Draco: Architectural and Operating System Support for System Call Security

Dimitrios Skarlatos, Qingrong Chen, Jianyan Chen, Tianyin Xu, Josep Torrellas

University of Illinois at Urbana-Champaign
{skarlat2, qc16, jianyan2, tyxu, torrella}@illinois.edu

***Abstract*—System call checking is extensively used to protect the operating system kernel from user attacks. However, existing solutions such as Seccomp execute lengthy rule-based checking programs against system calls and their arguments, leading to substantial execution overhead.**

To minimize checking overhead, this paper proposes *Draco*, a new architecture that *caches* system call IDs and argument values after they have been checked and validated. System calls are first looked-up in a special cache and, on a hit, skip all checks. We present both a software and a hardware implementation of Draco. The latter introduces a System Call Lookaside Buffer (SLB) to keep recently-validated system calls, and a System Call Target Buffer to preload the SLB in advance. In our evaluation, we find that the average execution time of macro and micro benchmarks with conventional Seccomp checking is 1.14× and 1.25× higher, respectively, than on an insecure baseline that performs no security checks. With our software Draco, the average execution time reduces to 1.10× and 1.18× higher, respectively, than on the insecure baseline. With our hardware Draco, the execution time is within 1% of the insecure baseline.

***Index Terms*—System call checking, Security, Operating system, Containers, Virtualization, Microarchitecture**

I. Introduction

Protecting the Operating System (OS) kernel is a significant concern, given its capabilities and shared nature. In recent years, there have been reports of a number of security attacks on the OS kernel through system call vectors [1]–[10].

A popular technique to protect OS kernels against untrusted user processes is *system call checking*. The idea is to limit the system calls that a given process can invoke at runtime, as well as the actual set of argument values used by the system calls. This technique is implemented by adding code at the OS entry point of a system call. The code compares the incoming system call against a list of allowed system calls and argument set values, and either lets the system call continue or flags an error. All modern OSes provide kernel support for system call checking, such as Seccomp for Linux [11], Pledge [12] and Tame [13] for OpenBSD, and System Call Disable Policy for Windows [14].

Linux's Seccomp (Secure Computing) module is the most widely-used implementation of system call checking. It is used in a wide variety of today's systems, ranging from mobile systems to web browsers, and to containers massively deployed in cloud and data centers. Today, every Android app is isolated using Seccomp-based system call checking [15]. Systemd, which is Linux's init system, uses Seccomp to support user process sandboxing [16]. Low-overhead virtualization technologies such as Docker [17], LXC/LXD [18], Google's gVisor [19], Amazon's Firecraker [20], [21], CoreOS rkt [22], Singularity [23], Kubernetes [24], and Mesos Containerizer [25] all use Seccomp. Further, Google's recent Sandboxed API project [26] uses Seccomp to enforce sandboxing for C/C++ libraries. Overall, Seccomp provides "*the most important security isolation boundary*" for containerized environments [21].

Unfortunately, checking system calls incurs overhead. Oracle identified large Seccomp programs as a root cause that slowed down their customers' applications [27], [28]. Seccomp programs can be application specific, and complex applications tend to need large Seccomp programs. Recently, a number of Seccomp overhead measurements have been reported [27]–[29]. For example, a micro benchmark that repeatedly calls `getppid` runs 25% slower when Seccomp is enabled [29]. Seccomp's overhead on ARM processors is reported to be around 20% for simple checks [30].

The overhead becomes higher if the checks include system call argument values. Since each individual system call can have multiple arguments, and each argument can take multiple distinct values, comprehensively checking arguments is slow. For example, Kim and Zeldovich [31] show that Seccomp causes a 45% overhead in a sandboxed application. Our own measurements on an Intel Xeon server show that the average execution time of macro and micro benchmarks is 1.14× and 1.25× higher, respectively, than without any checks. For this reason, current systems tend to perform only a small number of argument checks, despite being well known that systematically checking arguments is critical for security [32]–[34].

The overhead of system call and argument value checking is especially concerning for applications with high-performance requirements, such as containerized environments. The overhead diminishes one of the key attractions of containerization, namely lightweight virtualization.

To minimize this overhead, this paper proposes *Draco*, a new architecture that *caches* system call IDs and argument set values after they have been checked and validated. System calls are first looked-up in a special cache and, on a hit, skip the checks. The insight behind Draco is that the patterns of system calls in real-world applications have locality—the same system calls are issued repeatedly, with the same sets of argument values.

In this paper, we present both a software and a hardware implementation of Draco. We build the software one as a component of the Linux kernel. While this implementation is faster than Seccomp, it still incurs substantial overhead.

978-1-7281-7383-2/20/$31.00 ©2020 IEEE
DOI 10.1109/MICRO50266.2020.00017

Authorized licensed use limited to: University of Illinois. Downloaded on January 22,2021 at 22:12:22 UTC from IEEE Xplore. Restrictions apply.

The hardware implementation of Draco introduces novel microarchitecture to eliminate practically all of the checking overhead. It introduces the *System Call Lookaside Buffer* (SLB) to keep recently-validated system calls, and the *System Call Target Buffer* (STB) to preload the SLB in advance.

In our evaluation, we execute representative workloads in Docker containers. We run Seccomp on an Intel Xeon server, checking both system call IDs and argument values. We find that, with Seccomp, the average execution time of macro and micro benchmarks is 1.14× and 1.25× higher, respectively, than without performing any security checks. With our software Draco, the average execution time of macro and micro benchmarks reduces to 1.10× and 1.18× higher, respectively, than without any security checks. Finally, we use full-system simulation to evaluate our hardware Draco. The average execution time of the macro and micro benchmarks is within 1% of a system that performs no security checks.

With more complex checks, as expected in the future, the overhead of conventional Seccomp checking increases substantially, while the overhead of Draco's software implementation goes up only modestly. Moreover, the overhead of Draco's hardware implementation remains within 1% of a system without checks.

Overall, the contributions of this work are:

1) A characterization of the system call checking overhead for various Seccomp configurations.
2) The new Draco architecture that caches validated system call IDs and argument values for reuse. We introduce a software and a hardware implementation.
3) An evaluation of the software and hardware implementations of Draco.

II. Background

A. System Calls

System calls are the interface an OS kernel exposes to the user space. In x86-64 [35], a user space process requests a system call by issuing the `syscall` instruction, which transfers control to the OS kernel. The instruction invokes a system call handler at privilege level 0. In Linux, `syscall` supports from zero to six distinct arguments that are passed to the kernel through general-purpose registers. Specifically, the x86-64 architecture stores the system call ID in the `rax` register, and the arguments in registers `rdi`, `rsi`, `rdx`, `r10`, `r8`, and `r9`. The return value of the system call is stored in the `rax` register. The `syscall` instruction is a serializing instruction [35], which means that it cannot execute until all the older instructions are completed, and that it prevents the execution of all the younger instructions until it completes. Finally, note that the work in this paper is not tied to Linux, but applies to different OS kernels.

B. System Call Checking

A core security technique to protect the OS kernel against untrusted user processes is system call checking. The most widely-used implementation of such a technique is the Secure Computing (Seccomp) module [11], which is implemented in Linux. Seccomp allows the software to specify which system calls a given process can make, and which argument values such system calls can use. This information is specified in a *profile* for the process, which is expressed as a Berkeley Packet Filter (BPF) program called a *filter* [36]. When the process is loaded, a Just-In-Time compiler in the kernel converts the BPF program into native code. The Seccomp filter executes in the OS kernel every time the process performs a system call. The filter may allow the system call to proceed or, if it is illegal, capture it. In this case, the OS kernel may take one of multiple actions: kill the process or thread, send a `SIGSYS` signal to the thread, return an error, or log the system call [11]. System call checking is also used by other modern OSes, such as OpenBSD with Pledge [12] and Tame [13], and Windows with System Call Disable Policy [14]. The idea behind our proposal, Draco, can be applied to all of them.

Figure 1 shows an example of a system call check. In user space, the program loads the system call argument and ID in registers `rdi` and `rax`, respectively, and performs the system call. This ID corresponds to the `personality` system call. As execution enters the OS kernel, Seccomp checks if this combination of system call ID and argument value is allowed. If it is allowed, it lets the system call execution to proceed (Line 8); otherwise, it terminates the process (Line 11).

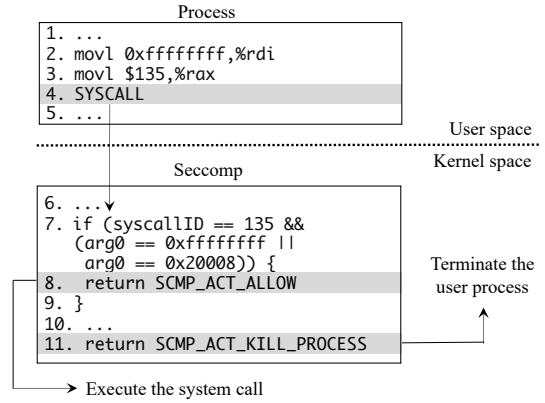


Fig. 1: Checking a system call with Seccomp.

