PKRU-Safe: Automatically Locking Down the Heap Between Safe and Unsafe Languages

Anonymous Author(s)
Submission Id: 318

Abstract

After more than twenty-five years of research, memory safety violations remain one of the major causes of security vulnerabilities in real-world software. Memory-safe languages, like Rust, have demonstrated that compiler technology can assist developers in writing efficient low-level code without the risk of memory corruption. However, many memory-safe languages still have to interface with unsafe code to some extent, which opens up the possibility for attackers to exploit memory-corruption vulnerabilities in the unsafe part of the system and subvert the safety guarantees provided by the memory-safe language.

In this paper, we present PKRU-Safe, an automated method for enforcing the principle of least privilege on unsafe components in mixed-language environments. PKRU-Safe ensures that unsafe (external) code cannot corrupt or otherwise abuse memory used exclusively by the safe-language components. Our approach is automated using traditional compiler infrastructure to limit memory accesses for developer-designated components efficiently. PKRU-Safe does not require any modifications to the program's original data flows or execution model. It can be adopted by projects containing legacy code with minimal effort, requiring only a small number of changes to a project's build files and dependencies, and a few lines of annotations for each untrusted library.

We apply PKRU-Safe to Servo, one of the largest Rust projects with approximately two million lines of Rust code (including dependencies) to automatically partition and protect the browser's heap from its JavaScript engine written in unsafe C/C++. Our detailed evaluation shows that PKRU-Safe is able to thwart real-world exploits, often without measurable overhead, and with a mean overhead under 11.55% in our most pessimistic benchmark suite. As the method is language agnostic and major prototype components operate directly on LLVM IR, applying our techniques to other languages is straightforward.

ACM Reference Format:
Anonymous Author(s). 2021. PKRU-Safe: Automatically Locking Down the Heap Between Safe and Unsafe Languages . In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Despite decades of research, memory-corruption bugs continue to plague real-world software. Google reported a range of 5-15 exploitable bugs in Chrome's renderer process across five major releases [22] and estimated that memory corruption accounts for 90% of the vulnerabilities in AOSP [64]. Similarly, Microsoft reported that 70% of security patches over the past 12 years addressed memory-corruption bugs [53].

New, memory-safe languages, like Rust [40], aim to combine safety and performance through a combination of advanced compile-time analysis and a rich type system, giving developers an option to write low-level code that is also secure. However, adoption of new programming languages takes time, and legacy parts of the code base force large software projects to interface with unsafe code [54]. Moreover, the abundance of unsafe code written before safe languages became a viable option makes it unlikely that all legacy code will be ported. This means that many applications written in safe languages will still interact with unsafe code, e.g., through libraries and other legacy components.

While operating systems traditionally provide abstractions to isolate applications from each other through virtual memory, there currently exists no standard way of isolating memory between components within an application. Although this has led to a number of proposed general isolation schemes in the related work, they all either require OS modifications [21, 36, 48, 65] or extensive rewriting of the application [34, 58, 66].

We present PKRU-Safe, the first intra-process isolation scheme for heap data in mixed-language environments that does not require OS modifications or rewriting of the application code, relying only on developer-provided annotations that operate at the library level. Our system automatically partitions the original program into two cooperative parts: the ***trusted compartment,*** $\mathcal{T}$, (i.e., Rust code and trusted dependencies) and the ***untrusted compartment,*** $\mathcal{U}$, (i.e., components written in an unsafe language which the developer does not wish to fully trust). Given this partitioning, PKRU-Safe prevents code that a developer specifically designates as *untrusted* (e.g., $\mathcal{U}$) from accessing data in $\mathcal{T}$ not directly shared with it.

PKRU-Safe works largely automatically by analyzing data flows and instrumenting the annotated application code as part of the compilation process. In our detailed evaluation we demonstrate the effectiveness of our approach using Rust as a memory-safe language environment which includes unsafe legacy components utilizing Intel's Memory-Protection Keys (MPK) as a fast, hardware-backed enforcement mechanism

to isolate unsafe memory domains.[1] Though previously only available on Xeon server chips, MPK PKU instruction extensions are now available in current generation AMD Ryzen and Intel 11th generation desktop client processors [23].

Our compiler analysis tracks data flows from allocation sites across the isolation boundary to determine which pieces of data are required to be shared across the boundary to maintain a functional program. Successfully implementing fine-grained heap isolation for large, real-world codebases requires us to solve a number of challenging problems, such as reliably tracking memory object provenances across compiler toolchains, overcoming the page-based protection granularity at run time, and scaling the annotation-based instrumentation to millions of code lines.

In summary, we make the following contributions. 1) We present PKRU-Safe, the first data-flow aware and fully automatic source-level compartmentalization framework that supports mixed-language environments without requiring OS modifications or application rewriting. 2) We implemented a prototype of PKRU-Safe and demonstrate our framework scales to large software projects such as Servo, a web browser which is written in Rust that also includes substantial portions of unsafe legacy code that requires least-privilege isolation. We will release our prototype as an open-source implementation. 3) We extensively evaluate and test our framework on a wide range of benchmarks with real-world applications, demonstrating PKRU-Safe is both effective and practical, showing a mean overhead under 11.55% in our most pessimistic benchmark, and often on par with unmodified applications.

2 Threat Model

PKRU-Safe's primary goal is to prevent unauthorized access and manipulation of heap data owned by the safe language through a memory corruption vulnerability in the unsafe components of the application. Since the safe and unsafe components share an address space, we must therefore consider a broad class of memory disclosure and corruption vulnerabilities. While the application itself may have exploitable bugs in the unsafe component, it is not itself malicious. Our standard assumptions are in line with the related work in this area [28, 35, 57, 59]:

- **Memory-corruption vulnerability.** The legacy part of the application contains a memory-corruption bug. This grants the attacker arbitrary read or write capabilities in any part of the application's address space.
- **Code-injection and code-reuse defense.** We assume a strong W⊕X policy [52] meaning no memory pages are ever simultaneously writable and executable. We assume that legacy code is built with control-flow integrity [9] including backward edge protection (e.g., using a shadow stack implementation [19]). This forces adversaries to resort to data-only attacks [20, 37, 39].
- **Hardware-backed Intra-Process Memory Protection.** We assume Intel Memory Protection Keys (MPK) [25] to be available on the target platform. PKRU[2] values, which control domain access privileges, remain in registers or MPK-protected memory so adversaries cannot directly manipulate them. Since CFI is in place an adversary cannot reuse MPK instructions inserted into the legacy part of the code.

Because our focus is on protecting heap data, we assume that the stack in $\mathcal{T}$ is protected with a Shadow Stack [19, 44] implementation that prevents tampering from $\mathcal{U}$, and that an adversary cannot corrupt stack data used by $\mathcal{T}$, even if it resides in memory accessible with $\mathcal{U}$.

As pointed out in Connor et al. [24], there are a variety of unexpected methods of circumventing MPK based protections. Addressing these is orthogonal to the problems addressed in this paper. Further, information disclosure attacks through side-channels such as transient execution are out of scope [29, 42, 47]. Likewise, we do not consider remote-fault injection attacks targeting hardware-backed isolation [17, 41, 62, 67].

3 Design

PKRU-Safe was designed to enforce fine-grained data access control without requiring developers to deeply understand the exact data flow of their application and all of its dependencies. Our system partitions the original program into two parts: the ***trusted compartment,*** $\mathcal{T}$, (i.e., Rust code and trusted dependencies) and the ***untrusted compartment,*** $\mathcal{U}$, (i.e., components written in an unsafe language which the developer does not wish to fully trust). PKRU-Safe prevents code that a developer specifically tags as *untrusted* (e.g., $\mathcal{U}$) from accessing data in $\mathcal{T}$ not directly shared with it. On its own, the code that makes up $\mathcal{U}$ is not malicious, but because it lacks sufficient safety guarantees, it should not be granted unrestricted access to all of the application's data. Limiting access in this manner more closely aligns with the principle of least privilege and reduces the available attack surface.

Because $\mathcal{T}$ is entirely composed of code from the memory-safe language and its trusted dependencies, we grant it an unrestricted view of the application's data, both its own private memory, which we refer to as ***trusted memory*** ($\mathcal{M}_{\mathcal{T}}$), and the memory used by $\mathcal{U}$, which we refer to as ***untrusted memory*** ($\mathcal{M}_{\mathcal{U}}$). This is a key factor in why we target memory-safe languages as the target for $\mathcal{T}$, since it

[1] Intel recently announced a supervisor mode feature for the MPK processor extensions, called PKS. Throughout the paper we refer to the userland feature of MPK, called PKU.

[2] In Intel's MPK architecture, the Protection Key Rights Register (PKRU) defines the access rights for each active key.

allows us to both simplify our reasoning regarding cross-compartment sharing and aligns well with our goal: restricting data-access rights of $\mathcal{U}$. In contrast, $\mathcal{U}$ only has access to $\mathcal{M}_{\mathcal{U}}$, which is accessible by all compartments. Thus $\mathcal{M}_{\mathcal{U}}$ holds the data shared between $\mathcal{T}$ and $\mathcal{U}$, and the private data of $\mathcal{U}$.

