École Polytechnique Fédérale de Lausanne

Performance Evaluation of Compartmentalized Systems

by Jonas Konrad

Master Project Report

Prof. Dr. sc. ETH Mathias Payer
Project Advisor

Florian Hofhammer
Project Supervisor

EPFL IC IINFCOM HEXHIVE
BC 160 (Bâtiment BC)
Station 14
CH-1015 Lausanne

January 6, 2023

Acknowledgments

I would like to thank Prof. Payer for allowing me to carry out my Master Project at the HexHive lab.

Furthermore, thank you to Florian Hofhammer for the supervision and proposal of the project. His guidance, technical skills, and willingness to help allowed me to approach the project with confidence.

Finally, another thank you to Prof. Payer for providing the LaTeX package that was used to create this document. It can be found at `https://github.com/HexHive/thesis_template`.

Lausanne, January 6, 2023 Jonas Konrad

Abstract

Compartmentalization is a technique intended to provide security for programs against bugs or malicious code by containing them in a controlled environment, from which they cannot influence the remainder of the system or application. It can be implemented in many ways and at various levels of abstraction. Every different approach to compartmentalization incurs some form of overhead. It then becomes a trade-off between security guarantees and performance costs. This project aims to benchmark the overhead of three compartmentalization architectures, namely SecureCells, ERIM, and Lightweight Contexts. To that end, we use a microbenchmark to measure compartment switch overhead, and a Memcached-like program to highlight expected real-world performance. We show that SecureCells outperforms both ERIM and Lightweight Contexts. Additionally, we discuss the three challenges of benchmarking, namely completeness, soundness, and reproducibility.

Contents

Chapter 1

Introduction

The core idea of compartmentalization is to separate two or more parts of a system to prevent malfunctions from spreading between or among them. To achieve this, a system needs to be split up into smaller parts. During this task, a well-defined, and smallest possible interface needs to be used. Additionally, to achieve the best results, these parts focus on one specific task, that is, they should follow the principle of least privilege. This approach brings protection against bugs or malicious code to the program because they cannot propagate to parts of the system outside the compartment they are in. To enforce this security policy, there needs to be a lower-level security mechanism in place. Another important consideration is that modern software is complex and contains many parts. During its execution, there will hence be a lot of switching between these parts. This is where the biggest issue with compartmentalization comes from. Every time execution goes from one compartment to another, there will be some overhead that impacts performance. Because of this, a compromise between performance, threat model, and usage complexity exists for every compartmentalization framework. Many approaches have been tried to reach the best solution. We distinguish four main categories: *operating system* (OS), language/compiler and runtime, virtualization, and hardware-based.

We distinguish three major differences between compartmentalization frameworks: the threat model, the usage complexity, and the performance overhead. They are elements that need to be taken into account when comparing compartmentalization frameworks. Comparisons of threat models can be done objectively since they are, by intent, well-defined and allow comparison between each other. Usage complexity, whilst arguably more subjective, is also not an issue, since it will inherently highlight which modifications need to be applied to software, and in any case, does not depend on external factors. Finally, the third aspect is where the issue lies. Benchmarking is not a trivial task. There is a multitude of factors to account for, like the influence of hardware, operating systems, scalability, etc. This implies that benchmarks need to be designed in a way to showcase the influence of every factor. Their results should have a meaning, and be a representation of real-world performance. Reproducibility is also important, to enable others to validate the result and get a prediction of the performance they can expect on their systems. All these elements play a role in how comparable different compartmentalization designs are.

Properly benchmarking systems is a difficult and intricate task. Kouwe et al. [12] identified 22 benchmarking flaws that threaten the validity of systems security evaluations in papers published in top venues. Whilst these flaws are not ill-meant, they are widespread and threaten the scientific process, which relies on reproducibility and comparability.

One proposed compartmentalization design is SecureCells [3], developed at the Hexhive lab. Hardware and OS based, it uses a combination of *virtual memory area* (VMA) granular permissions and userspace instructions to facilitate secure and fast compartment switching. In the context of SecureCells' development, there is a need to compare it to two other systems: ERIM [21] and *Lightweight Contexts* (lwC) [15]. These two were chosen because they are well-known compartmentalization works that do not have any prohibitive requirements which would have it difficult for us to use them. Notably, they do not require hardware changes. This project aims to meaningfully compare these three systems. There are multiple aspects to consider. Indeed, they use three different approaches and have different hardware requirements and functionalities. We use microbenchmarks as well as one macrobenchmark to do the performance comparison.

We show that the performance overhead introduced by lwC compartment switches is too high to allow its use in software that frequently switches between compartments executing a small amount of code. We also show that ERIM performs better in that regard, but its limited usability for compartmentalizing complex software is a disadvantage. SecureCells outperforms both ERIM and lwC thanks to its lower switching cost, and its use of a range-based *translation lookaside buffer* (TLB).

To come to these conclusions, we implemented a small library allowing us to count hardware events at an arbitrary point in the execution of the benchmark on both Ubuntu and FreeBSD systems. By reproducing the results stated in the lwC and ERIM papers, and by modifying the relevant benchmarks, we do a meaningful comparison of compartment switching performance overhead between lwC, ERIM, and SecureCells. We reuse the Memcached-based macrobenchmark introduced in SecureCells to do a performance comparison between the same three designs.

Chapter 2

Background

We aim to meaningfully compare the performance of three distinct compartmentalization systems. To understand the challenges involved, as well as understand which performance indicators should be considered, it is necessary to have sufficient knowledge of what compartmentalization is. We also need to understand the three systems we want to compare. The performance results mentioned here are taken from the papers and their benchmarks.

2.1 Compartmentalization

2.1.1 General concept

Modern software is getting increasingly complex and connected. It is typically composed of many subsystems and components that interact and constitute the application. Additionally, some of these parts are incorporated code from third parties, like libraries or plugins, which should be considered untrusted. All of this leads to a complex and interdependent system. Compartmentalization is a way to mitigate the impact of the vulnerabilities that such a system likely contains. The core idea is to split applications into least-privilege components, that are isolated from each other and communicate through a well-defined interface.

Compartmentalizing software provides a multitude of security benefits. A compromised component will not be able to compromise another, since the communication between them is defined and executed in such a way as to not allow any deviation from the expected behavior. Since each compartment operates under the principle of least privilege, the damage that can be done by compromising it is minimized. Indeed, in contrast to monolithic software, every compartment will only have the permissions it requires. It also allows the programmer to use third-party code in isolation, reducing the security risk of doing so. The same reasoning and guarantees also applies to bugs and logic errors in the code.

We distinguish two kinds of compartmentalization: extra- and intra-address space. Extra-address space is not used for individual programs, but to isolate different programs from each other. It could also be used to isolate parts of the same program, but the overhead incurred when switching compartments is very high. For example, since it typically utilizes OS-provided memory isolation, switching between compartments is equivalent to a process context switch. Intra-address space compartmentalization is more interesting. The issue with monolithic programs is that each of their parts share the same permissions to all data regions. Whenever an attacker gains access to any vulnerable part of the program, they can use techniques such as return-oriented programming to hijack program control flow. By introducing intra-address space per-compartment access rights in conjunction with policies for inter-compartment communication, compartmentalization reduces the impact of these kinds of attacks. Bugs or attacks are constrained to their compartment and cannot leave it, nor impact other compartments negatively.

We previously mentioned the security policy, but every design also needs a security mechanism. It is this mechanism that impacts program performance. In compartmentalization, the performance overhead comes from the program launch, checks during program execution (e.g., when using code instrumentation), or from switching between compartments. It is also worth noting that is important to consider the complexity of a design as well. If it is very hard to use it in production, if it has prohibitive hardware and software requirements, or if it is not flexible and cannot adapt to many kinds of programs, then that is also important. As long as it is not used, it is not preventing any attacks. When following the principle of least privilege, software can end up being split into many compartments that will often interact, and hence switching will occur often. The overhead of a round-trip can range from 82 cycles for XPC [6] to 12'000 cycles for lwC [15]. For example, if the concerned software is connected to a network, the compartment handling cryptographic operations will be switched to every time a packet is transmitted using TLS, and such an overhead adds up to be significant.

2.1.2 ERIM

ERIM [21] relies on the Intel x86 ISA extension called *memory protection keys* (MPK), also referred to as *protection keys for userspace* (PKU). Introduced in 2015 in the Skylake architecture, it allows managing memory permissions from userspace. Memory pages are tagged with 4-bit protection keys, and access checks are implemented in hardware, imposing no overhead on program execution. Once pages are tagged, the owning thread can change their permissions directly from userspace. Nonetheless, a system call is still required when initially tagging pages, as well as when allocating, and freeing keys. A register handles the rights associated with each key. Instructions are used to read (`rdpkru`) and write (`wrpkru`) that register. The `xrstor` instructions can also indirectly be used to write to this register. Since they are stored in a register, access rights are local to threads. Different threads can have different access rights to the same virtual memory page, which was not possible using e.g., `mprotect` on Linux. Additionally, using PKU is much faster than `mprotect`, around 50 times faster [19]. MPK does not provide strong security by itself since a compromised component can update its permissions. What ERIM adds on top is binary inspection to ensure that all occurrences of the `wrpkru` instruction are safe. One such instruction is defined as safe if it is immediately followed by either a pre-designated entry point of a trusted compartment or a specific sequence of instructions that checks that the permissions set do not include access to a trusted compartment. Binary inspection and instrumentation are used to enforce this.

ERIM can be used to provide either complete isolation (confidentiality and integrity) between two compartments, or only write protection (integrity). To achieve this, it reserves regions in the address space for exclusive access by one compartment each. Reservation is done by using MPK. The rest of the address space is the untrusted compartment, which holds the applications' regular heap and stack. It is completely accessible by trusted compartments. Call gates enable access to trusted compartments. Their code in assembly, taken from the paper, is visible in Listing 2.1. They transfer control to explicit or inline functions, which act as predefined entry points. Once execution of that function is finished, they transfer control back to the caller and disable access to the called functions' associated trusted compartment. They operate entirely in user mode. In a multi-threading context, each trusted component needs to allocate its private stack. When working with more than two components, care needs to be taken for the trust relations to be transitive. The simplest setting would be to have one untrusted and any number of trusted components that do not trust each other. ERIM provides only memory isolation. For privilege isolation, the instrumentation used for the `wrpkru` instructions and the call gates can be extended to support privilege separation concerning OS resources.

```
    xor ecx, ecx
    xor edx, edx
    mov PKRU_ALLOW_TRUSTED, eax
    WRPKRU ; copies eax to PKRU
    ; Execute trusted component's code
    xor ecx, ecx
    xor edx, edx
    mov PKRU_DISALLOW_TRUSTED, eax
    WRPKRU ; copies eax to PKRU
    cmp PKRU_DISALLOW_TRUSTED, eax
    je continue
    syscall exit ; terminate program
    continue:
    ; control returns to the untrusted application here
```

Listing 2.1: Assembly implementation of ERIM call gates.