Figure 1 shows that a Seccomp profile is a long list of `if` statements that are executed in sequence. Finding the target system call and the target combination of argument values in the list can take a long time, which is the reason for the often high overhead of Seccomp.

Seccomp does not check the values of arguments that are pointers. This is because such a check does not provide any protection: a malicious user could change the contents of the location pointed to by the pointer after the check, creating a Time-Of-Check-Time-Of-Use (TOCTOU) attack [37], [38].

Seccomp could potentially do advanced checks such as checking the value range of an argument, or the result of an operation on the arguments. However, our study of real-world Seccomp profiles shows that most real-world profiles simply check system call IDs and argument values based on a whitelist of exact IDs and values [19], [39]–[43].

Authorized licensed use limited to: University of Illinois. Downloaded on January 22,2021 at 22:12:22 UTC from IEEE Xplore. Restrictions apply.

C. System Call Checking in Modern Systems

System call checking with Seccomp is performed in almost all of the modern Linux-based system infrastructure. This includes modern container and lightweight VM runtimes such as Docker [17], Google's gVisor [19], Amazon's Firecracker [20], [21], and CoreOS rkt [22]. Other environments, such as Android and Chrome OS also use Seccomp [15], [44]–[46]. The systemd Linux service manager [16] has adopted Seccomp to further increase the isolation of user containers.

Google's Sandboxed API [26] is an initiative that aims to isolate C/C++ libraries using Seccomp. This initiative, together with others [47]–[50], significantly reduce the barrier to apply Seccomp profiles customized for applications.

Seccomp profiles. In this paper, we focus on container technologies, which have both high performance and high security requirements. The default Seccomp profiles provided by existing container runtimes typically contain hundreds of system call IDs and fewer argument values. For example, *Docker's default profile* allows 358 system calls, and only checks 7 unique argument values (of the `clone` and `personality` system calls). This profile is widely used by other container technologies such as CoreOS Rtk and Singularity, and is applied by container management systems such as Kubernetes, Mesos, and RedHat's RedShift. In this paper, we use Docker's default Seccomp profile as our baseline profile.

Other profiles include the default gVisor profile, which is a whitelist of 74 system calls and 130 argument checks. Also, the profile for the AWS Firecracker microVMs contains 37 system calls and 8 argument checks [20].

In this paper, we also explore more secure Seccomp profiles tailored for some applications. *Application-specific* profiles are supported by Docker and other projects [17], [24], [50]–[53].

III. Threat Model

We adopt the same threat model as existing system call checking systems such as Seccomp, where untrusted user space processes can be adversarial and attack the OS kernel. In the context of containers, the containerized applications are deployed in the cloud, and an adversary tries to compromise a container and exploit vulnerabilities in the host OS kernel to achieve privilege escalation and arbitrary code execution. The system call interface is the major attack vector for user processes to attack the OS kernel.

Prior work [54]–[56] has shown that system call checking is effective in defending against real-world attacks, because most applications only use a small number of different system calls and argument values [2]–[6]. For example, the mitigation of CVE-2014-3153 [3] is to disallow `FUTEX_REQUEUE` as the value of the `futex_op` argument of the `futex` system call.

Our focus is not to invent new security analyses or policies, but to minimize the execution overhead that hinders the deployment of comprehensive security checks. As will be shown in this paper, existing software-based security checking is often costly. This forces users to sacrifice either performance or security. Draco provides both high performance and a high level of security.

IV. Measuring Overhead & Locality

To motivate our proposal, we benchmark the overhead of state-of-the-art system call checking. We focus on understanding the overhead of: 1) using generic system call profiles, 2) using smaller, application-specific system call profiles, and 3) using profiles that also check system call arguments.

A. Methodology

We run macro and micro benchmarks in Docker containers with each of the following four Seccomp profiles:

insecure: Seccomp is disabled. It is an insecure baseline where no checks are performed.
docker-default: Docker's default Seccomp profile. It is automatically used in all Docker containers and other container technologies (e.g., CoreOS rkt and Singularity) as part of the Moby project [57]. This profile is deployed by container management systems like Kubernetes, RedHat RedShift, and Mesos Containerizers.
syscall-noargs: Application-specific profiles without argument checks, where the filter whitelists the exact system call IDs used by the application.
syscall-complete: Application-specific profiles with argument checks, where the filter whitelists the exact system call IDs and the exact argument values used by the application. These profiles are the most secure filters that include both system call IDs and their arguments.
syscall-complete-2x: Application-specific profiles that consist of running the above `syscall-complete` profile *twice in a row*. Hence, these profiles perform twice the number of checks as `syscall-complete`. The goal is to model a near-future environment that performs more extensive security checks.

Section X-B describes our toolkits for automatically generating the `syscall-noargs`, `syscall-complete`, and `syscall-complete-2x` profiles for a given application. The workloads are described in Section X-A, and are grouped into macro benchmarks and micro benchmarks. All workloads run on an Intel Xeon (E5-2660 v3) system at 2.60GHz with 64 GB of DRAM, using Ubuntu 18.04 with the Linux 5.3.0 kernel. We disable the `spec_store_bypass`, `spectre_v2`, `mds`, `pti`, and `l1tf` vulnerability mitigations due to their heavy performance impact—many of these kernel patches will be removed in the future anyway.

We run the workloads with the Linux BPF JIT compiler enabled. Enabling the JIT compiler can achieve 2–3× performance increases [30]. Note that JIT compilers are disabled by default in many Linux distributions to prevent kernel JIT spraying attacks [58]–[60]. Hence, the Seccomp performance that we report in this section represents the highest performance for the baseline system.

B. Execution Overhead

Figure 2 shows the latency or execution time of the workloads using the five types of Seccomp profiles described in Section IV-A. For each workload, the results are normalized to `insecure` (i.e., Seccomp disabled).

Authorized licensed use limited to: University of Illinois. Downloaded on January 22,2021 at 22:12:22 UTC from IEEE Xplore. Restrictions apply.

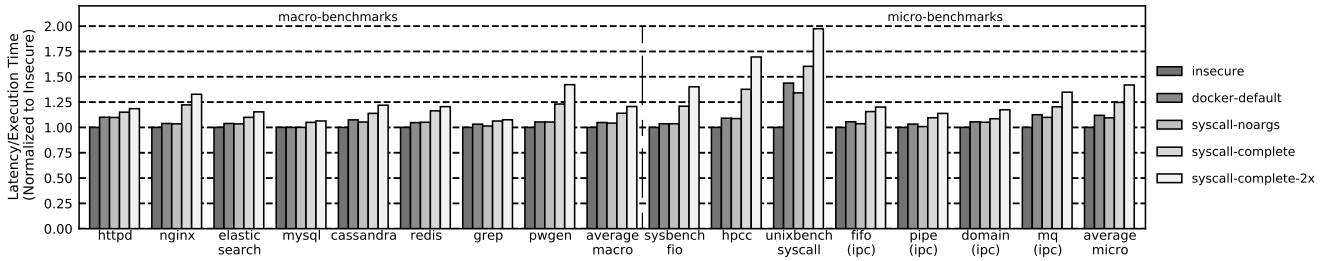


Fig. 2: Latency or execution time of the workloads using different Seccomp profiles. For each workload, the results are normalized to `insecure` (i.e., Seccomp disabled).

Overhead of checking system call IDs. To assess the overhead of checking system call IDs, we compare `insecure` to `docker-default`. As shown in Figure 2, enabling Seccomp using the Docker default profile increases the execution time of the macro and micro benchmarks by 1.05× and 1.12× on average, respectively.

Comparing `docker-default` to `syscall-noargs`, we see the impact of using application-specific profiles. Sometimes, the overhead is visibly reduced. This is because the number of instructions needed to execute the `syscall-noargs` profile is smaller than that of `docker-default`. Overall, the average performance overhead of using `syscall-noargs` profiles is 1.04× for macro benchmarks and 1.09× for micro benchmarks, respectively.

Overhead of checking system call arguments. To assess the overhead of checking system call arguments, we first compare `syscall-noargs` with `syscall-complete`. The number of system calls in these two profiles is the same, but `syscall-complete` additionally checks argument values. We can see that checking arguments brings significant overhead. On average, compared to `syscall-noargs`, the macro and micro benchmarks increase their execution time from 1.04× to 1.14× and from 1.09× to 1.25×, respectively.

We now double the number of checks by going from `syscall-complete` to `syscall-complete-2x`. We can see that the benchmark overhead often nearly doubles. On average, compared to `syscall-complete`, the macro and micro benchmarks increase their execution time from 1.14× to 1.21× and from 1.25× to 1.42×, respectively.