3.1 Overview

As depicted in Figure 1 our design relies on ① developer annotations which define the explicit interface between $\mathcal{T}$ and $\mathcal{U}$. Given that boundary, we use ② compiler-based instrumentation to partition the program along the compartment boundary in conjunction with our provenance tracking runtime to identify cross-compartment data sharing. Profiling inputs are then used to collect runtime profiles ③ that can determine if an allocation site from $\mathcal{T}$ is used by $\mathcal{U}$. The profiling information is then fed back into the ④ compiler toolchain and used to change shared allocations in *trusted memory* ($\mathcal{M}_{\mathcal{T}}$) to instead come from *untrusted memory* ($\mathcal{M}_{\mathcal{U}}$) in final version of the ⑤ application. This four stage compilation pipeline ensures that memory access permissions are correctly modified when transitioning between compartments and that each compartment is still able to correctly access program data.

3.2 Compartment Identification

To automate memory isolation, our system requires a definition of the compartment boundary. PKRU-Safe operates at the level of function calls and therefore needs some method for disambiguating calls to code in $\mathcal{U}$ from those in $\mathcal{T}$.

There are various ways this problem could be addressed. First, we could simply instrument *all* interfaces to libraries written in an unsafe language. This approach, however, would mean that we could not necessarily trust code in standard or system libraries which operate through the Foreign Function Interface (FFI). Further, it would force developers to distrust other dependencies that interface with external code, even when they do not impact security or may be required by the platform. We believe that it is far more common for a developer to want to selectively limit access to some subset of the application rather than to require such a drastic form of sandboxing, for example restricting data access privileges to core components that must process untrusted inputs. However, we see no fundamental reason why our approach could not work with a more restrictive policy.

Second, we could use some form of heuristic or static analysis to identify the boundary. However, we know of no satisfactory automated way to reason about a security partitioning, given the complex nature of modern applications. Prior work that has followed this approach have often made assumptions about the semantics of sensitive operations and well known library interfaces, but still rely on developers to provide new security rules [16, 32, 36] or to model portions of the system outside of the analysis [33, 61].

We chose to require developers to annotate their program to restrict data access to specific components. Annotations provide a nice middle ground between a heuristic approach and fine-grained manual refactoring. Our annotations operate at the level of library interfaces, and therefore only require a small number of changes to project or source files. In essence, the annotations we employ are declarative, and offer a less burdensome alternative to process level sandboxing, which often requires substantial refactoring.

The annotations provide a tractable way to reason about compartment transitions from $\mathcal{T}$ to $\mathcal{U}$, and allow us to reason about the legal, cross-compartment data flows the partitioning should enforce. Given that we now have a concrete mapping of compartment transitions from $\mathcal{T}$ to $\mathcal{U}$, we must address transitions from $\mathcal{U}$ to $\mathcal{T}$. We conservatively assume that any function from $\mathcal{T}$ that may be called from $\mathcal{U}$ needs to be able to return to $\mathcal{T}$. This includes any address-taken function from $\mathcal{T}$, since it may be used as a callback from $\mathcal{U}$.

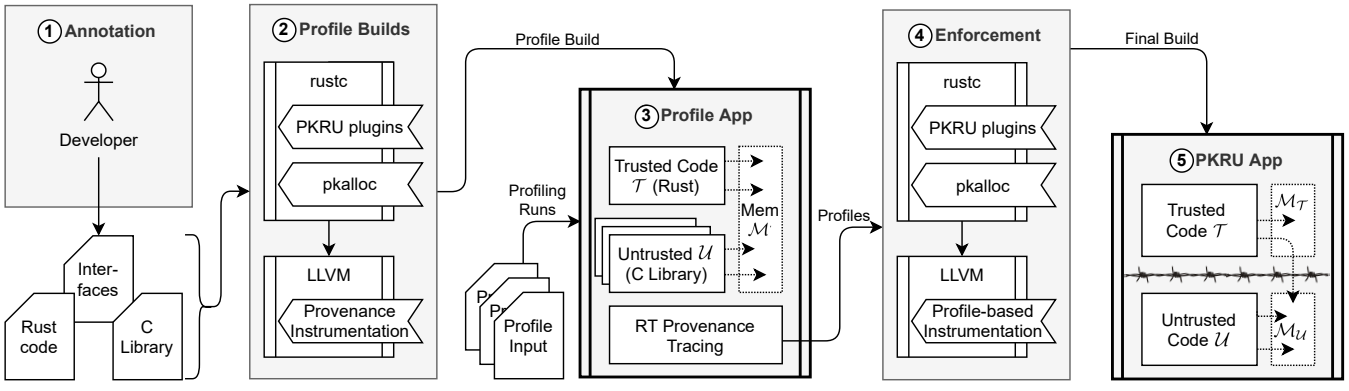

Figure 1. PKRU-Safe's dynamic workflow and main components: The developer annotates the intended interface between the safe and unsafe language environments. However, since allocations may propagate through the application at runtime via complicated data flows, we use dynamic profiling to identify allocations that must come from $\mathcal{M}_{\mathcal{U}}$. Finally, PKRU-Safe emits the instrumented executable that enforces the intended data flow policy at runtime.

3.3 Basic Instrumentation

In PKRU-Safe, each interface from $\mathcal{T}$ to $\mathcal{U}$ is transparently wrapped so that any call to the function from $\mathcal{T}$ must first revoke its ability to access $\mathcal{M}_{\mathcal{T}}$. When execution resumes in $\mathcal{T}$, the previous memory permissions are restored. Note that we do not assume the previous permissions allowed access to $\mathcal{M}_{\mathcal{T}}$, but instead track permissions in a per-thread compartment stack that ensures the permissions are correctly restored.

Our system does not attempt to modify normal program execution or data sharing beyond enforcing the access permissions described above. We allow $\mathcal{U}$ to directly call APIs from $\mathcal{T}$ in the normal fashion, and do not place any additional restrictions on the flow of function pointers from $\mathcal{T}$ to $\mathcal{U}$ beyond what the language or runtime may already impose. Because we do not attempt to analyze or reason about the call graph for $\mathcal{U}$, we cannot speculate about which functions in $\mathcal{T}$ may be called from $\mathcal{U}$ and therefore instrument all address-taken and externally visible APIs from $\mathcal{T}$ which may be called from $\mathcal{U}$. For these APIs, we change their permissions to enable access to $\mathcal{M}_{\mathcal{T}}$ and then to restore the previous level of access when they return. If $\mathcal{U}$ attempts to directly call a function from $\mathcal{T}$ that does not have this instrumentation, it will crash the program if it attempts to access data residing in $\mathcal{M}_{\mathcal{T}}$, which will still be inaccessible without correctly changing access permissions. This still leaves the challenge of how PKRU-Safe addresses cross-compartment data flow, as the existing cross-compartment data flows will now crash the program if they pass data from $\mathcal{M}_{\mathcal{T}}$ across the compartment boundary into $\mathcal{U}$.

3.4 Overcoming Page-based Protection Granularity

One of the key challenges in our system is that hardware-backed memory protection is fundamentally tied to the granularity of memory pages. However, the main focus of our work is to limit access to memory objects, which may, in general, be significantly smaller than a single memory page or, alternatively, span across multiple pages. This poses two issues. First, general-purpose heap allocators may not support (or respect) heap partitions. Second, how an object is allocated determines which memory page it resides on, and thus determines how it can be accessed.

Heap Partitioning. Modern general-purpose allocators commonly use a set of internal memory regions to manage heap objects. Generally, these are contiguous regions used to allocate objects within a particular size class. Normally, an allocator is free to place any two allocations of the same size in the same memory region. In our system, this would be problematic if those two objects actually should reside in separate compartments, since it may result in a program crash or expose private data from $\mathcal{M}_{\mathcal{T}}$ by placing it in the wrong memory region.

We chose to keep separate memory pools for each compartment, which makes disabling access to the entire pool straightforward. The key observation here is that each compartment's memory pool must be disjoint from all others, and that pages are never migrated between the pools, in particular through mechanisms such as an allocator's page cache. We tackle this challenge within PKRU-Safe by providing a compartment-aware heap allocator. Making an allocator that can cope with these changes also requires modifications to where and how it manages its own internal data in addition to supporting fine grained region control. For instance, our allocator keeps its internal data for each compartment in that compartment's memory region, which prevents other compartments from improperly accessing or modifying it. We discuss our specific implementation choices in greater detail in Section 4.

Data-Flow-Based Allocation. The final and most important challenge of PKRU-Safe is the ability to automatically determine if a memory object should be accessible from $\mathcal{U}$, and if so to ensure that it will reside in $\mathcal{M}_{\mathcal{U}}$. This is ultimately a question about the program's inter-procedural data flow, which must account for objects directly passed through the compartment boundary as well as those indirectly referenced by function parameters, such as objects reachable through the fields of aggregate types. As mentioned previously the root of this issue is the page-based nature of the enforcement mechanisms available in hardware. Therefore, when an object is allocated, we must be able to determine how it may be used in the future, since that will determine which memory pool it must reside in, and thus how it can be accessed.

