FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Lehrstuhl für Informatik 4 · Verteilte Systeme und Betriebssysteme

Luis Gerhorst

Flexible and Low-Overhead System-Call Aggregation using BPF

Masterarbeit im Fach Informatik

23. Dezember 2021

Please cite as:
Luis Gerhorst, "Flexible and Low-Overhead System-Call Aggregation using BPF", Master's Thesis, Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Dept. of Computer Science, December 2021.

Friedrich-Alexander-Universität Erlangen-Nürnberg
Department Informatik
Verteilte Systeme und Betriebssysteme
Martensstr. 1 · 91058 Erlangen · Germany

Flexible and Low-Overhead System-Call Aggregation using BPF

Masterarbeit im Fach Informatik

vorgelegt von

Luis Gerhorst

geb. am 1. Februar 1997
in Nürnberg

angefertigt am

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Department Informatik
Friedrich-Alexander-Universität Erlangen-Nürnberg

Betreuer:	**Benedict Herzog, M.Sc.**
	Stefan Reif, M.Sc.
	Prof. Dr.-Ing. Timo Hönig
Betreuender Hochschullehrer:	**Prof. Dr.-Ing. habil. Wolfgang Schröder-Preikschat**
Beginn der Arbeit:	**1. August 2021**
Abgabe der Arbeit:	**23. Dezember 2021**

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Declaration

I declare that the work is entirely my own and was produced with no assistance from third parties. I certify that the work has not been submitted in the same or any similar form for assessment to any other examining body and all references, direct and indirect, are indicated as such and have been cited accordingly.

(Luis Gerhorst)
Erlangen, 23. Dezember 2021

ABSTRACT

General-purpose operating systems (OSes) rely on hardware-based isolation to confine user processes to their own virtual address space. By doing so, they protect the system from malicious actors, maintain privacy, and achieve fault tolerance. However, user applications must still be able to communicate with each other and access the hardware for input/output (I/O). For this, OSes rely on system calls, which allow the user applications to invoke kernel code in a controlled manner. Every system call includes a switch from the unprivileged user mode to the privileged kernel mode. Although these processor-mode and address-space switches are the essential isolation mechanisms that guarantee the system's integrity, they induce direct and indirect performance costs as they invalidate parts of the processor state. In recent years, high-performance network and storage hardware have made the user/kernel transition overhead the bottleneck for input/output-heavy applications. To make matters worse, security vulnerabilities in modern processors (e.g., Meltdown) have prompted kernel mitigations that further increase the transition overheads. To decouple system calls from user/kernel transitions, I propose AnyCall, which uses an in-kernel bytecode compiler to execute safety-checked user code in kernel mode. This allows for multiple fast system calls interleaved with error checking and processing logic, using only a single user/kernel transition. I implement AnyCall using the Linux kernel's Berkeley Packet Filter (BPF) subsystem, extending it to support system-call invocation and user memory access. Being already supported in the kernel for flexible event handling and debugging, reusing BPF to implement system-call aggregation demonstrates that software-isolated processes are practical for modern general-purpose OSes. To demonstrate that porting real-world user applications to AnyCall is both practical and straight-forward, I port two real-world tools and document the code changes required. Finally, I evaluate AnyCall's performance on systems with OS-level mitigations against transient execution vulnerabilities active or inactive, including for example Kernel Page Table Isolation (KPTI) against Meltdown. On systems where KPTI is inactive, I demonstrate speedups of up to 10 % in compute-bound real-world applications. On the other hand, when KPTI is active, my evaluation demonstrates that system-call bursts are up to 98 % faster using AnyCall, and that real-world applications are sped up by 32 % to 40 %.

KURZFASSUNG

Betriebssysteme nutzen hardwarebasierte Isolationsmechanismen, um Benutzerprozesse auf ihren eigenen virtuellen Adressraum zu beschränken. Auf diese Weise schützen sie das System vor böswilligen Akteuren, wahren die Privatsphäre und verbessern die Fehlertoleranz. Benutzeranwendungen müssen jedoch weiterhin in der Lage sein, miteinander zu kommunizieren sowie auf die Hardware für Ein- und Ausgaben zuzugreifen. Um das zu ermöglichen, bieten Betriebssysteme Systemaufrufe an. Diese ermöglichen es den Benutzeranwendungen Kernelcode kontrolliert aufzurufen. Jeder Systemaufruf erzwingt jedoch einen Kontextwechsel vom unprivilegierten Benutzermodus in den privilegierten Kernelmodus. Obwohl diese Wechsel des Ausführungsmodi sowie des Adressraums zu den grundlegenden Isolationsmechanismen zur Sicherstellung der Systemintegrität gehören, verursachen sie direkte und indirekte Leistungseinbußen, da sie Teile des Prozessorzustandes ungültig machen. In den letzten Jahren hat hochleistungsfähige Netzwerk- und Speicherhardware diese Kontextwechsel zum Engpass für ein- und ausgabenintensive Anwendungen gemacht. Hinzu kommt, dass Sicherheitslücken in modernen Prozessoren (z. B. Meltdown) Gegenmaßnahmen im Betriebssystem-Kernel nötig machen, die die Kosten von Kontextwechseln weiter erhöhen. AnyCall löst dieses Problem, indem es Systemaufrufe von Kontextwechseln entkoppelt. Dazu verwendet AnyCall einen Kernel-internen Bytecode-Compiler, der die Ausführung von sicherheitsgeprüftem Anwendungscode im Kernelmodus möglich macht. Dies ermöglicht schnelle, mit Fehler- und Verarbeitungslogik verschachtelte Systemaufrufe-Stöße, mit nur einem einzigen Kontextwechsel. AnyCall's Implementierung erweitert das Berkeley Packet Filter (BPF) Subsystem des Linux Kernels um Schnittstellen zur Auslösung von Systemaufrufen und für den Zugriff auf Benutzerspeicher. Da BPF bereits im Kernel für flexible Ereignisbehandlung und Debugging benötigt wird, zeigt dessen Wiederverwendung zur Implementierung von Systemaufruf-Aggregationen, dass durch Software isolierte Prozesse für moderne Allzweck-Betriebssysteme praktikabel sind. Um zu demonstrieren, dass die Portierung von realen Benutzeranwendungen auf AnyCall unkompliziert machbar ist, portiere ich zwei reale Anwendungen und dokumentiere die erforderlichen Codeänderungen. Abschließend bewerte ich die Leistung von AnyCall auf Systemen mit aktiven oder inaktiven Abhilfemaßnahmen gegen Prozessorsicherheitslücken, wie zum Beispiel Kernel Page Table Isolation (KPTI) gegen Meltdown. Auf Systemen ohne KPTI beschleunigt AnyCall rechengebundene Anwendungen um bis zu 10 %. Wenn KPTI hingegen aktiv ist, zeigt meine Auswertung, dass AnyCall Systemaufruf-Stöße um bis zu 98 % beschleunigt, sowie, dass reale Anwendungen um 32 % bis 40 % beschleunigt werden.

CONTENTS

1 INTRODUCTION

General-purpose computing systems heavily rely on isolation of user applications to achieve fault-tolerance and guarantee security. First, isolation is required to keep systems usable even when individual applications experience faults. If desired, the applications can then be restarted with little overhead to return to normal operation. Further, fault isolation is not only required between user applications and the kernel, but also among user applications, as some provide fundamental system services (e.g., `systemd`). If these fail, the overhead of restarting them is equivalent to a system reboot. Second, isolation is required for privacy and to protect the user from malicious actors. The latter is required if a malicious party gains access to, or is able to hijack, certain user processes. In this case, isolation confines them to the compromised user account. It prevents them from spying on other users and from locking other users out of the computing system (i.e., denial of service). But even if no malicious party ever gains access to the computing system, isolation is beneficial for privacy on single-user systems (e.g., mobile phones). For example, it prevents social media applications (which often have a commercial incentive to collect data for personalized advertisements) from harvesting the user's data stored in other applications (e.g., the password manager, photo-, or banking applications).

To summarize, computing systems rely on isolation as the most fundamental security and safety mechanism that enables protection from malicious activities, maintains privacy, and confines faulty programs [Aik+06; Ge+19]. The necessary counterpart to isolation is a well-defined communication interface that provides kernel-level functionality safely to user programs. General-purpose operating systems typically implemented this using *system calls*.

System calls enable controlled hardware access and interprocess communication (IPC). Hardware accesses are, for example, required to display application data on the screen, to record audio from the microphone, or to play it on the speakers. The system has to control such hardware accesses to protect the user's privacy. Based on this, some systems, for example, implement confirmation dialogues before giving an application access to the microphone or camera. Similarly, IPC gives applications controlled access to services provided by other applications. For example, a browser can retrieve login credentials from a password manager, but only if the user approves the transmission in the password manager. To summarize, without input/output (I/O) and IPC through system calls, application's computations would be of no use, as they could never receive inputs from, or present results to, the user.

1.1 Motivation

The costs of system calls, in particular their execution-time overheads, have been a well-known performance-critical system property for decades [EH13]. An example for a system-call–intensive application is the Unix `find` tool that traverses directories and filters files by configurable criteria. Therefore, numerous system calls are executed to read the file-system tree, but the amount of computation in user space, in-between system calls, is often negligible. This is also the case with other fundamental Unix tools. Experiments on a 1.8 GHz desktop computer show that $7 \cdot 10^4$ to $7.3 \cdot 10^5$ system calls are performed per second[1] when unpacking archives, listing files, and estimating disk usage using the GNU coreutils.

Besides fundamental Unix applications, multiple works have demonstrated that server applications suffer from severe system-call overheads. For example, the performance of database servers like Memcached can be significantly improved by reducing system-call overheads [KS15]. Further, file servers have been shown to benefit from a dedicated `sendfile()` system call [Zad+05], which reduces invocation overheads by aggregating multiple system calls into one.

1.2 Problem Statement

System-call overheads are increasing relatively to application performance. This is caused both by an increasing system-call rate, but also by increasing per-call overheads. System-call rate is increasing in particular due to low-latency disks, large memories, and high-performance networking hardware. These devices reduce the I/O latency significantly or avoid it altogether. Thereby, system-call latencies are becoming more significant to the overall performance of the computing system.

Not only the rate at which system calls are performed is increasing, but also the overhead of each individual system call. When performing a system call, the processor must switch from user to kernel mode, execute the requested privileged operations, and finally switch back to user mode to resume the application. Each system call therefore involves two user/kernel transitions. Besides the traditional *direct* costs, these transactions also inflict *indirect* costs as the processor state must be (partially) invalidated [SS10]. The importance of such indirect costs is expected to grow since caches and hardware buffers are becoming increasingly performance-critical [Rup20]. With the discovery of Meltdown [Lip+18], Spectre [Koc+19], and further processor-level timing side-channels, the system-calls costs have increased even further, often causing a significant overhead on execution time [Pro+18; Ren+19] and energy demand [Her+21]. The required flushes of processor-internal buffers enforce isolation, but cause significant overheads, particularly for applications that execute system calls at a high rate.

Multiple works have demonstrated that hardware-based isolation techniques (i.e., virtual memory, processor protection rings) have significant overheads for applications. This includes experiments on Singularity operating system [HL07], which primarily relies on software techniques for isolation, thereby allowing all applications to share an address space and execute in ring 0. Here, enabling hardware-based isolation in addition to software-based isolation increases the isolation overhead from 4.9 % to 44.4 % for IPC-heavy applications. Recently, Li et al.[Li+21] have compared the overheads of hardware- and language-based (i.e., software-based) isolation for IPC. They found that even in an optimal scenario for hardware-based isolation, where the address space switch takes zero cycles, language-based IPC is still faster. By solely relying on hardware-based isolation, operating systems (OSes) therefore hurt application performance. This in turn complicates applications as they have to rely on alternative optimizations to achieve acceptable performance.

[1]The numbers were obtained by running Debian 10's `tar xf`, `find`, and `du -s` on the Linux 5.0 source tree. See Chapter 5 for details on the Intel i5-6260U evaluation setup.

1.3 Approach

Considering that applications with high system-call rate suffer most from hardware-based isolation, I propose AnyCall, a system where only a single user/kernel transition is required for an arbitrary number of system calls. To reduce the number of transitions between user and kernel space, the control flow migrates to kernel space, calls arbitrary system calls interleaved with application-specific logic, and eventually returns to user space. Motivated by the evaluations of software-based isolation from [HL07] and [Li+21], isolation of the user application logic is enforced using an in-kernel bytecode executor and static bytecode analysis. To execute the aggregation programs, AnyCall's implementation reuses the existing Berkeley Packet Filter (BPF) bytecode virtual machine (VM) included in recent versions of the Linux kernel.

1.4 Publication

This thesis contains research results from the following peer-reviewed workshop paper:

[Ger+21] Luis Gerhorst, Benedict Herzog, Stefan Reif, Wolfgang Schröder-Preikschat, and Timo Hönig. "AnyCall: Fast and Flexible System-Call Aggregation." In: *Proceedings of the 11th Workshop on Programming Languages and Operating Systems (PLOS'21)*. ACM, 2021, pp. 1–8. DOI: 10.1145/3477113.3487267

The contributions of [Ger+21] are three-fold. First, it presents the approach of AnyCall, which decouples the number of user/kernel transitions from the number of system calls while maintaining isolation using an in-kernel bytecode executor (i.e., BPF). Second, it presents AnyCall's implementation, extending the Linux kernel to support system calls from within BPF programs. For this, AnyCall enables verifiably safe accesses to user memory from inside BPF programs. This is used to construct system-call arguments and the access system-call results. Finally, we evaluate AnyCall by comparing the overheads of BPF-based and traditional system calls.

This thesis extends [Ger+21] with regard to various aspects. First, it covers the contents of the paper in greater detail, for example extending the discussions of BPF, user memory access, and the evaluation results. Second, it covers related works using software-based isolation (e.g., SPIN, Singularity, and RedLeaf) and compares the microkernel-approach against AnyCall's approach. Third, the design and implementation are extended, discussing signal handling, AnyCall's trust model, and interfaces for directory iteration. Finally, I have improved the evaluation, taking into account feedback received at the peer-reviewed workshop. I run each benchmark on a second hardware platform, evaluate I/O-bound workloads, and measure the performance of a directory iteration tool that uses AnyCall.

1.5 Overview

This thesis is structured into eight chapters. The following Chapter 2 introduces background knowledge about system calls, their overheads, and the BPF subsystem of the Linux kernel. Chapter 3 deduces the basic design of AnyCall, which lowers the system-call overheads for applications by aggregating multiple system calls into one `anycall()`. The design chapter discusses AnyCall's execution and programming model, signal handling, its trust model, and the in-kernel bytecode executor (i.e., BPF) used for AnyCall's implementation. Chapter 4 presents AnyCall's BPF-based implementation. In addition to showing how user applications invoke AnyCall, and how AnyCall invokes system calls, it discusses how AnyCall can safely access user memory. This allows it to construct system-call arguments and retrieve results. It further explores how AnyCall-specific kernel interfaces can overcome BPF's limitations and improve application performance. I demonstrate this by creating a BPF-based interface for (recursive) directory iteration. Following the implementation, Chapter 5 evaluates AnyCall, comparing it against traditional system calls in various micro- and real-world-benchmarks. Each experiment is executed on multiple hardware platforms, with or without mitigations against transient execution vulnerabilities (e.g., Meltdown) active. Chapter 6 presents related works. It discusses alternative approaches to software-based isolation (i.e., avoiding user/kernel transitions and microkernels) and compares AnyCall against multiple systems that have previously used software-based isolation to enhance dependability and performance. Chapter 7 summarizes the opportunities for future work and Chapter 8 concludes this thesis.

2 BACKGROUND

This chapter discusses system calls as well as recent hardware and software developments related to their overheads. Subsequently, it gives background on the Linux kernel's BPF technology.

2.1 System Calls

System calls enable isolated processes to communicate and perform I/O. This section discusses how general-purpose operating systems implement system calls for hardware-isolated processes.

2.1.1 Motivation

As outlined in the introduction, general-purpose OSes must isolate processes from each other to maintain fault isolation, protect them from malicious actors, and to ensure privacy. For this, OS kernels program the hardware memory management unit (MMU) to create separate virtual address spaces for all user processes. This prevents them from accessing physical memory owned by other processes or the kernel. If an unprivileged process attempts to do so anyways, a page fault is triggered, which redirects the control flow to the kernel for recovery. Programming the MMU and accessing similar hardware functions, are privileged operations the central processing unit (CPU) only permits if it runs in the privileged kernel mode. Therefore, to perform I/O (i.e., accessing the hardware) and to communicate with other processes (e.g., by reading/writing to their memory), applications must invoke the kernel. This is possible using *system calls*, through which the kernel gives the applications controlled access to privileged operations.

2.1.2 Implementation

The generic solution to implement system calls for hardware-isolated processes involves switching to the privileged execution mode, running the requested operation, and finally switching back to user context. Switching the processor mode while simultaneously transferring the control flow to the kernel is achieved using dedicated software trap instructions (e.g., `int 0x80` or `sysenter` on x86). Following the software trap, the kernel executes the desired operations as requested by the system-call arguments, which the user process places in registers or on the stack prior to the software trap. The application passes the system call to be executed to the kernel using registers, and encodes it using a numeric identifier. After performing the desired privileged operations, the kernel returns control to the application. For this, the kernel places the return value in a processor register from where the application can retrieve it after the mode switch. In summary, this implementation allows for arbitrary system calls by synchronously executing kernel code on the application's CPU core.

2.1.3 Discussion

This section discusses strengths and drawbacks of the system-call implementation described in the previous section. In particular, it is (a) implementation-agnostic, (b) allows for isolation, (c) is easy to use, (d) hides the implementation, and (e) allows for a stable application binary interface (ABI). However, it is also (f) inflexible and (g) slow.

Implementation-agnostic The implementation outlined in the previous section is generic in that it works for every system call regardless of the operations it performs. Inside the system call the kernel can access the hardware for I/O and read/write arbitrary user and kernel memory while remaining in full control. It can perform complex checks before, while, or after performing the requested system call. If it detects invalid parameters or forbidden operations, it can return an error code to the process at any time.

Maintain isolation Operating systems require isolation but also communication. In comparison to other solutions, system calls still retain a high level of isolation among the system components. Not considering alternatives such as shared memory, processes can only communicate with each other indirectly through the OS kernel, which decouples them from each other. As discussed in the previous paragraph, the kernel can place arbitrary limits on the allowed privileged operations and thereby protect itself and concurrently running applications. Also, the kernel has full control over the hardware state in which it resumes user processes. This includes basic security measures such as clearing values from processor registers, but also flushing processor caches to prevent information leaks through side-channels.

Subroutine-like programmming model Using system calls is straight-forward because they behave like subroutines. These are a well-understood basic programming tool every developer is familiar with.

Information-hiding System calls are abstract and interface with the application only through arguments and return values. This interface is narrow and allows for backwards-compatibility if this is a desired system property. The system-call implementations, however, are hidden and can be continuously optimized for non-functional properties like performance and energy usage without rewriting or recompilation of user applications.

Inflexible The strong isolation between kernel and applications also has drawbacks. Kernel extensions, for example, offer greater flexibility as they can directly access kernel-internal interfaces and data structures, thereby being able to offer functional and non-functional features that OS services in user space can not provide. Although system calls are composeable, doing so involves overheads. This manifests itself in a large, increasing number of system calls offered by modern operating systems [Lin21b].

Slow The generic system-call implementation involves two user/kernel transitions for every interaction, these include switching the processor mode but also flushing hardware registers and caches. This has been a well-known performance problem for decades [EH13], but no alternative implementation is as generic and straight-forward to use as the synchronous implementation involving two user/kernel transitions. For example, Xen's multicalls, which allow user space to call a batch of system calls using a single user/kernel transition, are unuseable if there are data or control-flow dependencies between the system calls. vDSO can avoid user/kernel transitions by mapping kernel data into user memory, but is limited to read-only data for security reasons. Any system call that has to write data to kernel memory can not use it. Finally, blocking system calls promote extensive multi-threading in applications thereby increasing the context-switching overhead in the computing system. Asynchronous input/output (AIO) was created to solve this but has a programming model many developers are not familiar with [Atl+16].

As processors achieve performance improvements through larger caches and architectural improvements, the relative overhead of user/kernel transitions for applications increases. This is amplified by high-performance I/O hardware and large memories, which cause even more frequent user/kernel transitions. The following Sections 2.2 and 2.3 provide detailed background on these trends.

2.2 System-Call Rate

Due to low-latency disks, large main memories, and high-performance networking hardware, the system-call frequency in modern computing systems is increasing. This amplifies the already-large user/kernel transition overheads for applications.

2.2.1 Low-Latency Disk

Traditional hard drives with high access latencies are now superseded by low-latency media such as NVME-SSDs and Optane storage. In particular the speed of non-volatile memory (NVM) is approaching the speed of dynamic random-access memory (DRAM) [ERT19], also allowing for very fast random-access I/O. As the device latency decreases, the speed of user/kernel transitions becomes more significant for the overall performance of the computing system [ERT19; Zho+21]. For example, Honda, Lettieri, Eggert, and Santry [Hon+18] have demonstrated that with non-volatile main memory (NVMM) the end-to-end latency in transactional storage systems (e.g., blob stores, key-value stores, and databases) is no longer dominated by the disk latency but instead by the software stack. Replacing a solid-state drive (SSD) with NVMM reduces the end-to-end transaction latency by two orders of magnitude from 1320 µs to 27.17 µs. If the disk access is omitted entirely, a transaction takes 23.32 µs, therefore the NVMM access only adds 3.85 µs to the transaction latency. In summary, systems using low-latency media are no longer bottlenecked by the device latency, but instead by the software stack and the user/kernel transitions the system performs.

2.2.2 Large Memories

The main memory of computing systems offers low-latency random access at the cost of volatility in case of a power failure. On most systems, the main memory is not used entirely by applications at all times. To make use of unused memory, OSes employ a page cache which holds file and directory contents recently read from disk. Accessing these files thereby happens at memory speed instead of disk speed, triggering very frequent user/kernel transitions. In summary, large memories cause the transitions to become even more critical to the computing system's performance.