ERIM does not make any assumptions about the untrusted component. Even in case of control-flow hijack vulnerabilities, the trusted components will not be accessible. This also means that trusted components cannot have such vulnerabilities and should be correctly implemented. Otherwise, an attacker could go into the untrusted compartment from there, breaking the compartmentalization. Trusted compartments should not compromise sensitive data through explicit information leaks, they should not call back into the untrusted component before the call gate does, and they should not map executable pages with unsafe occurrences of the `wrpkru` or `xrstor` instructions. The hardware, the OS kernel, dynamic program loader/linker (because it should invoke ERIMs initialization function before any other code in a new process), the compiler (for adding the entry points), and the ERIM library added to each process using it are trusted to be secure. Standard *data execution prevention* (DEP) also needs to be implemented. Trusted components need to carefully clean up everything they put onto the shared stack before finishing execution, such as to not leak anything to the untrusted component.

Since it relies on Intel MPK support, ERIM can only be used on appropriate hardware, that is Intel Skylake (6th gen) and later server CPUs (i.e. Xeon, etc.) or Intel Tiger Lake (11th gen) and later consumer CPUs (i.e. Core, etc.). Additionally, the operating system has to support this technology as well, which Linux kernel versions 4.9 and later do. Beside these two conditions, depending on the implementation chosen, it is simply a matter of either using the ERIM library or doing small (26 LoC) modifications to the Linux kernel.

The main overhead of using ERIM comes from the call gates used to switch between compartments, from 11 to 260 cycles per switch. The other overhead comes from binary inspection, taking between 3.5 and 6.2 microseconds per memory page.

One of the main advantages of ERIM is the hardware requirements. Since MPK is supported on the latest Intel processors, it can be deployed on most modern machines. The complexity of using ERIM is also not high, it is easy to isolate specific parts of a program, e.g. the cryptographic library, from the rest. Additionally, its codebase is small (less than 100 LoC for everything) and hence easier to check for correctness.

The main issue with ERIM is its limited flexibility. There is a limited amount of possible compartments, i.e, one untrusted monolithic compartment and up to 15 trusted compartments. Indeed, since memory protection keys are 4-bit and the key 0 is the default process key used by the untrusted compartment, there can be at most 15 trusted compartments. It is also important to keep in mind, that ERIM requires the operating system to be trusted, which is in most cases (e.g. Linux, Windows, etc.) a very large monolithic piece of software. Finally, if ERIM is used, then it "reserves" PKU for its own use. Because every memory page can only have one associated key, they cannot be used for other purposes at the same time.

2.1.3 Lightweight Contexts

Operating system processes provide memory and privilege isolation between programs, but the performance overhead of context switches is high. Lightweight Contexts aims to introduce compartmentalization inside an OS process by using a variant of processes with lower switching cost. One such entity is called a *lightweight context* (lwC). Threads and lwCs are orthogonal, an lwC is not schedulable. The same OS thread may start by running lwC *a*, switch to lwC *b*, etc. Each lwC is a separate unit of isolation, privilege, and execution state within a process.

Each OS process starts with one root lwC, which might create new lwCs (children), similar to POSIX forks, but they all share the same process id and no new OS thread is created. Instead, execution continues in the parent lwC, and the child is not automatically scheduled. These children can in turn also create new lwCs of their own, and so on. A lwC can switch to any other lwC it has knowledge of. By default, a new lwC gets a copy of the parents' state at the time it gets created, including per thread register values, virtual memory, file descriptors, and credentials. The parent may modify the visibility of these resources at creation or afterward. It can also set up shared memory regions. The behavior of system calls and signals can also be specified, e.g., to be redirected to the parent. Access capabilities associated with each lwC define who has access to what resources, which also includes other lwCs since they are identified by file descriptors. An operation allows switching to other lwCs. The file descriptor of the caller of this operation is available to the called-into lwC. Switching is semantically equivalent to a coroutine `yield`. Since lwCs do not have access to the state of other lwCs memory, file descriptors, and capabilities (unless explicitly shared), they can be used to provide isolation and privilege separation within a process. Additionally, they also provide mechanisms for sharing and control between lwCs of the same process. The idea is that parents drop as many privileges as possible after creation, to follow the principle of least privilege.

Different parts of the system are assumed to be trustworthy and uncompromised: the kernel, the tool chain used to build, link, and load the applications (otherwise, control could be hijacked before `main()` starts). The security of an individual lwC is only provided if the parent is not compromised at creation time, since it has full control over the child then. Since an lwC runs inside an OS thread, its privileges can only be a subset of that threads' privileges. This also means that the privileges of the root lwC are those of the OS thread it runs in. It is worth mentioning, that an application using lwC needs to make certain assumptions. Notably, an lwC can block or execute a thread indefinitely, or terminate the process prematurely by invoking exit.

Lightweight Contexts does not require specific hardware, but it does require implementation of kernel modifications.

The performance overhead of lwC comes from the context switches. It is half the time of OS process switches. This speed-up is because only virtual memory mappings need to be swapped, and no scheduling and synchronization are required.

Lightweight Contexts is a very flexible compartmentalization architecture. Additionally, it only requires a kernel modification, but no specific hardware. Using it to compartmentalize programs is also manageable. The logic is the same as when using POSIX forks, but with additional attention to be put into applying the principle of least privilege.

The big disadvantage of Lightweight Contexts is the performance overhead, with 2 ms for a switch. As with ERIM, its threat model also requires a very large, monolithic codebase to be trusted (namely the kernel and build tool chain).

2.1.4 SecureCells

In contrast with ERIM and Lightweight Contexts, SecureCells [3] does not aim to integrate easily into the current hardware and operating systems designs. This choice was made because they are not well adapted to compartmentalization, and will always require compromises if one wants to implement fast and secure compartmentalization on them, e.g., by tailoring them to a specific use case, or by limiting the number of compartments. SecureCells' approach is one of hardware-software co-design. Everything that can safely and efficiently be implemented in software is done as such, whilst performance and security critical operations are managed by hardware. It adds userspace instructions implementing compartmentalization primitives, such as compartment creation, permissions transfer, etc. The key insight in its design is that compartmentalization operations are secure if they go through *trusted computing base* (TCB) maintained permission checks. These checks must be implemented in hardware when a software implementation would be the bottleneck, otherwise, the performance overhead would be too high. By providing *VMA-granularity data regions* (cells) access control, SecureCells removes the overhead incurred when needing to check access rights at page granularity. The traditional page based *translation lookaside buffer* (TLB) is not used. SecureCells uses a range-based TLB because it makes checking permissions faster. Indeed, instead of having an entry for each page of the VMA, a range-based TLB will have one entry per VMA. This means that the permission will be checked only once per VMA access. One compartment is assigned to one *security division* (SD), which in turn has access to a set of cells. To track which SD is currently executing, a read-only register (named SDID) storing the currently executing compartment is added to each CPU core. A table storing the per cell, per compartment permissions (PTable) is also added. A different table tracks the granted permissions (GTable). The MMU uses these tables to enforce the access rights to cells. There is only restricted access to the PTable from userspace. Userspace instructions are introduced to accelerate common compartmentalization operations. They handle compartment switching with checked entry points, and permission surrender and transfer. The software mechanisms handle argument validation for call gates, maintaining the call stacks, and context switching for register context isolation. This is done because they would not be faster in hardware, are hence more flexible, and can be skipped if it is safe to do so.

The threat model only requires the hardware implementation and the supervisor to be trusted. Alongside a sound policy used for compartmentalization and well-designed interfaces between compartments, a compromised compartment cannot be used to compromise another.

The hardware requirements are very high. Since SecureCells relies heavily on hardware implementations, it cannot run on any commodity hardware. It requires implementations of the TLB, MMU, PTable, GTable, additional register, and additional instructions.

As with every compartmentalization architecture, the main performance overhead of SecureCells comes from the switching between SDs. The redesigned MMU and TLB enable a minimum amount of checks to be made for each switch. Additionally, since there is no data to be copied, but only permissions, switches are also executed quicker once validated. The use of custom hardware makes these operations faster. Finally, by making the common compartmentalization available to userspace, there is a limited need for transfer of control to the supervisor. This enables a faster switching time of 8 cycles, compared to ERIM at 99 and lwC at 6000.

As just mentioned, SecureCells achieves state-of-the-art switching overhead, but this is not its only advantage. Its design allows for a very flexible approach to compartmentalization, without any use case- or performance compromise-related constraints. Additionally, it requires a much smaller TCB than other designs. This allows it to be more confidently checked for correctness.

Nonetheless, it has two major downsides. The first one is the hardware requirement. On one hand, this is what enables its performance in the first place, but on the other hand, it constitutes a big barrier to wide adoption. The second one is the various issues related to the range-based TLB, notably fragmentation, which would need to be resolved. For example, Midgard [9] proposes an intermediate address space between the virtual and the physical address spaces, allowing for the benefit of one entry per VMA, as visible by the processor, without the need for associated physical memory to be contiguous.

2.2 Benchmarking

The performance of a defense mechanism or mitigation is almost as important as its security guarantees. Indeed, one cannot be considered without the other. Whilst in some applications, the acceptable performance overhead might be high, e.g., the mitigation of the Spectre vulnerability, in others, even the smallest performance hit is very expensive, e.g., in a high throughput web server. This means that the performance evaluation of a mitigation or design is equally important as the security guarantees. This is a real-world problem. Kouwe et al. [12] have shown that benchmarking mistakes are widespread, even amongst top venues. This appears constant through time, showing that there are not yet enough measures taken to improve this. The scientific process relies on reproducibility and comparability, otherwise, research cannot advance. If it is not possible to meaningfully compare two security systems, then it is also not possible to take an objective decision as to which is better suited to the desired application, or which design is more promising. We discuss in this section, what makes for a good benchmark, and more exactly, what is to avoid. It is important to note, that Kouwe et al. [12] did not identify any occurrence of cheating. They believe that most, if not all, benchmarking flaws are unintentional and most likely due to too low attention given to them. Note that benchmarking security is much harder, hence we focus on benchmarking performance. A benchmarking process should aim to fulfill four requirements [12]:

1. Completeness: it should verify all the claims about a system, as well as show any potential negative impact.
2. Relevancy: it should be meaningful, i.e., a benchmark should measure a metric that actually impacts performance in a real world use case.