Implications. Our results lead to the following conclusions. First, checking system call IDs can introduce noticeable performance overhead to applications running in Docker containers. This is the case even with Seccomp, which is the most efficient implementation of the checks.

Second, checking system call arguments is significantly more expensive than checking only system call IDs. This is because the number of arguments per system call and the number of distinct values per argument are often large.

Finally, doubling the security checks in the profiles almost doubles the performance overhead.

C. System Call Locality

We measured all the system calls and arguments issued by our macro benchmarks. Figure 3 shows the frequency of the top calls which, together, account for 86% of all the calls. For example, `read` accounts for about 18% of all system calls. We further break down each bar into the different argument sets used by the system call. Each color in a bar corresponds to the frequency of a different argument set (or none).

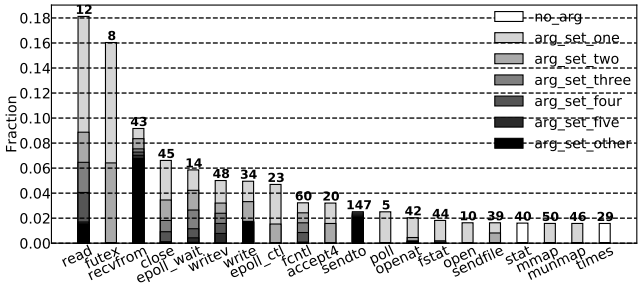


Fig. 3: Frequency of the top system calls and average reuse distance collected from the macro benchmarks.

We see that system calls have a high locality: 20 system calls account for 86% of all the calls. Further, individual system calls are often called with three or fewer different argument sets. At the top of each bar, we show the average *reuse distance*, defined as the number of other system calls between two system calls with the same ID and argument set. As shown in Figure 3, the average distance is often only a few tens of system calls, indicating high locality in reusing system call checking results.

V. Draco System Call Checking

To minimize the overhead of system call checking, this paper proposes a new architecture called *Draco*. Draco avoids executing the many `if` statements of a Seccomp profile at every system call (Figure 1). Draco *caches* system call IDs and argument set values after they have been checked and validated once. System calls are first looked-up in the cache and, on a hit, the system call and its arguments are declared validated, skipping the execution of the Seccomp filter.

Figure 4 shows the workflow. On reception of a system call, a table of validated system call IDs and argument values is checked. If the current system call and arguments match an entry in the table, the system call is allowed to proceed. Otherwise, the OS kernel executes the Seccomp filter and decides if the system call is allowed. If so, the table is updated with the new entry and the system call proceeds; otherwise the program is killed or other actions are taken (Section II-B).

This approach is correct because Seccomp profiles are stateless. This means that the output of the Seccomp filter

Authorized licensed use limited to: University of Illinois. Downloaded on January 22,2021 at 22:12:22 UTC from IEEE Xplore. Restrictions apply.

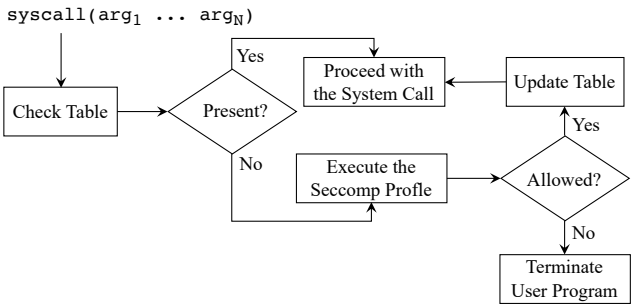


Fig. 4: Workflow of Draco system call checking.

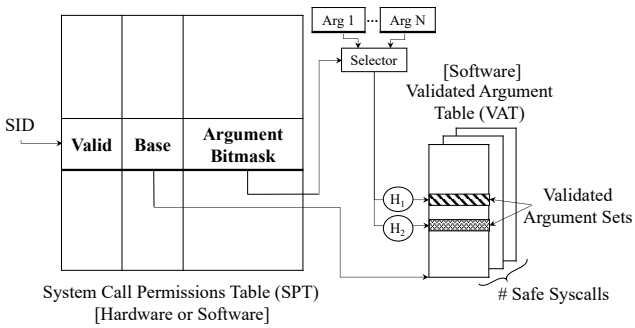


Fig. 5: Checking a system call and its arguments.

execution depends only on the input system call ID and arguments—not on some other state. Hence, a validation that succeeded in the past does not need to be repeated.

Draco can be implemented in software or in hardware. In the following, we first describe the basic design, and then how we implement it in software and in hardware. We start with a design that checks system call IDs only, and then extend it to check system call argument values as well.

A. Checking System Call IDs Only

If we only want to check system call IDs, the design is simple. It uses a table called *System Call Permissions Table* (SPT), with as many entries as different system calls. Each entry stores a single *Valid* bit. If the Valid bit is set, the system call is allowed. In Draco's hardware implementation, each core has a hardware SPT. In both hardware and software implementations, the SPT contains information for one process.

When the System Call ID (SID) of the system call is known, Draco finds the SPT entry for that SID, and checks if the Valid bit is set. If it is, the system call proceeds. Otherwise, the Seccomp filter is executed.

B. Checking System Call Arguments

To check system call arguments, we enhance the SPT and couple it with a *software structure* called *Validated Argument Table* (VAT). Both SPT and VAT are private to the process. The VAT is the same for both software and hardware implementations of Draco.

Figure 5 shows the structures. The SPT is still indexed with the SID. An entry now includes, in addition to the Valid bit, a *Base* and an *Argument Bitmask* field. The Base field stores the virtual address of the section of the VAT that holds information about this system call. The Argument Bitmask stores information to determine what arguments are used by this system call. Recall that a system call can take up to 6 arguments, each 1 to 8 bytes long. Hence, the Argument Bitmask has one bit per argument byte, for a total of 48 bits. A given bit is set if this system call uses this byte as an argument—e.g., for a system call that uses two arguments of one byte each, the Argument Bitmask has bits 0 and 8 set.

The VAT of a process has one structure for each system call allowed for this process. Each structure is a hash table of limited size, indexed with two hash functions. If an entry in the hash table is filled, it holds the values of an argument set that has been found to be safe for this particular system call.

When a system call is encountered, to find the correct entry in the VAT, Draco hashes the argument values. Specifically, when the system call's SID and argument set are known, the SPT is indexed with the SID. Draco uses the Arguments Bitmask to select which parts of the arguments to pass to the hash functions and generate hash table indices for the VAT. For example, if a system call uses two arguments of one byte each, only the eight bits of each argument are used to generate the hash table indices.

The address in Base is added to the hash table indices to access the VAT. Draco fetches the contents of the two entries, and compares them to the values of the arguments of the system call. If any of the two comparisons produces a match, the system call is allowed.

Draco uses two hash functions (H_1 and H_2 in Figure 5) for the following reason. To avoid having to deal with hash table collisions in a VAT structure, which would result in multiple VAT probes, each VAT structure uses 2-ary cuckoo hashing [61], [62]. Such a design resolves collisions gracefully. However, it needs to use two hash functions to perform two accesses to the target VAT structure in parallel. On a read, the resulting two entries are checked for a match. On an insertion, the cuckoo hashing algorithm is used to find a spot.

C. A Software Implementation

Draco can be completely implemented in software. Both the SPT and the VAT are software structures in memory. We build Draco as a Linux kernel component. In the OS kernel, at the entry point of system calls, we insert instructions to read the SID and argument set values (if argument checking is configured). Draco uses the SID to index into the SPT and decides if the system call is allowed or not based on the Valid bit. If argument checking is configured, Draco further takes the correct bits from the arguments to perform the hashes, then reads the VAT, compares the argument values, and decides whether the system call passes or not.

D. An Initial Hardware Implementation

An initial hardware implementation of Draco makes the SPT a hardware table in each core. The VAT remains a software structure in memory. The checks can only be performed when either the SID or both the SID and argument set values are available. To simplify the hardware, Draco waits until the system call instruction reaches the Reorder Buffer (ROB)

Authorized licensed use limited to: University of Illinois. Downloaded on January 22,2021 at 22:12:22 UTC from IEEE Xplore. Restrictions apply.

head. At that point, all the information about the arguments is guaranteed to be available in specific registers. Hence, the Draco hardware indexes the SPT and checks the Valid bit. If argument checking is enabled, the hardware further takes the correct bits from the arguments, performs the hashes, reads the VAT, compares the argument values, and determines if the system call passes. If the combination of system call and argument set are found not to have been validated, the OS is invoked to run the Seccomp filter.

VI. Draco Hardware Implementation

The initial hardware implementation of Section V-D has the shortcoming that it requires memory accesses to read the VAT—in fact, two accesses, one for each of the hash functions. While some of these accesses may hit in caches, the performance is likely to suffer. For this reason, this section extends the hardware implementation of Draco to perform the system call checks *in the pipeline* with very low overhead.