Even though memory accesses are much faster than disk accesses, there is still potential for further optimizations. When cached, files still live in kernel memory and have to be copied or mapped into user space before applications can access them [Gru+19]. This in either case incurs a loss of cache-locality, slowing down the access. Software-based isolation can resolve this by giving applications direct access to kernel memory while still preventing them from corrupting kernel data.

2.2.3 High-Performance Networking

Similarly to NVM disks, recent networking hardware is bottlenecked by the software stack [Rum+11]. While 10 Gbit/s Ethernet was already widespread in 2015 [Kau+16], 400 Gbit/s Ethernet is on the horizon as of 2019 [ERT19]. However, a 40 Gbit/s network interface controller (NIC) can already receive a packet of the size of a cache line every 12 ns while the last level cache (LLC) latency of an Intel Sandy Bridge processor is 15 ns [MHS14; Kau+16]. A single LLC access during packet processing is therefore not tolerable if the CPU wants to keep up with incoming packets unless other techniques to hide the latency are used (e.g., concurrent processing, instruction-level parallelism). In Memcached, for example, this is problematic as even with kernel-bypass, processing in the network stack and the application consumes about half of its server processing time [Pet+14].

Kernel-bypass gives network-heavy applications direct access to the networking hardware [Bel+14; Pet+14]. Kernel-bypass can avoid user/kernel transitions, but has the severe downside of limiting isolation among processes. For example, the sending of arbitrary internet protocol (IP) packets can not be prevented if the application has direct access to the NIC.

To avoid the loss of isolation incurred by kernel-bypass, alternative approaches have to apply sophisticated but *problem-specific* optimizations to the operating system's Transmission Control Protocol (TCP) stack, demonstrating the need for a solution which reduces user/kernel transition overheads. TCP Acceleration as an OS Service (TAS) [Kau+19], for example, separates the TCP slow path from the fast path and execute the latter on dedicated cores. Instead of using system calls to invoke privileged operations, they execute asynchronously on dedicated cores. Privileged and unprivileged processes communicate using shared queues to avoid mutual cache pollution.

In summary, to not bottleneck high-performance networking hardware, the software stack must avoid excessive transitions between user and kernel context. Existing solutions such as kernel-bypass or TAS either compromise on isolation or are specific to the TCP stack.

2.3 User/Kernel-Transition Overheads

The hardware-induced overheads of user/kernel transitions are currently increasing due to multiple trends in the architecture of modern computing systems. This section in particular focuses on increasing hardware buffers and cache sizes as well as mitigations against transient-execution attacks (e.g., Meltdown and Spectre).

The overhead of user/kernel transitions includes a direct and an indirect overhead. The direct overhead is the time for the transition itself, which includes switching the processor mode but also other software operations like switching of the virtual address space or the invocation of second-level interrupt handlers [Her+18]. The indirect overhead arises because many of the performed operations also slow down the code executing after the transition is complete. For example, a software interrupt executes code that is usually not located close to the invoking function, thereby triggering cache inlocality and stalling the processor pipeline. In addition to this, the change in privilege level invalidates hardware caches that contain privileged information as the kernel must protect these from access by unprivileged processes. Applications increasingly rely on caching to achieve acceptable performance, the indirect overheads of user/kernel transitions increase.

This section focuses on two major contributors for increasing direct and indirect user/kernel transition overheads: the continued reliance on hardware buffers/caches to achieve processor speedups and the ongoing discovery of transient execution attacks requiring OS-level mitigations.

2.3.1 Hardware Buffers and Caches

Hardware buffers and caches are increasing in size. Although advantageous to application performance in general, relying on them increases the relative user/kernel transition overheads for applications.

In particular, caches and buffers contribute to the direct switching overhead. The invalidation of caches[2] is yet another state change to be performed during the processor mode switch. Also, the more registers a CPU has, the more registers must be saved and restored when switching to or from a privileged execution context. Also, transferring the control flow using software interrupts disrupts processor pipelines.

However, more importantly, the more applications rely on hardware caches for performance, the greater the indirect user/kernel transition overhead for them. If caches are flushed during the transition, the required repopulation slows subsequent code execution. Due to the Spectre vulnerabilities, the number of branch misses scales with the system-call rate, even on recent processors [Hil+19]. This is also confirmed by the AnyCall microbenchmarks in Section 5.2 and Section 5.3. Also, with Kernel Page Table Isolation (KPTI) the Translation Lookaside Buffer (TLB) is invalidated on user/kernel transitions thereby triggering expensive page-table walks [ATW20].

2.3.2 Transient-Execution Attacks

Recently, the Meltdown [Lip+18] and Spectre [Koc+19] hardware vulnerabilities have made it necessary to develop mitigations in hardware, firmware, and software. As these vulnerabilities can circumvent the memory isolation between processes, mitigations at OS level have been developed. Although these mitigations are effective in fully or at least partly preventing the exploitation of these vulnerabilities, they come with potentially significant execution time and energy demand overheads [Her+21; AEA19; Pro+18]. Especially Linux' mitigations against Meltdown and attacks bypassing kernel address space layout randomization (KASLR) [HWH13], that is KPTI, introduce significant overheads for user/kernel transitions. The most important reason for this are additional TLB flushes before switching to user space, which can be reduced by the use of the Process Context Identifiers (PCIDs), but not fully avoided. Furthermore, PCIDs require specific CPU features and at least Linux kernel version v4.14. The mitigations' overhead for different variants of Spectre attacks apply more selectively depending on the workload but can nevertheless be significant [Her+21].

New transient execution attacks are discovered continuously, even in recent processors [MF21; Adv21]. Therefore, processors currently not considered vulnerable may require mitigations in the future, increasing the user/kernel transition overhead for applications running on them.

In summary, transient execution vulnerabilites as well as large hardware buffer and cache sizes increase the performance penalty imposed by user/kernel transitions. To avoid this overhead while maintaining isolation, systems can use software-based isolation instead of hardware-based isolation. In particular, software-based isolation avoids address spaces and processor mode switches, but still maintains fault isolation. Software-based isolation is available in Linux through BPF, which allows applications to execute small bytecode programs in the kernel's execution context under software-based isolation. The following Section 2.4 covers BPF in detail.

[2]To avoid having to invalidate caches on a mode switch, they can also be duplicated (i.e., split into separate caches, one for privileged and one for unprivileged processes) or tagged (e.g., using Process Context Identifiers). However, both approaches also impose microarchitectural overheads.

2.4 Berkeley Packet Filter (BPF)

BPF allows applications to inject small code fragments as event handlers into the Linux kernel. It thereby enables detailed but flexible configuration with relatively little overhead.

Initially, BPF was developed to filter network packets directly in the kernel without having to perform an expensive switch to user context [MJ93]. This is achieved by allowing user processes to submit small bytecode programs to the kernel for execution inside a VM (in kernel context). The instruction set of the original BPF is very limited (e.g., no loops) and intentionally not Turing complete. To improve the efficiency and expressiveness of BPF, an extended version with a redesigned instruction set was introduced in the 3.18 Linux kernel [Cor14; Cil21a]. The new instruction set is optimized for C interoperability and efficient compilation to native machine code (usually allowing a one-to-one mapping of BPF instructions to machine instructions). Also, recent versions of the kernel allow for safely bounded loops.

In this thesis, the term **BPF** refers to the current (extended) Berkeley Packet Filter implementation [Cor14; Cil21a] as this is common in the technical literature and Linux kernel documentation [Sta21; Zho+21; Lin21a]. In academic literature and older works, this version is sometimes also abbreviated eBPF [Mia+18; TDS21]. If it is required to refer to the original implementation [MJ93], this thesis does so explicitly using the term classic Berkeley Packet Filter (cBPF).

Section 2.4.1 first give a motivational example to illustrate why BPF was introduced. Section 2.4.2 then briefly discusses the BPF bytecode instruction set. Figure 2.1 displays the architecture of the Linux kernel's BPF implementation and is referenced throughout the remaining sections. Section 2.4.3 shows how BPF programs are loaded and invoked in the kernel, thereby explaining the steps ① to ⑤ of Figure 2.1. Section 4.4 section gives background on the kernel interfaces (⑥), and Section 2.4.6 presents the IPC capabilities available to BPF (⑦). Finally, Section 2.4.7 discussing how BPF affects security.

2.4.1 In-Kernel–VM Motivation

In today's computing systems, some kernel events happen at such a high rate that it is not possible to switch to a specific user process every time they occur. To still allow the processing of such events in a user-defined manner (e.g., for tracing and configuration), BPF was developed [MJ93].

For example, system administrators frequently have to monitor and debug the network packets sent and received by a system. On UNIX, this is possible using the `tcpdump` tool which prints network traffic to standard output. To avoid generating excessive amounts of data, `tcpdump` allows users to filter the output using a boolean expression supplied on the command line. However, copying all received packets to user space only to then discard most of them according to the applied criteria would not only be wasteful, but even impossible on systems receiving large amounts of data. Instead, it is desirable to apply the filter as early as possible, for example, directly in interrupt context when the OS is notified about the packet. However, executing user machine code in interrupt context is highly unsafe as it could corrupt privileged data structures or block the entire system with an endless loop. BPF solves this by not executing user-supplied code directly, but instead retrieving only VM bytecode from user programs, which the kernel then checks to ensure memory safety and a bounded execution time. The kernel achieves this by statically analyzing the bytecode. It checks every possible path of the control-flow graph (CFG) and ensures no unsafe memory access can occur. To execute the bytecode, the in-kernel VM either interprets it or, in recent implementations, uses just-in-time (JIT) compilation for near-native performance.

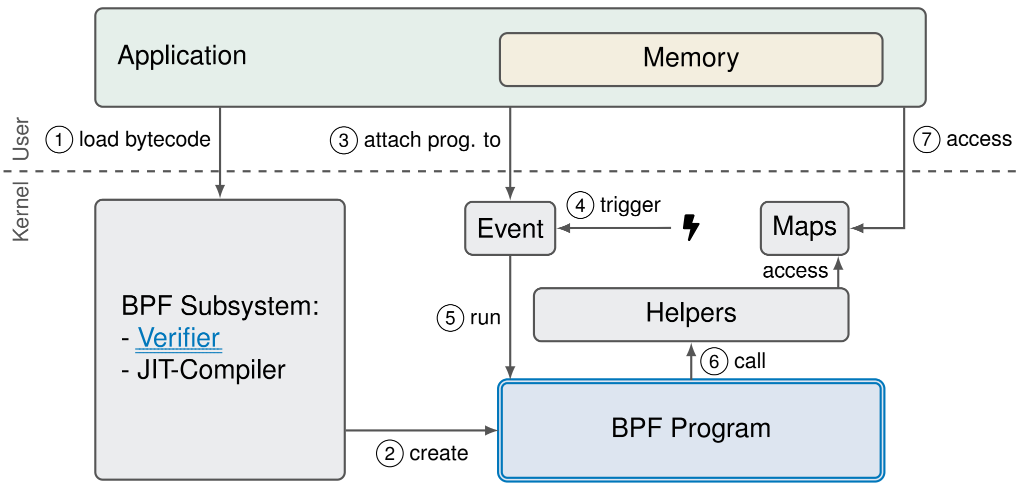

Figure 2.1 – Architecture of the Linux kernel's BPF implementation. BPF programs are only invoked asynchronously when the event to which they are attached is triggered, for example, on interrupts (⚡).

In addition to packet filtering and tracing [MJ93; Mia+18], BPF today has numerous applications. These include the tracing of arbitrary kernel events [IO 21], `seccomp` access-control policies [Dre12], file system sandboxing [BR18], caching [Ghi+21], and paravirtualization [AW18].

2.4.2 Instruction Set Design

This section presents the BPF instruction set thereby illustrating the expressiveness of BPF at a fundamental level. BPF's instruction set by itself is generic (i.e., Turing complete) and does not limit the expressiveness of BPF programs, limitations arise instead from the static analysis that is performed before the program is invoked. This allows BPF to become more expressive as the static analysis improves, without changing the instruction set.

BPF currently offers eleven 64-bit registers and 87 instructions. The bytecode uses fixed-size 64-bit encoding for the instructions, which map directly to machine instructions on the common architectures (e.g., AMD64, ARM) allowing for efficient JIT compilation. In addition to the registers, programs can also store arrays and variables to the BPF stack (512 Byte in size for Linux). Conditional jumps exists and backwards jumps are possible if they are not rejected by the verifier's implementation. No BPF instruction allows the direct modification of the program counter [Cil21a].

The VM is a register-based VM as these can be implemented more efficiently than stack-based VMs [MJ93]. Zandberg and Baccelli [ZB20] have shown that the BPF instruction set allows for very memory-efficient programs. However, small program text size is sacrificed to enable fast decoding (hence fixed-width 64-bit instruction encoding) and easier static analysis (instructions have simple semantics, no compound instructions as in complex instruction set computer exist).

2.4.3 Program Loading and Invocation

To invoke BPF programs quickly on performance-critical paths, the kernel separates program-loading from the invocation. This section presents both operations in detail and thereby covers the operations ① through ⑤ in Figure 2.1.

Initially, the user space process passes the BPF bytecode to the kernel (①). The BPF subsystem then first analyzes this bytecode to ensure memory safety and a bounded execution time. Both will be discussed in greater detail in Section 2.4.4. Second, the BPF program is compiled from the generic bytecode to native machine code. If loading is successful, the application receives a descriptor which it can use to reference the newly created program (②). It can use this descriptor to attach the program to events (③), which will subsequently invoke the program (⑤), whenever the events occur (④). In the current version of the Linux kernel, the available events, for example include, the arrival of network packets and tracepoints.

Compilation of the bytecode happens exclusively at load-time, therefore running the programs is very fast. User processes are only affected by the BPF program indirectly as the execution is always event-driven [Cil21a]. Either they are affected by the policy implemented by the program, for example the rejection of a filtered network packet, or they retrieve results from BPF invocations through dedicated safe IPC mechanisms called maps (see Section 2.4.6). BPF programs can receive arguments from the kernel when they are invoked and return a result to the kernel. The kernel can also give them direct, safe access to memory areas of a fixed size (e.g., network packet headers). Further, interaction happens through helpers which are kernel subroutines BPF programs can invoke. Helpers will be discussed in detail in Section 2.4.5. Each program has a type that is fixed at load-time, this determines the possible attachment points of the program and also the helpers it is allowed to invoke. In addition to the direct invocation by an event, BPF programs can also invoke other BPF programs through tail calls (resembling `execve()` in user processes). This however can only happen a limited number of times (currently 32 [Cil21b]) to prevent an unbounded execution time. In summary, BPF programs run asynchronously to user processes in various kernel contexts. Invoking them synchronously in user-thread context is not supported.

2.4.4 Bytecode Verification

Verification protects the kernel from malicious and buggy programs by statically analyzing the loaded bytecode for memory safety and bounded runtime [HJ+18]. Programs not passing this step are rejected. However, correct programs may still be rejected due to limits in the verifier. The expressiveness of BPF is therefore not limited by the bytecode format, but instead the capabilities of the verifier which can be improved continuously without making recompilation of BPF programs necessary. Verification in this context does not mean that passing BPF programs are correct in that they behave and provide the results as expected by their developer or user. Instead, verification means that programs can never block or crash the kernel.

Multiple measures are required to ensure a bounded execution time. The verifier imposes a limit on the total number of instructions in a program. It also builds and analyzes the CFG to ensure a static upper limit on the number of iterations of each loop. Tail calls also do not allow for endless loops as each call chain has a constant upper limit (currently 32 tail calls). This is enforced using a check on each tail call.

BPF programs can only access allocated memory locations (i.e., their stack and fixed-size allocations from helper functions) safely. Similarly to the Java Virtual Machine (JVM), each accessed memory location has a type that is tracked by the verifier. However, in addition to the general data types (e.g., integers or pointers), the verifier also tracks the range of possible values as it traverses each possible path through the CFG. Tracking the range of values a variable can contain enables safe pointer arithmetic as it can be verified that allocated areas are only dereferenced at valid offsets. The verifier also incorporates the conditions of jump instructions (e.g., from runtime error checks) into the analysis. These enable the verifier to limit the range of possible values in certain program branches and therefore allow accesses that would otherwise be rejected. For example, a BPF program is permitted to access an array living on its stack at a user-defined offset, as long as it includes a condition that ensures the input value is smaller than the array's size.

2.4.5 Helper Subroutines

Helpers are safe kernel functions callable by BPF programs (⑥ in Figure 2.1). They can receive up to six arguments in BPF registers (which directly map to machine registers on most architectures) and return a value. The arguments as well as the return value are typed. Pointer types include the size of the allocation, therefore helpers can implement routines for memory allocation or retrieve pointers to memory areas from BPF programs. For example, BPF programs can use helpers for zero-copy communication with user processes through ring buffers. First, the program allocates a ring-buffer entry of constant size using the `reserve()` helper. This returns a pointer which the BPF program can dereference directly (out-of-bounds accesses are prevented by the verifier). Finally, ownership over the buffer entry is passed back to the kernel which makes it consumable by user processes. Submitting the entry makes the previously retrieved memory area inaccessible to the BPF program, similarly to ownership transfers of references in Rust [Jun+17].

2.4.6 IPC using Maps

Maps enable safe IPC between BPF programs and user processes as well as in-between BPF programs. Synchronization is handled and enforced by the kernel. BPF programs can only access map contents through helpers (⑥ in Figure 2.1) and user space only through system calls (⑦). Both refer to the maps only using descriptors and synchronization is handled by the kernel. Not all types of maps have dictionary semantics. For example, arrays are also supported and, as mentioned in the previous section, ring buffers. Arrays and dictionaries offer the atomic reading/writing of entries. To summarize, maps offer safe IPC with rich semantics to BPF programs and user processes.

2.4.7 Security and Implications

BPF's security guarantees are based on software-based isolation through static bytecode analysis. While enabling low overhead, this approach is not without risks. The verifier and JIT compiler are complex software components which increase the kernel's attack surface significantly. Also, transient execution vulnerabilities can often be exploited from within BPF on today's processors [McI+19].

The verifier and JIT compiler required for BPF increase the kernel's attack surface as they can contain bugs like any other software component. For example, CVE-2021-29154 [Kry21] allowed the execution of arbitrary code in kernel mode and was caused by the incorrect computation of branch displacements. The BPF subsystem of the Linux kernel has 33 k physical source lines of code (SLoC), it is thereby the second-largest core kernel subsystem[3] after `trace` with 47 k SLoC [Lin21b; Whe04].

BPF is based on isolation using software. On today's processors, this technique is prone to side-channel attacks as processor caches are shared to a high degree among the isolated components. As of October 2021, BPF consequently does not support the execution of code from unprivileged sources for security reasons [Cor19a]. However, there exists a kernel capability which grants processes not running as `root` the ability to load and attach BPF programs. Still, several use-cases for unprivileged BPF do exist [Cor19a; Cor21b]. Foremost, unprivileged processes can use

[3]By *core kernel subsystem* I mean any subsystem located in the `kernel` subdirectory of the Linux source tree. `kernel` is not the largest directory of the tree, however, the other larger subdirectories either contain user space tools (i.e., `tools`), hardware abstractions that are not required in every configuration (i.e., `drivers`, `arch`, `sound`), or modules that are also only partially included depending on the configuration (i.e., `net`, `fs`).

BPF to efficiently filter network packets on sockets using the `setsockopt()` system call. Further, `seccomp()`'s implementation, which currently uses cBPF, could be simplified and its functionality extended by allowing it to reuse the existing BPF ecosystem. Finally, additional use-cases in other unprivileged processes (e.g., `systemd` and container runtimes) exist.

[McI+19] argue that transient execution vulnerabilities can not be solved exclusively using software. Transient execution vulnerabilities (e.g., Meltdown and Spectre) are usually exploitable in any Turing-complete language, and can not be effectively mitigated by limiting access to timers (e.g., reducing their accuracy or introducing jitter). Software-based isolation therefore does not currently offer *free lunch*, one does not get isolation in the presence of transient execution vulnerabilities while still having fast user/kernel transitions. BPF is still useful because it adds a new middle-ground with very fast transitions between kernel code and user bytecode while still guaranteeing strong fault-isolation. In addition, the isolation guaranteed by BPF may still be enhanced in the future, for example, using hardware support to prevent the exploitation of side-channels between software-isolated components [Nar+20].

DESIGN 3

This section presents the design of AnyCall which approaches the problem of increasing user/kernel transition overheads using programmable system-call aggregation. The approach is discussed in Section 3.1. Section 3.2 discusses the execution models I have considered for AnyCall and details why I have chosen a call-oriented execution model. Tightly related to this is the programming model, discussed in Section 3.3, as well as the handling of signals, detailed in Section 3.4. Section 3.5 discusses the trust model of AnyCall and Section 3.6 details why AnyCall executes user control flow in the kernel using BPF.

The design of AnyCall aims to retain the positive properties of traditional system calls, while improving on the negative properties (both have been discussed in Section 2.1.3). In particular, AnyCall is *implementation-agnostic*, maintains *fault-isolation*, has a *subroutine-like programmming model*, is *information-hiding*. The *flexibility* of the user/kernel interface is unchanged as it is orthogonal to AnyCall. AnyCall improves on the overhead of traditional system calls (i.e., AnyCall is fast not *slow*) and, on current processors, compromises on *isolation* from malicious actors.

3.1 Approach

AnyCall reduces the user/kernel transition overheads for applications using programmable system-call aggregation. Aggregation reduces the system-call rate as detailed in Section 3.1.1. Programmability, discussed in Section 3.1.2, avoids switches to user context for computation, and reduces implementation complexity for users.