3. Soundness: the benchmarking setup should be accurate enough for the results to be reproducible and meaningful, e.g., the standard deviation should not be too high.
4. Reproducibility: enough information about the benchmarking setup for others to reproduce the results need to be provided.

We distinguish two kinds of benchmarks. Microbenchmarks are designed to highlight specific aspects of a system. For example, in the context of compartmentalization, the number of cycles taken by a compartment switch, or the overhead of communication between compartments. In contrast, macrobenchmarks highlight the performance in a close-to real-world setting. The counterpart to the aforementioned microbenchmarks could be an encrypted video chat application, where the cryptographic library is in a separate compartment and the video feed is encrypted by it. Both of these kinds of benchmarks are needed, e.g., it is important to have performance measurements for precise operations for easy comparisons of specific aspects between systems. Nonetheless, the actual overhead created by these operations in a real program is also important, since it helps to understand which applications can benefit from using that system.

A benchmark inherently provides a number as a result. It is hence important to decide on a metric that represents a meaningful aspect of the system. It should also not be selectively chosen, otherwise, the results might be biased. For example, if a system allows for very fast compartment switching but adds significant overhead to each function call, then both switching overhead and function call overhead should be included in the metric. Hence, the metric can, and often should be multi-dimensional. This implies that there should be benchmarks evaluating all operations that are affected. Improvements in one operation might be overshadowed by the slow-down it introduces in others. This also means that no metric should be forgotten, e.g., maybe CPU overhead is very low, but memory overhead is very high.

Misinterpretation of benchmark results is one pitfall of benchmarking that is easy to fall into. Let us consider a benchmark which metric is throughput. It is often the case that a CPU idles during a workload, for example when waiting for I/O. If the CPU works during that previously idle time, then these cycles of overhead will be hidden when considering throughput as the metric.

Especially in the context of security systems, specialized hardware or modified operating systems are needed. This makes benchmarking harder because of limited functionality or unavailable tools. Sometimes this makes emulation unavoidable. This situation is of course not ideal because emulation performance might not be representative of real-world performance. This also implies that the chosen metrics should allow for comparison between different hardware. Raw measurements are rarely comparable, instead, the relative improvement compared to a baseline should be used. Nonetheless, absolute performance values can still be useful to the reader to perform a sanity check of the benchmark, for example, they might be unreasonable, or too low for the overhead to be visible. This baseline needs to be carefully chosen. Usually, it is the original system without the defense enabled. The hardware used needs to be specified as completely as possible.

As with every data visualization, it should be done in a non-misleading way. For example, results from a microbenchmark should not be presented as representing real-world performance. The standard rules for displaying data need to be respected. Scales of graphs should not be misleading, colors should be consistent, uncertainty should be shown, and so on. In a related manner, if the comparison is done with another system, this system should be the state of the art. This comparison should be done fairly, where none of the systems is put in a favorable position, or at least the differences in performance per use case should be highlighted.

Finally, many small but impactful mistakes can be made. There can be errors in overhead calculations, like saying that an increase in performance overhead is of 10% for an increase from 40% overhead to 50% overhead, when it is actually of $\frac{50-40}{40} = 25\%$. The variance of results is also important. If it is too high, the reproducibility and soundness of a benchmark are impacted. The geometric mean should be used [7] to average overhead ratios (i.e., normalized numbers), because the arithmetic mean depends on the baseline performance.

Chapter 3

Design

To compare three different compartmentalization architectures, namely SecureCells [3], ERIM [21], and Lightweight Contexts [15], we leverage a microbenchmark measuring the overhead of a compartment switch, and a macrobenchmark based on an in-memory key-value store.

3.1 Microbenchmarks

Microbenchmarks are inherently comparable between systems. Indeed, they measure one precise aspect of the design in isolation. As long as they are implemented correctly, and use the same metric, their results can be compared directly. For this reason, we keep the respective microbenchmarks presented in each paper. The ones measuring compartment switch cost overhead can be compared directly, whilst the others measure aspects unique to each design.

3.1.1 SecureCells

The microbenchmark measures the latency, in cycles, of `SDSwitch` instructions implemented in hardware. The implementation details of the benchmark are not specified in the paper. We reuse the results as they are presented in it.

3.1.2 ERIM

Since ERIM allows only the compartmentalization of function calls, the overhead of this addition is what is measured in the microbenchmark. Three kinds of function calls are measured: inlined, direct, and indirect. The paper says that the benchmark adds a constant to an integer and returns the result, but the available code artefact increments each number in a three element array by one. This does not matter for the result, but it is worth noting for completeness. A second microbenchmark measures the cost of binary inspection, but this cost is shown to be negligible (17.7ms for a program using 3918 4KB pages) when considering typical process runtime. Since this is the only design with such an "up front" overhead, we consider it irrelevant in this comparison. ERIM uses the Time Stamp Counter to count elapsed cycles. This is an issue because its behavior has changed in more recent architectures, for some processors. It now counts at a fixed rate, independent of the actual clock rate, and is not an ordered instruction. See chapter 17.15 of [5] for details. This makes it unfit to measure small amounts of cycles, as in a microbenchmark, and not consistently portable to other systems. For this reason, we use our library, presented in section 3.3, to get the results of this benchmark.

3.1.3 Lightweight Contexts

There are two microbenchmarks in the lwC paper. One measures the cost of creating, switching to, and destroying lwCs. It is not relevant to our discussion and hence we do not consider it. The other measure the cost of switching to an lwC, and compares it to other context switches. The exact design of the benchmark is not mentioned in the paper, but we can know it from the code artefact. For 1’000’000 iterations, context is switched between two threads, each incrementing a shared variable by 1. Time is measured in seconds and then divided by the number of iterations to get the median switch time. As we explain in section 3.3, whilst time can work to compare performance on the same hardware configuration, it is less than ideal for a microbenchmark. As for ERIM, we modify the benchmark to instead count execution cycles.

3.2 Macrobenchmark

We now focus on the macrobenchmark we use. It is an in-memory key-value caching service for small chunks of arbitrary data based on memcached [18] and is the same as used in the SecureCells paper [3]. The hashtable-based data store is considered trusted and put in a separate compartment than the interface, which is untrusted because it faces the outside world. Said interface handles incoming requests, switches compartments, and handles the response. In a more realistic scenario, there would also be cryptographic operations enabling TLS encryption for the data in transit. Due to software limitations of the SecureCells prototype, this element was dropped, and data is transmitted in plain. Each entry in the data store is 64B in size. A benchmark run works as follows:

1. The benchmark receives a data size it has to store, alongside a number of iterations.
2. The data store is set up.

3. Once for every 64B chunk of the requested data, dummy data is put into the store.
4. Once per iteration, all the 64B entries composing the dummy data are read.

We note that, in steps 3 and 4, the keys used for the key-value store are generated in a way analogous to pseudorandom with a fixed seed. This is intended to simulate random memory access. With this benchmark, we aim to:

- Show the overhead incurred by frequent compartment switches.
- Highlight the benefits of a range-based TLB compared to page-based one.

3.3 Getting performance measurements

We make our measurements in step 4. The number of iterations allows for a warm-up phase. Measurements are made once per iteration in step 4, accounting this way for the retrieval of the entire file. There are three performance indicators we want to measure:

- Execution cycles: CPU cycles elapsed during execution
- Retired instructions: instructions that were retired (i.e., completely executed) during execution
- TLB misses: misses when translating memory addresses of data or instructions

We chose them because, whilst they are influenced by hardware, they are also the most comparable between each system configuration. An alternative to cycles and instructions would have been time, but this allows only for comparisons of relative improvements, which is not compatible with the aim of microbenchmarks. Also, whilst it could reasonably be used for macrobenchmarks (although not ideal), it is only possible when comparing systems running on the exact same hardware, which is not our case.

Except for SecureCells, where we reuse the results from the paper, we take our own measurements using *hardware performance counters* (HPC), also called *performance monitoring counters* (PMC). They are registers whose purpose is to store the counts of hardware-related events. One big advantage of HPC is their low overhead on the performance of the benchmarked program. The exact way how we access those counters are different for each of ERIM and lwC, due to the different operating systems and hardware they run on, but the principle remains the same. We implement the respective library allowing us to initialize the counters, execute benchmark steps 1 to 3, and finally clear/read the counters before/after each iteration in 4.

3.3.1 ERIM

We run the ERIM benchmark on an Ubuntu 22.04.1 LTS system equipped with an Intel processor. The standard tool for gathering performance counter statistics on such a system would be `perf stat`. It allows to either sample or count performance counter values during the execution of a command, a process, or system-wide. This means, that it is not applicable in our case. Indeed, we cannot use it to gather precisely the desired values for every iteration, but only for the entire benchmarks' execution. Instead, we leverage the `perf_event_open` syscall, which `perf stat` uses in its implementation. It allows us to read and write HPC directly from our benchmark.

3.3.2 Lightweight Contexts

Lightweight Contexts requires a modified FreeBSD 11.0 kernel to function. We installed it on a system equipped with an Intel CPU. `perf` is not available on FreeBSD, but `pmcstat` is a drop-in replacement. This means that there are the same issues with it for our use case. As an alternative, we use the `pmc` library, which, analogous to `perf_event_open`, allows us to read the counters manually, and is also used by `pmcstat` in its implementation.

Chapter 4

Implementation

4.1 Reproducing paper results

One of the first things we did in this project is to reproduce benchmarking results from both ERIM and Lightweight Contexts. We faced challenges for both, which we highlight in this section.

4.1.1 ERIM

We use the ERIM implementation which does not require modifications to the kernel. As we already mentioned in subsection 2.1.2, ERIM requires MPK support, both in hardware and in the OS, to function. We can satisfy both, especially since MPK support is available on the more recent consumer CPUs. When reading the microbenchmark code, we realized that it did not include the entire call gate in its code, but it was missing some instructions, namely lines 12 to 14 in Listing 2.1. As we will discuss in the evaluation section, this does not impact the microbenchmark results, but it is worth mentioning and fixing in light of the comments made on benchmarking in section 2.2, which we did. Due to license requirements for reproducing the Levee macrobenchmark, we only tried to reproduce results for the nginx macrobenchmark. This benchmark compartmentalized the SSL session keys used in the web server. It aims to measure the overhead introduced by ERIM, and uses the throughput of the application as the metric. The provided artefact required some adaption and bug fixes for it to run. Nevertheless, simply running the benchmark on a single machine did not allow for meaningfully reproducing the effort required to do so, the benchmarks' runtime, and because of the limited added value for this project.