We model this as a taint tracking problem, in which allocation functions in $\mathcal{T}$ are sources of $\mathcal{M}_{\mathcal{T}}$ and the interfaces to $\mathcal{U}$ are sinks. Should any source ever flow into (or through) a sink, then we have determined that this memory object must be shared between the compartments, and that it must reside in $\mathcal{M}_{\mathcal{U}}$. Automatically accounting for data flow is a major departure from recent work in this area [30, 34, 66], which has required on developers to manually separate data.

By identifying which allocation sites must be shared, we can ensure that the necessary objects are placed into the shared region of memory and maintain the program's original functionality without requiring source code changes. PKRU-Safe uses dynamic profiling information to identify cross-compartment memory sharing, but is compatible with other suitable types of analysis. We describe our implementation of the dynamic reachability analysis in Section 4.

4 Implementation

Our implementation of PKRU-Safe is tightly integrated into the Rust compiler toolchain, making use of frontend annotations and instrumentation in the Rust compiler (`rustc`), as well as profiling instrumentation and analysis in its LLVM

backend. The compartmentalization runtime also makes use of our compartment-aware allocator, `pkalloc`, to separate the application data in two: $\mathcal{M}_{\mathcal{T}}$ and $\mathcal{M}_{\mathcal{U}}$. This allows us to provide a drop-in replacement for a normal Rust toolchain, and allows developers to adopt our memory protection scheme with only minimal changes to existing workflows.

4.1 Frontend Instrumentation

Our modifications to `rustc` provide the necessary developer annotations for identifying the boundary of $\mathcal{U}$. We provide this functionality chiefly through a compiler plugin, which allows developers to tag specific Rust crates as *untrusted* interfaces. The plugin will then annotate all FFI functions in those crates, and transparently wrap them to drop memory access permissions, as described in Section 3. This instrumentation happens during AST expansion, prior to type or borrow checking, and is transparent to all dependent code. Later compilation stages propagate the annotations as LLVM metadata so that we can reliably identify them in the LLVM backend.

We use Intel MPK to control the memory access permissions of each compartment, because it provides a suitable balance between performance and fine-grained access control. We use call gates consisting of small assembly stubs that modify compartment permissions held in the PKRU register. Each call gate verifies that the new PKRU value matches the target permission the gate is meant to enforce, and will otherwise exit the application if the values are mismatched. Further, because we assume code in $\mathcal{U}$ is compiled with CFI enabled, it should not be possible for an adversary to escalate their memory access capabilities, since PKRU-modifying gadgets are not valid control flow targets for untrusted code, and our call gates prevent whole-function reuse from incorrectly changing permissions.

4.2 Modifications to Rust and its Core Libraries

The other significant changes we made to Rust were to its `liballoc` and its Global Allocator trait. We extended `liballoc`'s allocation functions with equivalent untrusted variants that allocate from $\mathcal{M}_{\mathcal{U}}$ instead of $\mathcal{M}_{\mathcal{T}}$, such as `__rust_untrusted_alloc()` for `__rust_alloc()`. Additionally, we modified the `__rust_realloc()` implementation to ensure that memory is always reallocated from the same pool its base pointer originated from. This ensures that reallocations behave consistently regardless of the execution path.

4.3 Analysis

MPK-based protections are inherently tied to a memory object's provenance, e.g., if it resides in $\mathcal{M}_{\mathcal{T}}$ it is only accessible from $\mathcal{T}$. Hence, PKRU-Safe must understand how data may flow from $\mathcal{T}$ to $\mathcal{U}$ so that all of the memory objects that are passed to untrusted code can be allocated from the correct pool of memory. To overcome this challenge, we use a data flow analysis that can determine if a heap allocated object may eventually be passed through the compartment boundary.

4.3.1 Dynamic Analysis.

We opted to use a dynamic analysis approach that would allow us to identify the allocations that need to be shared between $\mathcal{T}$ and $\mathcal{U}$. Dynamic analysis has a number of well understood trade-offs when compared to static approaches. Since data-flow analysis is an active field of research, we provide a brief overview of the most prominent points and current developments in Sections 6 and 7. Our dynamic analysis tracks which heap allocations from $\mathcal{M}_{\mathcal{T}}$ are accessed from $\mathcal{U}$ in a profiling phase. The recorded allocation sites are then used later on in the instrumentation to automatically select which memory region an object should come from ($\mathcal{M}_{\mathcal{T}}$ or $\mathcal{M}_{\mathcal{U}}$). Note that because PKRU-Safe only changes which memory region an allocation comes from, it introduces no new allocation sites into the original program.

We use LLVM and compiler-rt to build a runtime profiler for our target code as depicted in Figure 2 in a simplified manner. When we profile an application we use compiler inserted instrumentation to track information about the program at runtime, and use the recorded data to generate our profiles. We use an LLVM pass to assign each call to the global allocator a unique Allocation Identifier (`AllocId`), which can be recorded and tracked as part of the runtime instrumentation. Our allocation identifiers are a tuple of the function ID, basic block ID, and the ID of the allocation call site, which allows us to later tie a specific `AllocId` to its origin location in the LLVM IR. All of these IDs are represented by the single `AllocId` field in Figure 2. At each call site to one of Rust's allocation APIs, we insert a callback into our runtime component to record metadata about the memory object which was just allocated. This metadata includes the `AllocId` of the object, the virtual memory address of the new object, and the size of the allocation. Further, we also instrument calls to Rust's reallocation and deallocation APIs to update the metadata or to stop tracking it respectively. In particular, reallocation calls associate the returned memory object with the original object's `AllocId`. We can safely do this, because our modified version of `__rust_realloc()` will not change the memory pool for an object on reallocation, so allocations will remain in the pool they were originally allocated in, i.e., $\mathcal{M}_{\mathcal{T}}$ or $\mathcal{M}_{\mathcal{U}}$. Tracking reallocation this way preserves the information about which objects may be used later by untrusted code, without concern for whether a reallocation occurs or not.

After instrumentation, a developer will profile the application to capture its expected behavior, which will generate a profile recording of all allocation identifiers accessed from $\mathcal{U}$. Because none of the instrumentation has attempted to

partition the heap yet, when profiles are collected all heap data is allocated in $\mathcal{M}_{\mathcal{T}}$. Therefore, we can be certain that any data originating from $\mathcal{T}$ and that is accessed by $\mathcal{U}$ during profiling will raise a fault. While profiling, our runtime interposes on all memory access violations by registering a fault handler for `SIGSEGV`, which records the `AllocIds` of memory objects from $\mathcal{M}_{\mathcal{T}}$ that are accessed from $\mathcal{U}$ when they trigger an MPK violation. Our fault handler looks up the metadata associated with the faulting memory address, and records it to the profile.

After profiling, the developer will recompile the application using the collected profiles, which will modify all the allocation sites in $\mathcal{T}$ that were accessed from $\mathcal{U}$ to come from $\mathcal{M}_{\mathcal{U}}$. This is done by simply matching the allocation identifiers to their corresponding allocation site in the LLVM IR, and updating the call to the allocator to use memory from $\mathcal{M}_{\mathcal{U}}$. If the profiling corpus does not record an allocation being used by $\mathcal{U}$, it will not be in the profile, and therefore it will reside in $\mathcal{M}_{\mathcal{T}}$. This makes it important for developers to have a good profiling corpus for their application, as missed cross-compartment data accesses will crash the program. We address this point in greater detail in Section 6.

One challenge of this approach is that some applications will register their own fault handlers. In particular, Servo and `rustc` both register specialized handlers for `SIGSEGV`, but discard previously registered handlers. To overcome this we initialize our handler as late as possible. If any conflicting fault handlers were registered before ours, we keep a reference to it, so faults unrelated to an MPK violation behave normally. While most programs do not require modification to use our instrumentation, some programs may require special treatment. For instance, Servo registers many fault handlers for `SIGSEGV` which may be registered late in program execution. To address this, we manually placed a call to register our fault handler directly after its last handler was installed. This limitation could be overcome with additional engineering effort to interpose on signal handling registration, similar to some of LLVM's sanitizer runtimes.

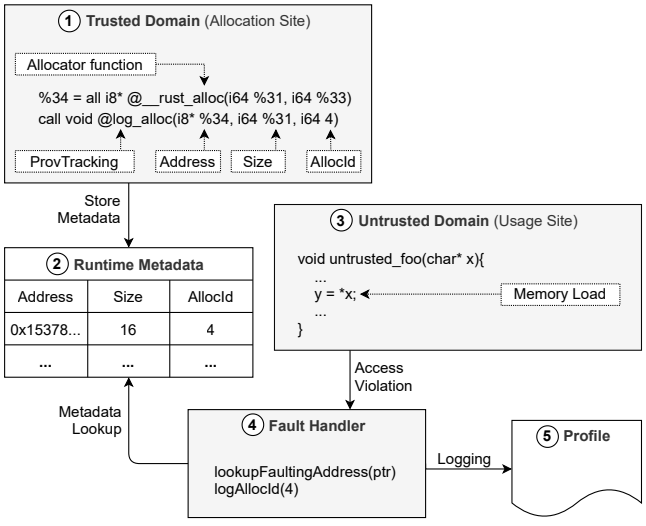

Figure 2. Dynamic Instrumentation. ① Provenance tracking calls are inserted in LLVM to track allocations from $\mathcal{M}_{\mathcal{T}}$ and store metadata into ②. ③ When untrusted code accesses $\mathcal{M}_{\mathcal{T}}$ it will trigger an access violation and call into our fault handler ④, which will record the `AllocId` in a profile record ⑤ before resuming execution.

