Washington, DC: USENIX Association, Aug. 2010. [Online]. Available: https://www.usenix.org/conference/usenixsecurity10/adapting-software-fault-isolation-contemporary-cpu-architectures

[89] L. Delshadtehrani, S. Canakci, M. Egele, and A. Joshi, "Sealpk: Sealable protection keys for risc-v," in *2021 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2021, pp. 1278–1281. [Online]. Available: https://ieeexplore.ieee.org/document/9473932

[90] T. Frassetto, P. Jauernig, C. Liebchen, and A.-R. Sadeghi, "IMIX: In-Process memory isolation EXtension," in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, pp. 83–97. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/frassetto

[91] Y. Chen, S. Reymondjohnson, Z. Sun, and L. Lu, "Shreds: Fine-grained execution units with private memory," in *2016 IEEE Symposium on Security and Privacy (SP)*, 2016, pp. 56–71. [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7546495

[92] Z. Tarkhani and A. Madhavapeddy, "μtiles: Efficient intra-process privilege enforcement of memory regions," arXiv:2004.04846 [cs.CR], 2020. [Online]. Available: https://arxiv.org/pdf/2004.04846.pdf

[93] J. Woodruff, R. N. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe, "The CHERI capability model: Revisiting RISC in an age of risk," in *Proceeding of the 41st Annual International Symposium on Computer Architecture*, ser. ISCA '14. Washington, DC, USA: IEEE, 2014, pp. 457–468. [Online]. Available: https://doi.org/10.1145/2678373.2665740

[94] R. N. M. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. Dave, B. Davis, K. Gudka, B. Laurie, S. J. Murdoch, R. Norton, M. Roe, S. Son, and M. Vadera, "CHERI: A hybrid capability-system architecture for scalable software compartmentalization," in *Proceedings of the 2015 IEEE Symposium on Security and Privacy*. Washington, DC, USA: IEEE, 2015, pp. 20–37. [Online]. Available: https://ieeexplore.ieee.org/document/7163016

[95] S. Kashyap, C. Min, B. Lee, T. Kim, and P. Emelyanov, "Instant OS updates via userspace Checkpoint-and-Restart," in *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. Denver, CO: USENIX Association, Jun. 2016, pp. 605–619. [Online]. Available: https://www.usenix.org/conference/atc16/technical-sessions/presentation/kashyap

[96] A. Webster, R. Eckenrod, and J. Purtilo, "Fast and service-preserving recovery from malware infections using CRIU," in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, pp. 1199–1211. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/webster

[97] S. Shillaker and P. Pietzuch, "FAASM: Lightweight isolation for efficient stateful serverless computing," in *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC'20. USA: USENIX Association, 2020.

[98] M. Alzayat, J. Mace, P. Druschel, and D. Garg, "Groundhog: Efficient request isolation in FaaS," arXiv:2205.11458 [cs.CR], 2022. [Online]. Available: http://arxiv.org/abs/2205.11458

[99] A. Voulimeneas, D. Song, F. Parzefall, Y. Na, P. Larsen, M. Franz, and S. Volckaert, "Distributed heterogeneous n-variant execution," in *Detection of Intrusions and Malware, and Vulnerability Assessment*. Cham: Springer International Publishing, 2020, pp. 217–237. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-030-52683-2_11