4.1.2 Lightweight Contexts

We first set up Lightweight Contexts using the QEMU emulator, since it requires a modified FreeBSD 11 RC4 kernel to function. Support for FreeBSD 11.0 stopped in November 2017, which made it more difficult to set everything up, especially since we had no prior experience with this operating system. Indeed, because of this, it is not possible to install prebuilt packages directly using the Ports package manager, and we are restricted to compiling and installing packages from source. This means that we have restrictions on which packages are available to us, as well as which versions of them. We had to fix a hard-coded path to get the kernel to compile. The user libraries were installed without any issues, but the microbenchmarks were not at the location given in the documentation, and once we found them they first had to be identified, i.e., we needed to find them amongst the functionality tests. We did not manage to get the nginx macrobenchmark to work and did not pursue this further, for the same reason as for ERIM. We used the pmc library to read HPC. This required the hwpmc kernel module, whose functionality is not available from inside a virtualized environment. Additionally, the hardware the system runs on needs to have been release before 2017. Indeed, since we use a kernel from that time and the hwpmc module cannot read counters from processors it does not support. This is because the exact configuration and specification of performance counters are dependent on the CPU architecture. For the reasons we listed above, we set everything up on an Intel Haswell CPU-based laptop.

4.2 Measuring performance values

We use HPC to get performance metrics. The available libraries are different on Ubuntu and FreeBSD, but our implementation shares the same interface for both. Our implementation allows for access to HPC from inside an application. The normal usage is as follows:

1. Initialize the desired counters
2. Clear the counters, before the code to benchmark
3. Read the counters, after the code to benchmark
4. Free the resources

where steps 2 and 3 can be repeated any number of times. Both implementations measure execution cycles, retired instructions, data TLB read and store misses, and instruction TLB misses, although they can be configured to use any counter available on the used CPU. We not that in the FreeBSD implementation, the counters are stopped sequentially, e.g., the instructions counter will count the instructions retired during the syscall stopping the cycles counter. This is not ideal but does not influence the results noticeably, since it does not add more than a minimal amount of cycles or instructions to the final result. In the Ubuntu implementation, all the counters are in one event group, which means that all operations on them are done at the same time. The entire group is scheduled onto the CPU as a unit. This means that they can be meaningfully compared, divided, etc. with each other, since they have counted during the same set of instructions.

4.3 Benchmark implementation

The macrobenchark is based on the memcached [18] software. It was originally implemented by Atri Bhattacharyya for the SecureCells [3] paper. We split it into two compartments. One untrusted, which is the interface part of the program, and one trusted, which is the data-store part. After our modifications, two functions constitute the interface between both compartments, namely `cache_set_wrapper(key)` for putting data into the cache, and `cache_get_wrapper(key, buffer)` for retrieving data from the cache. It is at these two functions that the compartment switch happens.

To ensure that our compartmentalization works as expected, and did not break the code's functionality, we added two different checks. The functionality check verifies that we indeed managed to retrieve the value we expected. To achieve this, we check that the length, and the content match. You can consult the implementation at Listing 4.1. To test that the compartmentalization works as expected, we try to access the protected part of memory, i.e., the in-memory key-value store, without switching to the secure compartment, i.e., from the "wrong" compartment. If the program crashes, then we know that the sensitive part of memory is isolated away. Note that this does not mean that there is no exploitable vulnerability, nor does it aim to show it. The intention behind this test is to ensure that we did not wrongly use the library and that we are indeed running a compartmentalized program.

```
    char *response;
    int response_length = cache_get(response, ...);
    int check = memcmp(CACHE_VALUE, response, response_length) || response_length
    != CACHE_VALUE_LENGTH;
    assert(check == 0);
```

Listing 4.1: Code snippet of the benchmark where we verify that the expected value was retrieved from cache.

4.4 Compartmentalizing the benchmark

4.4.1 ERIM

To implement the benchmark, we first need to familiarize ourselves with the usage of both ERIM and lwC. For ERIM, whilst there is no API described in the paper, there is one in the artefact, and the source code is well documented and contains multiple usage examples. One issue we faced is the initialization of the compartments' memory. The first step when using this library is to initialize it using `erim_init(shmemSize)`, where *shmemSize* is the size of the memory that the compartment can map. Due to metadata overhead, there need to be two virtual pages of overhead per planned allocation. This was not specified in the documentation. Once this was resolved, both inlined (see Listing 4.2) and overlayed (see Listing 4.3) versions were quick to implement. Note that only the usage of two compartments, i.e., one untrusted and one trusted, is supported in the currently available ERIM implementation.

```
    erim_switch_to_trusted;
    // compartmentalized code here
    erim_switch_to_untrusted;
```

Listing 4.2: ERIM inlined call example.

```
    // declare wrapper around the function we compartmentalize
    // ERIM_BUILD_BRIDGEn(return_type, function_symbol, n_arguments)
    ERIM_BUILD_BRIDGE4(int, cache_get, const char *, int, char *, int);

    ...

    // usage of the function as a call into the trusted compartment
    // ERIM_BRIDGE_CALL(function_symbol, function_arguments)
    return_value = ERIM_BRIDGE_CALL(cache_get, key->keybuf, key->len, rep,
   MAX_MSG_SIZE - sizeof(int));
```

Listing 4.3: ERIM overlayed call example.

4.4.2 Lightweight Contexts

In the paper, the authors describe the API that lwC uses. However, another API is implemented in the artefact, and this one is provided with lackluster documentation. This meant spending time digging through the code, implementing examples, and writing our basic documentation, to understand it well enough to use in our benchmark. Both in the paper and the code artefact, the API is similar to POSIX fork. Both the examples, and the documentation, are in this project's repository. The standard programming flow when using lwC is as follows:

1. Create the lwC, where the return value works like for fork: `Lwccreate(resource-spec, ...)`. Shared resources and buffers are specified here. Execution continues in the parent.

2. Switch between compartments: `Lwcsuspendswitch(to, ...)` (state of the lwC is saved) or `Lwcdiscardswitch(to, ...)` (state of the lwC is not saved)

3. Close lwCs: `lwcclose(descr)`

In our benchmark, we define the shared memory needed for *inter process communication* (IPC) when we initialize the data part of the program, as the registers that can be used for IPC are too small in our use case. A broad overview of the process is in Listing 4.4. Once that is done, the secure compartment enters an infinite loop, the "server", which handles incoming requests from the untrusted interface, the "client", for either setting into or getting values from the cache. You can see relevant code at Listing 4.5 Note that, at line 32, we switch back into the untrusted compartment using a suspending switch instead of a discarding switch, which would have followed the snapshot and rollback pattern intended by Lightweight Contexts. We cannot do this because here, since the data cache is in memory, the entries would be lost. After all, the heap of the compartment would be reset to what it was upon entry.

```
ipc_buffer = mmap(NULL, buffer_len, PROT_READ | PROT_WRITE, MAP_ANON, -1, 0);

...

struct lwc_resource_specifier specs[1];
specs[0].flags = LWC_RESOURCE_MEMORY | LWC_RESOURCE_SHARE;
specs[0].sub.memory.start = ipc_buffer;
specs[0].sub.memory.end = ipc_buffer + (buffer_len / sizeof(char));

...

descr = Lwccreate(specs, 1, &src, to_server, &cap, 0);
if (descr == LWC_SWITCHED)
{
  cache_init();
  cache_server();
}
else
{
  Lwcsuspendswitch(descr, to_args, 2, NULL, from_server, &cap);
}
```

Listing 4.4: Abriged code of the creation of the data compartment using Lightweight Contexts.

```
1  void cache_server()
2  {
3      Lwcsuspendswitch(src, NULL, 0, 0, to_server, &cap);
4      while (1)
5      {
6      operation = to_server[0];
7      if (operation == 0) // set into cache
8      {
9          char *key = ipc_buffer; // parse arguments
10         char *value = ipc_buffer + MAX_MSG_SIZE;
```

```
            ret_code = cache_set(key, NCHARS, value, CACHE_VALUE_LENGTH);
            to_args[0] = ret_code; // send value to client
        }
        else if (operation == 1) // get from cache
        {
            int key_length = to_server[1]; // parse arguments
            char key_value[MAX_MSG_SIZE];
            memcpy(key_value, ipc_buffer, key_length);
            int value_buffer[MAX_MSG_SIZE];
            int valn = cache_get(key_value, key_length, value_buffer, MAX_MSG_SIZE
 - sizeof(int));
            to_args[0] = valn; // send value to client
            memcpy(ipc_buffer, value_buffer, CACHE_VALUE_LENGTH);
        }
        Lwcsuspendswitch(src, to_args, 1, 0, to_server, &cap);
        }
    }
```

Listing 4.5: Abriged code of the server running in data compartment.

Chapter 5

Evaluation

In this section, we discuss the different benchmark results, as well as the ease with which we could produce them, if applicable. The Lightweight Contexts related experiments were run on the following system (A): ASUS BU201LA-DT028G laptop, running the FreeBSD 11.0-RELEASE operating system with the Lightweight Contexts evaluation kernel, an Intel Core i5-4210U CPU 1.70 GHz (2.70 GHz boost) with 3 MB cache, 8 GB of DDR3 RAM at 1600 MHz, and a 256 GB SanDisk SD7SB3Q256G1002 SATA 3 SSD at 600 MB/s read speed. All other experiments were run following system (B): Lenovo ThinkPad T14s Gen 3 laptop, running Ubuntu 22.04.1 LTS with Linux kernel v5.15.0, an Intel Core i7-1260P CPU 2.10 GHz (4.70 GHz boost) with 18 MB cache, 16 GB of LPDDR5 memory at 4800 MT/s, and a 512 GB SKHynix HFS512GDE9X081N NVMe 1.3 SSD at 3 GB/s read speed.

5.1 Microbenchmarks: reproducing paper results

One important aspect to consider when designing benchmarks and publishing results is reproducibility. There are different reasons why someone might want to reproduce performance results. Be it to verify claims, to run them on different hardware for a more precise comparison with another design, or to modify the benchmarks to make them more meaningful when comparing with another design. All three cases apply to this project. Indeed, we want to verify performance claims, not because we do not trust them, but because we want to better understand how they were computed, and, if necessary, adapt the relevant benchmarks to allow for a more meaningful comparison with SecureCells. Additionally, it also helps us better understand how to design a benchmark, and how to publish results to ensure reproducibility.