A. Caching the Results of Successful Checks

To substantially reduce the overhead of system call checking, Draco introduces a new hardware cache of recently-encountered and validated system call argument sets. The cache is called *System Call Lookaside Buffer* (SLB). It is inspired by the TLB (Translation Lookaside Buffer).

Figure 6 shows the SLB. It is indexed with the system call's SID and number of arguments that it requires. Since different system calls have different numbers of arguments, the SLB has a set-associative sub-structure for each group of system calls that take the same number of arguments. This design minimizes the space needed to cache arguments—each sub-structure can be sized individually. Each entry in the SLB has an SID field, a *Valid* bit, a *Hash* field, and a validated argument set denoted as $<Arg_1$ ... $Arg_N>$. The Hash field contains the hash value generated using this argument set via either the H_1 or H_2 hash functions mentioned in Section V-B.

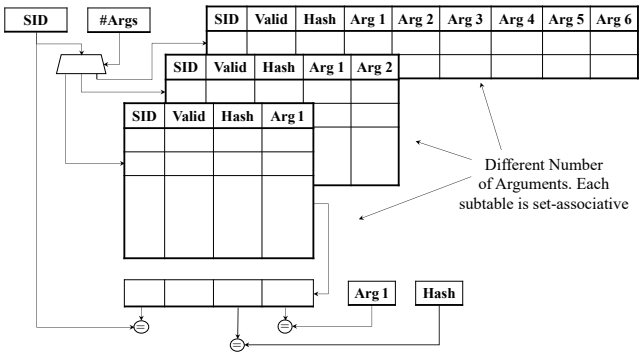


Fig. 6: System Call Lookaside Buffer (SLB) structure.

Figure 7 describes the SLB operations performed by Draco alongside the SPT and VAT. When a system call is detected at the head of the ROB, its SID is used to access the SPT (①). The SPT uses the Argument Bitmask (Figure 5) to generate the argument count used by the system call. This information, together with the SID, is used to index the SLB (②).

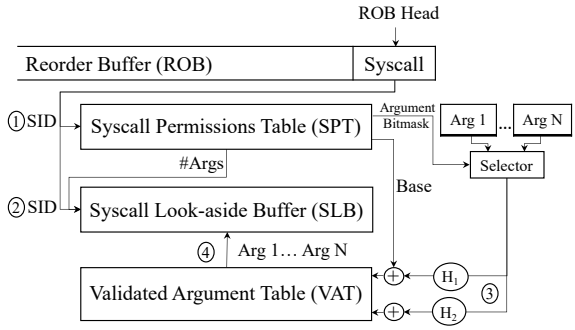


Fig. 7: Flow of operations in an SLB access.

On an SLB miss, the corresponding VAT is accessed. Draco takes the current system call argument set and the Argument Bitmask from the SPT and, using hash functions H_1 and H_2, generates two hash values (③). Such hash values are combined with the Base address provided by the SPT to access the VAT in memory. The resulting two VAT locations are fetched in parallel, and their contents are compared to the actual system call argument set. On a match, the SLB entry is filled with the SID, the validated argument set, and the one hash value that fetched the correct entry from the VAT (④). The system call is then allowed to resume execution.

On subsequent accesses to the SLB with the same system call's SID and argument set, the SLB will hit. A hit occurs when an entry is found in the SLB that has the same SID and argument set values as the incoming system call. In this case, the system call is allowed to proceed *without requiring any memory hierarchy access*. In this case, which we assume is the most frequent one, the checking of the system call and arguments has negligible overhead.

B. Preloading Argument Sets

The SLB can significantly reduce the system call checking time by caching recently validated arguments. However, on a miss, the system call stalls at the ROB head until its arguments are successfully checked against data coming from the VAT in memory. To avoid this problem, we want to hide all the stall time by *preloading the SLB entry early*.

This function is performed by the *System Call Target Buffer* (STB). The STB is inspired by the Branch Target Buffer. Its goal is to preload a potentially useful entry in the SLB as soon as a system call is placed in the ROB. Specifically, it preloads in the SLB an argument set that is in the VAT and, therefore, has been validated in the past for the same system call. When the system call reaches the head of the ROB and tries to check its arguments, it is likely that the correct argument set is already preloaded into the SLB.

Fundamentally, the STB operates like the BTB. While the BTB predicts the target location that the upcoming branch will jump to, the STB predicts the location in the VAT that stores the validated argument set that the upcoming system call will require. Knowing this location allows the hardware to preload the information into the SLB in advance.

Authorized licensed use limited to: University of Illinois. Downloaded on January 22,2021 at 22:12:22 UTC from IEEE Xplore. Restrictions apply.

System Call Target Buffer and Operation. The STB is shown in Figure 8. Each entry stores the program counter (PC) of a system call, a *Valid* bit, the SID of the system call, and a *Hash* field. The latter contains the one hash value (of the two possible) that fetched this argument set from the VAT when the entry was loaded into the STB.

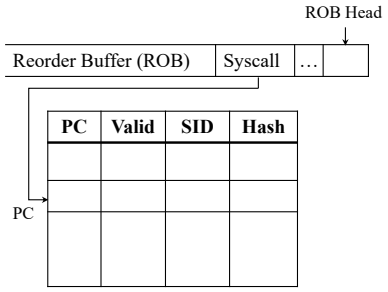


Fig. 8: System Call Target Buffer (STB) structure.

The preloading operation into the SLB is shown in Figure 9. As soon as an instruction is inserted in the ROB, Draco uses its PC to access the STB (①). A hit in the STB is declared when the PC in the ROB matches one in the STB. This means that the instruction is a system call. At that point, the STB returns the SID and the predicted hash value. Note that the SID is the correct one because there is only one single type of system call in a given PC. At this point, the hardware knows the system call. However, it does not know the actual argument values because, unlike in Section VI-A, the system call is not at the ROB head, and the actual arguments may not be ready yet.

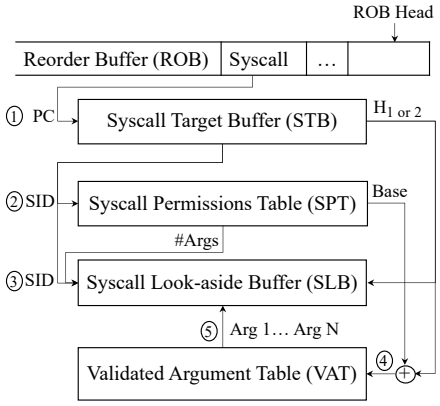


Fig. 9: Flow of operations in an SLB preloading.

Next, Draco accesses the SPT using the SID (②), which provides the Base address of the structure in the VAT, and the number of arguments of the system call.

At this point, the hardware has to: (i) check if the SLB already has an entry that will likely cause an SLB hit when the system call reaches the ROB head and, (ii) if not, find such an entry in the VAT and preload it in the SLB. To perform (i), Draco uses the SID and the number of arguments to access the set-associative subtable of the SLB (③) that has the correct argument count (Figure 6). Since we do not yet know the actual argument set, we consider a hit in the SLB when the hash value provided by the STB matches the one stored in the SLB entry. This is shown in Figure 6. We call this an *SLB Preload hit*. No further action is needed because the SLB likely has the correct entry. Later, when the system call reaches the head of the ROB and the system call argument set is known, the SLB is accessed again with the SID and the argument set. If the argument set matches the one in the SLB entry, it is an *SLB Access hit*; the system call has been checked without any memory system access at all.

If, instead, no SLB Preload hit occurs, Draco performs the action (ii) above. Specifically, Draco reads from the SPT the Base address of the structure in the VAT, reads from the STB the hash value, combines them, and indexes the VAT (④). If a valid entry is found in the VAT, its argument set is preloaded into the SLB (⑤). Again, when the system call reaches the head of the ROB, this SLB entry will be checked for a match.

C. Putting it All Together

Figure 10 shows the Draco hardware implementation in a multi-core chip. It has three per-core hardware tables: SLB, STB, and SPT. In our conservatively-sized design, they use 8KB, 4KB, and 4KB, respectively. The SLB holds a process' most popular *checked* system calls and their *checked* argument sets. When a system call reaches the head of the ROB, the SLB is queried. The SLB information is backed up in the per-process software VAT in memory. VAT entries are loaded into the SLB on demand by hardware.

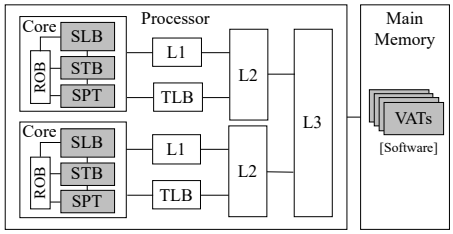


Fig. 10: Multicore with Draco structures in a shaded pattern.

As indicated, Draco does not wait for the system call to reach the ROB head before checking the SLB. Instead, as soon as a system call enters the ROB, Draco checks the SLB to see if it can possibly have the correct entry. If not, it accesses the VAT and loads a good-guess entry into the SLB.