4.3.2 Fault Handler.

The key component of the dynamic analysis is our fault handler. Each time an address from $\mathcal{M}_{\mathcal{T}}$ is used from $\mathcal{U}$, an access violation will be raised by the hardware. Our fault handler services these faults when they are the result of an MPK violation, but will otherwise fall back to any preexisting fault handler for `SIGSEGV`. The fault handler operates as part of $\mathcal{T}$, and is able to modify compartment permissions to record profiles and inspect trusted memory. Since our runtime tracks all live heap objects and their valid address ranges, we use the faulting address to look up the object's metadata and record it if it's `AllocId` has not been seen before. This limits our profile to a set of *unique* faulting allocation sites. After recording the fault, we need to resume the application. This poses a challenge, as simply exiting the handler and allowing the program to resume will cause another fault when the memory access is retried, but simply advancing the instruction pointer would skip the memory access altogether. Similarly, only reseting the PKRU register and resuming execution would effectively perform a compartment transition, and would cause all other memory accesses while executing in $\mathcal{U}$ to be missed in the profile. We opted to use single stepping to allow us to temporarily disable the MPK protection in the fault handler, and then restore it after the memory access completes. We achieved this by setting the trap flag on the faulting instruction, and setting the PKRU register to allow the memory access (i.e., temporarily switch compartments back to $\mathcal{T}$). After the instruction completes, the processor will trap into the OS and a `SIGTRAP` signal will be raised. We service the fault for `SIGTRAP` in our runtime, and restore the previous PKRU value. While we could have alternatively emulated the memory access, we wished to avoid decoding the faulting instruction, and correctly emulating the access, when single stepping would suffice.

4.4 `pkalloc`

It is imperative that we ensure that the global allocator does not improperly share memory pages between its memory pools for $\mathcal{M}_{\mathcal{T}}$ and $\mathcal{M}_{\mathcal{U}}$. To address this issue we developed `pkalloc`, an allocator interface, which manages the memory for both $\mathcal{M}_{\mathcal{T}}$ and $\mathcal{M}_{\mathcal{U}}$ originating from $\mathcal{T}$. It allows us to provide a uniform interface to both memory pools through use of the extended Global Allocator trait, which we modified to support allocating memory for use in $\mathcal{U}$ (i.e., $\mathcal{M}_{\mathcal{U}}$).

pkalloc wraps two separate heap allocators, je_malloc and malloc from libc. Internally, all allocations from $\mathcal{M}_{\mathcal{T}}$ are allocated by je_malloc, which we modified to only use pages from a special pool of memory reserved from the OS during application startup, and which are marked with a PKEY to ensure they are only accessible from $\mathcal{T}$. By default we reserve 46-bits of the address space for $\mathcal{T}$, but this value can be tuned on a per-application basis. We leverage the Copy-on-Write semantics of mmap to map the entire region at once, which has virtually no cost if those pages are never used. All memory outside of this pool of memory is a part of $\mathcal{M}_{\mathcal{U}}$, and therefore is accessible from $\mathcal{U}$. To supply allocations from the shared region of memory, $\mathcal{M}_{\mathcal{U}}$, we use libc's malloc implementation which uses a completely disjoint set of memory pages from those in $\mathcal{M}_{\mathcal{T}}$. This split-allocator design allowed us to avoid modifying allocator internals to prevent memory sharing.

5 Evaluation

We evaluated PKRU-Safe on two dimensions: security and performance.

5.1 Experimental configuration

We compiled and tested the performance of our system on a Dell Precision 7820 workstation, running Ubuntu 18.04.4 LTS with kernel version 4.15.0. The system has a Intel Xeon Silver 4110 CPU running at 2.10 GHz and 48 GB of DDR4 ECC memory. Both our experiments and build jobs were run inside of a Docker container based on Debian Buster. While we tried to follow best practices when running experiments, there are some forms of system variability that are difficult to completely eliminate [55]

We used a set of micro-benchmarks with different memory access patterns and variable workloads between compartment transitions to understand the overhead our proposed system would impose. We also evaluated our system in the context of the Servo web browser. Servo is written almost completely in Rust but uses the SpiderMonkey JavaScript engine written in unsafe C/C++[3], which is why SpiderMonkey is placed in $\mathcal{U}$ in our experiments. Servo is perhaps the most complex Rust program available today, having 265K lines of Rust code with an additional 1.8M lines of Rust from its dependencies. We chose it as our primary target for evaluation because Servo is exactly the type of application we believe developers will want to use memory sandboxes on: it is mostly written in Rust, but has a key component that is written in unsafe C++ and that processes untrusted inputs. Another key factor is that Servo is a large, complex program with approximately 2 million lines of Rust code, with sophisticated, hard to track data flows. Our reasoning here is that any automated methods of hardening a Rust application that will work on Servo will likely scale well to most other Rust applications, due to both Servo's large size and complexity. Additionally, though Servo will remain a standalone browser for the foreseeable future, several of its components have already been integrated into Firefox [54], which may also benefit from adopting this style of memory access control.

[3]CVE-2019-11707, CVE-2019-9792, CVE-2019-9813,CVE-2019-9810 [1–4]

5.2 Micro-Benchmarks

We created a series of micro-benchmarks to better understand the types of overhead our instrumentation was introducing. We use a set of three workloads to measure overhead. Each benchmark uses a controllable workload, so we could understand how call gate overhead scaled with the work done in each compartment transition. Each workload is duplicated in a *trusted* and an *untrusted* version. Trusted workloads do not have call gates, but otherwise behave identically. Trusted and untrusted workloads live in separate libraries, with the untrusted library marked as *untrusted*. Each API in the untrusted library has our call gate instrumentation.

Our workloads are as follows:

- **Empty.** The FFI function has no body. This benchmark measures the maximum overhead call gates can impose per instrumented function call.
- **Read-One.** The FFI fuction performs a single memory read to a heap address. This benchmark is designed to understand how call gates scale with work size, but also help quantify allocator overhead.
- **Callback.** The FFI function performs a callback that triggers a compartment transition back to the trusted compartment, which both exercises external call gates, but also increases the number of compartment transitions per unit of work.

All call gate micro-benchmarks ran for 100 million iterations, and are reported as the mean overhead averaged across all iterations. The call gates in the **Empty** benchmark had an overhead of 8.55x, which aligns with our expectations for overhead when compared to the execution time of a single ret instruction. The **Read-One** benchmark had slightly better performance, with 7.61x on average, while the **Callback** benchmark had only a 6.17x overhead, despite going through twice the number of call gates. The decreased overhead in these benchmarks stems from the more favorable ratio of the call gates to the workload.

To confirm this, we simulated increasing amounts of work between each compartment transition by increasing the number of loop iterations executed in the target function. We can see from the plot in Figure 3 that the overhead decreases quickly as the execution time between compartment transitions increases.

5.3 Performance

Configuration. We compiled Servo in three different configurations: configuration (`base`) is our baseline configuration with a completely unmodified Servo built using our version of `rustc` and with LTO enabled, configuration (`alloc`) adds our allocator `pkalloc`, but does not add other instrumentation to the application (i.e., no call gates between $\mathcal{T}$ and $\mathcal{U}$), and configuration (`mpk`) adds call gate instrumentation between $\mathcal{T}$ and $\mathcal{U}$ to configuration (`alloc`).

The `base` configuration was built using our modified Rust compiler, which is based on the same version of the nightly Rust compiler that the revision of Servo we built against required, `nightly-2019-06-19`. The only other changes we made were to build files, which we altered to compile using LTO and to reference a local checkout of the crates containing the SpiderMonkey JavaScript engine (`mozjs` and `rust-mozjs`) that we would instrument in later tests. This version is otherwise unmodified, and serves as our baseline for comparison. Configuration (`alloc`) was modified to replace the Global Allocator implementation in Servo (`jemalloc`) with `pkalloc`, which required us to modify approximately 90 lines of Servo's code. While this sounds significant, the bulk of this was simply commenting out Servo's definitions for its Global Allocator, and using our `pkalloc` crate in its place. This configuration allows us to capture the overhead introduced by using our allocator, as it is the only meaningful change from the baseline configuration. Our final configuration, `mpk`, enables the full instrumentation of untrusted APIs which makes the memory pages in $\mathcal{M}_{\mathcal{T}}$ inaccessible when not executing in $\mathcal{T}$. We added our library level annotations to the `mozjs` and `rust-mozjs` crates to enable the call gates between the browser and the JavaScript engine.

We found that we needed to add one more change to the `rust-mozjs` crate because it rewrites the rust bindings generated by bindgen, a Rust library used to automatically generate Rust bindings to FFI code, after they are created through macro expansion. While PKRU-Safe normally works with bindgen, this new rebinding bypassed our instrumentation by linking directly to the C/C++ APIs after our wrappers were generated, and changing their names and signatures.