5.1.1 ERIM

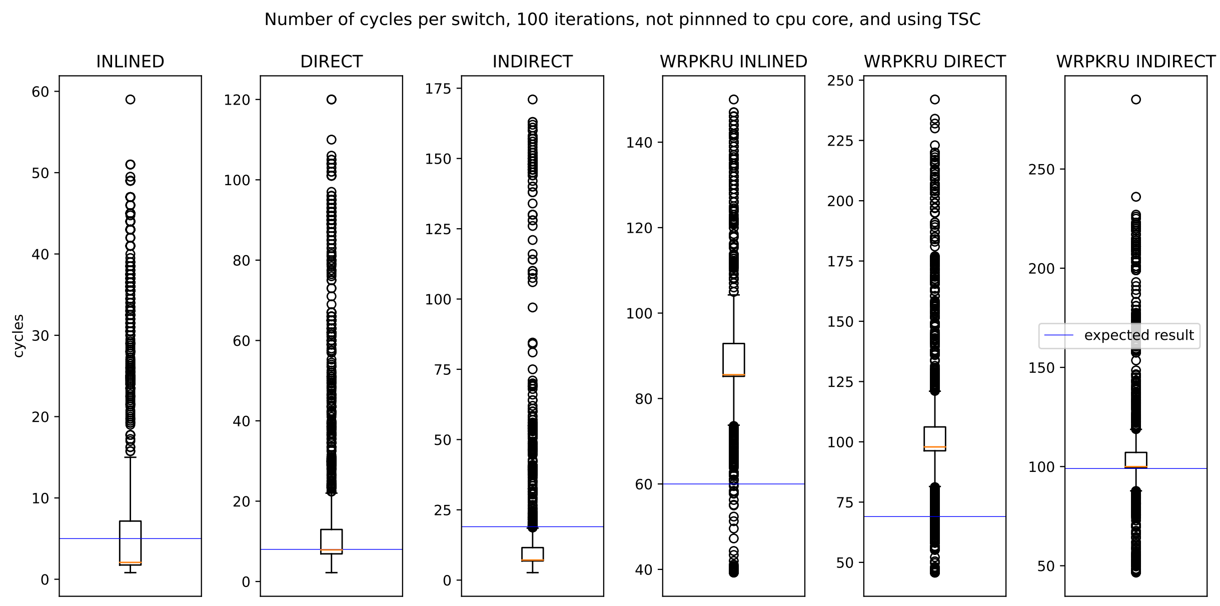
There were no particular issues to get the switch cycles microbenchmark running on our machine with configuration B. In the paper, there is no mention of how often the benchmark was run, nor of how the final results were derived or what the uncertainty is. When running it for ourselves, we noticed that the results where not very consistent, as can be seen in the bar plots of Figure 5.1a. To try and stabilize the benchmark results, we pinned it to a specific CPU core, which sure that the process where the benchmark runs does not change CPU core. It also has additional importance with our specific hardware configuration. Indeed, our Intel processor has a hybrid architecture, with both performance (core) and efficiency (atom) cores. By pinning benchmarks to a performance core, we avoid variability due to processor architecture differences. The stability improvements are visible in Figure 5.1b. As we explained in subsection 3.1.2, the time stamp counter is no longer appropriate for measuring execution cycles. To get more accurate results, we use hardware counters with the implementation describe at section 3.3. The new results can be seen Figure 5.1c. Finally, we also implemented the missing instructions in the call gate. Even though we measure cycles, and the missing `cmp` and `je` instructions are expected to add at least 2 of them to the measurements, the uncertainty of the benchmark is too high for this change to be significant, as can be seen in Table 5.1. Given the very high uncertainty, both using TSC and HPC, we use the median as the benchmark result and not the mean. The Lightweight Contexts paper gives 55 to 80 cycles of overhead for a simple switch using a call gate. Taking the results from part 5 in Table 5.1 we measure 73 to 84 cycles, which is an increase of 24% in the lower bound. The biggest issue here is the difficulty to reproduce the paper's results, mainly because of the high uncertainty in the results, and because of the lacking information on how exactly the results in the paper were computed. There are other, minor issues in comparison, but worth noting, like different work functions between benchmark versions (i.e., the one measuring indirect calls uses a different work function as the two others). The benchmarking design relevance of differentiating between, the three kinds of function calls can also be debated. Indeed, the overhead in cycles is not dependent on how the functions are called, it only depends on the call gates, which are the same in all three cases.

5.1.2 Lightweight Contexts

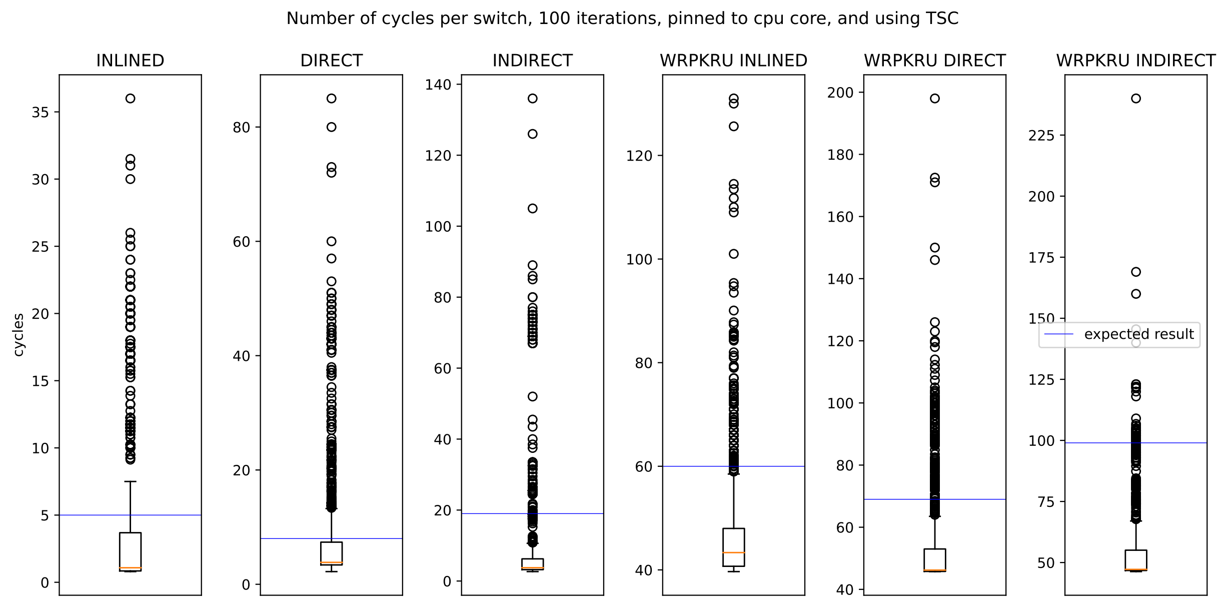
The microbenchmark used in the Lightweight Contexts paper measures and compares the cost of switching execution contexts of four different designs. Once we located the benchmarks in the artefact, they were not in the location given in the documentation, they could be reproduced without any issues. The results are in Table 5.2a. The paper mentions the uncertainty, which is low for both lwC and user threads, but comparatively high for kernel threads and processes. We did not get the same absolute results as in the paper due to the chosen metric, which is time in microseconds. The result is hence dependent on the hardware performance of the system that runs the benchmark. Nonetheless, we consider the benchmark to be reproducible because we measured the same relative differences of the systems compared to lwC, as well

Configuration	Type of function call	Expected	Mean	Median	Std. dev.
TSC for measurements	INLINED	5	8.38	2.07	12.45
Partial call gate	DIRECT	8	17.43	7.88	22.49
No CPU pinning	INDIRECT	19	20.30	7.14	33.86
	WRPKRU INLINED	60	90.50	85.57	18.46
	WRPKRU DIRECT	69	104.20	97.90	31.01
	WRPKRU INDIRECT	99	108.04	99.96	31.30
TSC for measurements	INLINED	5	4.28	1.08	6.21
Partial call gate	DIRECT	8	9.04	3.84	11.65
CPU pinned	INDIRECT	19	10.33	3.75	17.10
	WRPKRU INLINED	60	47.42	43.34	10.96
	WRPKRU DIRECT	69	54.05	46.21	16.27
	WRPKRU INDIRECT	99	54.82	47.20	16.26
TSC for measurements	INLINED	5	4.25	1.07	6.15
Complete call gate	DIRECT	8	8.96	3.81	11.55
CPU pinned	INDIRECT	19	10.26	3.66	17.05
	WRPKRU INLINED	60	48.26	42.79	13.95
	WRPKRU DIRECT	69	54.67	46.21	17.99
	WRPKRU INDIRECT	99	56.38	47.36	19.72
HPC for measurements	INLINED	5	144.94	5.00	362.20
Partial call gate	DIRECT	8	155.83	9.00	382.24
CPU pinned	INDIRECT	19	157.41	9.00	388.97
	WRPKRU INLINED	60	214.59	78.00	348.82
	WRPKRU DIRECT	69	232.24	89.00	373.98
	WRPKRU INDIRECT	99	236.22	92.00	381.52
HPC for measurements	INLINED	5	144.85	5.00	364.00
Complete call gate	DIRECT	8	155.08	9.00	379.72
CPU pinned	INDIRECT	19	156.97	9.00	388.36
	WRPKRU INLINED	60	218.21	78.00	360.30
	WRPKRU DIRECT	69	236.14	88.00	388.30
	WRPKRU INDIRECT	99	243.62	93.00	404.68