For a core to access its SLB, Draco includes two local hardware structures: the STB—which takes an instruction PC and provides an SID—and the SPT—which takes the SID and provides its argument count and its VAT location.

Draco execution flows. Table I shows the possible Draco execution flows. Flow ① occurs when the STB access, SLB preload, and SLB access all hit. It results in a fast execution. Flow ② occurs when the STB access and SLB preload hit, but the actual SLB access does not find the correct argument set. In this case, Draco fetches the argument set from the VAT, and fills an entry in the STB with the correct hash, and in the SLB with the correct hash and argument set. This flow is marked as slow in Table I. Note however, that the argument

Authorized licensed use limited to: University of Illinois. Downloaded on January 22,2021 at 22:12:22 UTC from IEEE Xplore. Restrictions apply.

Execution Flow	STB Access	SLB Preload	SLB Access	Action	Speed
①	Hit	Hit	Hit	None.	Fast
②	Hit	Hit	Miss	After the SLB access miss, fetch the argument set from the VAT, and fill an entry in the STB with the correct hash, and in the SLB with the correct hash and arguments.	Slow
③	Hit	Miss	Hit	After the SLB preload miss, fetch the argument set from the VAT, and fill an entry in the SLB with the correct SID, hash and arguments.	Fast
④	Hit	Miss	Miss	After the SLB preload miss, do as ③. After the SLB access miss, do as ②.	Slow
⑤	Miss	N/A	Hit	After the SLB access hit, fill an entry in the STB with the correct SID and hash.	Fast
⑥	Miss	N/A	Miss	After the SLB access miss, fetch the argument set from the VAT, and fill an entry in the STB with correct SID and hash, and in the SLB with correct SID, hash, and arguments.	Slow

TABLE I: Possible Draco execution flows. The *Slow* cases can have different latency, depending on whether the VAT accesses hit or miss in the caches. If the VAT does not have the requested entry, the OS is invoked and executes the Seccomp filter.

set may be found in the caches, which saves accesses to main memory. Finally, if the correct argument set is not found in the VAT, the OS is invoked and executes the Seccomp filter. If such execution validates the system call, the VAT is updated as well with the validated SID and argument set.

Flow ③ occurs when the STB access hits, the SLB preload misses, and Draco initiates an SLB preload that eventually delivers an SLB access hit. As soon as the preload SLB miss is declared, Draco fetches the argument set from the VAT, and fills an entry in the SLB with the correct SID, hash, and arguments. When the system call reaches the ROB head and checks the SLB, it declares an SLB access hit. This is a fast case.

Flow ④ occurs when the STB access hits, the SLB preload misses, Draco's SLB preload brings incorrect data, and the actual SLB access misses. In this case, after the SLB preload miss, Draco performs the actions in Flow ③; after the SLB access miss, Draco performs the actions in Flow ②.

Flows ⑤ and ⑥ start with an STB miss. Draco does not preload the SLB because it does not know the SID. When the system call reaches the head of the ROB, the SLB is accessed. In Flow ⑤, the access hits. In this case, after the SLB access hit, Draco fills an entry in the STB with the correct SID and hash. In Flow ⑥, the SLB access misses, and Draco has to fill an entry in both STB and SLB. ⑤ is fast and ⑥ is slow.

VII. System Support

A. VAT Design and Implementation

The OS kernel is responsible for filling the VAT of each process. The VAT of a process has a two-way cuckoo hash table for each system call allowed to the process. The OS sizes each table based on the number of argument sets used by corresponding system call (e.g., based on the given Seccomp profile). To minimize insertion failures in the hash tables, the size of each table is over-provisioned two times the number of estimated argument sets. On an insertion to a table, if the cuckoo hashing fails after a threshold number of attempts, the OS makes room by evicting one entry.

During a table lookup, either the OS or the hardware (in the hardware implementation) accesses the two ways of the cuckoo hash table. For the hash functions, we use the $ECMA$ [63] and the $\neg\ ECMA$ polynomials to compute the Cyclic Redundancy Check (CRC) code of the system call argument set (Figure 5).

The base addresses in the SPT are stored as virtual addresses. In the hardware implementation of Draco, the hardware translates this base address before accessing a VAT structure. Due to the small size of the VAT (several KBs for a process), this design enjoys good TLB translation locality, as well as natural caching of translations in the cache hierarchy. If a page fault occurs on a VAT access, the OS handles it as a regular page fault.

B. Implementation Aspects

Invocation of Software Checking. When the Draco hardware does not find the correct SID or argument set in the VAT, it sets a register called *SWCheckNeeded*. As the system call instruction completes, the system call handler in the OS first checks the SWCheckNeeded register. If it is set, the OS runs system call checking (e.g., Seccomp). If the check succeeds, the OS inserts the appropriate entry in the VAT and continues; otherwise the OS does not allow the system call to execute.

Data Coherence. System call filters are not modified during process runtime to limit attackers' capabilities. Hence, there is no need to add support to keep the three hardware structures (SLB, STB, and SPT) coherent across cores. Draco only provides a fast way to clear all these structures in one shot.

Context Switches. A simple design would simply invalidate the three hardware structures on a context switch. To reduce the start-up overhead after a context switch, Draco improves on this design with two simple supports. First, on a context switch, the OS saves to memory a few key SPT entries for the process being preempted, and restores from memory the saved SPT entries for the incoming process. To pick which SPT entries to save, Draco enhances the SPT with an *Accessed* bit per entry. When a system call hits on one of the SPT entries, the entry's Accessed bit is set. Periodically (e.g., every 500μs), the hardware clears the Accessed bits. On a context switch, only the SPT entries with the Accessed bit set are saved.

The second support is that, if the process that will be scheduled after the context switch is the same as the one that is being preempted, the structures are not invalidated. This is safe with respect to side-channels. If a different process is to be scheduled, the hardware structures are invalidated.

Simultaneous Multithreading (SMT) Support. Draco can support SMT by partitioning the three hardware structures

Authorized licensed use limited to: University of Illinois. Downloaded on January 22,2021 at 22:12:22 UTC from IEEE Xplore. Restrictions apply.

and giving one partition to each SMT context. Each context accesses its partition.

VIII. Generality of Draco

The discussions so far presented a Draco design that is broadly compatible with Linux's Seccomp, so we could describe a whole-system solution. In practice, it is easy to apply the Draco ideas to other system call checking mechanisms such as OpenBSD's Pledge and Tame [12], [13], and Window's System Call Disable Policy [64]. Draco is generic to modern OS-level sandboxing and containerization technologies.

Specifically, recall that the OS populates the SPT with system call information. Each SPT entry corresponds to a system call ID and has argument information. Hence, different OS kernels will have different SPT contents due to different system calls and different arguments.

In our design, the Draco hardware knows which registers contain which arguments of system calls. However, we can make the design more general and usable by other OS kernels. Specifically, we can add an OS-programmable table that contains the mapping between system call argument number and general-purpose register that holds it. This way, we can use arbitrary registers.

The hardware structures proposed by Draco can further support other security checks that relate to the security of transitions between different privilege domains. For example Draco can support security checks in virtualized environments, such as when the guest OS invokes the hypervisor through hypercalls. Similarly, Draco can be applied to user-level container technologies such as Google's gVisor [19], where a user-level guardian process such as the Sentry or Gofer is invoked to handle requests of less privileged application processes. Draco can also augment the security of library calls, such as in the recently-proposed Google Sandboxed API project [26].

In general, the Draco hardware structures are most attractive in processors used in domains that require both high performance and high security. A particularly relevant domain is interactive web services, such as on-line retail. Studies by Akamai, Google, and Amazon have shown that even short delays can impact online revenue [65], [66]. Furthermore, in this domain, security in paramount, and the expectation is that security checks will become more extensive in the near future.

IX. Security Issues

To understand the security issues in Draco, we consider two types of side-channels: those in the Draco hardware structures and those in the cache hierarchy.

Draco hardware structures could potentially provide side channels due to two reasons: (i) Draco uses speculation as it preloads the SLB before the system call instruction reaches the head of the ROB, and (ii) the Draco hardware structures are shared by multiple processes. Consider the first reason. An adversary could trigger SLB preloading followed by a squash, which could then speed-up a subsequent benign access that uses the same SLB entry and reveal a secret. To shield Draco from this speculation-based attack, we design the preloading mechanism carefully. The idea is to ensure that preloading leaves no side effect in the Draco hardware structures until the system call instruction reaches the ROB head.

Specifically, if an SLB preload request hits in the SLB, the LRU state of the SLB is not updated until the corresponding non-speculative SLB access. Moreover, if an SLB preload request misses, the requested VAT entry is not immediately loaded into the SLB; instead, it is stored in a Temporary Buffer. When the non-speculative SLB access is performed, the entry is moved into the SLB. If, instead, the system call instruction is squashed, the temporary buffer is cleared. Fortunately, this temporary buffer only needs to store a few entries (i.e., 8 in our design), since the number of system call instructions in the ROB at a time is small.