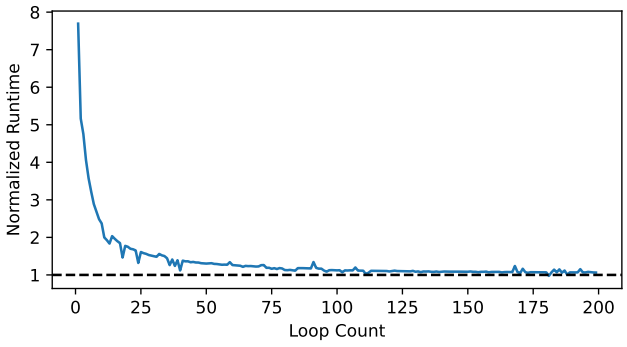

Figure 3. Call Gate Overhead vs. Work Load

Table 1. Servo Mean Benchmark Overhead and Statistics

	alloc	mpk	Transitions	% $\mathcal{M}_{\mathcal{U}}$
Dromaeo	5.89%	11.55%	1775338812	4.13%
JetStream2	-1.48%	0.61%	7025902	42.41%
Kraken	-0.11%	-0.41%	5831503	48.59%
Octane	-2.25%	3.28%	425426	16.57%

This is a non-standard use of bindgen, and predates its normal facilities for mapping types and rebinding signatures. As a result we had to modify the macros used in the `rust-mozjs` crate to use our call gates, and therefore ensure that *all* of the browsers interfaces to the JavaScript engine had the correct call gates. Without this change we would have missed nearly 10% of the bindings to critical APIs in the JavaScript engine. This, however, could be overcome with some additional engineering to ensure that our wrappers were generated at a later time during AST expansion.

To begin our experiment, we first profiled Servo using its internal test suite to capture the normal, expected behavior of the browser. This gave us a suitable memory profile to partition the browser heap into $\mathcal{M}_{\mathcal{T}}$ and $\mathcal{M}_{\mathcal{U}}$. Specifically, we used the test suites for the Web Platform Tests, jQuery, and Web-IDL. Additionally, we used Selenium to browse a selection of common web pages, such as google.com, reddit.com, wikipedia.org, and youtube.com. We confirmed that the browser was functional by visiting a number of websites outside of our corpus, and exercising functionality we had not recorded, such as watching a video in the browser and following links to other websites. After the final instrumentation, our toolchain had changed 274 of Servo's 12088 allocation sites in $\mathcal{T}$ to come from $\mathcal{M}_{\mathcal{U}}$. This means our system is able to modify as little as 2.26% of the allocation sites, across the entire program and its Rust dependencies to separate the Rust portion of the browser's heap data from its JavaScript engine.

Standard Benchmarks. We evaluated Servo using the Dromaeo [5], Kraken [7], Octane [8], and JetStream2 [6] benchmark suites. These benchmarking suites are well known and widely used for benchmarking browser performance. We provide a quick overview of the results in Table 1. Note that although these are separate benchmarking suites, there is a large overlap in their testing corpus. For each benchmark we served a copy of the benchmark via a local http server. We made no modifications to Dromaeo, Kraken, or Octane, but we had to disable the `WASM`-based test suite in JetStream, because the base revision of Servo we used for our evaluation had a bug that would not allow that benchmark to complete.

Dromaeo benchmarks were run with the *recommended* set of benchmarks in the same configuration used by the Servo developers. The results from the experiment, which can be found in Figure 4 and Table 2, demonstrate that the overhead for this style of instrumentation is highly correlated with the

Table 2. Dromaeo Benchmark Overhead and Statistics

	alloc	mpk	Transitions	% $\mathcal{M}_{\mathcal{U}}$
dom	7.85%	30.74%	734083388	50.30%
v8	-2.31%	0.53%	339698	4.59%
dromaeo	15.87%	4.64%	730295	0.57%
sunspider	-1.34%	-0.81%	893923	3.11%
jslib	9.39%	22.65%	1017275385	13.93%
mean	5.89%	11.55%	-	-

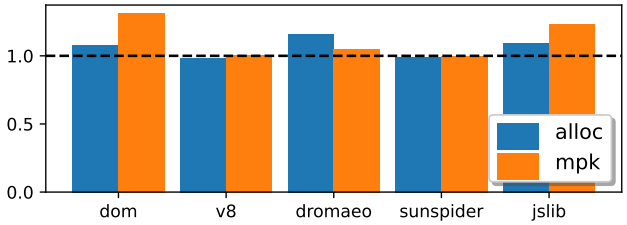

Figure 4. Dromaeo Benchmark Normalized Runtime Overhead

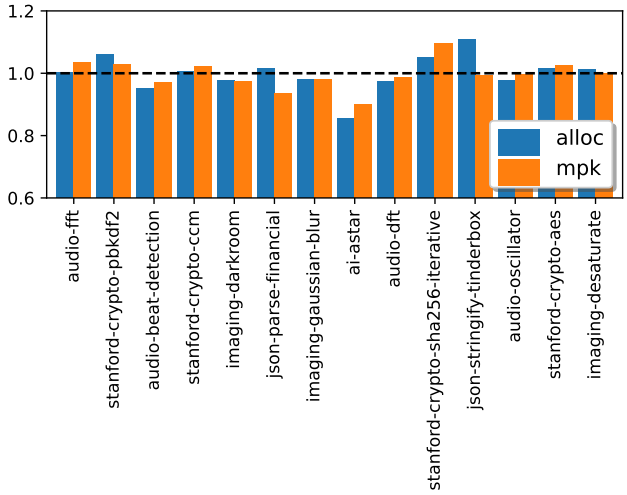

Figure 5. Kraken Benchmark Normalized Runtime Overhead

specific workload. In general, our split allocator design fared well, with respect to the performance impact, with its mean scores within 6% of the baseline. We hypothesized that the majority of this overhead comes from the less performant allocator used for $\mathcal{M}_{\mathcal{U}}$, which is the libc version of `malloc`. To verify this we changed `pkalloc`'s implementation to use memory from $\mathcal{M}_{\mathcal{T}}$ for both trusted and untrusted allocation sites, but with our call gates disabled to avoid program errors. This modification removed any detectable overhead from our benchmarking numbers. This result gives us confidence to expect that choosing a more performant allocator for $\mathcal{M}_{\mathcal{U}}$ would remove most of the overhead associated with `pkalloc`. Adding our call gate instrumentation shows an average overhead across all of Dromaeo of roughly 11.55% compared to the baseline, but only has approximately 6% additional overhead on top of the allocator.

The data shows that the `dom` and `jslib` tests exhibit a significant overhead due to the introduction of call gates, having overheads for the MPK-based call gates of approximately 22.89% and 13.26% over the allocator only versions respectively. Given our micro-benchmarking results, we wanted to understand exactly why this happened. We were able to determine that these benchmarks cause Servo to spend a significant amount of time switching rapidly between $\mathcal{T}$ and $\mathcal{U}$, but do very little work before transitioning back. This occurs for two reasons: 1) a function with a call gate is called in a loop, and 2) several of the callbacks registered with the JavaScript engine also call functions that trigger transitions between compartments. This can occasionally result in a deeply nested stack of compartment transitions where only a small amount of work is performed before the compartment stack unwinds.

As noted in our micro-benchmarks, the runtime overhead of each call gate is proportional to the number of instructions executed between compartment switches, and both the `dom` and `jslib` test suites highlight this limitation. We ran the individual benchmark suites with instrumentation to capture the number of compartment transitions in each sub-suite during normal benchmarking. The **Transitions** column in Table 2 shows the number of compartment transitions that the `dom` and `jslib` suites execute during the benchmark. The data shows that these portions of the suite have disproportionately large numbers of compartment transitions when compared to the other workloads. Further, the `dom` test suite has significantly fewer benchmarks than the `jslib` suite (26 vs 59). If we average the number of transitions evenly across the number of tests in each suite, we can see that each `dom` test has almost 63% more transitions per test than `jslib` on average. We believe that this per test view of transitions more closely matches the overheads we see in the benchmarks, as the larger number of tests increase the total number of transitions, but do not contribute as much overhead to any one test.

The Kraken benchmarks in Figure 5 show low overhead on average, with performance results on par with the baseline. Similar to Kraken, the Octane benchmarks had low overhead, with an average overhead under 4% for our `mpk` configuration. Experimental results for Octane are presented in Figure 6.