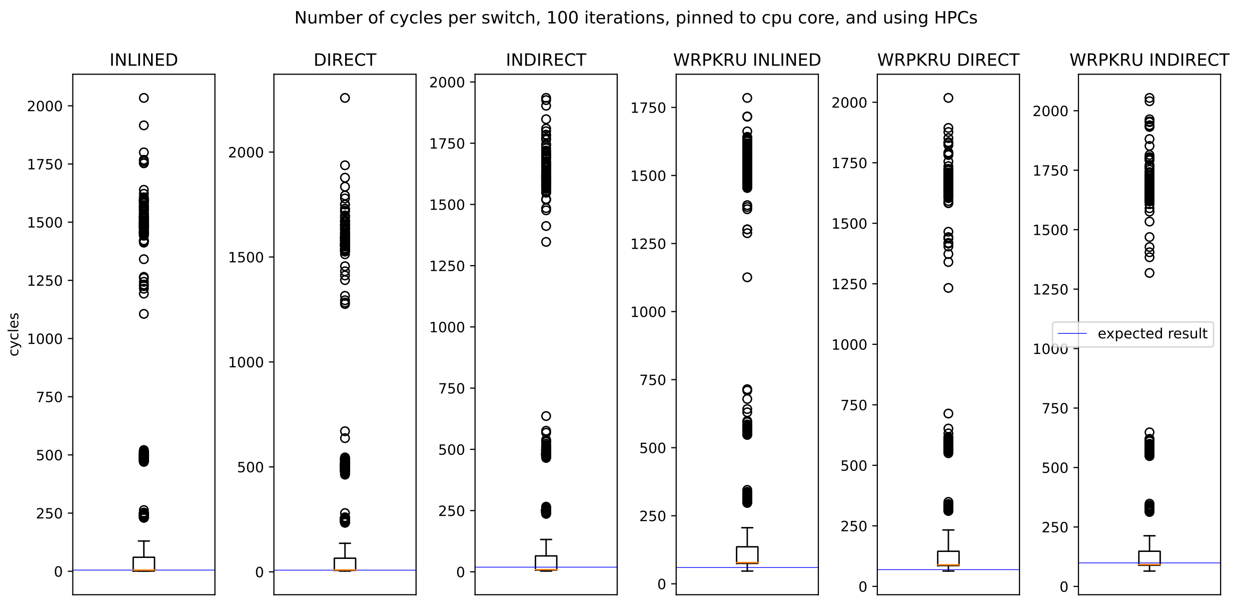
Table 5.1: Results of the ERIM microbenchmark measuring overhead of compartment switches.



(a)



(b)



(c)

Figure 5.1: Results of the ERIM microbenchmark. Note the different y-axis scales.

Type	Metric	Expected	Computed
lwC	median	2.01	3.80
	std dev	0.03	0.00
	relative	1.00	1.00
	mean	n/a	3.81
process	median	4.25	7.55
	std dev	0.86	0.02
	relative	2.11	1.98
	mean	n/a	7.55
u-thread	median	1.71	3.32
	std dev	0.06	0.23
	relative	0.85	0.83
	mean	n/a	3.17
k-thread	median	4.12	7.64
	std dev	0.98	0.02
	relative	2.05	2.00
	mean	n/a	7.63

(a) Original version measuring time in microseconds.

Type	Metric	Expected	Computed
lwC	median	n/a	6068.20
	std dev	n/a	2.54
	relative	1.00	1.00
	mean	n/a	6068.13
process	median	n/a	5518.94
	std dev	n/a	30.26
	relative	2.11	0.91
	mean	n/a	5509.13
u-thread	median	n/a	4552.55
	std dev	n/a	1.72
	relative	0.85	0.75
	mean	n/a	4551.66
k-thread	median	n/a	11041.67
	std dev	n/a	61.67
	relative	2.05	1.82
	mean	n/a	11023.76

(b) Modified version measuring cycles.

Table 5.2: Results of the lwC microbenchmark measuring overhead of compartment switches, over 10 trials, of four different designs: Lightweight Contexts (lwC), user threads (u-thread), kernel threads (k-thread), and processes (process).

as lower uncertainty than in the paper. The big drawback of this benchmark is the choice of time as metric. This makes it not comparable to other designs than the three done here. This is especially unfortunate because it is a microbenchmark, and hence intended to allow quick and easy comparison of low-level metrics of different designs. To alleviate this issue, we modified the benchmark to use our performance metric measurements described in section 3.3. The results are listed in Table 5.2b. We maintained very low uncertainty, validating the reproducibility argument of the benchmark. We also observed the theoretical cycle count of 6000 (6068 in our measurement) cycles of overhead for a Lightweight Context switch. The relative differences between kernel threads and user threads are also of the same magnitude. Note that the relative difference with processes is very different. Instead of 98% slower than lwC, it is 9% faster. This difference does not make sense. The overhead of process switches should be higher, alone given the additional scheduling operations of OS processes. The issue may come from the way we count CPU cycles, which appears to not count during these operations. Since we observe close to half the expected number of cycles, we believe that, although explicitly enabled, the cycles occurring in the child process are not counted in this case. Indeed, this microbenchmark executes the same operations in two different processes, then divides by the total number of iterations, which would lead to half the expected cycles with this problem. We did not verify this hypothesis.

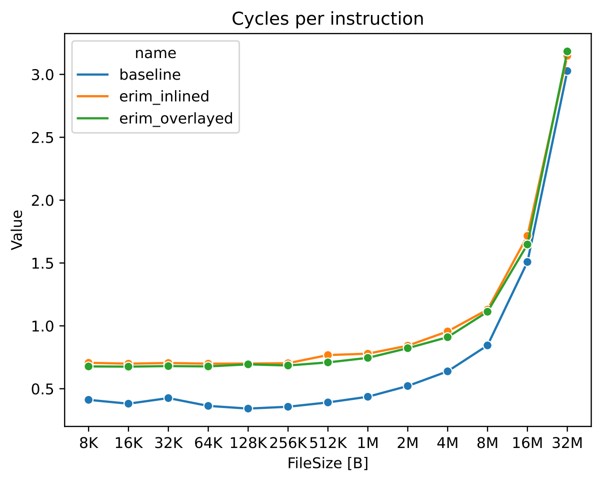
5.2 Macrobenchmark performance results

We ran our memcached-based macrobenchark, described in section 3.2 for both ERIM and Lightweight Contexts. Each time, the baseline is the uncompartmentalized version of that benchmark. We have different baselines, since the systems we used are different, as described at the beginning of this chapter. The results for SecureCells are taken from the paper [3].

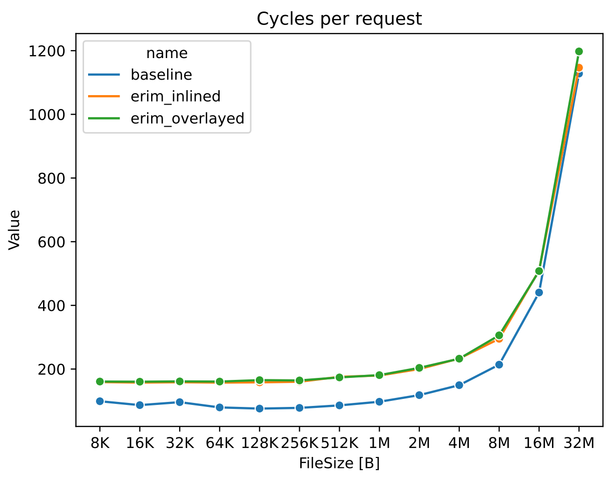
As explained at section 3.3, we need a way to get precise measurements of performance metrics from inside our C implementation of the benchmark. To do this, we implemented our library, using the lower-level functions used by tools like `perf`. We refer to section 3.3 for details. As we can see in Figure 5.2a and Figure 5.3a, there seems to be an issue with this library. Indeed, a *cycles per instructions* (CPI) count of less than one is not possible on a single-threaded application, which our benchmark is. This is because an instruction cannot be executed in less than one cycle, i.e., a single CPU cannot retire more than one instruction per cycle. We distinguish two cases. The first is when using configuration B. To try and fix the issue, we compared our implementation of the `perf_event_open` syscall with the one used by `perf stat`, since this tool gave us a CPI of more than one when used to take measurements for the full execution of the benchmark. We found its usage to be the same as ours. We performed a sanity check by measuring the metrics for the entire execution of the benchmark, instead of just the iterations of the value retrieval, but the result was also a CPI of less than one. Luckily, this behavior of our library is constant, and there is very low uncertainty in our measurements. This means that the relative performance between the baseline and ERIM is still meaningful, which is the most important aspect in our case. The second case, that of measurements for lwC, i.e., on configuration A, is trickier. We can again positively note the low uncertainty in our measurements. Also, the cycle counts for the baseline, see Figure 5.3a, make sense, and are as expected in theory. Sadly, we have again the issue of too low a CPI for the compartmentalized runs. We discuss this issue further in subsection 5.2.2.

5.2.1 ERIM

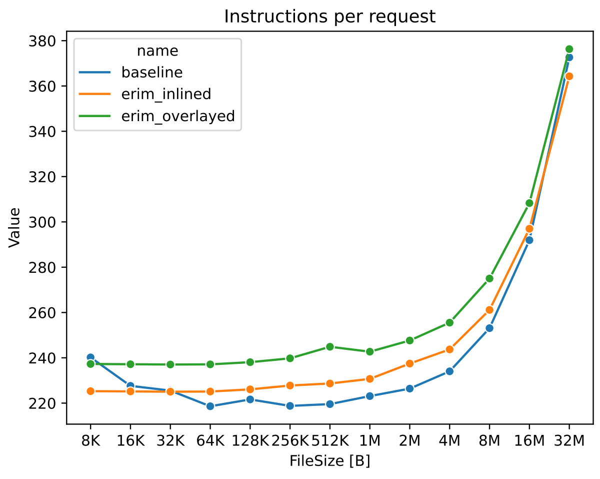
Our configuration B, on which this system runs, has a traditional, page-based TLB. As the data size exceeds 4 MB, TLB misses start to occur. This is apparent in Figure 5.2d. When looking at the CPI in Figure 5.2a, we can see that the call gates introduced by ERIM, notably the PKU-related instructions like `wrpkru`, cause a higher CPI. The bigger the data size gets, the less impact this fixed overhead has, as the TLB misses start to influence the CPI. The same behavior can be observed when looking at cycles per request in Figure 5.2b. When considering these three metrics, we notice that the inlined and the overlayed variants of ERIM behave similarly. This is no longer true when looking at instructions per request. As is visible in Figure 5.2c, the additional instructions needed by the dynamic link time overlay implementation are apparent. We have no explanation for the descending number of instructions per request until data size of 64 KB for the baseline. We believe it might be due to architectural optimizations.