The second potential reason for side channels is the sharing of the Draco hardware structures by multiple processes. This case is eliminated as follows. First, in the presence of SMT, the SLB, STB, and SPT structures are partitioned on a per-context basis. Second, in all cases, when a core (or hardware context) performs a context switch to a different process, the SLB, STB, and SPT are invalidated.

Regarding side channels in the cache hierarchy, they can be eliminated using existing proposals against them. Specifically, for cache accesses due to speculative SLB preload, we can use any of the recent proposals that protect the cache hierarchy against speculation attacks (e.g., [67]–[72]). Further, for state left in the cache hierarchy as Draco hardware structures are used, we can use existing proposals like cache partitioning [73], [74]. Note that this type of potential side channel *also occurs in Seccomp*. Indeed, on a context switch, Draco may leave VAT state in the caches, but Seccomp may also leave state in the caches that reveals what system calls where checked. For these reasons, we consider side channels in the cache hierarchy beyond the scope of this paper.

X. Evaluation Methodology

We evaluate both the software and the hardware implementations of Draco. We evaluate the software implementation on the Intel Xeon E5-2660 v3 multiprocessor system described in Section IV-A; we evaluate the hardware implementation of Draco with cycle-level simulations.

A. Workloads and Metrics

We use fifteen workloads split into macro and micro benchmarks. The macro benchmarks are long-running applications, including the Elasticsearch [75], HTTPD, and NGINX web servers, Redis (an in-memory cache), Cassandra (a NoSQL database), and MySQL. We use the Yahoo! Cloud Serving Benchmark (YCSB) [76] using `workloada` and `workloadc` with 10 and 30 clients to drive Elasticsearch and Cassandra, respectively. For HTTPD and NGINX, we use `ab`, the Apache HTTP server benchmarking tool [77] with 30 concurrent requests. We drive MySQL with the OLTP workload of SysBench [78] with 10 clients. For Redis, we use the redis-benchmark [79] with 30 concurrent requests. We also evaluate Function-as-as-Service scenarios, using functions similar to

Authorized licensed use limited to: University of Illinois. Downloaded on January 22,2021 at 22:12:22 UTC from IEEE Xplore. Restrictions apply.

the sample functions of OpenFaaS [80]. Specifically, we use a `pwgen` function that generates 10K secure passwords and a `grep` function that searches patterns in the Linux source tree.

For micro benchmarks, we use FIO from SysBench [78] with 128 files of a total size of 512MB, GUPS from the HPC Challenge Benchmark (HPCC) [81], Syscall from UnixBench [82] in mix mode, and fifo, pipe, domain, and message queue from IPC Bench [83] with 1000B packets.

For macro benchmarks, we measure the mean request latency in HTTPD, NGINX, Elasticsearch, Redis, Cassandra, and MySQL, and the total execution time in the functions. For micro benchmarks, we measure the message latency in the benchmarks from IPC Bench, and the execution time in the remaining benchmarks.

B. System Call Profile Generation

There are a number of ways to create application-specific system call profiles using both dynamic and static analysis. They include system call profiling (where system calls not observed during profiling are not allowed in production) [49], [56], [84]–[86] and binary analysis [47], [48], [54].

We build our own software toolkit for automatically generating the `syscall-noargs`, `syscall-complete`, and `syscall-complete-2x` profiles described in Section IV-A for target applications. The toolkit has components for (1) attaching `strace` onto a running application to collect the system call traces, and (2) generating the Seccomp profiles that only allow the system call IDs and argument sets that appeared in the recorded traces. We choose to build our own toolkit because we find that no existing system call profiling tool includes arguments in the profiles.

For `syscall-complete-2x`, we run the `syscall-complete` profile twice in a row. Hence, the resulting profile performs twice the number of checks as `syscall-complete`. The goal of `syscall-complete-2x` is to model a near-future environment where we need more extensive security checks.

C. Modeling the Hardware Implementation of Draco

We use cycle-level full-system simulations to model a server architecture with 10 cores and 32 GB of main memory. The configuration is shown in Table II. Each core is out-of-order (OOO) and has private L1 instruction and data caches, and a private unified L2 cache. The banked L3 cache is shared.

We integrate the Simics [87] full-system simulator with the SST framework [88], together with the DRAMSim2 [89] memory simulator. Moreover, we utilize Intel SAE [90] on Simics for OS instrumentation. We use CACTI [91] for energy and access time evaluation of memory structures and the Synopsys Design Compiler [92] for evaluating the RTL implementation of the hash functions. The system call interface follows the semantics of x86 [35]. The Simics infrastructure provides the actual memory and control register contents for each system call. To evaluate the hardware components of Draco, we model them in detail in SST. For the software components, we modify the Linux kernel and instrument the system call handler and Seccomp.

Processor Parameters	
Multicore chip	10 OOO cores, 128-entry Reorder Buffer, 2GHz
L1 (D, I) cache	32KB, 8 way, write back, 2 cyc. access time (AT)
L2 cache	256KB, 8 way, write back, 8 cycle AT
L3 cache	8MB, 16 way, write back, shared, 32 cycle AT
Per-Core Draco Parameters	
STB	256 entries, 2 way, 2 cycle AT
SLB (1 arg)	32 entries, 4 way, 2 cycle AT
SLB (2 arg)	64 entries, 4 way, 2 cycle AT
SLB (3 arg)	64 entries, 4 way, 2 cycle AT
SLB (4 arg)	32 entries, 4 way, 2 cycle AT
SLB (5 arg)	32 entries, 4 way, 2 cycle AT
SLB (6 arg)	16 entries, 4 way, 2 cycle AT
Temporary Buffer	8 entries, 4 way, 2 cycle AT
SPT	384 entries, 1 way, 2 cycle AT
Main-Memory Parameters	
Capacity; Channels	32GB; 2
Ranks/Channel	8
Banks/Rank	8
Freq; Data rate	1GHz; DDR
Host and Docker Parameters	
Host OS	CentOS 7.6.1810 with Linux 3.10
Docker Engine	18.09.3

TABLE II: Architectural configuration used for evaluation.

For HTTPD, NGINX, Elasticsearch, Redis, Cassandra, and MySQL, we instrument the applications to track the beginning of the steady state. We then warm-up the architectural state by running 250 million instructions, and finally measure for two billion instructions. For functions and micro benchmarks, we warm-up the architectural state for 250 million instructions and measure for two billion instructions.

The software stack for the hardware simulations uses CentOS 7.6.1810 with Linux 3.10 and Docker Engine 18.09.03. This Linux version is older than the one used for the real-system measurements of Section IV and the evaluation of the software implementation of Draco in Section XI-A, which uses Ubuntu 18.04 with Linux 5.3.0. Our hardware simulation infrastructure could not boot the newer Linux kernel. However, note that the Draco hardware evaluation is mostly independent of the kernel version. The only exception is during the cold-start phase of the application, when the VAT structures are populated. However, our hardware simulations mostly model steady state and, therefore, the actual kernel version has negligible impact.

Appendix A repeats the real-system measurements of Section IV and the evaluation of the software implementation of Draco for CentOS 7.6.1810 with Linux 3.10.

XI. Evaluation

A. Performance of Draco

Draco Software Implementation. Figure 11 shows the performance of the workloads using the software implementation of Draco. The figure takes three Seccomp profiles from Section IV-A (`syscall-noargs`, `syscall-complete`, and `syscall-complete-2x`) and compares the performance of the workloads on a conventional system (labeled *Seccomp* in the figure) to a system augmented with software Draco (labeled *DracoSW* in the figure). The figure is organized as

Authorized licensed use limited to: University of Illinois. Downloaded on January 22,2021 at 22:12:22 UTC from IEEE Xplore. Restrictions apply.

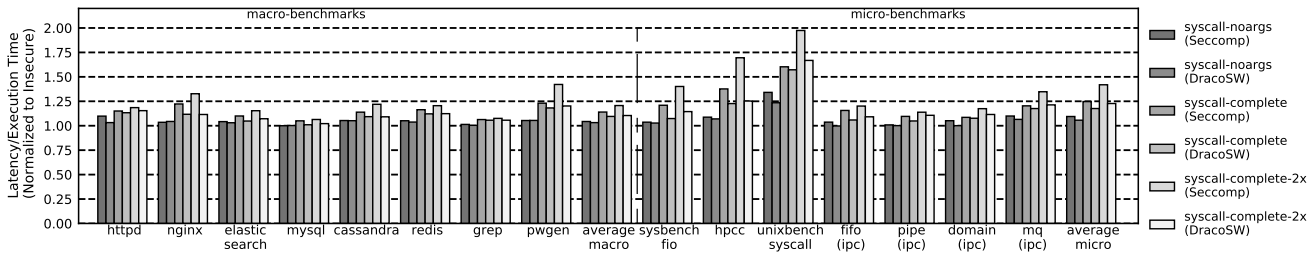


Fig. 11: Latency or execution time of the workloads using the software implementation of Draco and other environments. For each workload, the results are normalized to `insecure`.