3.1.1 System-Call Aggregation

Aggregation solves the problem of increasing system-call overheads in applications (compare Section 2.2), by executing multiple kernel calls using a single user/kernel transition. This approach reduces the system-call rate by the aggregation factor. For example, aggregating only two system calls on average, already halves the application's system-call rate. Therefore, users only have to aggregate few system calls to achieve significant speedups.

Inside the aggregated section, regular system-call implementations execute without the overheads of frequent user/kernel transitions. AnyCall therefore is as *implementation-agnostic* as traditional system calls because it supports all system calls the OS kernel offers.

3.1.2 Programmability

In-between system calls, user applications often perform error handling or compute the inputs for successive system calls based on the result of preceding calls. AnyCall accounts for this using programmability, that is, users can execute their control flow in-between the aggregated system calls. The user control flow retrieves the results of preceding system calls and computes the arguments for successive system calls.

In comparison to batched system-call interfaces (e.g., Xen multicall [Pan+11]), this has multiple advantages. First, having to batch multiple system calls increases user program's complexity. The user can no longer check for errors directly after individual system calls, but has to delay the check until the whole batch is complete. The delayed detection can even make it necessary to roll back operations that were part of the batch. Second, inside a batch a user program can not compute system-call arguments based on the results of preceding system calls. If such a computation is required, the application can not use batching but has to perform a full switch to user context in order to execute its control flow. AnyCall however, can still be used in this case. By allowing for programmable aggregation, AnyCall therefore improves efficiency and reduces the complexity of applications using system-call aggregation.

To implement programmability, we can not allow the application to execute user machine code directly in kernel mode, as this would be highly unsafe. Instead, the user code still has to be isolated from the kernel to prevent simple programming errors from crashing the whole system. For this, I am proposing software-based isolation as it incurs a lower overhead than the hardware-based techniques modern OSes use for user processes. Software-based isolation can, for example, be implemented using VMs that interpret or JIT compile type-safe application (byte-)code (compare JVM) or, by inserting runtime checks into application machine code using binary-rewriting.

Without loss of generality, the remainder of this chapter refers to the user control flow executing in-between the aggregated system calls as **VM programs**. That is, because VMs usually provide the isolation between the user code and the kernel we require for AnyCall. However, other software-based approaches such as binary-rewriting can also isolate the user code from the kernel. A more general term for VM program, as used in this chapter, would be software-isolated process, or program. The implementation of the isolated execution environment (e.g., VM) is orthogonal to the design of AnyCall.

3.2 Execution Model

I have considered two alternative execution models for AnyCall, a return-oriented and a call-oriented approach. AnyCall uses a call-oriented approach because it offers better safety and efficiently implements a straight-forward programming model.

3.2.1 Return-Oriented

In the return-oriented execution model, VM programs are executed in response to system calls which are initiated by the application. With this model, the control flow is similar to *return-oriented programming* [Roe+12], an exploitation technique where chunks are executed sequentially, and the *return* instruction at the end of an instruction sequence leads to the execution of the following sequence. For system-call aggregation, such sequences are either VM programs, or system calls. In the return-oriented execution model, a VM program can perform kernel calls to request subsequent system calls which, on termination, trigger other VM programs. It thus creates a chain of system calls, interleaved with VM programs.

The return-oriented model is very similar to the way the Linux kernel currently uses BPF. That is, the kernel exclusively invokes BPF programs in response to events. In the case of system-call aggregation, this is the completion of a system call. The return-oriented model for aggregation has been recently proposed for integration into the Linux `io_uring` subsystem [Axb20; Cor21a]. Future work will compare this implementation to AnyCall.

3.2.2 Call-Oriented

In a call-oriented execution model, VM programs invoke system calls synchronously, using stubs provided by the kernel and exposed inside the VM. This thesis refers to system calls performed in this manner as **kernel calls**. Equivalently to function calls, the original VM program receives the result upon system-call completion and resumes execution.

Figure 3.1 displays this model by example. The left side displays an application performing three identical system calls from user space and the right side displays the same three calls executed using AnyCall. The system calls from user context each incur a full user/kernel transition, including a processor mode switch from user to kernel mode. The kernel calls from within the VM, however, have subroutine-like overheads. Therefore, the total direct user/kernel transition overheads are reduced. Furthermore, avoiding the user/kernel transitions reduces the indirect overhead because the VM program as well as the kernel execute with hot caches. This reduces the execution time of the second and third kernel call as well as the runtime of the user control flow executing inside the VM. Therefore, the total execution time t is reduced to t'.

3.2.3 Discussion

The return-oriented and call-oriented execution models differ in multiple aspects, in particular, with respect to safety and programmability.

For *safety*, the system-call aggregation needs to manage its state, but protect it from other contexts. In the return-oriented model, the local state has to be preserved along a call chain, but protected from concurrent call chains. In consequence, this model demands for concurrency control. Furthermore, preserving the state correctly (including garbage collection at each end of the call chain) is non-trivial and cannot be safely ensured by the kernel.

Regarding *programmability*, the call-oriented approach is straight-forward as it resembles the way applications have interacted with the kernel traditionally, which is consequently well-understood by many programmers. In comparison, the return-oriented approach scatters control flow over multiple chunks and embeds control flow decisions in helper calls.

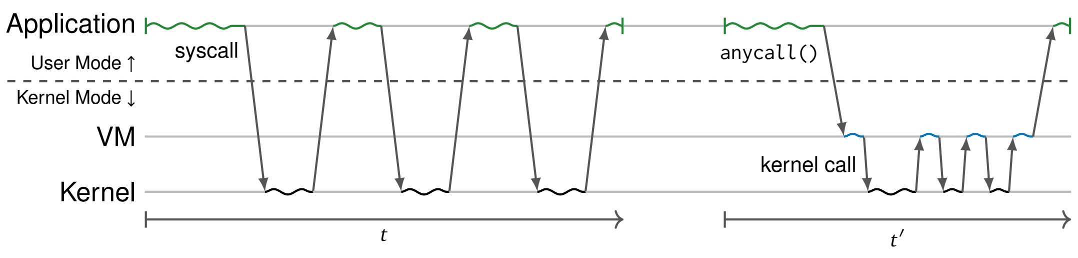

Figure 3.1 – Interaction diagram for three identical synchronous system calls from user space (left) and using AnyCall (right). The aggregated execution time t' is smaller than t as direct and indirect transition overheads are reduced.

In summary, AnyCall implements the call-oriented execution model because it allows for efficient state-management and straight-forward programmability.

3.3 Programming Model

AnyCall should be straight-forward to use for system programmers. Therefore, control flow is transferred to the kernel using subroutine calls and the VM program accesses user memory explicitly using interfaces that resemble `memcpy()` and `mmap()` [Lin17; Lin20c].

3.3.1 Control Flow

To ease adoption, AnyCall must implement a straight-forward programming model. Traditional system calls are invoked as subroutines, therefore many programmers are familiar with this model. AnyCall consequently also implements this model.

In this subroutine-like programming model, the user application invokes the `anycall()` as a subroutine, and the VM program in turn invokes the kernel calls as subroutines. This further has the advantage that call-oriented execution models implements this programming model efficiently and with low complexity. To summarize, the subroutine-like programming model makes AnyCall as straight-forward to use as traditional system calls, while still improving upon their overhead (compare Section 2.1.3).

3.3.2 User Memory Access

Many system calls receive or return values by reference. To access or supply these values, the VM requires a method for reading and writing user memory. This can happen either transparently or explicitly. AnyCall implements the latter approach because it encourages more efficient applications.

The transparent approach to user memory access completely hides the difference between VM memory and user memory from the VM program. The VM program would then access the user memory transparently, just like any other memory it accesses. However, behind the scenes the VM still has to differentiate between user and VM memory, as only the latter is located inside the kernel address space. While accessing kernel memory is natural in kernel context, accessing user memory is an architecture- and configuration-specific operation that sometimes incurs overheads (e.g., switching of the virtual address space, paging in memory that has been swapped to disk). Hiding this performance-critical system property from the programmer would encourage them to write programs that do not perform well. Consequently, AnyCall does not implement a programming model for transparent user memory access.

A preferable approach is to access user memory explicitly in AnyCall. System programmers are familiar with two interfaces that are well-suited for this, `memcpy()` and `mmap()` [Lin17; Lin20c]. First, subroutines that copy memory to or from a supplied user address efficiently access large memory areas. In AnyCall, this is implemented by `copy_to/from_user(void *, size_t, __-user void *)` subroutines the VM program can invoke. Second, for repeated accesses to smaller memory areas, AnyCall should not enforce the overhead of one subroutine invocation per access on the VM program. In this case, it is desirable to allow the VM program to map the user memory into the kernel address space, making it directly accessible with low overhead. In AnyCall, this is implemented by the `void *map(__user void *, size_t)` and `unmap(void *)` subroutines.

3.4 Signal Handling

To enable user processes the timely handling of asynchronous events, OSes offer signals. A signal invokes a user-defined handler whenever a certain event occurs. To still allow the timely handling of events when applications call long-running, blocking system calls, signals abort blocking system calls and cause them to return to user context with an error code. This enables timely handling of the signal. The required restarting of the interrupted system calls can be automated using the `SA_RESTART` mechanism [Lin20e].

With ANYCALL however, returning from a kernel call does not directly return to user context but instead to the VM program. There are two approaches to handle signals in ANYCALL, one *implicit*, the other *explicit*. With the *implicit* approach, an arriving signal not only aborts the kernel call, but also the `anycall()`. With an *explicit* approach, the signal only aborts the kernel call. Return to user context is then only performed due to explicit error handling in the VM program. ANYCALL implements this latter approach because it is less complex to implement and handles signals with a reasonable latency.

The *implicit* approach is more complex to implement because aborting the `anycall()` transparently requires that there is also a way to transparently resume it after the signal-handler executed in user context. The *explict* approach requires the VM program to handle errors from kernel calls and end the `anycall()` prematurely if a signal is detected. It therefore makes the user responsible for timely signal handling. In a prototype, this can be ensured, therefore ANYCALL implements the latter approach.

Future work may allow for the automatic restarting of the `anycall()` after it has been interrupted by a signal-handler installed with the `SA_RESTART` flag [Lin20e]. Alternatively, the application can install VM programs as signal handlers instead of user space routines. This would enable low-overhead, user-defined signal handling with minimal latency.

3.5 Trust Model

ANYCALL enable *programmable* system-call aggregation. To allow for low-overhead programmability while preventing faulty or malicious applications from crashing the entire system, ANYCALL uses software-based isolation.

This approach is not without risks, as software-based isolation is weaker than hardware-based isolation on today's systems [Nar+20]. While the design of ANYCALL is orthogonal to the isolation technique used, it still compromises on the *strong isolation* system calls offer (compare Section 2.1.3). This compromise however, only concerns the isolation from malicious actors as only these may exploit the side-channel vulnerabilities in today's processors to retrieve confidential information. *Fault isolation* in turn is completely retained by ANYCALL as VMs can guarantee that a sandboxed program can never block or crash the enclosing software system (in ANYCALL's case the kernel). Still, to account for the former, ANYCALL is only made available to semi-trusted processes (e.g., using capabilities). Semi-trusted VM programs can be made available to completely untrusted processes through system services.

It is subject of future work to create VMs that allow ANYCALL to be used by untrusted processes, possibly enabled by new hardware features [Nar+20]. Also, the design of ANYCALL ensures that the VM implementation can be changed to trade isolation for expressiveness.

3.6 In-Kernel VM

There exist multiple VMs (or runtime environments) which achieve the software-based isolation AnyCall requires. These include BPF [MJ93], the LuaJIT runtime environment [KS15], the JVM [Gol+02], WebAssembly (Wasm) [Haa+17; ZB20], and Microsoft's Common Language Runtime (CLR) [HL07]. By reusing an existing VM instead of developing a new VM that is solely useful for AnyCall, I reduce the implementation complexity and benefit from the existing development and compiler ecosystem. AnyCall uses BPF because it was designed for use inside the OS kernel and because it has a predictably low memory and execution time overhead.

3.6.1 Memory

BPF achieves memory-safety with predictably low overhead in comparison to systems using garbage collection (e.g., Wasm [ZB20]). Static analysis of the bytecode ensures that dynamic allocations are always freed before the program terminates. This solution is preferable to running a garbage collector inside the kernel, as collection algorithms are complex to implement and unpredictable [Emm+19]. In addition, the optimal garbage collection algorithm is application-specific. Therefore, multiple algorithms have to be implemented in order to achieve optimal results for each application. This further increases the complexity of the implementation, which is not acceptable if the trusted computing base is to be kept small. AnyCall therefore uses BPF with manual memory management safeguarded by static analysis.

3.6.2 Execution Time

In addition to guaranteeing memory-safety, the kernel has to ensure that the VM program does not consume excess CPU time. BPF currently achieves this by statically analyzing the bytecode for a bounded execution time (e.g., all loops are bounded). However, support for preemption using interrupts can be added in future work.

The runtime overhead of BPF also relates to its memory-safety solution. By avoiding the use of garbage collection, BPF ensures that programs have a predictably low execution time overhead and are not subject to unpredictable delays due to garbage collection. In addition, the overhead of manual memory management is low if dynamic allocations are rare.

3.6.3 Ecosystem

BPF was created for user-defined event handling inside the kernel that is fast and flexible. By showing that AnyCall can reuse a VM created for this existing use-case, I keep the implementation complexity of OS kernels low. This is crucial as the continued growth of the trusted computing base is a widely studied problem [Tan06].

In addition to keeping the trusted computing base small, AnyCall further immediately benefits from the existing BPF ecosystem. BPF offers stable compilers from many programming languages and a production-ready bytecode executor inside the kernel. In contrast, creating a new VM or porting a VM designed for execution in user space, would be error-prone and lead to security vulnerabilities. BPF supports multiple front-end programming language and thereby makes it possible to write the AnyCall code in the same programming language as the user application. BPF therefore not only simplifies the implementation of AnyCall, but also makes it straight-forward and convenient to use.

4 IMPLEMENTATION

This section presents my BPF-based implementation of AnyCall for Linux. First, Section 4.1 gives details on the AnyCall Linux kernel patch and Section 4.2 gives an overview of its architecture. The following sections 4.3, 4.4, and 4.5 give details on AnyCall's invocation, its ability to invoke system-call implementations, and its interfaces for user memory access. Section 4.6 presents an AnyCall-specific alternative to the POSIX interface for directory iteration. Finally, Section 4.7 presents how I have ported three user applications to AnyCall.

4.1 Patch Size

One of AnyCall's design goals is a small implementation that does not significantly increase the size of the trusted computing base. Excluding a 375-line system-call table (mostly a copy of the x86 system-call table), my changes consist of 687 addition and 9 deletions across 19 files of the v5.11 Linux kernel. The patch size could be further reduced by removing BPF helper functions (e.g., those for directory iteration) that are only of interest for the prototype.

AnyCall also integrates into the `libbpf` user space library [Lin20a]. The library allows for easy usage of BPF in applications by offering library routines to handle and load BPF bytecode. It is in part generated from the Linux kernel sources. Aside from having to re-run the generation script, only three lines of code had to be added for AnyCall. These merely inform the library about the new BPF program type.

The user space benchmarks and tests for AnyCall consist of 1.7 k SLoC. This, however, also includes duplicate boilerplate code that could be eliminated, and older benchmarks and tests that are not covered in this thesis.

4.2 Overview

Figure 4.1 displays how AnyCall integrates into the existing BPF ecosystem of the Linux kernel.

Loading and Invocation ① ② ③ ⑤ AnyCall BPF programs no longer require the event-driven invocation regular BPF programs use. Instead, applications directly invoke the program using the `anycall()` system call. The BPF program therefore executes synchronously in the context of the user thread but still in kernel mode. Upon completion, the BPF program returns to the invoking application thread. Section 4.3 covers loading and invocation in detail.

Kernel Calls ④ To allow for system-call aggregation, the AnyCall patch adds new BPF helper functions that allow the program to invoke regular system calls as subroutines. Section 4.4 covers these helpers in detail.

User Memory Access Because many system calls retrieve or return values by reference, it is helpful if the AnyCall BPF program can efficiently read and write user memory. This allows the program to access system-call results and arguments. The patch includes multiple interfaces (resembling POSIX' `memcpy()` and `mmap()`) for user memory access. Section 4.5 presents these interfaces in detail.

In comparison to the existing BPF implementation (displayed in Figure 2.1), AnyCall modifies the invocation model and the available helper functions. The BPF verifier and JIT compiler are not modified.

4.3 Loading and Invocation

This section presents how application threads transfer control to AnyCall BPF programs. The AnyCall patch adds a new BPF program type to the Linux kernel. Programs of the AnyCall type can be loaded using the standard `bpf()` system call and therefore also integrate with existing libraries that simplify BPF bytecode handling (e.g., `libbpf` [Lin20a]). Usually, BPF programs are attached to kernel events after loading and are then invoked asynchronously. To enable AnyCall's synchronous execution model, the patch adds a new system call which invokes the BPF program referenced by a file descriptor. Attaching the program to an event is not required. The `anycall()` system call returns once the BPF program has finished executing. AnyCall BPF programs therefore execute in the context of the application thread but in kernel mode. In this context it is possible to block the invoking thread and access the memory of the process. Therefore, AnyCall can reuse the regular system-call implementations which were created under the assumption of executing in this context.

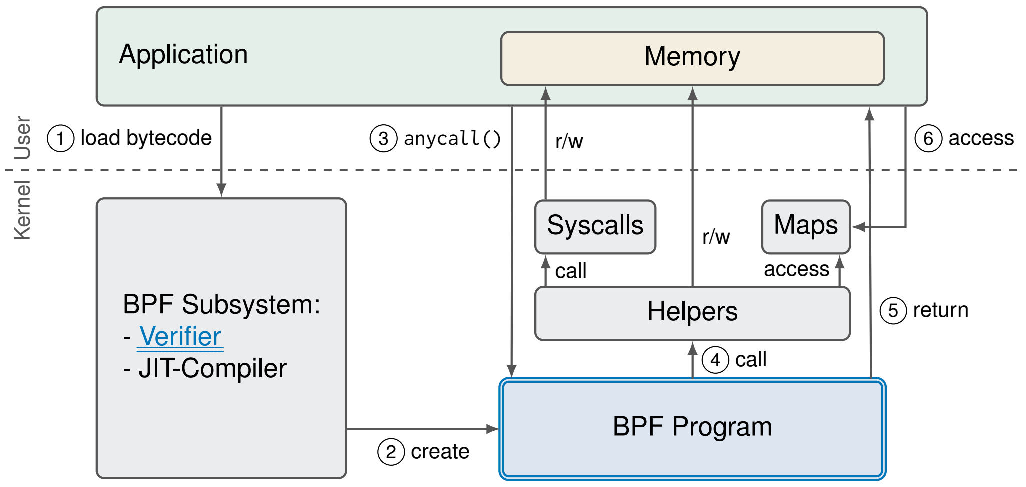

Figure 4.1 – Architecture of AnyCall's Linux implementation. BPF programs of the AnyCall type are invoked synchronously and execute in the context of the invoking application thread. They can invoke system calls as subroutines and access user memory using special BPF helper functions.

4.4 Kernel Calls

To allow for system-call aggregation, AnyCall exposes system-call implementations to BPF programs as helper functions called *kernel calls*. A similar but more limited[4] approach (i.e., only supporting the `bpf()` and `close()` system calls) was recently introduced by the kernel developers in the context of code signing [Cor21c; Sta21]. Regarding the system calls available to BPF, AnyCall uses a dedicated system-call table because many platforms offer non-portable architecture-specific system calls. BPF bytecode however should be fully portable, therefore these architecture-specific system calls must not be used. In practice, this does not affect the expressiveness of AnyCall BPF programs as architecture-specific system calls are rarely used in portable real-world applications.

In BPF bytecode, helpers are identified by numbers which are replaced with the function address during compilation to machine code. Therefore, the call-time overhead is comparable to a subroutine call in machine code. Each kernel call is invoked using a dedicated helper function receiving up to five arguments in BPF registers (which directly map to machine registers on most architectures). Additional arguments can be passed by reference using an array on the BPF stack.

4.5 User Memory Access

Many system calls receive or return values by reference. The existing code-base of the Linux kernel expects that these references point to user space. The numerous widely-scattered checks that system-call arguments do not reference kernel memory are required for security reasons, so disabling or bypassing them is not an option. Hence, AnyCall requires a way to read and write user memory in order to construct system-call arguments and process their results.

The BPF static analyzer already has the ability to track fixed-size dynamic memory allocations from helper functions. This is used to allow BPF to allocate and write records into ring buffers shared with user space. This can be used like `malloc/free` [Lin21a] in BPF programs, however, the available helpers allocate kernel memory which is unusable for system-call arguments. Therefore, a new mechanism is required.

I have considered three alternative solutions: one resembles `memcpy()`, one is based on page-faults, and the other is based on page pinning. All three give BPF the ability to access arbitrary user memory if it is safe, but they differ regarding their efficiency and application programming interface (API). The C interfaces are compared in Listing 4.1. AnyCall implements the approaches resembling `memcpy()` and the one based on page pinning, because these are the most efficient. The `memcpy`-approach is the most efficient if memory is only accessed once and the pinning-approach is the most efficient if memory is accessed repeatedly.

4.5.1 Copy Helpers

The first interface is straight-forward in that it offers two helpers which copy memory to or from a user address. This is very efficient if the user memory has to be copied anyway, or if the access only happens once. In these cases, the interface only requires a single BPF helper call.

[4]The solution discussed to enable signature checks for BPF programs adds a `syscall` BPF program type to the kernel. Like `anycall()`s, these run in the context of the invoking user process. However, unlike `anycall()`, they are limited to two system calls (`bpf()` and `close()`) and do not offer special interfaces for user memory access.

```
// Copy Helpers
void copy_to_user(void __user *dst, void *src, size_t n);
void copy_from_user(void *dst, void __user *src, size_t n);

// Page-Fault Handler (not implemented)
handle_t *user_access_begin(void __user *addr, size_t m);
bool copy_to_handle(handle_t *dst, size_t o, void *src, size_t n);
bool copy_from_handle(void *dst, handle_t *src, size_t o, size_t n);
void user_access_end(handle_t *);

// Page Pinning
void *map(void __user *addr, size_t m);
void unmap(void *ptr);
```

Listing 4.1 – Synopsis of the AnyCall interfaces for user memory access. For safety, the verifier prevents arbitrary pointers from being dereferenced directly. This includes the pointers to user memory which are marked with `__user` in the synopsis.