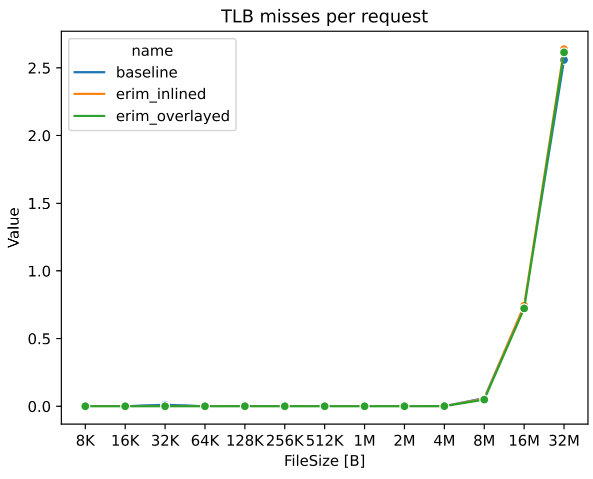
(a)



(b)



(c)



(d)

Figure 5.2: Results of our macrobenchmark for the ERIM compartmentalization design.

5.2.2 Lightweight context

As we already noted, there is an issue with the CPI measurements for Lightweight Contexts, as is apparent in Figure 5.3a. Whilst the values make sense for the baseline, they do not for the compartmentalized runs. When performing a sanity check using the command in Listing 5.1, which under the hood uses the same `pmc` library as we do, the result is also a CPI of less than one for the compartmentalized version. However, the CPI for the baseline is theoretically sound with this command, as is ours. We believe that the `pmc` library gets confused when an lwC other than the root lwC is running and stops counting during its execution and only resumes once the thread is again running the root lwC. Notices that this does not seem to be the case for instructions, which are counted correctly, as can be seen in Figure 5.3c (notice the scaling of the y-axis). Since we attach the counters to one process, it might be the case that the Lightweight Contexts implementation available to us assigns different process ids to different lwCs. Because of this, the counters we attached to the process when inside the root lwC stop counting when in the child lwC. We also instructed the counter to count in descendants of the process. We did not explore this further. From what we can know from the measurements we have, lwC and the baseline behave as expected. There is a massive overhead introduced by lwC context switches, as is apparent in Figure 5.3b. Indeed, the scaling makes the baseline look like having zero cycles per request, despite the cycles of the compartmentalized run being counted too low due to `pmc` limitations, as we established. This overhead gets even worse once the TLB misses increase, which can be seen when put in relationship with the results from Figure 5.3d. The reason is the same as for ERIM, we start to get misses due to the reach limitations of the page-based TLB.

```
$ pmcstat -C -d -p INSTR_RETIRED_ANY -p CPU_CLK_UNHALTED_CORE [memcached
benchmark]
```

Listing 5.1: Command to measure performance metrics using HPC: `-C` specifies counting mode, `-d` instructs to also count children of the starting thread, and `-p` the desired counters (here retired instructions and cpu cycles respectiveley).

5.3 Comparing the performance

When comparing the raw switching cost in cycles, as measured using the microbenchmarks, we can see in Table 5.3 that the difference is of orders of magnitude between the three designs. SecureCells benefits from the hardware implementation, whilst ERIM has the benefit of requiring only a call gate centered around a syscall. lwC has the largest switching cost because it needs to swap the virtual memory every time.

As for the performance of SecureCells on the memcached-based macrobenchmark, we see that the performance overhead is also the lowest compared to its baseline. These results are represented in Figure 5.4. Thanks to the low cycle overhead per switch, there is no visible impact on the raw cost in cycles. It even outperforms the baseline once the data size goes above 4 MB.

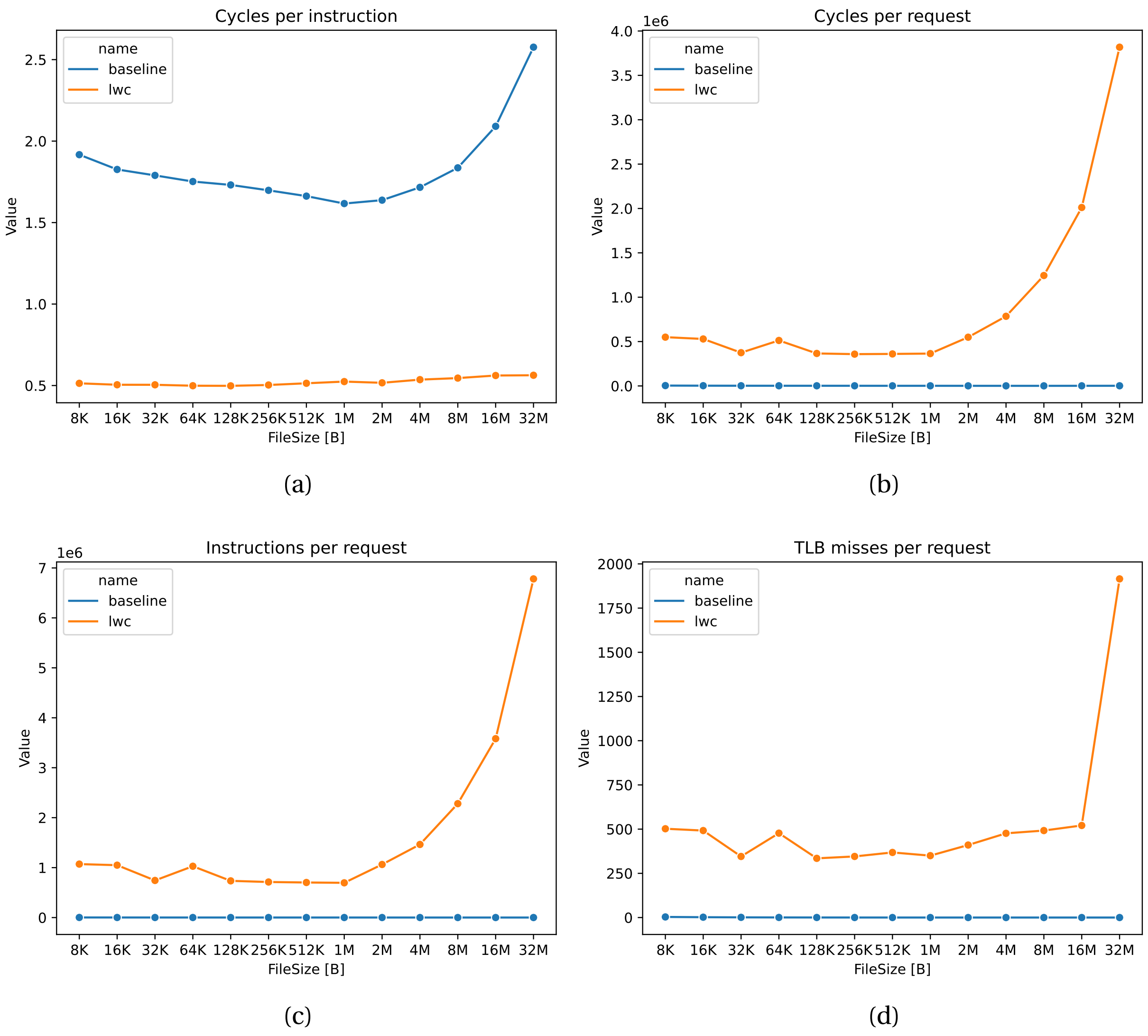


Figure 5.3: Results of our macrobenchmark for the lwC compartmentalization design.

SecureCells	ERIM	lwC
8	42	6068

Table 5.3: Overhead in cycles of a single compartment switch.

This is thanks to SecureCells' range-based TLB, which incurs no misses, and hence no additional cycles. We can also see that the theoretical performance of another architecture, namely CHERI, would be prohibitive in a use case such as our benchmark because the switching cost is much higher than the number of cycles executed by the code run in the compartment at each request. The situation is the same for ERIM and CHERI. lwC has a cost per switch that is two orders of magnitude higher, which means that it dwarfs the cycles required to serve a request. Finally, all except SecureCells begin to have performance impacted due to TLB misses when the data size becomes too large.

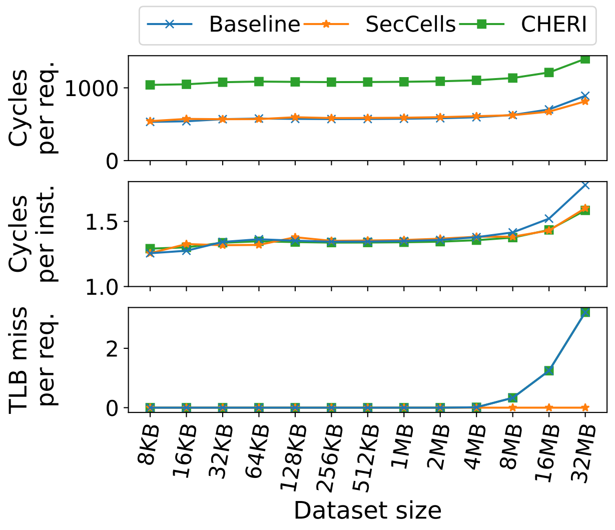


Figure 5.4: Results of our macrobenchmark for SecureCells and CHERI, taken from [3].

Chapter 6

Related Work

Compartmentalization techniques are usually overlaid on top of existing systems to allow for easier adoption, leveraging existing technologies and hardware. However, they sometimes have hardware or software requirements that make them difficult to adopt but enable more flexible designs or stronger threat models. There are many approaches and designs. We distinguish four main categories. Although we present each design in the category their core design uses, they often leverage other aspects, e.g., both operating system and hardware elements.

The operating system-based approach typically leverages OS processes for isolation. Whilst it integrates well with widely used computers, it incurs high overhead at each compartment switch, making it unsuitable for software with frequent switches. Lightweight Contexts [15], which we present in more detail in subsection 2.1.3, is a good example of this. Secure Memory Views [11] allows concurrent threads running in the same process to partially or fully isolate their memory space. It allows these threads to selectively manage the access rights to their memory by other threads by using a user-space programming interface, and a kernel space privilege enforcement mechanism. Hodor [10] introduces an OS abstraction called *protected libraries*. It focuses on enabling the secure sharing of data-plane libraries with the help of Intel PKU. Cross Process Call (XPC) [6] is a hardware-assisted OS primitive for both microkernel and monolithic kernels. It improves the cost of *inter-process communication* (IPC) by enabling zero-copy IPC, without the need to trap into the kernel. FlexOS [14] is a new OS design aiming to provide a modular approach to compartmentalization, where the used technique can be tailored to a specific use case at build time. The user can decide which OS components are compartmentalized, as well as the isolation and protection primitives for each compartment. The goal is to make security-performance trade-offs easier to explore by researchers and developers.