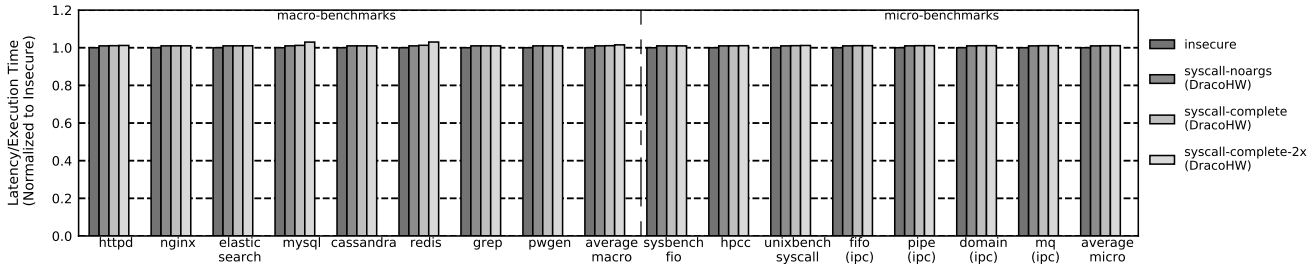


Fig. 12: Latency or execution time of the workloads using the hardware implementation of Draco. For each workload, the results are normalized to `insecure`.

Figure 2, and all the bars are normalized to the `insecure` baseline that performs no security checking.

We can see that software Draco reduces the overhead of security checking relative to Seccomp, especially for complex argument checking. Specifically, when checking both system call IDs and argument sets with `syscall-complete`, the average execution times of the macro and micro benchmarks with Seccomp are 1.14× and 1.25× higher, respectively, than `insecure`. With software Draco, these execution times are only 1.10× and 1.18× higher, respectively, than `insecure`. When checking more complex security profiles with `syscall-complete-2x`, the reductions are higher. The corresponding numbers with Seccomp are 1.21× and 1.42×; with software Draco, these numbers are 1.10× and 1.23× only.

However, we also see that the checking overhead of software Draco is still substantial, especially for some workloads. This is because the software implementation of argument checking requires expensive operations, such as hashing, VAT access, and argument comparisons. Note that our software implementation is extensively optimized. In particular, we experimented with highly-optimized hash functions used by the kernel (which use the x86 hashing instructions) and selected the most effective one.

Draco Hardware Implementation. Figure 12 shows the workload performance using the hardware implementation of Draco. The figure shows `insecure` and hardware Draco running three Seccomp profiles from Section IV-A, namely `syscall-noargs`, `syscall-complete`, and `syscall-complete-2x`. The figure is organized as Figure 2, and all the bars are normalized to `insecure`.

Figure 12 shows that hardware Draco eliminates practically all of the checking overhead under all the profiles, both when only checking system call IDs and when checking system call IDs and argument set values (including the double-size checks). In all cases, the average overhead of hardware Draco over `insecure` is 1%. Hence, hardware Draco is a secure, overhead-free design.

B. Hardware Structure Hit Ratios

To understand hardware Draco's performance, Figure 13 shows the hit rates of the STB and SLB structures. For the SLB, we show two bars: Access and Preload. *SLB Access* occurs when the system call is at the head of the ROB. *SLB Preload* occurs when the system call is inserted in the ROB and the STB is looked-up. An SLB Preload hit means only that the SLB likely contains the desired entry. An SLB Preload miss triggers a memory request to preload the entry.

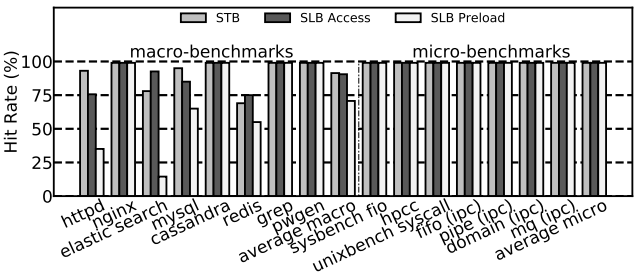


Fig. 13: Hit rates of STB and SLB (access and preload).

The figure shows that the STB hit rate is very high. It is over 93% for all the applications except for Elasticsearch and Redis. The STB captures the working set of the system calls being used. This is good news, as an STB hit is a requirement for an SLB preload.

Consider the *SLB Preload* bars. For all applications except HTTPD, Elasticsearch, MySQL, and Redis, the hit rates are

Authorized licensed use limited to: University of Illinois. Downloaded on January 22,2021 at 22:12:22 UTC from IEEE Xplore. Restrictions apply.

close to 99%. This means that, in most cases, the working set of the system call IDs and argument set values being used fits in the SLB.

However, even for these four applications, the *SLB Access* hit rates are 75–93%. This means that SLB preloading is successful in bringing most of the needed entries into the SLB on time, to deliver a hit when the entries are needed. Hence, we recommend the use of SLB preloding.

C. Draco Resource Overhead

Hardware Components. Hardware Draco introduces three hardware structures (SPT, STB, and SLB), and requires a CRC hash generator. In Table III, we present the CACTI analysis for the three hardware structures, and the Synopsys Design Compiler results for the hash generator implemented in VHDL using a linear-feedback shift register (LFSR) design. For each unit, the table shows the area, access time, dynamic energy of a read access, and leakage power. In the SLB, the area and leakage analysis includes all the subtables for the different argument counts and the temporary buffer. For the access time and dynamic energy, we show the numbers for the largest structure, namely, the three-argument subtable.

Parameter	SPT	STB	SLB	CRC Hash
Area (mm2)	0.0036	0.0063	0.01549	0.0019
Access time (ps)	105.41	131.61	112.75	964
Dyn. rd energy (pJ)	1.32	1.78	2.69	0.98
Leak. power (mW)	1.39	2.63	3.96	0.106

TABLE III: Draco hardware analysis at 22 nm.

Since all the structures are accessed in less that 150 ps, we conservatively use a 2-cycle access time for these structures. Further, since 964 ps are required to compute the CRC hash function, we account for 3 cycles in our evaluation.

SLB Sizing. Figure 14 uses a violin plot to illustrate the probability density of the number of arguments of system calls. The first entry (`linux`) corresponds to the complete Linux kernel system call interface, while the remaining entries correspond to the different applications. Taking HTTPD as an example, we see that, of all the system calls that were checked by Draco, most have three arguments, some have two and only a few have one or zero. Recall that, like Seccomp, Draco does not check pointers. For generality, we size the SLB structures based on the Linux distribution of arguments per system call.

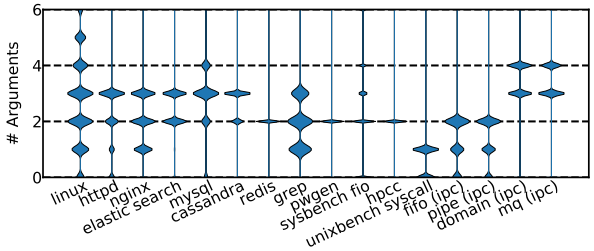


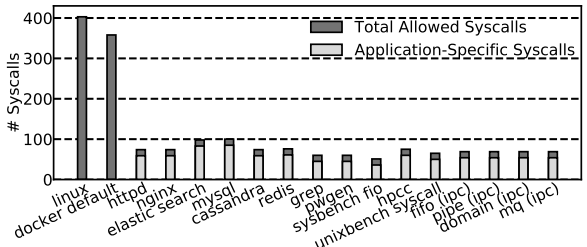
Fig. 14: Number of arguments of system calls.

VAT Memory Consumption. In a VAT, each system call has a hash table that is sized based on the number of argument sets used by that system call (e.g., based on the given Seccomp profile). Since the number of system calls is bounded, the total size of the VAT is bounded. In our evaluation, we find that the geometric mean of the VAT size for a process is 6.98KB across all evaluated applications.

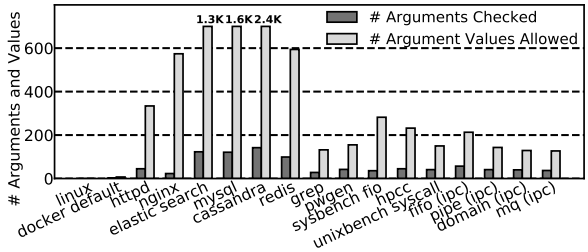
D. Assessing the Security of Application-Specific Profiles

We compare the security benefits of an application-specific profile (`syscall-complete` from Section IV-A) to the generic, commonly deployed `docker-default` profile. Recall that `syscall-complete` checks both syscall IDs and arguments, while `docker-default` only checks syscall IDs.