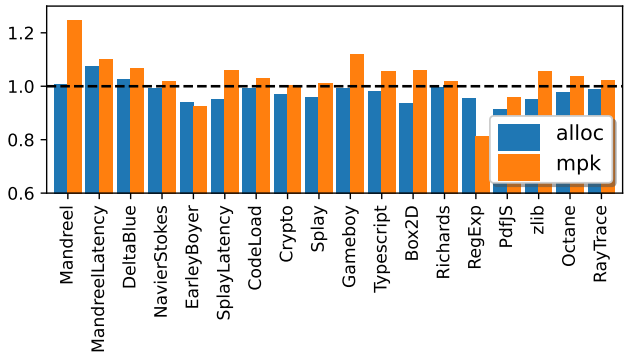

Figure 6. Octane Normalized Overhead

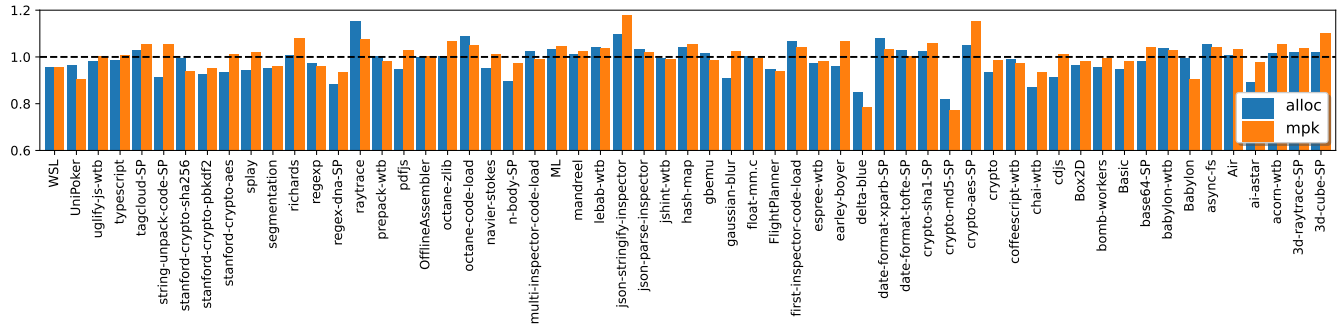

Figure 7. JetStream2.0 Benchmark Overhead

Table 3. JetStream2 Overall Scores and Overhead

	base	alloc	mpk
Score	60.31	61.20	59.94
Overhead	-	-1.48%	0.61%

Like Kraken and Octane, the results from JetStream2 have scores on par with the baseline for the entire test suite. Plots of the normalized runtime overhead for each of the Jetstream2 benchmarks can be found in Figure 7. Note that JetStream2 scores each of its benchmarks individually, but computes an overall score as the geometric mean of the individual scores [26], which we present in Table 3.

5.4 Security

For our security evaluation, we leveraged a known type confusion vulnerability in Servo's JavaScript engine, SpiderMonkey (CVE-2019-11707), which is present in SpiderMonkey versions prior to 60.8.0. This exploit was previously shown to be the basis for an arbitrary read/write primitive [15], which we use in a fashion similar to Park et al. [59]. Specifically, we use the read/write primitive to access browser data that should logically be inaccessible from the JavaScript engine.

As part of our evaluation, we modified Servo to hold a secret at a fixed address, which it allocates at program start, and logs its value to console on program exit. We chose to use a fixed address for ease of implementation, as prior work has shown that it is possible to determine the location of in memory secrets in a variety of ways. Servo was otherwise identical to our `mpk` configuration, using the same profiling corpus and build configuration we used in our benchmark experiments. Our exploit uses the type confusion vulnerability mentioned earlier to write a value to this address. The important aspect of this exploit is that it is targeting some private data in memory at a valid address within $\mathcal{M}_{\mathcal{T}}$. In our experiment, the attack successfully writes to data in Servo's private memory without our memory protection, which can be confirmed by examining the program output. In contrast, once Servo's memory has been compartmentalized by PKRU-Safe the attacker's attempt to write to private memory fails, resulting in a memory access violation from the OS, and crashing the application.

This demonstrates PKRU-Safe's practicality in enforcing fine-grained privilege separation for heap data automatically and with low developer effort. This success, however, does not cover the breadth of potential exploits an adversary may try to use. We consider several attack vectors out of scope due to the assumptions in our threat model (see Section 2). ROP-style code reuse is mitigated through our assumption of forward and backward edge CFI in $\mathcal{U}$. Code pointers, however, may be corrupted or abused if they are passed to $\mathcal{U}$ as part of aggregate data types. Memory corruption of this type, however, occurs within the shared region of $\mathcal{M}_{\mathcal{U}}$, and therefore cannot be stopped by compartmentalization. This affects all compartmentalization schemes: cross-compartment sharing of data structures must either be forbidden, or otherwise have a method for verifying their integrity. Lastly, it is possible for untrusted code to corrupt Rust objects that reside in $\mathcal{M}_{\mathcal{U}}$ and perform COOP-style code reuse through `vtable` corruption. This, however, is an uncommon situation, as Rust generally forbids passing Rust data types, like Trait objects through its FFI, generating compiler warnings if such objects or pointers to them cross the FFI boundary. We also stress that confused deputy style attacks are not the focus of this work, since the confused deputy is operating within its own compartment.

6 Discussion

Static vs. Dynamic Analysis. Principally, a purely static inter-procedural data flow analysis would fit well with Rust's compile-time guarantees related to ownership and object lifetimes. However, while working towards a prototype instrumentation for Servo we found that state-of-the-art data-flow analyses struggle to provide sufficient precision, soundness, and/or scalability: in our initial experiments all of the existing analyses failed to instrument our MPK-enabled build correctly. Upon closer investigation, we found a variety of different reasons for this. First, some pointer analyses yielded unsound results, missing several allocation sites, thus resulting in program crashes, because data would not be correctly

shared between compartments. Second, some pointer analyses dramatically over-approximated the amount of data that could be used by untrusted code, resulting in memory from completely unrelated components to be exposed to untrusted code. Third, some pointer analyses were unable to complete at all due to the exponential space complexity of the nature of the problem[4].

While efficient, demand-driven analyses similar to Synchronized Push Down Systems (SPDS) proposed by Späth et al. [63] represent an attractive alternative, at the time of writing there exists no comparable analysis for LLVM based languages. In fact, this may be one of the fundamental limitations of LLVM IR, which lacks a reliable way to reason about potential access paths through pointers, due to loss of information through optimization, type casting, or pointer arithmetic [46]. For this reason, we leverage dynamic analysis to instrument Servo, although PKRU-Safe supports instrumentation entirely based on static analysis in principle, which we tested using various small programs.

Lastly, while incomplete dynamic profiling with PKRU-Safe may lead to a program crash, similar policies are widely used in production systems. For instance, Android now includes UBSan instrumentation in its production builds, which by default will abort when encountering undefined behavior [11, 43, 51]. If developers fail to exercise these behaviors prior to release, then these applications will be terminated, in a similar fashion to those in our system. In our view crashes due to missed inter-compartment dataflows are bugs that stem from a lack of testing, and are similar to other memory related errors, like dereferencing invalid pointers. Further, operating systems and applications often test and profile applications and collect telemetry and performance information using a subset of their installation base [27]. In principle, PKRU-Safe could be deployed using similar approaches, while minimizing negative impact on its users.

Number of Compartments. We ensure that whenever code in the safe language is executing it has full access to the program's data, which we consider safe since this part of the application is both trusted and memory safe. In contrast, when untrusted components are running, they should only be able to access their own data and data from the safe language that has been shared with them. Our choice of using only two domains is a policy decision, one chosen to keep both the reasoning about our partitioning scheme and the management of memory resources as simple as possible. Though we have tried to introduce the minimal amount of change from the original program by keeping our partitioning simple, we see no fundamental issue using a more complicated partitioning scheme that uses more than two domains.

[4]Specifically, they consistently exhausted more than 192 GB of RAM with an additional 192 GB of swap space on disk in a matter of hours.

Stack Protection. PKRU-Safe is focused, first and foremost, on protecting heap data. Currently, our threat model (Section 2) assumes some level of stack protection is in place that prevents $\mathcal{U}$ from corrupting any of the stack data owned by $\mathcal{T}$. Ideally, we would also be able to classify stack data from $\mathcal{T}$ in the same way we tackle heap data. In principle, this could be addressed by our current analysis with some significant engineering effort to augment provenance tracking for stack data. We could mark the stack used by $\mathcal{T}$ also to be part of $\mathcal{M}_{\mathcal{T}}$, and rely on profiling to identify each stack allocation that was affected in the same way that we track heap allocations. This change would take a non-trivial effort to adopt but requires no methodology change over our approach with heap data.

7 Related Work

The idea of limiting access to potentially sensitive resources, including memory, is not new and has been an active area of research for decades and we provide an overview comparison to the most relevant approaches in Table 4. Software fault isolation (SFI), process-level sandboxing, and in-process sandboxing are prominent examples in this space. However, similarities can be found across a wide swath of computer science research. Region-based memory, privilege separation, memory capabilities, SFI, and memory domains are examples of techniques that use memory partitioning or access control methods to manage how essential resources, such as memory, are used. Manually restructuring legacy applications to reliably and effectively isolate trusted from untrusted components can constitute a significant, multi-million dollar effort [60].

For this reason, PKRU-Safe takes inspiration from prior work in *automated* compartmentalization, such as Privtrans [18], Wedge [16], PtrSplit [50], ProgramCutter [70], and recently CALI [13]. In particular, we leverage a dynamic tracing component to make fine-grained resource-sharing decisions, similar to Wedge and ProgramCutter. However, in contrast to these works, PKRU-Safe does not attempt to restrict access to OS-level resources, relying on the OS to provide suitable primitives such as `seccomp` filters to restrict access to sensitive system calls, for example. Moreover, PKRU-Safe provides in-process memory separation at the level of individual allocation sites. In contrast, automatic compartmentalization has been primarily focused on complete process level isolation, using more coarse-grained memory separation. While SOAAP [32] uses dynamic analysis test compartmentalization hypotheses, it does not automatically compartmentalize the application, but is intended to support developers as they manually re-architect their program.