It is also possible to use a language, compiler, and runtime approach to implement compartmentalization. Well-known examples are SFI [22], CPI [13], and CFI [1]. Wedge [4] utilizes OS primitives (tagged memory in combination with threads) to enforce compartments, in combination with a Linux tool that assists the programmer in applying the primitives in applications. It operates in two phases. First, a run-time instrumentation phase logs memory allocations and accesses made by the target software. The second phase is the static analysis of the resulting trace, which looks for memory access patterns. It can give insight into which memory items a procedure, and all its descendants, access, and with what modes of access. The analysis can also return the procedures accessing a given list of data items. Finally, it can also find where a given procedure writes data to. Enclosure [8] is a programming language construct that allows the isolation of external libraries. This is useful in modern languages, where it is very common to use a large quantity of third-party, potentially unchecked, code. It introduces a dynamically scoped programming language construct. Users can define access policies on code invoked within it, at package granularity, on its access rights to data and code of the program. To enforce these permissions, it leverages Intel VT-x and PKU.

Virtualization-based techniques make use of a hypervisor to provide data encapsulation. TrustVisor [17] is a hypervisor providing data integrity and secrecy to selected portions of an application. It can, in addition, provide code integrity. SeCage [16] leverages static and dynamic analysis to define which parts of an application should be compartmentalized. It then uses `VMFUNC` and nested pages to enforce the memory views of the different compartments. Dune [2] is a kernel module using the virtualization hardware of the processor to provide user-level management of privileged hardware features.

We already presented ERIM [21] at subsection 2.1.2. CHERI [23] is another design using hardware to enable compartmentalization. It extends the 64-bit BERI RISC *instruction set architecture* (ISA), compiler, and OS with a capability system that supports the compartmentalization of C-language TCBs. Donky [20] introduces a hardware extension providing memory protection keys, similar to Intel PKU, but with stronger security guarantees.

For their performance evaluation, the designs we mentioned in this chapter make use of microbenchmarks to compare the cost of a compartment switch with other designs ([20] [17] [15]), while others provide this information, but do not compare their result with other designs ([21] [23] [2] [16] [8] [4] [10]). FlexOS [14] does not use any microbenchmarks, though such a benchmark is arguably not useful regarding their design. XPC [6] does not use context switch as a microbenchmark, because it is not a useful metric in its design. All the papers we mentioned make use of macrobenchmarks to measure the performance overhead once the proposed compartmentalization is applied.

Chapter 7

Conclusion

In this project, we have seen that benchmarking is not a trivial aspect of designing new secure systems. Nonetheless, it is an important aspect of publishing to take into account. The scientific process requires reproducible, meaningful, and sound numbers to function. Otherwise, comparisons between systems can lead to wrong conclusions, which could, in extreme cases, lead to discarding ideas that would have improved the status quo. There are many variables to account for when measuring performance, and many obstacles, like hardware requirements. Different use cases will highlight different aspects of a system. Macrobenchmarks are used to get performance numbers representative of what can be expected in real-world usage and if different usage scenarios. Nonetheless, some specific performance indicators are useful to meaningfully compare the same kind of system. Benchmarks designed to measure these indicators are called microbenchmarks. They depend on the type of secure system, for example when benchmarking compartmentalization switching costs between compartments is important. Ideally, there would be a standard defining, for each system category, what needs to be measured, using which metric, and how it has to be measured. Sadly, this does not exist and is arguably impractical to create. Guidelines on how to create benchmarks and present results are a more realistic way of improving performance evaluations of new and existing systems. We faced some of these challenges when benchmarking ERIM [21] and Lightweight Contexts [15]. Hardware requirements for ERIM and software requirements for lwC each lead to issues with benchmarking. Our macrobenchmark had to be portable to each of the systems, and we needed to implement our low-level tool for measuring HPC from inside C code. Additionally, we had to reproduce results from the respective papers, which, whilst doable, was a good example of the reproducibility problem in benchmarking.

Using the microbenchmark and macrobenchark results, we compared the performance results of ERIM, lwC, and SecureCells. Whilst these results clearly show SecureCells to be more performant, benchmark numbers should not be considered in isolation. Indeed, they are not the only important aspect of a secure system. Whilst SecureCells outperforms the other two systems, it is also the one with the most restrictive requirements, essentially requiring custom hardware to run. This makes it difficult to deploy widely. It is important to note, that this is a design choice of this system. Indeed, there are always trade-offs to consider between security, performance, ease of use, and ease of implementation. In this case, the choice was made to neglect the ease of implementation in favor of the other three. ERIM performed better than lwC. It has lower hardware and software requirements than SecureCells, but it suffers from usage limitations, like a limited amount of compartments. In contrast, lwC causes a prohibitive performance overhead in our macrobenchark, but its design allows the compartmentalization of any application.

Our results were reproducible, sound, and complete enough to meaningfully compare the performance of ERIM, lwC and SecureCells. Nonetheless, there are many ways to improve this project. Firstly, we only used one macrobenchmark. Others could be designed to highlight other aspects of these compartmentalization systems. For example, a benchmark focusing more on throughput, similar to the virtual network function pipeline of the SecureCells paper [3]. Our implementation of HPC readings also should be improved, and the identified issues fixed. Future work could also expand on it to make reading HPC easier from inside benchmarks. The issue of benchmarking is complex, and merits careful consideration in every publication. Nonetheless, it is a time-consuming process and fully depends on what is being benchmarked. The most important aspect of a publication remains the system that is presented. Whilst, benchmarking is necessary for the publication to be useful, it cannot exist without the underlying design. To make published performance numbers better follow the principles of completeness, soundness, and reproducibility, we believe that the computer security research field could benefit from well-defined benchmarking guidelines and portable tools for measuring performance.

Bibliography

[1] Martin Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. “Control-Flow Integrity”. In: *ACM Conference on Computer and Communication Security*. 2005.

[2] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. “Dune: Safer User-level Access to Privileged CPU Features”. In: *USENIX Symposium on Operating Systems Design and Implementation*. 2020.

[3] Atri Bhattacharyya, Florian Hofhammer, Yuanlong Li, Siddharth Gupta, Andrés Sanchez, Babak Falsafi, and Mathias Payer. “SecureCells: A Secure Compartmentalized Architecture”. Submitted to IEEE S&P. 2022.

[4] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. “Wedge: Splitting Applications into Reduced-Privilege Compartments”. In: *USENIX Symposium on Networked Systems Design and Implementation*. 2008.

[5] Intel Corporation. *Intel® 64 and IA-32 Architectures Developer’s Manual: Vol. 3B.*

[6] Dong Du, Zhichao Hua, Yubin Xia, Binyu Zang, and Haibo Chen. “XPC: Architectural Support for Secure and Efficient Cross Process Call”. In: *International Symposium on Computer Architecture*. 2019.

[7] Philip J. Fleming and John J. Wallace. “How Not to Lie with Statistics: The Correct Way to Summarize Benchmark Results”. In: *Commun. ACM*. 1986.

[8] Adrien Ghosn, Marios Kogias, Mathias Payer, James R. Larus, and Edouard Bugnion. “Enclosure: Language-Based Restriction of Untrusted Libraries”. In: *ACM Conference on Computer and Communication Security*. 2021.

[9] Siddharth Gupta, Atri Bhattacharyya, Yunho Oh, Abhishek Bhattacharjee, Babak Falsafi, and Mathias Payer. “Rebooting Virtual Memory with Midgard”. In: *International Symposium on Computer Architecture*. 2021.

[10] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, and Michael L. Scott. “Hodor: Intra-Process Isolation for High-Throughput Data Plane Libraries”. In: *Usenix Annual Technical Conference*. 2019.

[11] Terry Ching-Hsiang Hsu, Kevin Hoffman, Patrick Eugster, and Mathias Payer. “Enforcing Least Privilege Memory views for Multithreaded Applications”. In: *ACM Conference on Computer and Communication Security*. 2016.

[12] Erik van der Kouwe, Gernot Heiser, Dessins Andriesse, Herbert Bos, and Cristiano Giuffrida. “SoK: Benchmarking Flaws in Systems Security”. In: *IEEE European Symposium on Security and Privacy*. 2019.

[13] Volodymyr Kuznetsov, László Szekeres, Mathis Payer, George Candea, R. Sekar, and Dawn Song. “Code-Pointer Integrity”. In: *USENIX Symposium on Operating Systems Design and Implementation*. 2014.

[14] Hugo Lefeuvre, Vlad-Andrei Bădoiu, Ştefan Teodorescu, Pierre Olivier, Tiberiu Mosnoi, Răzvan Deaconescu, Felipe Huici, and Costin Raiciu. “FlexOS: Making OS Isolation Flexible”. In: *Proceedings of the Workshop on Hot Topics in Operating Systems*. 2021.

[15] James Litton, Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, Bobby Bhattacharjee, and Peter Druschel. “Light-Weight Contexts: An OS Abstraction for Safety and Performance”. In: *USENIX Symposium on Operating Systems Design and Implementation*. 2016.

[16] Yutao Liu, Tianyu Zhou, Kexin Chen, Haibo Chen, and Yubin Xia. “Thwarting Memory Disclosure with Efficient Hypervisor-enforced Intra-domain Isolation”. In: *ACM Conference on Computer and Communication Security*. 2015.

[17] Jonatahn M. McCune, Ning Qu, and Yanlin Li. “TrustVisor: Efficient TCB Reduction an Attestation”. In: *IEEE Symposium on Security and Privacy*. 2009.

[18] *Memcached*. `https://memcached.org/`.

[19] Soyeon Park, Sangho Lee, Wen Xu, Hyungon Moon, and Taesoo Kim. “libmpk: Software Abstraction for Intel Memory Protection Keys (Intel MPK)”. In: *Usenix Annual Technical Conference*. 2019.

[20] David Schrammel, Samuel Weiser, Stefan Steinegger, Martin Schwarzl, Michael Schwarz, Stefan Mangard, and Daniel Gruss. “Donky: Domain Keys - Efficient In-Process Isolation for RISC-V and x86”. In: *Usenix Security Symposium*. 2020.

[21] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. “ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK)”. In: *Usenix Security Symposium*. 2019.

[22] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. “Efficient Software-Based Fault Isolation”. In: *ACM Symposium on Operating Systems Principles*. 1993.

[23] Robert N.M. Watson, Jonathan Woodruff, Peter G. Neumann, Simon W. Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, Steven J. Murdoch, Robert Norton, Michael Roe, Stacey Son, and Munraj Vadera. “CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization”. In: *IEEE International Symposium on Security and Privacy*. 2015.