4.5.2 Page-Fault Handler

Most traditional system calls access data in user space directly, swapping-in non-present pages on demand using the kernel's page-fault handler. The same technique can be used in BPF programs, However, because user memory access is architecture-specific BPF helper function calls are required for every memory access. These read or write from the memory referenced by the opaque `handle_t` pointer type at the given offset (`size_t o`). On x86, present user memory is accessed directly but special operations are required on other architectures. If the access triggers a page-fault which can not be handled, the helper functions return an error to the BPF program.

To be safe, the lifetime of the `handle_t` pointer returned from `user_access_begin()` must be tracked by the BPF verifier. This ensures that the `copy_to/from_handle()` helpers are only called before the BPF program invokes `user_access_end()`. If the program does not guarantee this invariant, the verifier rejects it at load-time. The verifier also ensures that `user_access_end()` is called for every `handle_t` pointer before the program terminates.

Discussion. This interface has the disadvantage of requiring one BPF helper call for every user memory access. It therefore does not have any advantage over the interface resembling `memcpy()`, while arguably being more complicated to use and implement. AnyCall therefore does not use this approach. Future work may extend the BPF JIT compiler to inline the `copy_to/from_handle()` calls, thereby possibly optimizing them on architectures where user memory can be accessed directly. This would enable efficient user memory access from BPF without pinning the pages into memory.

4.5.3 Page Pinning

To avoid a helper call for every user memory access, AnyCall implements an alternative solution that only requires a `map()` and an `unmap()` helper call for each memory area. `map()` pins the pages into memory, thereby preventing page-faults, and *maps* them to kernel virtual addresses to guarantee direct, portable access to the memory from BPF.

Listing 4.2 illustrates the use of this interface and the invariants enforced by the BPF verifier. The BPF program passes a user address and the size of the mapping to `map()`. The verifier ensures that the program only accesses the returned pointer if it has previously performed an error check

```
1 int *buf = map(user_addr, sizeof(int));
2 if (!buf) return -1;         // Check enforced by loader
3 *buf = 4;                    // Valid, access within bounds
4 *((long *) buf) = 4;         // Invalid, out-of-bounds
5 unmap(buf);                  // unmap-call enforced by loader
6 *buf = 4;                    // Invalid, unmap() invalidated buf
```

Listing 4.2 – BPF program accessing an integer at the virtual user address `user_addr` using `map()` and `unmap()`. **Invalid** accesses to the memory area, which must be of constant size, are detected and prevented by the BPF verifier. Dereferencing `user_addr` directly is prevented by the verifier for safety.

to ensure `map()` was successful. If this is the case, the program can dereference the pointer (and thereby access the user memory backing it) at any offsets that do not exceed the static size of the mapping. Therefore, the access in line 4 would be rejected at load-time. Finally, calling `unmap()` makes any following dereferences illegal (e.g., the access on line 6 would be rejected).

Discussion. In comparison to the page-fault and copy-approaches, the pinning-approach has a lower overhead if the program accesses the area repeatedly. The overhead is reduced because the pinning-approach only requires two helper calls, no matter how often the memory is accessed. However, pinning the pages into memory also has the downsides. It increases the amount of unswappable kernel memory, therfore risking memory bottlenecks if used excessively. To prevent the system from running low on memory, only small areas should be mapped. The `map()` helper can perform a check to ensure no single process pins excessive amounts of memory to guarantee this.

4.6 Directory Iteration

This section presents an AnyCall-specific interface for directory iteration which eliminates almost all user/kernel transitions. I motivate the interface in Section 4.6.1, present its core idea in Section 4.6.2, and finally its implementation in Section 4.6.3. Section 4.7.3 will present a real-world application using this interface, which is also part of my evaluation.

4.6.1 Motivation

For every BPF program, the verifier must prove a bounded execution time to accept it. If the termination of the program depends on the results of kernel calls, this is not possible. For example, the verifier can not know that the `getdents` system call must, at some point, return no further directory contents. This can be solved by setting a static upper limit on the number of iterations in the BPF program, and having user space restart the program as needed.

While this works almost always and has negligible overhead because it still aggregates many system calls, I also wanted to explore interfaces that take advantage of BPF's specific properties to create more efficient kernel interfaces. The system-call implementations AnyCall reuses are all designed for use from user processes: they often receive or return data in large chunks in an attempt to reduce the number of user/kernel transitions. To demonstrate how this can be achieved more efficiently using BPF, I have created the `iterdents` interface which allows for user-defined directory iteration using only a single user/kernel transition to recursively process an entire directory tree.

4.6.2 Design

The core idea of the `iterdents` interface is to invoke a BPF callback program for each individual directory entry instead of copying the entries to user memory. The BPF program receives a pointer to the directory entry in kernel memory which it accesses directly. For this, it reuses the mechanism that allows BPF to directly access fields in network packet headers. Thereby, it avoids copying the directory entries to user memory. Further, by iterating over the directory contents in the kernel, the interface avoids the need for an unbounded loop in the BPF program.

4.6.3 Implementation

Listing 4.3 compares the `getdents` [Lin20b] and `iterdents` C APIs. The `getdents` interface allows user processes to read a chunk of directory entries from a directory file descriptor. The entries are copied into the user-supplied buffer and the read offset of the directory file is advanced. Subsequent calls return the following entries to the user process.

The AnyCall `iterdents` interface also receives a directory file descriptor, but in addition also receives a descriptor referencing a loaded BPF callback program as second argument. The kernel iterates over the directory entries and invokes the BPF callback for every entry. When invoked, the program receives a pointer to the `bpf_dirent64` structure which contains information about the current entry and is allocated and initialized by the kernel.

Frequently, it is helpful to invoke additional system calls inside the callback itself. For example, to print the directory entry to standard out (using `write()`) or to open the entry (using `openat()`). These system calls expect pointers to user memory containing the directory entry's name. To enable these operations, `iterdents` optionally receives two additional arguments: a user address (`void __user *buf`) and the amount of memory writable at this address (`size_t n`). Before invoking the callback, the kernel copies the entry's name to the user address (unless it is `NULL`). The callback can

```
ssize_t getdents64(int fd, struct linux_dirent64 *dirp, size_t count);
struct linux_dirent64 {
  ino64_t        d_ino;     /* Inode number */
  off64_t        d_off;     /* Offset to next structure */
  unsigned short d_reclen;  /* Size of this dirent */
  unsigned char  d_type;    /* File type */
  char           d_name[];  /* Filename (null-terminated) */
}

int iterdents64(int fd, int cb_fd, void __user *buf, size_t n);
/* cb_fd: int bpf_callback(const struct bpf_dirent64 *) */
struct bpf_dirent64 {
  const char *name;      /* Filename (null-terminated kernel mem.) */
  void       *name_end;  /* End of filename, for verification */
  u32         name_len;  /* Length of filename */
  u64         ino;       /* Inode number */
  u32         type;      /* File type */
};
```

Listing 4.3 – Synopsis comparing the Linux `getdents` and AnyCall `iterdents` interfaces for user-defined directory iteration.

therefore use this address as an argument to system calls like `write()`, `openat()`, and `fstatat()`. With `iterdents64()`, this enables user-defined recursive directory iteration entirely in the context of the kernel. To prevent the kernel stack from overflowing, `iterdents64()` can maintain a per-thread counter to limit the depth of the recursion.

To ensure memory safety, the BPF verifier prevents the callback from writing to the `bpf_dirent64` structure and ensures that `const char *name` is only ever dereferenced at offsets up to `void *name_end`. The callback can therefore never corrupt kernel data even though it can directly access the directory entry. While iterating over the directory contents, the directory's inode is locked thereby preventing concurrent accesses.[5]

The current implementation of `iterdents` in AnyCall copies parts of the directory entry into the `bpf_dirent64` structure. In addition, if `void __user *name` is not null, `iterdents` also copies each entry name to user memory. The interface is therefore not zero-copy and could be improved further by allowing the callback to directly access the entries in the format in which they are already be present in the page cache. Also, the number of subroutine calls can be reduced by passing an array of entries to the callback. Finally, a more traditional approach could extend the `getdents` interface to also accept BPF callbacks which determine whether certain entries should be skipped.

4.7 Porting Effort

This section presents three applications I have ported to AnyCall. Even though AnyCall is still a prototype, the knowledge required to use it can be acquired quickly. To move C code to AnyCall, the programmer places the code in AnyCall's `libbpf`-based framework to compile it to BPF bytecode instructions. If required at all, modifying the code is straight-forward. It includes adding calls to access user memory areas, ensuring bounded loops, limiting the program size, and making sure all pointer arithmetic is analyzable. In any case, the need for modifications arises from the limits of the BPF execution environment and not from AnyCall. As kernel developers improve BPF's expressiveness continuously, fewer modifications are required for AnyCall with every kernel release. Removing the need for manual modifications using compiler-integration is a possible subject for future work.

This section is structured as follows. Section 4.7.1 first gives a small example which illustrates the most common modifications in detail. The following Section 4.7.2 and Section 4.7.3 detail how I have ported two real-world applications to AnyCall. Chapter 5 compares the performance of the different implementations. Finally, Section 4.7.4 summarizes the lessons learned while porting the two tools to AnyCall.

4.7.1 Illustrative Example

Listing 4.4 illustrates how a small C program for disk usage estimation can be ported to AnyCall. The program has two global variables, `int fd[N]` and `size_t total`. The first variable contains a set of open files for which `estimate_disk_usage()` should store the aggregated disk usage into `size_t total`. For this, the algorithm iterates over the files and invokes the `fstat()` system call

[5] Note that this may be unacceptable in some systems as it allows processes using `iterdents` to lock out competing processes while iterating over the directory (which may take considerable CPU time as it can include multiple BPF callbacks and recursive iteration over subdirectories). To solve this, the system can release the lock every M entries allowing for concurrent accesses to the directory. It thereby would effectively replicate the guarantees provided by `getdents`, which also allows concurrent accesses in-between the individual `getdents` calls, but prevents them in `getdents` while the kernel copies the requested M entries to user memory. This approach may be extended to support nested iteration by allowing recursive iterators to release and the re-acquire the locks held by parent iterators.

```
 1 #define N 256
 2 int fd[N]; // Input files.
 3 size_t total; // Output.
 4 struct stat __user *user_addr;
 5
 6 void estimate_disk_usage(size_t n) {
 7   struct stat s;
 8   for (size_t i = 0; i < n && i < N; i++) {
 9     fstat(fd[i], user_addr);
10     copy_from_user(&s, user_addr, sizeof(s));
11     total += s.st_size;
12   }
13 }
```

Listing 4.4 – C Program excerpt for disk usage estimation, ported to AnyCall using the **modifications** marked bold blue. Error handling is omitted for brevity. $n < N$, where N is a compile-time constant.

for each descriptor. The system call stores the file's disk usage to the user memory buffer from where the algorithm retrieves it and adds it to the total. To port this function to AnyCall, two changes are required:

`copy_from_user()` *(Lines 4, 9, and 10)* When the algorithm executes in user space, the program passes a pointer to `stat` structure directly to `fstat()`. This is not possible if the algorithm is executed using BPF in the kernel: the `stat` structure is allocated on the BPF stack in kernel memory, passing its address to `fstat()` would be an error as the system call expects a user address. Instead, the program has to pass `user_addr` to the system call which points to a memory buffer the invoking C program has allocated for the BPF program. The system call stores the result to this user address and the BPF program copies it to the BPF stack using the `copy_from_user()` helper.[6]

`i < N` *(Line 8)* The function only estimates the disk usage for the first n file descriptors. The runtime as well as the memory safety of the code therefore depends on the input parameter. To be memory safe, n must always be less than the array size N. The BPF verifier requires that this is guaranteed by the BPF bytecode while user space machine code can trigger a runtime error. The programmer can solve this by inserting a runtime check to ensure `estimate_disk_usage()` never iterates past the end of the file descriptor array.

To summarize, both changes are relatively straight-forward and in the second case the BPF verifier even helps in preventing runtime errors. The following section presents a real-world file searching tool that uses an algorithm similar to the one discussed in this section.

4.7.2 File Searching Tool

Many file types user *magic values* at predefined offsets for identification. To demonstrate that AnyCall can speed up a real-world application, I have applied it to a tool that filters files by such

[6]The example uses `copy_from_user()` for brevity but the performance could be further improved by using the pinning-approach for accessing user memory. For this, the BPF program maps the user address once, before starting the iteration, and retrieves each system-call result using the pointer returned from `map()`. After finishing the iteration, the program calls `unmap()`.

```
 1: allocate paths[N]
 2: repeat
 3:     for all i < N do
 4:         fgets() into paths[i]
 5:         0-terminate path
 6:     end for
 7:     for all i < N do
 8:         int fd = open(paths[i], O_RDONLY, 0777)
 9:         lseek(fd, offset, SEEK_SET)
10:         read() from fd into buffer
11:         if magic value equals the value in buffer then
12:             calculate path length using strlen(paths[i])
13:             write() path length bytes from paths[i] to standard out
14:         end if
15:     end for
16: until EOF not reached
```

Algorithm 4.1 – Algorithm of a traditional C program which filters files by their magic value.

```
 1: allocate paths[N]
 2: allocate shared user memory
 3: load AnyCall
 4: repeat
 5:     for all i < N do
 6:         fgets() into paths[i]
 7:         0-terminate path
 8:         calculate path length, write to shared user memory
 9:     end for
10:     anycall()
11: until EOF reached
```

Algorithm 4.2 – Algorithm to filter files by magic values using AnyCall. `anycall()` invokes Algorithm 4.3. **Changes** with regard to Algorithm 4.1 are marked bold blue.

```
 1: map() shared user memory to shared kernel memory
 2: for all i < N do
 3:     int fd = open(paths[i], O_RDONLY, 0777)
 4:     lseek(fd, offset, SEEK_SET)
 5:     read() from fd into shared user memory
 6:     if magic value equals the value in shared kernel memory then
 7:         read path length from shared kernel memory
 8:         write() path length bytes from paths[i] to standard out
 9:     end if
10: end for
```

Algorithm 4.3 – BPF program used to filter files by magic values using AnyCall. *Shared kernel memory* and *shared user memory* reference the same physical memory but have different virtual addresses. **Changes** with regard to the lines 7–15 of Algorithm 4.1 are marked bold blue.

magic values. For this tool, the list of files is received on standard input (e.g., generated by `find -type f` [GNU21]) and each file is opened, `seek`-ed, read, and closed. If the contents read from the offset match the magic value, the file path is redirected to standard out.

In total, I have implemented four variants of the tool. One implementation uses AnyCall, the others (`libc`, `sys`, and `sys-burst`) use traditional system calls. This section compares the algorithms of the `sys-burst` and AnyCall implementations. For details on the other two implementations as well as the evaluation, refer to Section 5.4.1. Algorithm 4.1 displays the algorithm executed by the `sys-burst` implementation. The AnyCall implementation is displayed in Algorithm 4.2 and Algorithm 4.3. `sys-burst` and AnyCall both read file paths from standard input in chunks of N. They check each file path for the magic value and conditionally write the path to standard output. The AnyCall implementation performs the latter using a single `anycall()` (which executes Algorithm 4.3) while the `sys-burst` implementation uses multiple traditional system calls. For brevity, error handling is omitted in this presentation. Also, we assume that the number of file paths read from standard input is always a multiple of N. Extending the algorithms to remove these restrictions is straight-forward.

The AnyCall implementation uses the `map()` interface to efficiently access the user memory. First, this enables AnyCall to retrieve parameters of the chunk from the user space algorithm (e.g., the length of the file paths). Second, the BPF program can use the mapping to quickly access the data read from the file. For this, it passes the user address which was mapped to the `read()` system call, thereafter it can immediately access the result (i.e., the bytes to be compared against the magic value) by reading from the kernel address of the mapping (i.e., the pointer returned from `map()`).

Iterating over the contents of zero-terminated C strings is a possibly-unbounded operation which must be avoided in BPF programs. Still, the BPF program requires the length of each file path in order to write it to standard output. Therefore, the length of all file paths is calculated in user space and stored to the *shared memory* buffer from where the BPF program reads them (line 7 in Algorithm 4.3).

4.7.3 Disk Usage Estimation Tool

This section presents the port of a real-world disk usage estimation tool to AnyCall. It is thereby similar to the example presented in Section 4.7.1, but also includes the algorithms to find and open the files in the first place. For this, it uses the AnyCall-specific `iterdents` interface to recursively iterate over the directory tree entirely inside the kernel.

Algorithm 4.5 displays the BPF program used to recursively estimate the disk usage for a directory tree. The iteration happens recursively using a single `anycall()` from user space. For this, the callback itself invokes `openat()` to open each directory entry and then iterates over the contents of any subdirectories recursively using `iterdents`. In comparison, Algorithm 4.4, which shows the equivalent user space program, has to allocate a buffer into which it lets `getdents` copy the directory entries in chunks of M. A second, minor difference, between the two algorithms is the use of `copy_from_user()` after `fstatat()` in the BPF program. In summary, the two algorithms are very similar, therefore porting the user application to use AnyCall is straight-forward.

Regarding performance, there is still potential for optimizations. The `iterdents` interface does not avoid copying each entry's name as `openat()` still requires a user address. Also, meta information about each entry (e.g., its inode number) is copied to the `bpf_dirent64` structure. Future work may develop interfaces that eliminate these copy operations.

```
size_t total = 0
procedure ESTIMATE_DISK_USAGE(int parent_fd, struct linux_dirent64 *entry)
    if entry is a regular file then
        allocate struct stat statbuf
        fstatat(parent_fd, entry->d_name, &statbuf)
        total += statbuf.st_size
    else if entry is a directory other than "." or ".." then
        int fd = openat(parent_fd, entry->d_name)
        allocate buf[M]
        repeat
            getdents64(fd, buf, M)
            for struct linux_dirent64 child in buf do
                ESTIMATE_DISK_USAGE(fd, &child)
            end for
        until EOF reached
        close(fd)
    end if
end procedure
```

Algorithm 4.4 – Recursive user space algorithm to estimate the disk usage for a directory tree. The recursion is initiated by passing `AT_FDCWD` (which references the current working directory) as `parent_fd`, and a dummy directory entry with the name of the subdirectory to be traversed as `entry`. The output is written to `total`.

```
size_t total = 0
int parent_fd = AT_FDCWD
procedure CALLBACK(struct bpf_dirent64 *entry)
    if entry is a regular file then
        allocate struct stat statbuf
        fstatat(parent_fd, u_name, u_statbuf)
        copy_from_user(&statbuf, u_statbuf, sizeof(struct stat))
        total += statbuf.st_size
    else if entry is a directory other than "." or ".." then
        int fd = openat(parent_fd, u_name)
        save parent_fd and set it to fd
        iterents64(parent_fd, callback_fd, u_name, NAME_MAX)
        restore parent_fd
        close(fd)
    end if
end procedure
```

Algorithm 4.5 – Recursive BPF program to estimate the disk usage for a directory tree. When invoked, the procedure writes the total disk usage to `total`, from where the user program can retrieve it. Pointer variables prefixed with `u_` are allocated by the user program. `callback_fd` references the BPF program itself. **Changes** with regard to Algorithm 4.4 are marked bold blue.

4.7.4 Lessons Learned

This section summarizes the lessons learned while porting the file searching and disk usage estimation tools to AnyCall. In general, simple programs can be ported quickly using to the transparent `libc`-like system-call stubs my framework provides. Further, BPF's static analysis supports the programmer in writing reliable programs. In multiple instances, it prevented bugs in my implementations by catching missing error handling and edge case in the control flow. On the other hand, modifying null-terminated C strings is often best performed in user space. That is, because iterating over their contents requires an upper limit anyways to satisfy the BPF verifier, therefore, a fall-back slow path in user space has to be implemented if this can not be guaranteed. However, BPF can still pass the C strings around (e.g., as system-call arguments) as long as it treats them as opaque pointers. For this, user space must only prepare them in the required format (e.g., stripping newlines from file paths) before passing control to the BPF program.

Porting the file searching tool to AnyCall has shown that one of the main challenges in writing AnyCall programs arises from the required handling of both user and kernel addresses. Future work may address this using hybrid pointers (i.e., structures that contain both a user address and a kernel address to which the memory is mapped). The appropriate way to access the area could then be chosen automatically based on the context. In addition, mapping the user memory into the kernel can be performed lazily on demand. Another approach is to eliminate the need for user addresses in BPF entirely by changing the system calls to also accept references to kernel memory.

In conclusion, porting modern, system-call intensive C code to AnyCall can be accomplished quickly by experienced systems programmers. In contrast, tasks that make heavy use of pointer arithmetic, or that are computation-intensive, are better performed in user space.

5 EVALUATION

This section presents ANYCALL's evaluation. It is designed to answers the following research questions:

Q1 How many system calls are required to amortize for the overhead of JIT compiling and verifying the BPF programs required for using ANYCALL?

Q2 How does the type and number of kernel calls the `anycall()` performs, influence the amortization time?

Q3 To which extent does the processor architecture influence the amortization time?

Q4 How is the performance influenced by the system's software configuration? For this, I consider mitigations against transient execution vulnerabilities (e.g., Meltdown) and the configured processor frequency.

Q5 Are the system-call rates in real-world applications high enough to benefit from ANYCALL?

Q6 Which types of real-world applications benefit most from ANYCALL? Are these CPU-, memory-, or I/O-bound?