Similarly, in-process memory isolation has been previously explored using a variety of isolation primitives: kernel extensions for private memory regions [69], double mapping

Table 4. Comparison of Selected Related Approaches.

	FideliusCharm [10]	RLBox [56]	XRust [49]	Hodor [34]	ERIM [66]	PtrSplit [50]	PKRU-Safe
In-Process Isolation	✓	✓	✗	✓	✓	✗	✓
HW Enforcement	✓	✗	✗	✓	✓	✗	✓
No App Modification	✗	✗	✓	✓	✗	✓	✓
No OS Modification	✗	✓	✓	✗	✗	✓	✓
Data-Flow Analysis	✗	✗	✓	✓	✗	✓	✓
Mixed-Language Support	✓	✓	✗	✗	✗	✗	✓
Runtime Overhead	5%-13.69%	3%-13%	21.57%-51.21%	2.78%-9.85%	1x-3x	79.3%	1%-11.55%

virtual address ranges [36], ARM memory domains [21, 65], linker-assisted isolation [12], and virtualization [14]. Recently, new processor features, like Intel MPK, have revived interest in intra-process sandboxing [30, 34, 66]. PKRU-Safe differs from these works, primarily in its approach to data flow and its ability to automatically perform most of the partitioning without relying on developers to rewrite large portions of their application. While PKRU-Safe uses similar sandboxing mechanics to other MPK-based approaches, it does not attempt to address the issue of stray MPK-related instructions as in ERIM or Hodor. Instead, it focuses on solving the largely ignored problem of correctly managing inter-compartmental data flow. In principle, our system could reuse infrastructure from these systems, and work just as well, at the cost of additional engineering effort to reconcile their different environments, build systems, and libraries.

PKRU-Safe is focused on isolating untrusted code at the level of a library interface, similar to the approaches found in Google's Sandboxed API [31], RLBox [56], and Enclosures [30]. While Google's Sandboxed API and RLBox are focused on using process-level isolation and WASM, Ghosn et al. [30] use MPK based isolation that is directly integrated into the toolchains and runtimes for Go and Python, but still directly relies on developers to correctly account for data flow manually and to make source level changes to support the desired isolation policy. Similar approaches have been explored for sandboxing Rust libraries[45] in a manner similar to the approaches used in BreakApp[68] and ART [38]. Other Rust-centric approaches, such as Fidelius Charm [10], require changes to the OS kernel, are not fully automated, and do not fully account for data flow. PKRU-Safe operates at a similar granularity but automatically accounts for data flow when separating memory access privileges, though with fewer features.

XRust [49] aims to isolate the unsafe parts of Rust from the safe parts of Rust by allocating memory for safe and unsafe code from two different heaps. This is primarily achieved through the use of inline reference monitors to bounds check sensitive memory accesses, though they additionally provide an option to use guard pages in place of the reference monitors. Compared to PKRU-Safe XRust does not aim to provide a general mechanism for in-process memory isolation. We note that XRust could be used in tandem with our approach to strengthen the trust in our compartmentalization scheme for unsafe code blocks in the Rust language part of the program. For this, PKRU-Safe could adopt pointer analysis similar to XRust, however, the analysis would need to be *sound*, highly *scalable*, and *precise*. As explained in Section 6, unfortunately, no such analysis currently exists, which is why we exploit dynamic analysis for our instrumentation. PKRU-Safe is focused on protecting heap data across both language and trust boundaries. It is not an attempt to enforce partial memory safety on the trusted application but to enforce sensible memory access controls across these boundaries.

8 Conclusion

We presented PKRU-Safe, the first data-flow aware and fully automatic source-level compartmentalization framework that supports multi-threaded mixed-language environments without requiring OS modifications or application rewriting. It allows developers to gain safety now, as they transisiton the rest of their codebase to safer alternatives. Our prototype of PKRU-Safe demonstrates its scalability to large software projects, such as web browsers, which require least privilege isolation. Our evaluation on a wide range of benchmarks and real-world applications further shows that PKRU-Safe is both effective and practical, yielding low overhead while adding significant additional memory isolation for mixed-language programs.

References

[1] 2019. CVE-2019-11707. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-11707.
[2] 2019. CVE-2019-9792. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-9792.
[3] 2019. CVE-2019-9810. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-9810.
[4] 2019. CVE-2019-9813. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-9813.
[5] 2021. Dromaeo: JavaScript performance testing. https://wiki.mozilla.org/Dromaeo.
[6] 2021. JetStream 2. https://browserbench.org/JetStream.
[7] 2021. Kraken Benchmarks. http://krakenbenchmark.mozilla.org/.
[8] 2021. Octane Benchmarks. https://developers.google.com/octane/.

[9] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. 2005. Control-Flow Integrity. In *ACM Conference on Computer and Communications Security (CCS)*.

[10] Hussain M. J. Almohri and David Evans. 2018. Fidelius Charm: Isolating Unsafe Rust Code. In *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy* (Tempe, AZ, USA) *(CODASPY '18)*. Association for Computing Machinery, New York, NY, USA, 248–255. https://doi.org/10.1145/3176258.3176330

[11] Dan Austin. 2017. Android Bug Swatting with Sanitizers. https://android-developers.googleblog.com/2017/08/android-bug-swatting-with-sanitizers.html.

[12] Julian Bangert, Sergey Bratus, Rebecca Shapiro, Jason Reeves, Sean W. Smith, Anna Shubina, Maxwell Koo, and Michael E. Locasto. 2016. Sections are Types, Linking is Policy: Using the Loader Format for Expressing Programmer Intent. In *BlackHat USA*. https://www.blackhat.com/us-16/briefings.html#sergey-bratus

[13] Markus Bauer and Christian Rossow. 2021. Cali: Compiler Assisted Library Isolation. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security (ASIA CCS'21)*. Association for Computing Machinery.

[14] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. 2012. Dune: Safe User-level Access to Privileged CPU Features. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. USENIX Association, Hollywood, CA, 335–348. https://www.usenix.org/conference/osdi12/technical-sessions/presentation/belay

[15] bi0s. 2019. Writeup for CVE-2019-11707. https://blog.bi0s.in/2019/08/18/Pwn/Browser-Exploitation/cve-2019-11707-writeup/.

[16] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. 2008. Wedge: Splitting Applications into Reduced-Privilege Compartments. In *5th USENIX Symposium on Networked Systems Design and Implementation (NSDI 08)*. USENIX Association, San Francisco, CA. https://www.usenix.org/conference/nsdi-08/wedge-splitting-applications-reduced-privilege-compartments

[17] Ferdinand Brasser, Lucas Davi, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. 2017. CAn't touch this: Software-only mitigation against Rowhammer attacks targeting kernel memory. In *26th USENIX Security Symposium (USENIX Security 17)*. 117–130.

[18] David Brumley and Dawn Song. 2004. Privtrans: Automatically Partitioning Programs for Privilege Separation. In *13th USENIX Security Symposium (USENIX Security 04)*. USENIX Association, San Diego, CA. https://www.usenix.org/conference/13th-usenix-security-symposium/privtrans-automatically-partitioning-programs-privilege

[19] Nathan Burow, Xinping Zhang, and Mathias Payer. 2019. SoK: Shining light on shadow stacks. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 985–999.

[20] Shuo Chen, Jun Xu, Emre Can Sezer, Prachi Gauriar, and Ravishankar K Iyer. 2005. Non-Control-Data Attacks Are Realistic Threats. In *USENIX Security Symposium*.

[21] Y. Chen, S. Reymondjohnson, Z. Sun, and L. Lu. 2016. Shreds: Fine-Grained Execution Units with Private Memory. In *2016 IEEE Symposium on Security and Privacy (SP)*. 56–71. https://doi.org/10.1109/SP.2016.12

[22] Chromium. 2019. Chromium Site Isolation. https://www.chromium.org/Home/chromium-security/site-isolation.

[23] The GNU Compiler Collection. 2021. x86 Options. https://gcc.gnu.org/onlinedocs/gcc/x86-Options.html.

[24] R. Joseph Connor, Tyler McDaniel, Jared M. Smith, and Max Schuchard. 2020. PKU Pitfalls: Attacks on PKU-based Memory Isolation Systems. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 1409–1426. https://www.usenix.org/conference/usenixsecurity20/presentation/connor

[25] Jonathan Corbet. 2015. Intel Memory Protection Keys. https://lwn.net/Articles/643797/.

[26] Webkit Developers. 2021. JetStream2 in depth analysis. https://browserbench.org/JetStream/in-depth.html.

[27] Firefox. 2019. Telemetry/Experiments - Mozilla Wiki. https://wiki.mozilla.org/Telemetry/Experiments.

[28] Tommaso Frassetto, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. 2017. JITGuard: Hardening Just-in-time Compilers with SGX. In *ACM Conference on Computer and Communications Security (CCS)*.

[29] David Gens, Orlando Arias, Dean Sullivan, Christopher Liebchen, Yier Jin, and Ahmad-Reza Sadeghi. 2017. Lazarus: Practical side-channel resilient kernel-space randomization. In *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 238–258.

[30] Adrien Ghosn, Marios Kogias, Mathias Payer, James R Larus, and Edouard Bugnion. 2021. *Enclosure: Language-Based Restriction of Untrusted Libraries*. Technical Report.

[31] Google Inc. 2021. Sandboxed API. https://github.com/google/sandboxed-api.

[32] Khilan Gudka, Robert N.M. Watson, Jonathan Anderson, David Chisnall, Brooks Davis, Ben Laurie, Ilias Marinos, Peter G. Neumann, and Alex Richardson. 2015. Clean Application Compartmentalization with SOAAP. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (Denver, Colorado, USA) *(CCS '15)*. Association for Computing Machinery, New York, NY, USA, 1016–1031. https://doi.org/10.1145/2810103.2813611

[33] Arie Gurfinkel and Jorge A. Navas. 2017. A Context-Sensitive Memory Model for Verification of C/C++ Programs. In *Static Analysis*, Francesco Ranzato (Ed.). Springer International Publishing, Cham, 148–168.

[34] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L. Scott, Kai Shen, and Mike Marty. 2019. Hodor: Intra-Process Isolation for High-Throughput Data Plane Libraries. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 489–504. https://www.usenix.org/conference/atc19/presentation/hedayati-hodor

[35] Andrei Homescu, Stefan Brunthaler, Per Larsen, and Michael Franz. 2013. Librando: transparent code randomization for just-in-time compilers. In *ACM Conference on Computer and Communications Security (CCS)*.

[36] Terry Ching-Hsiang Hsu, Kevin Hoffman, Patrick Eugster, and Mathias Payer. 2016. Enforcing Least Privilege Memory Views for Multithreaded Applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. Association for Computing Machinery, New York, NY, USA, 393–405. https://doi.org/10.1145/2976749.2978327

[37] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang. 2016. Data-Oriented Programming: On the Expressiveness of Non-control Data Attacks. In *2016 IEEE Symposium on Security and Privacy (SP)*. 969–986. https://doi.org/10.1109/SP.2016.62

[38] Jie Huang, Oliver Schranz, Sven Bugiel, and Michael Backes. 2017. The art of app compartmentalization: Compiler-based library privilege separation on stock android. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 1037–1049.

[39] Kyriakos K. Ispoglou, Bader AlBassam, Trent Jaeger, and Mathias Payer. 2018. Block Oriented Programming: Automating Data-Only Attacks. In *ACM Conference on Computer and Communications Security (CCS)*.

[40] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. RustBelt: Securing the Foundations of the Rust Programming Language. *Proc. ACM Program. Lang.* 2, POPL, Article 66 (Dec. 2017), 34 pages.

[41] Zijo Kenjar, Tommaso Frassetto, David Gens, Michael Franz, and Ahmad-Reza Sadeghi. 2020. V0ltpwn: Attacking x86 processor integrity from software. In *29th USENIX Security Symposium (USENIX Security 20)*. 1445–1461.

[42] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. 2019. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1–19.

[43] Nick Kralevich. 2016. The Art of Defense: How vulnerabilities help shape security features and mitigations in Android. *BlackHat. https://oreil. ly/16rCq. Security and Reliability by Default* 263 (2016).

[44] Volodymyr Kuznetzov, László Szekeres, Mathias Payer, George Candea, R Sekar, and Dawn Song. 2018. Code-pointer integrity. In *The Continuing Arms Race: Code-Reuse Attacks and Defenses*. 81–116.

[45] Benjamin Lamowski, Carsten Weinhold, Adam Lackorzynski, and Hermann Härtig. 2017. Sandcrust: Automatic Sandboxing of Unsafe Components in Rust. In *Proceedings of the 9th Workshop on Programming Languages and Operating Systems* (Shanghai, China) *(PLOS'17)*. Association for Computing Machinery, New York, NY, USA, 51–57. https://doi.org/10.1145/3144555.3144562

[46] Juneyoung Lee, Chung-Kil Hur, Ralf Jung, Zhengyang Liu, John Regehr, and Nuno P Lopes. 2018. Reconciling high-level optimizations and low-level code in LLVM. In *Proceedings of the ACM on Programming Languages*, Vol. 2. ACM New York, NY, USA, 1–28.

[47] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, et al. 2018. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*. 973–990.

[48] James Litton, Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, Bobby Bhattacharjee, and Peter Druschel. 2016. Light-weight contexts: An OS abstraction for safety and performance. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 49–64.

[49] Peiming Liu, Gang Zhao, and Jeff Huang. 2020. Securing Unsafe Rust Programs with XRust. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 234–245. https://doi.org/10.1145/3377811.3380325

[50] Shen Liu, Gang Tan, and Trent Jaeger. 2017. PtrSplit: Supporting General Pointers in Automatic Program Partitioning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17)*. Association for Computing Machinery, New York, NY, USA, 2359–2371. https://doi.org/10.1145/3133956.3134066

[51] René Mayrhofer, Jeffrey Vander Stoep, Chad Brubaker, and Nick Kralevich. 2021. The Android Platform Security Model. *ACM Trans. Priv. Secur.* 24, 3, Article 19 (April 2021), 35 pages. https://doi.org/10.1145/3448609

[52] Microsoft. 2006. Data Execution Prevention (DEP). http://support.microsoft.com/kb/875352/EN-US.

[53] Matt Miller. 2019. Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape. https://github.com/microsoft/MSRC-Security-Research/blob/master/presentations/2019_02_BlueHatIL/2019_01-BlueHatIL-Trends,challenge,andshiftsinsoftwarevulnerabilitymitigation.pdf.

[54] Mozilla. 2019. Oxidation. https://wiki.mozilla.org/Oxidation.

[55] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. 2009. Producing Wrong Data without Doing Anything Obviously Wrong!. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems* (Washington, DC, USA) *(ASPLOS XIV)*. Association for Computing Machinery, New York, NY, USA, 265–276. https://doi.org/10.1145/1508244.1508275

[56] Shravan Narayan, Craig Disselkoen, Tal Garfinkel, Nathan Froyd, Eric Rahm, Sorin Lerner, Hovav Shacham, and Deian Stefan. 2020. Retrofitting Fine Grain Isolation in the Firefox Renderer. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 699–716. https://www.usenix.org/conference/usenixsecurity20/presentation/narayan

[57] Ben Niu and Gang Tan. 2014. RockJIT: Securing Just-In-Time Compilation Using Modular Control-Flow Integrity. In *ACM Conference on Computer and Communications Security (CCS)*.

[58] Soyeon Park, Sangho Lee, Wen Xu, HyunGon Moon, and Taesoo Kim. 2019. libmpk: Software Abstraction for Intel Memory Protection Keys (Intel MPK). In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 241–254. https://www.usenix.org/conference/atc19/presentation/park-soyeon

[59] Taemin Park, Karel Dhondt, David Gens, Yeoul Na, Stijn Volckaert, and Michael Franz. 2020. NoJITsu: Locking Down JavaScript Engines. In *NDSS*. https://doi.org/10.14722/ndss.2020.24262

[60] Charles Reis, Alexander Moshchuk, and Nasko Oskov. 2019. Site isolation: Process separation for web sites within the browser. In *28th USENIX Security Symposium (USENIX Security 19)*. 1661–1678.

[61] Philipp Dominik Schubert, Ben Hermann, and Eric Bodden. 2019. Phasar: An inter-procedural static analysis framework for c/c++. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 393–410.

[62] Mark Seaborn and Thomas Dullien. 2015. Exploiting the DRAM rowhammer bug to gain kernel privileges. In *BlackHat USA*.

[63] Johannes Späth, Karim Ali, and Eric Bodden. 2019. Context-, Flow- and Field-Sensitive Data-Flow Analysis using Synchronized Pushdown Systems. In *46th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2019, January 16-18, 2019, Lisbon, Portugal*. 10:1–10:27.

[64] Jeff Vander Stoep and Chong Zhang. 2019. Queue the Hardening Enhancements. https://android-developers.googleblog.com/2019/05/queue-hardening-enhancements.html.

[65] Zahra Tarkhani and Anil Madhavapeddy. 2020. μTiles: Efficient Intra-Process Privilege Enforcement of Memory Regions. *arXiv preprint arXiv:2004.04846* (2020).

[66] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. 2019. ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK). In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 1221–1238. https://www.usenix.org/conference/usenixsecurity19/presentation/vahldiek-oberwagner

[67] Victor Van Der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clémentine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. 2016. Drammer: Deterministic rowhammer attacks on mobile platforms. In *ACM Conference on Computer and Communications Security (CCS)*.

[68] Nikos Vasilakis, Ben Karel, Nick Roessler, Nathan Dautenhahn, André DeHon, and Jonathan M Smith. 2018. BreakApp: Automated, Flexible Application Compartmentalization.. In *NDSS*.

[69] Jun Wang, Xi Xiong, and Peng Liu. 2015. Between Mutual Trust and Mutual Distrust: Practical Fine-Grained Privilege Separation in Multithreaded Applications. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference* (Santa Clara, CA) *(USENIX ATC '15)*. USENIX Association, USA, 361–373.

[70] Y. Wu, J. Sun, Y. Liu, and J. S. Dong. 2013. Automatically partition software into least privilege components using dynamic data dependency analysis. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 323–333. https://doi.org/10.1109/ASE.2013.6693091