To answer these questions, I have implemented multiple micro- and real-world benchmarks. Using the microbenchmarks, I compare ANYCALL to the equivalent implementation performing traditional system calls from user space. My real world benchmarks are the applications presented in the sections 4.7.2 and 4.7.3, which, respectively, search files by magic values and estimate disk usage. Every experiment is executed on multiple systems in different software configurations to determine the influence of the execution environment.

This chapter is structured as follows. Section 5.1 describes the evaluation hardware and the software configuration. Section 5.2 and Section 5.3 cover the microbenchmarks while Section 5.4 presents the results from the real-world benchmark. Section 5.5 summarizes the answers to the research questions *Q1* through *Q6*.

5.1 Setup

To enable reproducibility, this section describes the evaluation systems and their configuration. Section 5.1.1 describes the hardware and software used, with Section 5.1.2 focusing on the processor vulnerability mitigations active in my experiments. Section 5.1.3 describes my measurement routine in detail.

5.1.1 Hardware and Software

The evaluation machines run version 5.11 of the Linux kernel [Lin21b] with the AnyCall patch and Debian GNU/Linux 10.2.1 (Buster). The disks use the ext4 file system and the system configuration is left to the default except for the processor frequency and the active vulnerability mitigations. The processors are run at their fixed base frequency instead of using dynamic voltage and frequency scaling (DVFS), for two reasons. First, disabling DVFS aids reproducibility as the maximum frequency depends on the system's environment (e.g., the temperature). Second, the AnyCall and `sys-burst` implementations evaluated in Section 5.4.1 trigger higher CPU frequencies than the other implementations if the default Linux frequency governor is used. Therefore, disabling DVFS ensures fairness. To summarize, both processors run at their constant base frequency. I have also evaluated the benchmarks with the frequency pinned to the maximum and with DVFS enabled, but neither had an effect on the conclusions drawn from the evaluation results.

Table 5.1 compares the two machines used in the evaluation. The AMD machine runs at a higher frequency[7] and uses a faster SSD than the Intel machine. Most notably, the AMD 3950X was released after Meltdown and Spectre had been disclosed, while the Intel i5-6260U had been released before. Consequently, the AMD machine mitigates against most transient execution vulnerabilities in hardware while the Intel machine requires OS-level mitigations. The following Section 5.1.2 discusses this difference in detail.

5.1.2 Processor Vulnerability Mitigations

I execute my tests in various system configurations to analyze the effect on the results. As expected, enabling or disabling software mitigations for transient execution vulnerabilities has a measurable effect. The mitigations are activated/deactivated using the standard `mitigations` kernel parameter supplied at boot-time. Parameters to enable or disable specific mitigations are not used. For reference, Table 5.2 lists the transient execution vulnerabilities mitigated on each system in the different configurations. The mitigations used against these vulnerabilities are listed in Table 5.3.[8]

The Intel i5-6260U is vulnerable to Meltdown, therefore Linux enables KPTI by default. This is not the case for recent processors (including the AMD 3950X), but I still believe the results obtained on the machine with KPTI active are relevant. That is for two reasons. First, I expect that there is still a significant number of processors vulnerable to Meltdown deployed, simply because of the large number of devices affected when the vulnerability was disclosed. Second, as the discovery of

[7]Note that the AMD machine is pinned to 2.8 GHz even though the advertised base frequency is 3.5 GHz. That is due to limitations in the Linux DVFS driver for recent AMD processors. On these processors, pinning the frequency to the base frequency also enables automatic frequency scaling up to the maximum frequency (in my case up to 4.7 GHz depending on the environment and used CPU core [Rei+20]). This makes the frequency at which the experiments execute unpredictable and hinders reproducibility. To solve this without installing a custom DVFS driver which would also hinder reproducibility, I pin the processor to 2.8 GHz as it is the highest frequency below the base frequency to which the processor can be fixed.

[8]Obtained by reading from `/sys/devices/system/cpu/vulnerabilities`.

Table 5.1 – Overview over the evaluation hardware. The Intel machine uses the Transcend MTS800 (TS128GMTS800) [Tra21] SSD, while the AMD machine uses the Samsung PM961 (MZVLW128HEGR-00000) SSD. Meltdown was disclosed in 2018.

Processor	Release	CPU Freq.	RAM	Disk Seq. Read	Disk Random 4 kByte Read
Intel i5-6260U	2015	1.8 GHz	16 GB	560 MB/s	70 k IOPS
AMD 3950X	2019	2.8 GHz	32 GB	2800 MB/s	140 k IOPS

Table 5.2 – Comparison of active mitigations in the different evaluation setups. `mitigations=on/off` refers to the kernel boot parameter to disable mitigations against transient execution vulnerabilities. Refer to Table 5.3 for details on the respective OS-level mitigations.

	Mitigated Vulnerabilities	
Processor	`mitigations=on`	`mitigations=off`
Intel i5-6260U	Meltdown, Spectre, SSB, L1TF, MDS	L1TF
AMD 3950X	Spectre, SSB	-

Table 5.3 – List of transient execution vulnerabilities and their mitigations in Linux. Indirect branch prediction barrier (IBPB) and single threaded indirect branch prediction (STIBP) are conditional in that they are only active for SECCOMP or indirect branch restricted tasks.

Vulnerability	**Mitigation**
Meltdown	Kernel Page Table Isolation (KPTI)
Spectre v1	usercopy/swapgs barriers; `__user` pointer sanitization
Spectre v2	full generic/AMD retpoline; conditional IBPB; conditional STIBP; IBRS_FW (Intel-only); RSB filling
Speculative Store Bypass (SSB)	disabled via `prctl` and `seccomp`
L1 Terminal Fault (L1TF)	page table entry (PTE) inversion
Microarchitectural Data Sampling (MDS)	clear CPU buffers

new Meltdown-type attacks and development of respective mitigations is an ongoing process [MF21; HWH13; KPK12], it is likely that the latency of user/kernel transitions increases further. This is in line with the general development of an increasing latency of core OS functionalities (e.g., system calls) [Ren+19].

5.1.3 Measurement Routine

The measurements presented in this evaluation are structured as follows. A **test** executes a certain benchmark program in a specific system configuration. Each test is possibly executed multiple times in a row to determine the influence of the caches, I call this a test **burst**. To analyze the run-to-run variance, each test burst can in addition be **repeated** multiple times with arbitrary other tests and even reboots in-between the burst's repetitions.

To prepare a system for a test, it is booted with the desired kernel parameter and the processor frequency is pinned to the base frequency. Thereafter, test bursts for different benchmark programs execute without reboots. To still make the results reproducible, I use the following measurement routine. Immediately before executing the first test in a burst, the system runs `sync`, drops the page caches[9], and then stays idle for one second. Thereafter, a `bash` loop executes the test burst. Therefore, in the first iteration, the caches are cold while the other iterations execute with warm caches. During the burst, the performance measurements are stored to the local disk and only transmitted to the control system after the burst. The order in which test bursts for different configurations/programs execute is randomized in each repetition. Therefore, the machine state in which tests execute is as diverse as possible. To record performance metrics (e.g., execution time, CPU time, and hardware performance counters) I use the Linux kernel's `perf stat` tool. I record at most as many hardware

[9]I drop the page caches by writing to `/proc/sys/vm/drop_caches`.

performance counters (i.e., events) as supported by the processor, therefore `perf`'s scaling feature is not used.

5.2 Microbenchmark

This section presents the performance of `getpid()` executed using AnyCall or traditional system calls. This experiment contributes to answering the research questions *Q1* through *Q4*.

5.2.1 Number of `anycall()`s

To analyze the degree to which AnyCall reduces direct user/kernel transition overheads, I measure the execution time of `getpid()` performed with AnyCall and in user space. I refer to the user-space implementation as `sys`. The execution time of the AnyCall implementation includes the time for loading the BPF program into the kernel. The motivating experiment in the introduction has demonstrated that it is not uncommon for real-world applications to perform $6.97 \cdot 10^4$ to $7.26 \cdot 10^5$ system calls per second. Therefore, my first experiment, displayed in Figure 5.1, executes between 0 and $2 \cdot 10^5$ `getpid()` calls in user space or using AnyCall (`anycall()`s with 200 kernel calls are invoked 0 to 1,000 times).

In this experiment as well as the one described in the following Section 5.3, the burst length is set to two and each burst is only repeated once. Therefore, every test is executed twice, once with cold and once with warm caches. Both runs are shown in Figure 5.1. As the workload is CPU- and memory-bound, I report the CPU time instead of the wall time to normalize for page faults and background tasks.

As expected, the time required for static analysis and to-native compilation of the bytecode causes the user-space implementation to be faster if few system calls are performed. However, if AnyCall performs the equivalent of $2.15 \cdot 10^4$ to $1.90 \cdot 10^5$ traditional system calls, it is faster than the `sys` implementation. The specific number of system calls required for amortization of the loading-overhead depends on the processor and OS configuration. All benchmarks show a linear relation between the executed work and the execution time.

I have further analyzed the CPU time using linear models displayed in Table 5.4. AnyCall has the most benefit on the Intel i5-6260U with mitigations active. Here, the initial overhead of AnyCall is 19.2 ms to prepare and load the BPF program into the kernel. Thereafter, the implementation using traditional system calls requires 176.2 µs for 200 `getpid()` calls, while AnyCall uses 3.5 µs, it is therefore 98 % faster. In the system configurations without KPTI, AnyCall is 91.8 % to 92.6 % faster.

To analyze *how* the reduced system-call rate affects the execution time, I have recorded the values of hardware performance counters. The number of instructions per kernel call (i.e., system calls for `sys`) is reduced significantly by AnyCall (i.e., by 90.4 % to 90.7 %). I therefore conclude that the speedup in systems without KPTI is mostly due to a reduced number of instructions. On the system with KPTI active (i.e., Intel i5-6260U booted with `mitigations=on`), measuring the number of instruction and data address TLB misses reveals that each traditional system call triggers at least one miss while kernel calls in the `anycall()` trigger none. This is not the case in the other system configurations, therefore I conclude that the additional speedup on this system is a result of the reduced TLB misses.

Using the wall time in the analysis (i.e., `perf`'s `duration_time`) leads to the same conclusions but increase the variance when the caches are cold. As mentioned in Section 5.1, the processor frequency is fixed to the base frequency. I have also executed this microbenchmark with DVFS

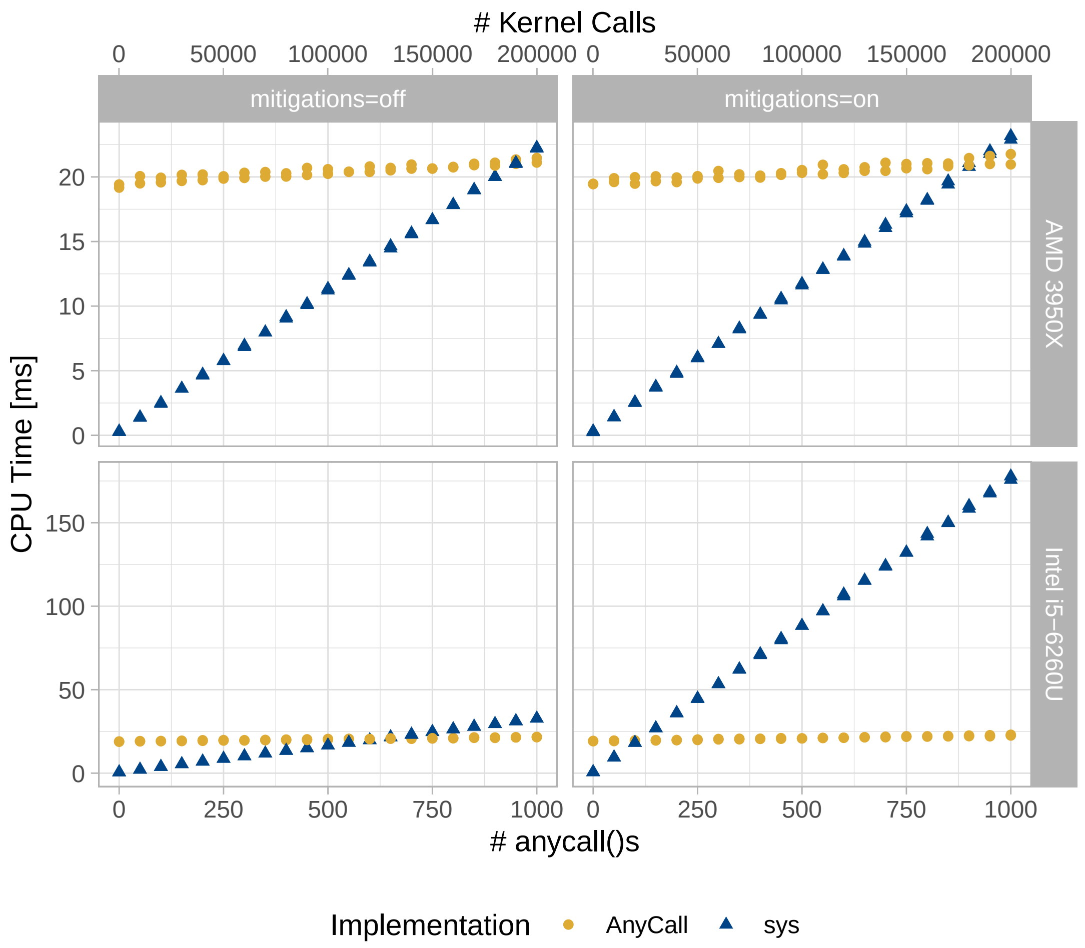

Figure 5.1 – CPU time for a program invoking an `anycall()` 0 to 1,000 times. Each `anycall()` aggregates 200 `getpid()` kernel calls. The `sys` implementation executes the equivalent number of traditional system calls from user space. Note the different y axis scales between the facet rows.

Table 5.4 – Linear models fitting the `getpid()` benchmark data (displayed in Figure 5.1) using the method of least squares. The parameter x is the number of `anycall()`s performed (or equivalent number of traditional system calls for `sys`). f gives the predicted CPU time and x^* the number of `anycall()`s after which the loading overhead is amortized (multiplying it by 200 gives the number of system calls after which loading is amortized). The difference in slope is given by Δ_{max}.

Processor	mitigations	$f_{\text{AnyCall}}(x)$	$f_{\texttt{sys}}(x)$	x^*	Δ_{max}
AMD 3950X	`off`	19.6 ms + $x \cdot$ 1.63 µs	301 µs + $x \cdot$ 21.9 µs	951	92.6 %
	`on`	19.5 ms + $x \cdot$ 1.77 µs	283 µs + $x \cdot$ 22.7 µs	919	92.2 %
Intel i5-6260U	`off`	18.9 ms + $x \cdot$ 2.64 µs	571 µs + $x \cdot$ 32.2 µs	622	91.8 %
	`on`	19.2 ms + $x \cdot$ 3.53 µs	577 µs + $x \cdot$ 176 µs	108	98.0 %

enabled or the processor pinned to its maximum and minimum frequency, however, the conclusions of this section are not affected by this change.

5.2.2 Number of Kernel Calls per `anycall()`

The previous experiment varied the number of `anycall()`s and always performed the same number of kernel calls in every `anycall()`. The complementary experiment, that is, vary the number of kernel calls performed per `anycall()`, shows almost identical results. Therefore, a plot is omitted in this thesis as the results for the Intel i5-6260U with `mitigations=on` can also be found in [Ger+21].

5.3 Vector `anycall()`s

This section presents the performance of `anycall()`s more complex than the `getpid()` `anycall()`. It contributes to answering *Q1*, *Q3*, and *Q4*. By comparing the results from this experiment to the `getpid()` microbenchmark, *Q2* is answered.

5.3.1 Motivation

In real-world applications, it is common to execute the same system call repeatedly on different data [Lin20d]. Using AnyCall, one can easily create such *vector* versions for arbitrary system calls, even incorporating user-defined error handling. To demonstrate this, I have created vector versions of the `open()` and `close()` system calls. The vector `open()` `anycall()` creates a requested number of unnamed temporary files and stores the file descriptors into an array (Opening named files is also possible by passing an array of strings to the `anycall()`). The vector `close()` `anycall()` receives an array of file descriptors and executes `close()` for each. In comparison to the `getpid()` experiment, more memory is accessed in kernel and user space.

In this benchmark, the program therefore loads two `anycall()`s and invokes each `anycall()` 0 to 1,000 times (each invocation performs 200 kernel calls). Each test program therefore executes twice as many `anycall()`s as in the previous experiment, however, the number of times each loaded BPF program is reused is the same.

5.3.2 Results

Figure 5.2 displays the aggregated execution time of the vector `open()` and `close()` `anycall()`s and compares it to the equivalent implementation using traditional system calls. I analyze the data using linear models which are displayed in Table 5.5. Depending on the system configuration, the initial overhead to load the two BPF programs is 25.0 ms to 30.0 ms.

On the Intel i5-6260U with `mitigations=on`, the execution time for the two `anycall()`s increases by 0.74 ms with each processed file while the runtime of the variant doing traditional system calls increases by 1.16 ms with each file. In the system with KPTI active, the two `anycall()`s are therefore 36.4 % faster. On the systems without KPTI, the speed up is still significant as the CPU time per file is reduced by 12.6 % to 15.4 %. In comparison to my `getpid()` experiment the CPU-time difference is smaller, as the user/kernel transition overhead dominates the `getpid()` execution time. However, the number of calls required to compensate for the loading overhead is reduced because more time is saved per system call. Depending on the system configuration, $1.30 \cdot 10^4$ to $6.61 \cdot 10^4$ system calls suffice to justify the use of AnyCall.

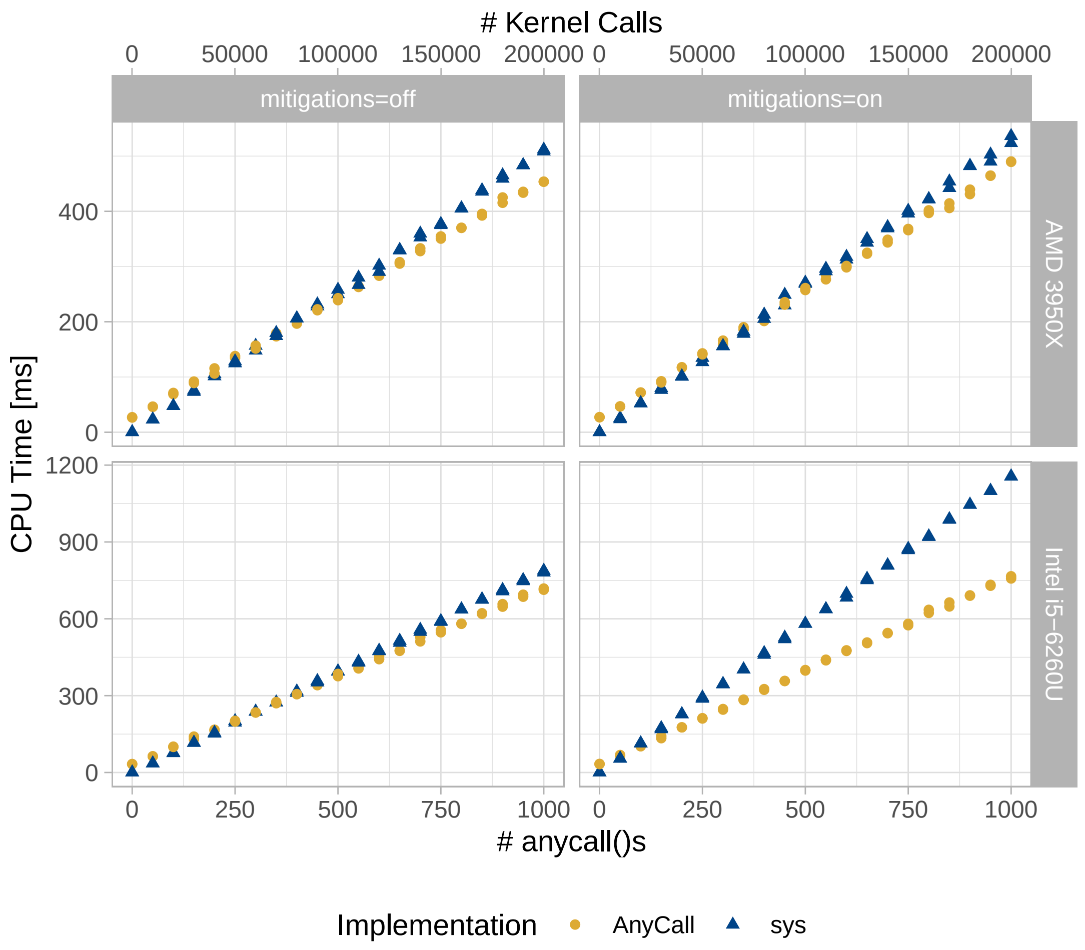

Figure 5.2 – CPU time for two `anycall()`s that are *each* invoked 0 to 1,000 times. Each `anycall()` aggregates 200 kernel calls which either open or close temporary files. The `sys` implementation opens and closes the equivalent number of temporary files from user space. Note that the x axis in this benchmark gives the number of `anycall()`s or kernel calls *per* loaded `anycall()`, therefore the total numbers of `anycall()`s or kernel calls are twice as high.

Table 5.5 – Linear models fitting the vector benchmark data (displayed in Figure 5.2). The parameter x is the number of files processed (or equivalent number of traditional `open()` and `close()` calls for `sys`). f gives the predicted CPU time and x^* the number of `anycall()`s required to amortize for loading *each* BPF programs. Δ_{max} gives the maximum amount by which AnyCall if faster if x is large.

Processor	mitigations	$f_{\text{AnyCall}}(x)$	$f_{\text{sys}}(x)$	x^*	Δ_{max}
AMD 3950X	`off`	$25.9\,\text{ms} + x \cdot 432\,\mu\text{s}$	$1.64\,\text{ms} + x \cdot 511\,\mu\text{s}$	309	15.4 %
	`on`	$25.0\,\text{ms} + x \cdot 460\,\mu\text{s}$	$1.56\,\text{ms} + x \cdot 531\,\mu\text{s}$	331	13.4 %
Intel i5-6260U	`off`	$30.0\,\text{ms} + x \cdot 692\,\mu\text{s}$	$1.85\,\text{ms} + x \cdot 792\,\mu\text{s}$	282	12.6 %
	`on`	$29.0\,\text{ms} + x \cdot 737\,\mu\text{s}$	$1.59\,\text{ms} + x \cdot 1.16\,\text{ms}$	65	36.4 %

Again, I have recorded the values of hardware performance counters for each test run to analyze how the system-call rate affects the performance. In contrast to the getpid() benchmark, AnyCall does not significantly affect the number of instructions for vector anycall()s. They are only reduced by 9.5 % to 15.1 %, which is expected as more code executes between the user/kernel transitions. Still, this explains a significant portion of the CPU-time reductions achieved on all systems. Another reason is possibly the reduction in L1 instruction cache load misses (but not L1 data cache load misses). On the KPTI-system, I expect the additional performance gains to be a result of the significant reduction in TLB misses (by 92.1 % to 99.8 %).

5.4 Real-World Benchmarks

This section presents two real-world applications I have sped up using AnyCall. Section 5.4.1 presents a tool that searches files by their magic values and Section 5.4.2 presents the evaluation of a tool for estimating disk usage. This section addresses the research questions *Q5* and *Q6*.

5.4.1 File Searching

Many file types use *magic values* at predefined offsets for identification. To demonstrate that AnyCall can speed up a real-world application, I have applied it to a tool that filters files by such magic values. For this tool, the list of files is received on standard input (generated by `find -type f`) and each file is opened, `seek`-ed, read, and closed. If the contents read from the offset match the magic value, the file path is written to standard output. In total, I have implemented four variants. One uses `anycall()`s and three use traditional system calls:

`libc` checks the file contents using buffered I/O based on C-library `FILE` pointers.

`sys` checks the file contents using raw traditional system calls. Like `libc`, it processes one file path at a time.

AnyCall To allow for a larger BPF program it is beneficial to read in a chunk of file paths and prepare an array of zero-terminated strings to be passed to the BPF program. The BPF program checks each path in the chunk using `open()`, `lseek()`, `read()`, and `close()`. The AnyCall BPF program conditionally calls `write()` to print matches.

`sys-burst` executes the same algorithm as the AnyCall implementation but uses repeated traditional system calls.

The algorithms executed by the AnyCall and `sys-burst` implementations are covered in greater detail in Section 4.7.2.

To evaluate these implementations I run them in bursts of 10 and repeat each test burst 10 times to analyze the run-to-run variance. From every burst, I only analyze the first and last iteration which execute with cold and hot caches respectively. In contrast to the microbenchmarks from the previous section, this benchmark involves I/O and may use multiple threads. Therefore, I report the wall time, not the CPU time. (An evaluation of the CPU time for the Intel i5-6260U with hot caches and KPTI active is found in [Ger+21].) The directory traversal by `find` and the processing of the input is always performed in user space and included in the execution time. The best chunk sizes for AnyCall and `sys-burst` were empirically determined to be 512 and 1024. Chunk sizes above 512 were not possible for AnyCall, because the BPF program became too large for static analysis. However, with hot caches and KPTI active, `sys-burst` is already outperformed if the chunk size is set to 4. This translates to only 24 kernel calls per `anycall()` [Ger+21]. In the following I always compare AnyCall to the fastest traditional implementation (i.e., `sys`) using the median, because I assume outliers are a result of system noise.

Figure 5.3 displays the runtime when the Linux v5.0 sources are checked for files with the `/bin/sh` shebang using the different implementations. Even though the AnyCall implementation issues only a single `anycall()`, it outperforms the fastest traditional implementation by 31.8 % on the Intel i5-6260U with hot caches and KPTI active. However, in the other configurations the AnyCall implementation is not able to amortize for the loading overhead, causing the median to be 0.2 % to 12.8 % higher. (The AMD 3950X with cold caches and `mitigations=off` is the only other exception, here AnyCall is insignificantly faster by 0.8 % / 48.2 ms).

Notably, AnyCall is slower than `sys` on the system with KPTI active if the caches are cold. I assume that this is caused by the frequent I/O operations which causes the CPU to enter sleep modes frequently, thereby flushing its caches. These flushes void the improved cache usage AnyCall

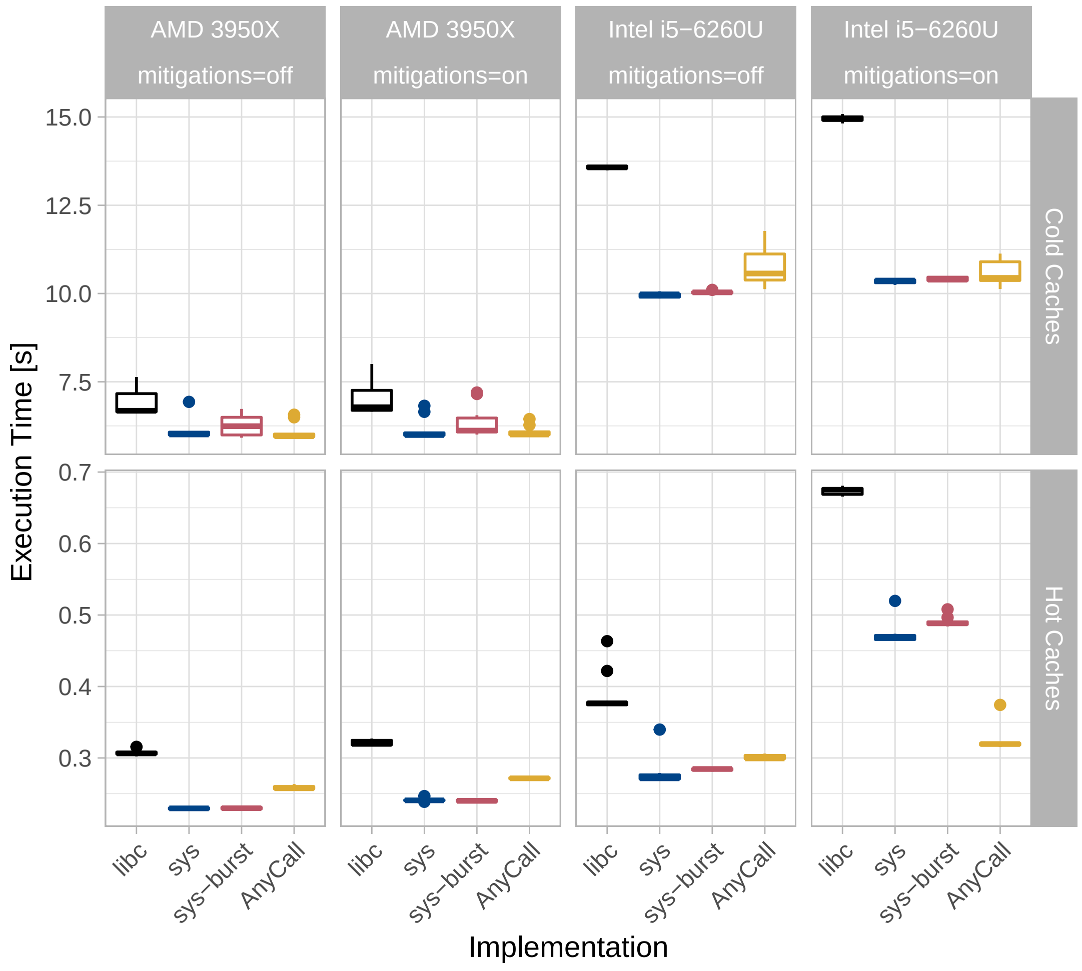

Figure 5.3 – Execution time to search files with the `/bin/sh` shebang in the Linux source tree. The implementation using AnyCall outperforms the other implementations on the Intel i5-6260U if the caches are hot and KPTI is active.

enables. In detail, blocking I/O operations suspend the calling thread when it performs the kernel call, therefore flushing the caches similarly to processor mode switches. This causes AnyCall to perform worse if the caches are cold. However, I observe that this effect is larger on the Intel machine which also uses a slower SSD. I therefore assume that faster I/O devices yield better performance for AnyCall, as it is the case on the AMD machine.

In addition, I expect that the I/O access patterns AnyCall and `sys-burst` use are not optimal, thereby explaining why they often perform worse than `sys` (especially on the Intel machine with cold caches where they are slower by up to 590.9 ms). In detail, the `sys` implementation may be able to hide a part of the I/O latency by alternating reads from standard input, file-system accesses, and writes to standard output. `sys-burst` and AnyCall however, perform both operations in bursts of 512 to 1024 paths. This thesis is supported by the fact that `sys-burst` consistently performs better than `sys` if they are compared by CPU time instead of wall time.

5.4.2 Disk Usage Estimation

When systems run low on available disk space, it is common for administrators to estimate the disk usage of certain directory trees in order to decide whether they need cleanup. I have implemented two versions of such a tool, one using traditional system calls, the other using `anycall()`s. In addition, I compare both to the standard GNU implementation of POSIX's `du -s`. I run these tools on a part of the Poky Linux distribution build tree containing $1.09 \cdot 10^6$ files in $4.62 \cdot 10^5$ directories.[10] The burst size and the number of repetitions for each test is the same as in the previous section (10 repetitions; a burst size of 10; only the first and last run from each burst are analyzed). The first run executes with cold caches and is therefore I/O-bound. The last run is CPU- and memory-bound as both evaluation systems have enough random-access memory (RAM) to fit the entire directory structure. I compare AnyCall to the fastest traditional implementation (here `sys`) using the median, because I expect that variation is caused by system noise.

Section 5.4.2 displays the execution time for each implementation in the different system configurations. AnyCall outperforms the fastest user space implementation (i.e., `sys`) in all configurations but one (that is, the Intel machine with cold caches). Again, I expect that this is due to the frequent idle periods the slower SSD triggers. The GNU implementation always performs worse than the smaller `sys` and AnyCall implementations. As in the previous experiment, the most notable speedup is achieved on the Intel machine with hot caches and KPTI active. Here AnyCall is 40 % faster. In the other configurations the speedup is more moderate, being 0.7 % to 10.1 %. In all configurations but the one with KPTI and hot caches, the difference between having mitigations active and inactive is small (e.g., for `sys` only 3.3 % to 5.0 %).

5.5 Summary

This section summarizes the results of this evaluation, explicitly answering the research questions:

Q1 Section 5.2 and Section 5.3 answer this question for two different workloads: To be faster than the equivalent implementation using traditional system calls (i.e., to amortize for the overhead of loading and verifying the BPF program), AnyCall has to perform $1.30 \cdot 10^4$ to $1.90 \cdot 10^5$ kernel calls per loaded BPF program on the systems used in this evaluation.

[10]To prevent excessive benchmarking overhead, I do not transfer a full copy of the directory tree to the evaluation systems, but instead a shallow copy (i.e., the files are all empty). This does not affect the benchmark as the `du -s` implementations still have to check each individual file. Specifically, I transfer all files or directories that either match `find . -maxdepth 15 -type d` or `find . -maxdepth 15 -type f -links 1` in Poky `honister`'s `build/tmp/work` directory.

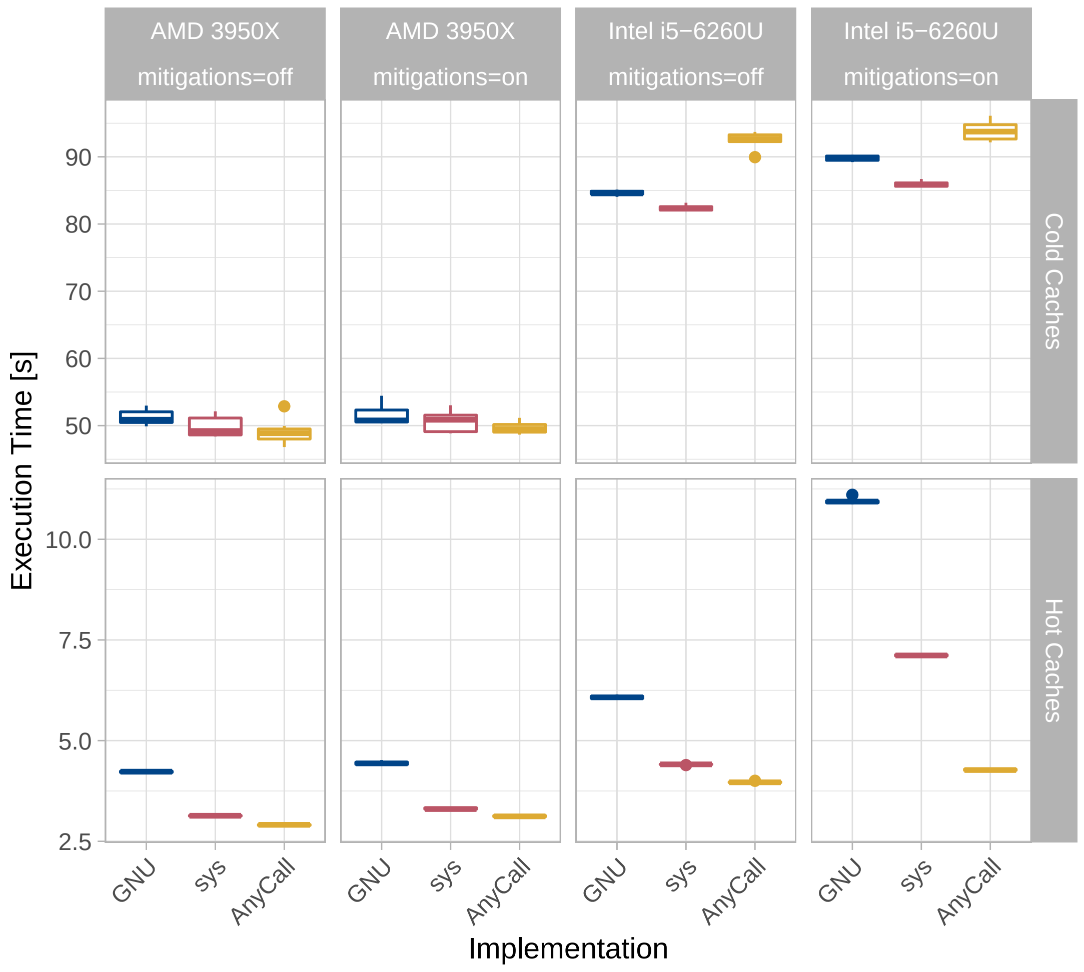

Figure 5.4 – Execution time to estimate the disk usage of the Poky Linux distribution build tree using different implementations. AnyCall outperforms the other implementations in all system configurations expect the Intel machine with cold caches.

Q2 By comparing the results from Section 5.2 and Section 5.3, I conclude that the absolute amount of time saved per `anycall()` increases as more data is accessed inside and around the kernel calls. Therefore, the amortization time decreases if the amount of data accessed by the kernel or the user increases. This can be explained by the fact that AnyCall has a greater potential for reducing the number of cache misses if more data is accessed between the user/kernel transitions. In contrast, the relative performance improvement is larger, if little code executes in and around the kernel calls, because then, the runtime is dominated by the user/kernel transitions.

Q3 The processor architectures used in my evaluation (these are Intel 6th generation Skylake and AMD Zen 2) do not significantly influence the number of kernel calls required to amortize for the overhead of loading AnyCall BPF programs.

Q4 I have analyzed how the system configuration influences the performance improvements AnyCall achieves. First, systems with KPTI active benefit from AnyCall significantly in both microbenchmarks (up to 98 % faster) and real-world benchmarks (31.8 % to 40 % faster). Aside from this, the processor architecture and generation, as well as the processor frequency do not significantly impact AnyCall's performance characteristics in my benchmarks.

Q5 I have demonstrated that real-world applications frequently perform system calls at high rates, enough to justify the use of AnyCall.

Q6 CPU- and memory-bound benchmarks benefit most from AnyCall in my evaluation. I/O-bound workloads also benefit if the accessed device is fast (e.g., the SSD of the AMD machine). Systems with slower I/O devices (e.g., the SSD of the Intel machine) do not benefit from AnyCall in my benchmarks, likely because the frequent transitions to processor C-states flush the caches.

To summarize, AnyCall achieves significant speedups across a diverse set of system configurations and benchmarks.

RELATED WORK 6

This section presents AnyCall's related work. These use three different approaches to solving the problem of slow user/kernel transitions. First, Section 6.1 covers approaches that avoid the transitions in the first place, for example using shared memory and multicall interfaces. Second, Section 6.2 covers microkernels, which still perform frequent user/kernel transitions under hardware isolation, but attempt to make the transitions as cheap as possible. Finally, I cover systems using software-based isolation to allow for cheap user/kernel transitions. This also includes AnyCall, which Section 6.3 compares against SPIN, Singularity, Cosy, [KS15], RedLeaf, and BPF-controlled io_uring.

6.1 User/Kernel-Transition Avoidance

In special cases, AIO APIs, multicall interfaces, or vDSO can be used to avoid user/kernel transitions. This section presents them in the subsections 6.1.1, 6.1.2, and 6.1.3.

6.1.1 io_uring

io_uring submits tasks to (and receives results from) the kernel asynchronously using shared memory queues [SS10; Cor19b; Axb19]. By handling these submissions concurrently on another core, the system can completely avoid user/kernel transitions on the application core. Still, if there are many control flow or data dependencies between the system calls, this approach becomes impractical as the user and kernel core have to communicate and wait for each other frequently. This is problematic as even a minimal cross-core call and reply can require four cache-line transactions, taking 448 to 1988 cycles (depending on wether the cores are on the same or different sockets) [Nar+19]. AnyCall solves this by allowing for low-overhead communication between user and kernel code on a single CPU core. My evaluation has demonstrated that this is efficient even if there are many data-dependencies between the system calls (in particular, Section 5.4.2).

Further, using io_uring efficiently requires a programming model many developers are not familiar with [Atl+16]. Recent projects to integrate io_uring into modern programming languages are promising [Ler+21], but even if successful they still require rewriting the application code to use `async/await`. AnyCall, in contrast, has the potential of being completely transparent to the user if it is integrated into the compiler.

Finally, to support a specific system call in io_uring, the system call's implementation first has to be modified to support asynchronous invocation. As a consequence, io_uring currently only supports a subset of the available Linux system calls. AnyCall, in contrast, supports every hardware-unspecific system call without having its implementation modified.

6.1.2 Xen Multicall

Multicall interfaces as implemented by the Xen hypervisor are an alternative to AIO that are easier to implement but also limited. Consequently, they suffer from the same constraints as io_uring in that they are inefficient if there are tight data and control flow dependencies between the system calls. In practice, they are therefore limited to batched page-table updates and networking hardware control [Pan+11]. AnyCall, in contrast, can still be used even if there are data and control-flow dependencies between the aggregated system calls.

6.1.3 Virtual Dynamically-linked Shared Object (vDSO)

In Linux, some system calls only read a small amount of information. They can be implemented using vDSO [Lin21c], where the data is mapped into user space, making it directly readable without a processor mode switch. However, vDSO is limited to read-only data for security reasons. AnyCall in contrast is not limited in this way. It can be used with arbitrary system calls, including those that write to kernel memory.

6.2 Microkernels

An alternative to avoiding user/kernel transitions is to make them as cheap as possible while still using hardware for isolation. Many microkernels, in particular seL4 [EH13], take this approach as IPC is naturally more performance critical to them than for monolithic systems. As microkernels implement many systems services as user processes, invoking them not only requires switching to the kernel but also to another user process. This in turn allows the OS to be kept as minimal as possible (e.g., kernel extensions should never be required as the OS does not implement any policies). Minimality as a main driver of microkernel design also helps in reducing the IPC overheads, however, it is ultimately still limited by the hardware. In 2013 seL4 was achieving one-way IPC between two processes in 188 to 316 cycles while entering and leaving the kernel today (without switching the process) still takes 96 to 140 cycles on modern x86 hardware. And even in a hypothetical scenario, where the address space switch consumes no cycles at all, hardware-based isolation is still outperformed by software-based isolation which can reduce the overhead to a subroutine call [Li+21].

6.3 Software-Based Isolation

Most operating systems isolate processes using hardware mechanisms, for example, using the MMU to run them in different address spaces. However, recent advances in compilers and verification make it feasible to reconsider this approach [Nar+20]. This section compares AnyCall against other systems using specification and verification in software to isolate applications sharing an address space.

An overview similar to this one can also be found in [Nar+20]. This section contributes a discussion of Cosy, [KS15], and BPF-controlled io_uring. Further, each system is compared against AnyCall. [Nar+20] in comparison adds a discussion of J-Kernel [Eic+99] and KaffeOS [BH05].

Table 6.1 gives an overview over the systems covered by this section. Section 6.3.1 covers SPIN [Ber+95], Section 6.3.2 covers the Singularity [HL07] project, and Section 6.3.3 covers Cosy [Zad+05] as well as a LuaJIT-based system [KS15] by Koomsin and Shinjo. Section 6.3.4 compares AnyCall to RedLeaf [Nar+20] which is a recent system employing Rust. Finally, Section 6.3.5

compares AnyCall and BPF-controlled io_uring [Beg21], the latter being currently integrated into the upstream Linux kernel.

6.3.1 SPIN

SPIN [Ber+95] is an extensible OS which allows applications to change the OS interface and implementation by loading extensions at runtime. Both the kernel and the untrusted extensions are written in type-safe Modula 3 and compiled by a trusted compiler (e.g., to prevent the extensions from using Modula 3's `UNSAFE` and `LOOPHOLE` directives to escape from their isolation module). As a consequence, isolation is guaranteed at compile-time but not at load-time. The execution time of the extensions is limited using interrupt-driven preemption. Memory safety is achieved using types and garbage collection. SPIN does not offer fault isolation as the state of one extension can be passed to, and directly accessed by, other extensions using shared pointers. If one extension crashes, its shared data may therefore be left in an inconsistent state allowing the fault to propagate [Nar+20].

Comparing SPIN to AnyCall, the most notable difference is that SPIN's extensions execute completely under software-isolation while AnyCall only executes (in itself Turing-incomplete) subroutines of each application using software-based isolation. Further, AnyCall offers fault isolation and supports multiple programming languages (through LLVM [Sta14]) without requiring a trusted compilation environment while SPIN does not isolate faults and requires trusted compilation by a Modula 3 compiler. Regarding memory safety, SPIN uses garbage collection while AnyCall relies on static analysis to guarantee memory reclaim.

6.3.2 Singularity

Singularity [HL07; Nig+09] was started in 2003 with the primary goal of improving systems dependability. Better performance was only a secondary goal. To achieve this, Singularity executes programs compiled to type-safe (possibly virtual) instruction sets (e.g., Microsoft's CLR bytecode, typed assembly language was also considered but not implemented) in VMs. Singularity's processes are thereby isolated using verification and specification in software, instead of using hardware checks at runtime. For example, they can share an address space with each other or even with the kernel, without interfering with each other. Singularity therefore calls them software-isolated processes (SIPs).

Allowing SIPs to share an address space enables very fast IPC. To still isolate them from each other, their code is verified and JIT compiled (the latter helps in preventing hardware exceptions

Table 6.1 – Comparison of systems using software-based isolation for fast user/kernel transitions. They differ with regard to the runtime they build on (*foundation*), the mechanisms used for memory safety, fault isolation, and whether they require a trusted compilation environment. Cosy is excluded from the table, because the paper does not clearly state whether memory safety or fault isolation are *guaranteed*, or only supported.

System	Foundation	Memory Safety	Fault Isolation	Untrusted Compilation
SPIN	Modula 3	garbage collection	✗	✗
Singularity	CLR	garbage collection	✓	✓
[KS15]	LuaJIT	garbage collection	✓	✓
RedLeaf	Rust	static analysis	✓	✗
AnyCall	BPF	static analysis	✓	✓

by inserting checks into the machine code). Communication exclusively happens through typed channels that are synchronized by the OS. This allows Singularity to guarantee fault isolation using statically enforced ownership semantics [Nar+20].

The design of Singularity allows for multiple user-process runtimes to co-exist in one system. The standard runtime would be Turing-complete and run bytecode in a VM that uses garbage collection to reclaim dynamic allocations (i.e., Microsoft's CLR). In addition, the design considers more constrained domain-specific execution environments that allow for less complex and hence more efficient memory management mechanisms, avoiding the overhead of garbage collection. In retrospective, an example for such a runtime could have been BPF although the paper does not discuss this. A third option considered (but also not implemented) was to fall back to hardware-based isolation if code not written in a type-safe language must run on Singularity. This would allow running legacy applications (e.g., written in C/C++) in their own address space under hardware isolation. To summarize, Singularity's design allows for a diverse set of execution environments, each optimal for different kinds of applications.

In comparison to AnyCall, Singularity offers Turing-completeness while AnyCall intentionally does not. I argue that this is not a problem for AnyCall, as my evaluation has shown that complex applications can still be implemented using it. As second major difference is the reliance on garbage collection in Singularity [Emm+19].

6.3.3 System-Call Aggregation

This section presents two approaches, Cosy [Zad+05] and a LuaJIT-based system [KS15]. Both focus on system-call aggregation similarly to AnyCall.

Cosy [Zad+05] was created with the goal of allowing for the efficient and safe execution of user-level code in the kernel, thereby reducing the isolation-overhead. They use their system to create both more efficient, composed, system calls but also to run sections of an application in the kernel. While AnyCall uses BPF, they use a custom runtime environment called Cosy. Cosy uses in-kernel preemption to limit the user's CPU time while executing in the kernel. AnyCall primarily uses static analysis to limit the runtime of the BPF program, but in addition also uses interrupt-driven, in-kernel preemption if the respective Linux configuration option is active. Cosy uses x86 segments for memory safety which is no longer available on x86-64. It therefore remains unclear if Cosy could be ported to modern hardware. AnyCall in contrast supports modern hardware, works with KASLR (which is not clear for Cosy), and reuses an existing execution environment that has already been integrated into Linux.

The system created by Koomsin and Shinjo in 2015 reuses the LuaJIT runtime to execute system-call *scripts* in the kernel (these scripts effectively aggregate multiple system calls). It therefore ports a runtime for user application extensions into the OS kernel. AnyCall in contrast uses a bytecode executor created from scratch for use by the kernel. Due to the existing LLVM backend [Sta14], AnyCall supports various programming languages while [KS15] only supports Lua. To access user memory, Koomsin and Shinjo use subroutines similar to the copy helpers described in Section 4.5.1. AnyCall contributes the page-fault and pinning-based access methods to the discussion. While AnyCall supports more programming languages, [KS15]'s scripts may in some cases be easier to use as the code must not be statically analyzed but can rely on garbage collection and runtime checks for safety. [KS15] also supports AnyCall as they evaluate their approach using Memcached and thereby show how system-call aggregation can speed up database server workloads. This suggests, that evaluating AnyCall using Memcached is a promising subject for future work.

6.3.4 RedLeaf

SPIN, Singularity, and [KS15] all use garbage collection to safely reclaim dynamic allocation. RedLeaf [Nar+20] in contrast offers memory safety without relying on garbage collection, but instead uses the ownership model of the Rust programming language. The compiler checks the model statically at compile-time, therefore it has no direct runtime-overheads. RedLeaf further uses mechanisms similar to Singularity's typed channels to guarantee strong fault isolation. Among SPIN, Singularity, Cosy, and [KS15], it is the only system that was published after the Meltdown and Spectre vulnerabilities had been published. Like AnyCall, RedLeaf considers solving this to be a subject of future work. Using Rust's ownership model to ensure memory safety is conceptually similar to the static analysis by the BPF verifier on which AnyCall relies. Like SPIN, RedLeaf's implementation requires a trusted compilation environment, AnyCall however does not. RedLeaf could resolve this by adding support for typed assembly language or proof-carrying code (PCC) to Rust, however, I argue that it is unclear whether this is possible. While BPF bytecode was designed to be quickly analyzable by a low-complexity verifier, Rust's ownership model was created with the assumption that it will be checked by regular compilers. For these, having a small implementation and low latency is not critical, because the compiler usually executes in user space on good-performing desktop workstations and build-servers.

6.3.5 BPF-Controlled io_uring

BPF-controlled io_uring [Beg21; TDS21] was developed in parallel to AnyCall by the Linux io_-uring maintainers. AnyCall's design is aware of it, aiming to contribute and evaluate alternative approaches instead of replicating features that are already being integrated into the upstream Linux kernel. In particular, Section 3.2 and Section 4.5 are motivated by this. Like AnyCall, BPF-controlled io_uring executes user code in the kernel using BPF. However, instead of calling regular system-call implementations from within the BPF program, the BPF programs execute in response to the completion of asynchronous io_uring system calls. When invoked, the BPF handler program can then request subsequent asynchronous system calls using dedicated BPF helper functions.

Comparing AnyCall to BPF-controlled io_uring, AnyCall uses a call-oriented execution model while BPF-controlled io_uring's execution model is return-oriented. AnyCall does not require familiarity with AIO, which has shown low adoption rates in the past [Atl+16]. In addition, AnyCall supports any Linux system call while BPF-controlled io_uring is limited to the operations supported by io_uring. Summarizing these aspects, AnyCall focuses on straight-forward usability while BPF-controlled io_uring focuses on high-performance computing.

Regarding performance, AnyCall uses synchronous I/O while BPF-controlled io_uring is based on asynchronous I/O. The latter may be beneficial if used properly as previous work has demonstrated that io_uring can achieve remarkable performance improvements, even without BPF-control. However, being return-oriented, BPF-controlled io_uring scatters the control-flow and state-management of a call chain across multiple BPF programs. This not only complicates development, but may also hurt performance as the state has to be communicated using IPC instead of being directly accessible on the BPF stack. Invoking a new BPF program may also be slower than returning from a subroutine in an already-running BPF context (i.e., as it happens after kernel calls in AnyCall).

Having one BPF program instead of multiple is also related to the interfaces for user memory access BPF-controlled io_uring and AnyCall offer. Both offer an interface based on copy-helpers (Section 4.5.1), but AnyCall in addition offers pinning and page-fault-based interfaces (although the latter is not implemented, see Section 4.5.2 and Section 4.5.3). It is unclear whether these two can be supported in BPF-controlled io_uring as the completion of a system call starts a new BPF program instead of returning to an existing context. To safely reuse mappings across system calls, the verifier would have to prove that the mappings are cleaned up by other programs. If possible at all, this is significantly more complex than the checks required for AnyCall as passing the mappings between the programs involves IPC.

To summarize, BPF-controlled io_uring is based on AIO while AnyCall retains the synchronous programming model many developers are familiar with. While the latter is easier to use and supports all Linux system calls, io_uring offers additional performance benefits but also overheads as multiple BPF programs have to share state and communicate. Comparing the two is therefore a major subject of future work. This work may also evaluate the potential for using io_uring (without BPF-control) from within AnyCall (e.g., submitting AIO requests from within the AnyCall BPF program running in processor kernel mode).

7 FUTURE WORK

Throughout this thesis I have identified numerous opportunities for future work. These include implementing mechanisms to speed up the loading and verification of `anycall()`s, and to make them accessible to untrusted processes. As hardware characteristics change, the implementation of the BPF bytecode executor can be optimized without changing AnyCall's ABI. To further reduce system-call overheads, AnyCall can be extended to avoid copies between user and kernel memory. Finally, related work motivates evaluating AnyCall using server and database workloads (e.g., Memcached) and comparing it against BPF-controlled io_uring.

Chapter 5 has demonstrated that the main overhead associated with using AnyCall is the time and energy spent verifying and JIT-compiling the BPF bytecode. To avoid this overhead when a program is loaded repeatedly (e.g., by different processes executing the same binary), the kernel can cache the hashes of BPF programs passing verification [Hei21]. When the same program is loaded a second time, the kernel then is able to skip verification. If remembering the hashes for verified programs is considered too memory-intensive, the kernel could alternatively hand out cryptographic signatures upon having verified a program.

BPF programs can currently only be loaded by privileged processes, as current processors do not prevent side-channel attacks on the kernel from within the BPF VM. To still make AnyCall usable to untrusted processes, a privileged systems service can load common library `anycall()`s (e.g., `sendfile()`) and share descriptors with untrusted applications upon request [Cor21b; Zad+05]. It may even be possible to verify that certain BPF programs (e.g., those having a simple structure, only aggregating a fixed number of system calls into a multicall) can never be used for side-channel attacks on the kernel. If this is the case, a systems service can load dynamically generated BPF library programs for unprivileged applications on demand.

AnyCall's design is orthogonal to the in-kernel VM implementation used to execute the aggregation code under software-based isolation. Adopting it to new hardware characteristics is therefore possible without changing the ABI. If switching the processor to user mode is possible with little overhead on an architecture, BPF can be changed to relax the static analysis and instead rely on hardware-based isolation on these systems. Similarly, if future architectures offer a processor mode protecting against side-channel attacks that has smaller overheads than today's user mode, BPF can run unprivileged programs in this mode. This would allow unprivileged processes to safely load BPF programs without putting the system at risk.

AnyCall eliminates frequent user/kernel transitions from user applications, however, data is still frequently copied between user and kernel memory areas. To further improve AnyCall's performance [KA21], a mechanism can eliminate these copy operations by taking advantage of the guarantees from static analysis. All system calls include checks to verify that supplied pointers refer user memory (which can only be accessed directly by the kernel in special cases), therefore changing those widely scattered checks would be impractical. However, most system calls reuse the same kernel-internal libraries to perform the security checks, therefore changing them may be possible using only few modifications to the kernel.

Finally, related work has demonstrated that AnyCall may also speed up server and database workloads. For example, Koomsin and Shinjo [KS15] have found that a similar system speeds up Memcached significantly. Future work will therefore evaluate AnyCall using Memcached (or another database server) and also compare it to BPF-controlled io_uring to better understand the trade-offs between the two designs.

CONCLUSION 8

General-purpose operating systems (OSes) must isolate user processes for fault tolerance, to confine malicious actors, and to maintain privacy. For this, they typically rely on hardware-based isolation to run user applications in an unprivileged processor mode, and to confine them to their own virtual address spaces. To still allow the processes to communicate with each other and perform input/output (I/O), OSes offer *system calls*. Recently, system-call overheads have been increasing due to multiple factors. Caused by high-performance I/O hardware and large memories, system-call rates in applications are increasing. At the same time, the per-call overheads are increasing because of larger hardware buffers and caches, as well as the continued discovery of transient execution vulnerabilities in modern processors (e.g., Meltdown).

To reduce system-call overheads in real-world applications, I propose AnyCall, which leverages software-based isolation to reduce the per-call overheads significantly. For this, AnyCall aggregates system calls and application-specific control logic, and executes as Berkeley Packet Filter (BPF) bytecode in the Linux kernel. By reusing the existing BPF virtual machine (VM), the patch size is kept small as the kernel already includes it for flexible event handling, debugging, and configuration. Further, AnyCall maintains isolation while decoupling the number of user/kernel transitions from the number of system calls. My implementation provides helper functions to access system calls from within BPF, as well as memory management. To determine whether my approach is practical, I port multiple real-world applications to AnyCall and document the required changes. Further, to evaluate its performance, I use compute- and I/O-bound real-world- and microbenchmarks. Each benchmark is executed on two systems using different processor architectures and I/O devices. Using AnyCall, I speed up a real-world disk usage estimation tool on systems without Kernel Page Table Isolation (KPTI) active, if they have fast I/O devices. Here, I demonstrate speedups by 1 % to 10 %. AnyCall is the most beneficial to compute-bound workloads on systems with KPTI active. Using a `getpid()` microbenchmark, I demonstrate that kernel calls inside the AnyCall environment are up to 98 % faster than system calls in user space. I measure that vector `anycall()`s on systems with KPTI active are 36 % faster than the user-space equivalent, and finally, speed up a real-world file searching tool by 32 % and a real-world disk usage estimation tool by 40 %. In conclusion, AnyCall demonstrates that BPF-based system-call aggregation is both an efficient but also practical approach to reduce system-call overheads for real-world user applications.

GLOSSARY

VM program In this thesis: A software-isolated program executing in the kernel's hardware context, but with user privileges. On Linux, for example, such programs are BPF programs.

hardware-based isolation A technique where OS processes are isolated from each other using the memory management unit (MMU) and processor's execution mode.

hardware-isolated process Process isolated from other processes and the kernel using hardware-based techniques. On Linux for example, this includes the processor ring (or execution mode) and the MMU.

implementation-agnostic Synchronous system calls implemented using two user/kernel transitions allow for arbitrary operations in the kernel.

inflexible System calls are inflexible as they can only be composed with overheads.

information-hiding System calls are information-hiding in that the implementation can be changed without changing their interface.

kernel call Invocation of kernel code from within a hardware- or software-isolated user process. Kernel calls are also referred to as system calls when they are performed by hardware-isolated user processes.

kernel context Hardware and software state in which system calls and other kernel routines execute.

kernel space Virtual memory only accessible to the OS kernel.

maintain isolation System calls maintain isolation between user processes.

slow System calls are slow if used frequently, because they involve transitions between user and kernel context.

software-based isolation A technique where OS processes are isolated from each other using specification and verification in software.

software-isolated process Process isolated from other processes and the kernel using software-based techniques. For example, by executing its code in a VM.

subroutine-like programmming model System calls are straight-forward to use, because they have a subroutine-like programming model.

system call Kernel call from within a hardware-isolated user process.

transition overhead Overhead of transitions between user and kernel context in hardware- or software-isolated software components.

user context Hardware and software state in which user applications execute.

user space Virtual memory accessible to user processes.

user/kernel transition Transition between user and kernel context, for example, as part of a traditional system call. Usually includes switching the processor mode and the virtual address space.

LIST OF ACRONYMS

ABI	Application Binary Interface
AIO	Asynchronous Input/Output
API	Application Programming Interface
BPF	Berkeley Packet Filter (extended implementation, also abbreviated eBPF)
cBPF	classic Berkeley Packet Filter (original implementation)
CFG	Control-Flow Graph
CISC	Complex Instruction Set Computer
CLR	Common Language Runtime
CPU	Central Processing Unit
DRAM	Dynamic Random-Access Memory
DVFS	Dynamic Voltage and Frequency Scaling
I/O	Input/Output
IBPB	Indirect Branch Prediction Barrier
IBRS_FW	Indirect Branch Restricted Speculation when Calling Firmware
IP	Internet Protocol
IPC	Interprocess Communication
JIT	Just-in-Time
JVM	Java Virtual Machine
KASLR	Kernel Address Space Layout Randomization
KPTI	Kernel Page Table Isolation
LLC	Last Level Cache
MMU	Memory Management Unit
NIC	Network Interface Controller

NVM	Non-Volatile Memory
NVMM	Non-Volatile Main Memory
OS	Operating System
PCC	Proof-Carrying Code
PCID	Process Context Identifier
POSIX	Portable Operating System Interface
PTE	Page Table Entry
RAM	Random-Access Memory
RSB	Return Stack Buffer
SIP	Software-Isolated Process
SLoC	Physical Source Lines of Code
SSD	Solid-State Drive
STIBP	Single Threaded Indirect Branch Prediction
TAS	TCP Acceleration as an OS Service
TCP	Transmission Control Protocol
TLB	Translation Lookaside Buffer
vDSO	virtual Dynamically-linked Shared Object
VM	Virtual Machine
Wasm	WebAssembly

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

LIST OF ALGORITHMS

REFERENCES

[Adv21] Advanced Micro Devices, Inc. *Transient Execution of Non-canonical Accesses*. Retrieved 2020-09-29. 2021. URL: https://www.amd.com/en/corporate/product-security/bulletin/amd-sb-1010.

[AEA19] Omar Alhubaiti and El-Sayed M. El-Alfy. "Impact of Spectre/Meltdown Kernel Patches on Crypto-Algorithms on Windows Platforms." In: *Proceedings of the 2nd International Conference on Innovation and Intelligence for Informatics, Computing, and Technologies (3ICT'19)*. IEEE, 2019, pp. 1–6. DOI: 10.1109/3ICT.2019.8910282.

[Aik+06] Mark Aiken, Manuel Fähndrich, Chris Hawblitzel, Galen Hunt, and James Larus. "Deconstructing Process Isolation." In: *Proceedings of the 2006 Workshop on Memory System Performance and Correctness (MSPC'06)*. ACM, 2006, pp. 1–10. DOI: 10.1145/1178597.1178599.

[Atl+16] Vaggelis Atlidakis, Jeremy Andrus, Roxana Geambasu, Dimitris Mitropoulos, and Jason Nieh. "POSIX Abstractions in Modern Operating Systems: The Old, the New, and the Missing." In: *Proceedings of the 11th European Conference on Computer Systems (EuroSys'16)*. ACM, 2016. DOI: 10.1145/2901318.2901350.

[ATW20] Nadav Amit, Amy Tai, and Michael Wei. "Don't Shoot down TLB Shootdowns!" In: *Proceedings of the 15th European Conference on Computer Systems (EuroSys'20)*. ACM, 2020. DOI: 10.1145/3342195.3387518.

[AW18] Nadav Amit and Michael Wei. "The Design and Implementation of Hyperupcalls." In: *Proceedings of the 2018 USENIX Annual Technical Conference (USENIX ATC'18)*. Retrieved 2021-05-21. USENIX, 2018, pp. 97–112. URL: https://www.usenix.org/system/files/conference/atc18/atc18-amit.pdf.

[Axb19] Jens Axboe. *Efficient IO with io_uring*. Retrieved 2021-05-21. 2019. URL: https://kernel.dk/io_uring.pdf.

[Axb20] Jens Axboe. *Re: [PATCHSET v2 0/6] io_uring: add support for open/close*. Retrieved 2021-05-22. 2020. URL: https://lwn.net/ml/linux-fsdevel/7324bbb7-8f7b-c0c6-6a45-48b8b77c4be8@kernel.dk/.

[Beg21] Pavel Begunkov. *[RFC v2 00/23] io_uring BPF requests*. Retrieved 2021-10-07. 2021. URL: https://lore.kernel.org/io-uring/cover.1621424513.git.asml.silence@gmail.com/.

[Bel+14] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. "IX: A Protected Dataplane Operating System for High Throughput and Low Latency." In: *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI'14)*. Retrieved 2021-05-21. USENIX, 2014, pp. 49–65. URL: https://www.usenix.org/system/files/conference/osdi14/osdi14-paper-belay.pdf.

[Ber+95] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. "Extensibility Safety and Performance in the SPIN Operating System." In: *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP'95)*. ACM, 1995, pp. 267–283. DOI: 10.1145/224056.224077.

[BH05] Godmar Back and Wilson C. Hsieh. "The KaffeOS Java Runtime System." In: *ACM Trans. Program. Lang. Syst.* 27.4 (2005), pp. 583–630. ISSN: 0164-0925. DOI: 10.1145/1075382.1075383.

[BR18] Ashish Bijlani and Umakishore Ramachandran. "A Lightweight and Fine-grained File System Sandboxing Framework." In: *Proceedings of the 9th Asia-Pacific Workshop on Systems (APSys'18)*. ACM, 2018, pp. 1–7. DOI: 10.1145/3265723.3265734.

[Cil21a] Cilium Authors. *Cilium Version 1.10 Documentation - BPF and XDP Reference Guide*. Retrieved 2021-10-28. 2021. URL: https://docs.cilium.io/en/v1.10/bpf/.

[Cil21b] Cilium Authors. *eBPF Documentation - What is eBPF?* Retrieved 2021-10-29. 2021. URL: http://web.archive.org/web/20211016061305/https://ebpf.io/what-is-ebpf/.

[Cor14] Jonathan Corbet. *BPF: the universal in-kernel virtual machine*. Retrieved 2021-10-26. 2014. URL: https://lwn.net/Articles/599755/.

[Cor19a] Jonathan Corbet. *Reconsidering unprivileged BPF*. Retrieved 2020-11-03. 2019. URL: https://lwn.net/Articles/796328/.

[Cor19b] Jonathan Corbet. *Ringing in a new asynchronous I/O API*. Retrieved 2021-05-21. 2019. URL: https://lwn.net/Articles/776703/.

[Cor21a] Jonathan Corbet. *BPF meets io_uring*. 2021. URL: https://lwn.net/Articles/847951/.

[Cor21b] Jonathan Corbet. *eBPF seccomp() filters*. Retrieved 2021-07-27. 2021. URL: https://lwn.net/Articles/857228/.

[Cor21c] Jonathan Corbet. *Toward signed BPF programs*. Retrieved 2021-05-12. 2021. URL: https://lwn.net/Articles/853489/.

[Dre12] Will Drewry. *dynamic seccomp policies (using BPF filters)*. Retrieved 2021-05-20. 2012. URL: https://lwn.net/Articles/475019/.

[EH13] Kevin Elphinstone and Gernot Heiser. "From L3 to seL4: What Have We Learnt in 20 Years of L4 Microkernels?" In: *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP'13)*. ACM, 2013, pp. 133–150. DOI: 10.1145/2517349.2522720.

[Eic+99] Thorsten von Eicken, Chi-Chao Chang, Grzegorz Czajkowski, Chris Hawblitzel, Deyu Hu, and Dan Spoonhower. "J-Kernel: A Capability-Based Operating System for Java." In: *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*. Ed. by Jan Vitek and Christian D. Jensen. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 369–393. ISBN: 978-3-540-48749-4. DOI: 10.1007/3-540-48749-2_17.

[Emm+19] Paul Emmerich, Simon Ellmann, Fabian Bonk, Alex Egger, Esaú García Sánchez-Torija, Thomas Günzel, Sebastian di Luzio, Alexandru Obada, Maximilian Stadlmeier, Sebastian Voit, and Georg Carle. "The Case for Writing Network Drivers in High-Level Programming Languages." In: *Proceedings of the 2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS'19)*. ACM, 2019, pp. 1–13. DOI: 10.1109/ANCS.2019.8901892.

[ERT19] Pekka Enberg, Ashwin Rao, and Sasu Tarkoma. "I/O Is Faster Than the CPU: Let's Partition Resources and Eliminate (Most) OS Abstractions." In: *Proceedings of the 17th Workshop on Hot Topics in Operating Systems (HotOS'19)*. ACM, 2019, pp. 81–87. DOI: 10.1145/3317550.3321426.

[Ge+19] Qian Ge, Yuval Yarom, Tom Chothia, and Gernot Heiser. "Time Protection: The Missing OS Abstraction." In: *Proceedings of the 14th EuroSys Conference (EuroSys'19)*. ACM, 2019, pp. 1–17. DOI: 10.1145/3302424.3303976.

[Ger+21] Luis Gerhorst, Benedict Herzog, Stefan Reif, Wolfgang Schröder-Preikschat, and Timo Hönig. "AnyCall: Fast and Flexible System-Call Aggregation." In: *Proceedings of the 11th Workshop on Programming Languages and Operating Systems (PLOS'21)*. ACM, 2021, pp. 1–8. DOI: 10.1145/3477113.3487267.

[Ghi+21] Yoann Ghigoff, Julien Sopena, Kahina Lazri, Antoine Blin, and Gilles Muller. "BMC: Accelerating Memcached using Safe In-kernel Caching and Pre-stack Processing." In: *Proceedings of the 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI'21)*. Retrieved 2021-05-21. USENIX, 2021, pp. 487–501. URL: https://www.usenix.org/system/files/nsdi21-ghigoff.pdf.

[GNU21] GNU Findutils contributors. *find(1) – findutils – Debian bullseye – Debian Manpages*. Retrieved 2021-11-04. 2021. URL: https://manpages.debian.org/bullseye/findutils/find.1.en.html.

[Gol+02] Michael Golm, Meik Felser, Christian Wawersich, and Jürgen Kleinöder. "The JX Operating System." In: *Proceedings of the USENIX Annual Technical Conference, General Track (USENIX ATC'02)*. Retrieved 2021-12-09. USENIX, 2002, pp. 45–58. URL: https://www.usenix.org/legacy/event/usenix02/full_papers/golm/golm.pdf.

[Gru+19] Daniel Gruss, Erik Kraft, Trishita Tiwari, Michael Schwarz, Ari Trachtenberg, Jason Hennessey, Alex Ionescu, and Anders Fogh. "Page Cache Attacks." In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS'19)*. ACM, 2019, pp. 167–180. DOI: 10.1145/3319535.3339809.

[Haa+17] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. "Bringing the Web up to Speed with WebAssembly." In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'17)*. ACM, 2017, pp. 185–200. DOI: 10.1145/3062341.3062363.

[Hei21] Bernhard Heinloth. *Personal communication on October 29*. 2021.

[Her+18] Benedict Herzog, Luis Gerhorst, Bernhard Heinloth, Stefan Reif, Timo Hönig, and Wolfgang Schröder-Preikschat. "INTspect: Interrupt Latencies in the Linux Kernel." In: *Proceedings of the 2018 VIII Brazilian Symposium on Computing Systems Engineering (SBESC'18)*. IEEE, 2018, pp. 83–90. DOI: 10.1109/SBESC.2018.00021.

[Her+21] Benedict Herzog, Stefan Reif, Julian Preis, Timo Hönig, and Wolfgang Schröder-Preikschat. "The Price of Meltdown and Spectre: Energy Overhead of Mitigations at Operating System Level." In: *Proceedings of the 14th European Workshop on Systems Security (EuroSec'21)*. ACM, 2021, pp. 8–14. DOI: 10.1145/3447852.3458721.

[Hil+19] Mark D. Hill, Jon Masters, Parthasarathy Ranganathan, Paul Turner, and John L. Hennessy. "On the Spectre and Meltdown Processor Security Vulnerabilities." In: *IEEE Micro* 39.2 (2019), pp. 9–19. DOI: 10.1109/MM.2019.2897677.

[HJ+18] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. "The EXpress Data Path: Fast Programmable Packet Processing in the Operating System Kernel." In: *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies (CoNEXT'18)*. ACM, 2018, pp. 54–66. DOI: 10.1145/3281411.3281443.

[HL07] Galen C. Hunt and James R. Larus. "Singularity: Rethinking the Software Stack." In: *SIGOPS Oper. Syst. Rev.* 41.2 (2007), pp. 37–49. ISSN: 0163-5980. DOI: 10.1145/1243418.1243424.

[Hon+18] Michio Honda, Giuseppe Lettieri, Lars Eggert, and Douglas Santry. "PASTE: A Network Programming Interface for Non-Volatile Main Memory." In: *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI'18)*. Retrieved 2021-12-09. USENIX, 2018, pp. 17–33. URL: https://www.usenix.org/conference/nsdi18/presentation/honda.

[HWH13] Ralf Hund, Carsten Willems, and Thorsten Holz. "Practical Timing Side Channel Attacks against Kernel Space ASLR." In: *Proceedings of the 2013 IEEE Symposium on Security and Privacy (SP'13)*. IEEE, 2013, pp. 191–205. DOI: 10.1109/SP.2013.23.

[IO 21] IO Visor Project. *BCC - Tools for BPF-based Linux IO analysis, networking, monitoring, and more*. Retrieved 2021-05-21. 2021. URL: https://github.com/iovisor/bcc.

[Jun+17] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. "RustBelt: Securing the Foundations of the Rust Programming Language." In: *Proc. ACM Program. Lang.* 2.POPL (2017). DOI: 10.1145/3158154.

[KA21] Giorgos Kappes and Stergios V. Anastasiadis. "Asterope: A Cross-Platform Optimization Method for Fast Memory Copy." In: *Proceedings of the 11th Workshop on Programming Languages and Operating Systems (PLOS'21)*. ACM, 2021, pp. 9–16. DOI: 10.1145/3477113.3487264.

[Kau+16] Antoine Kaufmann, Simon Peter, Naveen Kr. Sharma, Thomas Anderson, and Arvind Krishnamurthy. "High Performance Packet Processing with FlexNIC." In: *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'16)*. ACM, 2016, pp. 67–81. DOI: 10.1145/2872362.2872367.

[Kau+19] Antoine Kaufmann, Tim Stamler, Simon Peter, Naveen Kr. Sharma, Arvind Krishnamurthy, and Thomas Anderson. "TAS: TCP Acceleration As an OS Service." In: *Proceedings of the 14th EuroSys Conference (EuroSys'19)*. ACM, 2019, pp. 24:1–24:16. DOI: 10.1145/3302424.3303985.

[Koc+19] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. "Spectre Attacks: Exploiting Speculative Execution." In: *Proceedings of the 2019 IEEE Symposium on Security and Privacy (SP'19)*. IEEE, 2019, pp. 1–19. DOI: 10.1109/SP.2019.00002.

[KPK12] Vasileios P. Kemerlis, Georgios Portokalidis, and Angelos D. Keromytis. "kGuard: Lightweight Kernel Protection against Return-to-user Attacks." In: *Proceedings of the 21st USENIX Security Symposium*. Retrieved 2021-12-09. USENIX, 2012, pp. 459–474. URL: https://www.usenix.org/system/files/conference/usenixsecurity12/sec12-final143.pdf.

[Kry21] Piotr Krysiuk. *[CVE-2021-29154] Linux kernel incorrect computation of branch displacements in BPF JIT compiler can be abused to execute arbitrary code in Kernel mode*. Retrieved 2021-09-29. 2021. URL: https://www.openwall.com/lists/oss-security/2021/04/08/1.

[KS15] Ake Koomsin and Yasushi Shinjo. "Running Application Specific Kernel Code by a Just-in-Time Compiler." In: *Proceedings of the 8th Workshop on Programming Languages and Operating Systems (PLOS'15)*. ACM, 2015, pp. 15–20. DOI: 10.1145/2818302.2818305.

[Ler+21] Carl Lerche, Matthias Beyer, Tom Kaitchuck, Cameron "SkamDart", Alex Saveau, and "songzhi". *tokio-uring*. Retrieved 2021-11-30. 2021. URL: https://github.com/tokio-rs/tokio-uring/tree/fda163945ff69d0de44e965fe219eb4d94ed45a2.

[Li+21] Zhaofeng Li, Tianjiao Huang, Vikram Narayanan, and Anton Burtsev. "Understanding the Overheads of Hardware and Language-Based IPC Mechanisms." In: *Proceedings of the 11th Workshop on Programming Languages and Operating Systems (PLOS'21)*. ACM, 2021, pp. 53–61. DOI: 10.1145/3477113.3487275.

[Lin17] Linux man-pages project contributors. *memcpy(3) – manpages-dev – Debian bullseye – Debian Manpages*. Retrieved 2021-11-04. 2017. URL: https://manpages.debian.org/bullseye/manpages-dev/memcpy.3.en.html.

[Lin20a] Linux kernel contributors. *libbpf v0.3 (Revision e05f9be, Released 2020-12-21)*. Retrieved 2021-11-11. 2020. URL: https://github.com/libbpf/libbpf/tree/e05f9be4f4bed72cd2c4942bab972da8d693df08.

[Lin20b] Linux man-pages project contributors. *getdents(2) – manpages-dev – Debian bullseye – Debian Manpages*. Retrieved 2021-11-04. 2020. URL: https://manpages.debian.org/bullseye/manpages-dev/getdents64.2.en.html.

[Lin20c] Linux man-pages project contributors. *mmap(2) – manpages-dev – Debian bullseye – Debian Manpages*. Retrieved 2021-11-04. 2020. URL: https://manpages.debian.org/bullseye/manpages-dev/mmap.2.en.html.

[Lin20d] Linux man-pages project contributors. *readv(2) – manpages-dev – Debian bullseye – Debian Manpages*. Retrieved 2021-11-04. 2020. URL: https://manpages.debian.org/bullseye/manpages-dev/readv.2.en.html.

[Lin20e] Linux man-pages project contributors. *SA_RESTART – sigaction(2) – manpages-dev – Debian bullseye – Debian Manpages*. Retrieved 2021-11-04. 2020. URL: https://manpages.debian.org/bullseye/manpages-dev/sigaction.2.en.html#SA_RESTART.

[Lin21a] Linux kernel contributors. *BPF ring buffer*. Retrieved 2021-05-20. 2021. URL: https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/Documentation/bpf/ringbuf.rst?h=v5.11.22#n112.

[Lin21b] Linux kernel contributors. *Linux kernel v5.11*. Retrieved 2021-05-20. 2021. URL: https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tag/?h=v5.11.

[Lin21c] Linux kernel contributors. *vdso(7) - Linux manual page*. Retrieved 2021-05-18. 2021. URL: https://man7.org/linux/man-pages/man7/vdso.7.html.

[Lip+18] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. "Meltdown: Reading Kernel Memory from User Space." In: *Proceedings of the 27th USENIX Security Symposium*. Retrieved 2021-05-21. USENIX, 2018, pp. 973–990. URL: https://www.usenix.org/system/files/conference/usenixsecurity18/sec18-lipp.pdf.

[McI+19] Ross McIlroy, Jaroslav Sevcík, Tobias Tebbi, Ben L. Titzer, and Toon Verwaest. *Spectre is here to stay: An analysis of side-channels and speculative execution*. Retrieved 2021-12-09. 2019. URL: http://arxiv.org/abs/1902.05178.

[MF21] Saidgani Musaev and Christof Fetzer. *Transient Execution of Non-Canonical Accesses*. Retrieved 2020-09-18. 2021. URL: https://arxiv.org/abs/2108.10771.

[MHS14] Daniel Molka, Daniel Hackenberg, and Robert Schöne. "Main Memory and Cache Performance of Intel Sandy Bridge and AMD Bulldozer." In: *Proceedings of the workshop on Memory Systems Performance and Correctness (MSPC'14)*. ACM, 2014, pp. 1–10. DOI: 10.1145/2618128.2618129.

[Mia+18] Sebastiano Miano, Matteo Bertrone, Fulvio Risso, Massimo Tumolo, and Mauricio Vásquez Bernal. "Creating Complex Network Services with eBPF: Experience and Lessons Learned." In: *Proceedings of the 2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR'18)*. IEEE, 2018, pp. 1–8. DOI: 10.1109/HPSR.2018.8850758.

[MJ93] Steven McCanne and Van Jacobson. "The BSD Packet Filter: A New Architecture for User-level Packet Capture." In: *Proceedings of the Winter 1993 USENIX Conference*. Retrieved 2021-05-21. USENIX, 1993, pp. 259–269. URL: https://vodun.org/papers/net-papers/van_jacobson_the_bpf_packet_filter.pdf.

[Nar+19] Vikram Narayanan, Abhiram Balasubramanian, Charlie Jacobsen, Sarah Spall, Scott Bauer, Michael Quigley, Aftab Hussain, Abdullah Younis, Junjie Shen, Moinak Bhattacharyya, and Anton Burtsev. "LXDs: Towards Isolation of Kernel Subsystems." In: *Proceedings of the 2019 USENIX Annual Technical Conference (USENIX ATC'19)*. Retrieved 2021-12-09. USENIX, 2019, pp. 269–284. URL: https://www.usenix.org/conference/atc19/presentation/narayanan.

[Nar+20] Vikram Narayanan, Tianjiao Huang, David Detweiler, Dan Appel, Zhaofeng Li, Gerd Zellweger, and Anton Burtsev. "RedLeaf: Isolation and Communication in a Safe Operating System." In: *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*. Retrieved 2021-10-24. USENIX, 2020, pp. 21–39. URL: https://www.usenix.org/conference/osdi20/presentation/narayanan-vikram.

[Nig+09] Edmund B. Nightingale, Orion Hodson, Ross McIlroy, Chris Hawblitzel, and Galen Hunt. "Helios: Heterogeneous Multiprocessing with Satellite Kernels." In: *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP'09)*. ACM, 2009, pp. 221–234. DOI: 10.1145/1629575.1629597.

[Pan+11] Ying-Shiuan Pan, Jui-Hao Chiang, Han-Lin Li, Po-Jui Tsao, Ming-Fen Lin, and Tzi-cker Chiueh. "Hypervisor Support for Efficient Memory De-duplication." In: *Proceedings of the 2011 IEEE 17th International Conference on Parallel and Distributed Systems (ICPADS'11)*. IEEE, 2011, pp. 33–39. DOI: 10.1109/ICPADS.2011.71.

[Pet+14] Simon Peter, Jialin Li, Irene Zhang, Dan Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. "Arrakis: The Operating System is the Control Plane." In: *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI'14)*. USENIX, 2014, pp. 1–16. DOI: 10.1145/2812806.

[Pro+18] Andrew Prout, William Arcand, David Bestor, Bill Bergeron, Chansup Byun, Vijay Gadepally, Michael Houle, Matthew Hubbell, Michael Jones, Anna Klein, Peter Michaleas, Lauren Milechin, Julie Mullen, Antonio Rosa, Siddharth Samsi, Charles Yee, Albert Reuther, and Jeremy Kepner. "Measuring the Impact of Spectre and Meltdown." In: *Proceedings of the 22nd High Performance Extreme Computing Conference (HPEC'18)*. IEEE, 2018, pp. 1–5. DOI: 10.1109/HPEC.2018.8547554.

[Rei+20] Stefan Reif, Benedict Herzog, Fabian Hügel, Timo Hönig, and Wolfgang Schröder-Preikschat. "Nearly Symmetric Multi-Core Processors." In: *Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys'20)*. ACM, 2020, pp. 42–49. DOI: 10.1145/3409963.3410486.

[Ren+19] Xiang Ren, Kirk Rodrigues, Luyuan Chen, Camilo Vega, Michael Stumm, and Ding Yuan. "An Analysis of Performance Evolution of Linux's Core Operations." In: *Proceedings of the 27th Symposium on Operating Systems Principles (SOSP'19)*. ACM, 2019, pp. 554–569. DOI: 10.1145/3341301.3359640.

[Roe+12] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. "Return-Oriented Programming: Systems, Languages, and Applications." In: *ACM Trans. Inf. Syst. Secur.* 15.1 (2012). ISSN: 1094-9224. DOI: 10.1145/2133375.2133377.

[Rum+11] Stephen Rumble, Diego Ongaro, Ryan Stutsman, Mendel Rosenblum, and John Ousterhout. "It's Time for Low Latency." In: *Proceedings of the 13th Conference on Hot Topics in Operating Systems (HotOS'11)*. Retrieved 2021-07-29. USENIX, 2011, pp. 1–5. URL: https://www.usenix.org/legacy/event/hotos11/tech/final_files/Rumble.pdf.

[Rup20] Karl Rupp. *48 Years of Microprocessor Trend Data*. Retrieved 2021-10-11. 2020. URL: https://github.com/karlrupp/microprocessor-trend-data/tree/47382e2e3c653d71ebae66d8e8aecc088866543d.

[SS10] Livio Soares and Michael Stumm. "FlexSC: Flexible System Call Scheduling with Exception-Less System Calls." In: *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI'09)*. Retrieved 2021-05-21. USENIX, 2010, pp. 33–46. URL: https://static.usenix.org/event/osdi10/tech/full_papers/Soares.pdf.

[Sta14] Alexei Starovoitov. *LLVM BPF backend*. Retrieved 2021-10-06. 2014. URL: https://reviews.llvm.org/D6494.

[Sta21] Alexei Starovoitov. *[PATCH bpf-next 00/15] bpf: syscall program, FD array, loader program, light skeleton*. Retrieved 2021-12-12. 2021. URL: `https://lwn.net/ml/bpf/20210417033224.8063-1-alexei.starovoitov@gmail.com/`.

[Tan06] Andy Tanenbaum. *Tanenbaum-Torvalds Debate Part II*. Retrieved 2021-11-04. 2006. URL: `https://web.archive.org/web/20200818111003/https://www.cs.vu.nl/~ast/reliable-os/`.

[TDS21] Franz-Bernhard Tuneke, Christian Dietrich, and Horst Schirmeier. *Programmable Asynchronous I/O with io_uring and eBPF in Linux*. Retrieved 2021-10-07. 2021. URL: `https://www.betriebssysteme.org/aktivitaeten/treffen/2021-trondheim/programm/abstract-tuneke/`.

[Tra21] Transcend Information Inc. *MTS800 & MTS800I SATA III M.2 SSDs*. Retrieved 2021-11-09. 2021. URL: `https://www.transcend-info.com/products/images/modelpic/1054/Transcend-MTS800_MTS800I.pdf`.

[Whe04] David A. Wheeler. *SLOCCount*. Retrieved 2020-10-01. 2004. URL: `https://manpages.debian.org/bullseye/sloccount/sloccount.1.en.html`.

[Zad+05] Erez Zadok, Sean Callanan, Abhishek Rai, Gopalan Sivathanu, and Avishay Traeger. “Efficient and Safe Execution of User-Level Code in the Kernel.” In: *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS’20)*. IEEE, 2005, p. 221.1. DOI: `10.1109/ipdps.2005.189`.

[ZB20] Koen Zandberg and Emmanuel Baccelli. “Minimal Virtual Machines on IoT Microcontrollers: The Case of Berkeley Packet Filters with rBPF.” In: *Proceedings of the 2020 9th IFIP International Conference on Performance Evaluation and Modeling in Wireless Networks (PEMWN’20)*. IEEE, 2020, pp. 1–6. DOI: `10.23919/pemwn50727.2020.9293081`.

[Zho+21] Yuhong Zhong, Hongyi Wang, Yu Jian Wu, Asaf Cidon, Ryan Stutsman, Amy Tai, and Junfeng Yang. “BPF for Storage: An Exokernel-Inspired Approach.” In: *Proceedings of the 18th Workshop on Hot Topics in Operating Systems (HotOS’21)*. ACM, 2021, pp. 128–135. DOI: `10.1145/3458336.3465290`.