Figure 15(a) shows the number of different system calls allowed by the different profiles. First, `linux` shows the total number of system calls in Linux, which is 403. The next bar is `docker-default`, which allows 358 system calls. For the remaining bars, the total height is the number allowed by `syscall-complete` for each application. We can see that `syscall-complete` only allows 50–100 system calls, which increases security substantially. Moreover, the figure shows that not all of these system calls are application-specific. There is a fraction of about 20% (remaining in dark color) that are required by the container runtime. Note that, as shown in Section IV, while application-specific profiles are smaller, their checking overhead is still substantial.



(a) Number of system calls allowed.



(b) Number of system call arguments checked & values allowed.

Fig. 15: Security benefits of an application-specific profile over the default Docker profile.

Figure 15(b) shows the total number of system call arguments checked, and the number of unique argument values allowed. Linux does not check any arguments, while `docker-default` checks a total of three arguments and allows seven unique values (Section II-C). The `syscall-complete` profile checks 23–142 arguments per application, and allows 127–2458 distinct argument values per application. All these checks substantially reduce the attack surface.

Authorized licensed use limited to: University of Illinois. Downloaded on January 22,2021 at 22:12:22 UTC from IEEE Xplore. Restrictions apply.

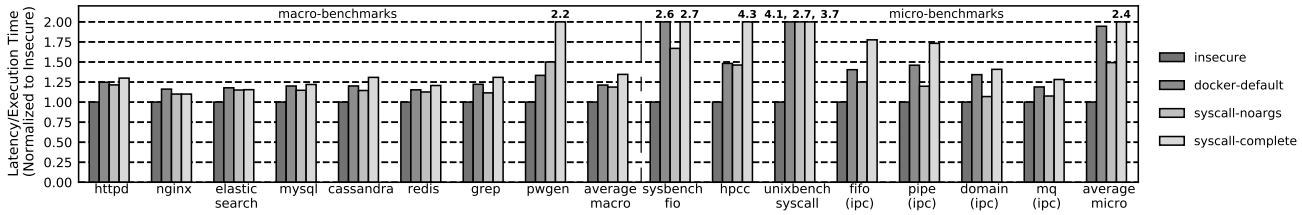


Fig. 16: Latency or execution time of the workloads using different Seccomp profiles. For each workload, the results are normalized to `insecure` (i.e., Seccomp disabled). This expertiment uses the older CentOS 7.6.1810 with Linux 3.10.

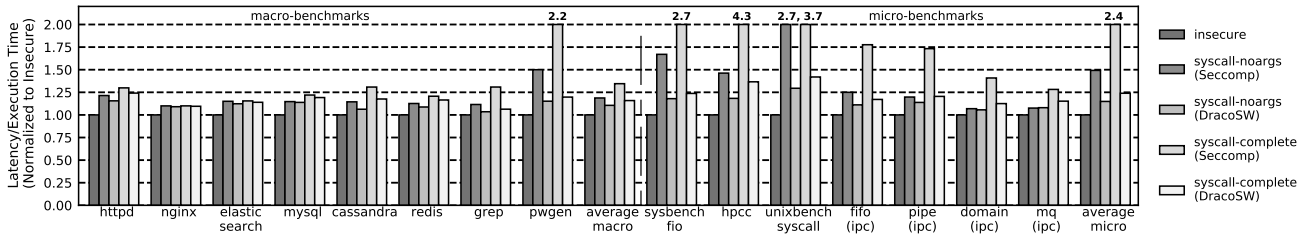


Fig. 17: Latency or execution time of the workloads using the software implementation of Draco and other environments. For each workload, the results are normalized to `insecure`. This expertiment uses the older CentOS 7.6.1810 with Linux 3.10.

XII. Other Related Work

There is a rich literature on system call checking [93]–[104]. Early implementations based on kernel tracing [93]–[99] incur excessive performance penalty, as every system call is penalized by at least two additional context switches.

A current proposal to reduce the overhead of Seccomp is to use a binary tree in `libseccomp`, to replace the branch instructions of current system call filters [27]. However, this does not fundamentally address the overhead. In Hromatka's own measurement, the binary tree-based optimization still leads to 2.4× longer system call execution time compared to Seccomp disabled [27]. Note that this result is without any argument checking. As shown in Section IV, argument checking brings further overhead, as it leads to more complex filter programs. Speculative methods [105]–[107] may not help reduce the overhead either, as their own overhead may surpass that of the checking code—they are designed for heavy security analysis such as virus scanning and taint analysis.

A few architectural security extensions have been proposed for memory-based protection, such as CHERI [108]–[110], CODOMs [111], [112], PUMP [113], REST [114], and Califorms [115]. While related, Draco has different goals and resulting design—it checks system calls rather than load/store instructions, and its goal is to protect the OS.

XIII. Conclusion

To minimize system call checking overhead, we proposed *Draco*, a new architecture that caches validated system call IDs and argument values. System calls first check a special cache and, on a hit, are declared validated. We presented both a software and a hardware implementation of Draco. Our evaluation showed that the average execution time of macro and micro benchmarks with conventional Seccomp checking was 1.14× and 1.25× higher, respectively, than on an insecure baseline that performs no security checks. With software Draco, the average execution time was reduced to 1.10× and 1.18× higher, respectively, than on the insecure baseline. Finally, with hardware Draco, the execution time was within 1% of the insecure baseline.

We expect more complex security profiles in the near future. In this case, we found that the overhead of conventional Seccomp checking increases substantially, while the overhead of software Draco goes up only modestly. The overhead of hardware Draco remains within 1% of the insecure baseline.

Acknowledgments

This work was funded in part by NSF under grants CNS-1956007, CNS-1763658, CCF-1725734, CCF-1816615, CCF-2029049, and a gift from Facebook. The authors thank Andrea Arcangeli from RedHat, Hubertus Franke and Tobin Feldman-Fitzthum from IBM for discussions on Seccomp performance, and Seung Won Min from UIUC for assisting with the Synopsys toolchain.

Appendix

This appendix repeats the real-system measurements of Section IV and the evaluation of software Draco of Section XI-A for the older CentOS 7.6.1810 with Linux 3.10 (except for the `syscall-complete-2x` profiles). The KPTI and Spectre patches are enabled. The BPF JIT is enabled but Seccomp does not make use of its functionality. These experiments were performed for the initial submission of this paper.

If we compare Figure 16 to Figure 2, we see that the newer kernel significantly improves the performance of Seccomp. Several pathological cases were eliminated and the overhead of `docker-default` is reduced. Despite these improvements, the overhead of `syscall-complete` remains significant.

If we compare Figure 17 to Figure 11, we see that the improvement attained by software Draco over Seccomp with `syscall-complete` has decreased. However, software Draco still significantly reduces overhead, especially for `syscall-complete-2x`.

Authorized licensed use limited to: University of Illinois. Downloaded on January 22,2021 at 22:12:22 UTC from IEEE Xplore. Restrictions apply.

REFERENCES

[1] "CVE-2017-5123," https://nvd.nist.gov/vuln/detail/CVE-2017-5123.
[2] "CVE-2016-0728," https://nvd.nist.gov/vuln/detail/CVE-2016-0728.
[3] "CVE-2014-3153," https://nvd.nist.gov/vuln/detail/CVE-2014-3153.
[4] "CVE-2017-18344," https://nvd.nist.gov/vuln/detail/CVE-2017-18344.
[5] "CVE-2018-18281," https://nvd.nist.gov/vuln/detail/CVE-2018-18281.
[6] "CVE-2015-3290," https://nvd.nist.gov/vuln/detail/CVE-2015-3290.
[7] "CVE-2016-5195," https://nvd.nist.gov/vuln/detail/CVE-2016-5195.
[8] "CVE-2014-9529," https://nvd.nist.gov/vuln/detail/CVE-2014-9529.
[9] "CVE-2014-4699," https://nvd.nist.gov/vuln/detail/CVE-2014-4699.
[10] "CVE-2016-2383," https://nvd.nist.gov/vuln/detail/CVE-2016-2383.
[11] J. Edge, "A seccomp overview," https://lwn.net/Articles/656307/, Sep. 2015.
[12] "OpenBSD Pledge," https://man.openbsd.org/pledge.
[13] "OpenBSD Tame," https://man.openbsd.org/OpenBSD-5.8/tame.2.
[14] "PROCESS MITIGATION SYSTEM CALL DISABLE POLICY structure," https://docs.microsoft.com/en-us/windows/win32/api/winnt/ns-winnt-process_mitigation_system_call_disable_policy.
[15] P. Lawrence, "Seccomp filter in Android O," https://android-developers.googleblog.com/2017/07/seccomp-filter-in-android-o.html, Jul. 2017.
[16] J. Corbet, "Systemd Gets Seccomp Filter Support," https://lwn.net/Articles/507067/.
[17] Docker, "Seccomp security profiles for Docker," https://docs.docker.com/engine/security/seccomp/.
[18] Ubuntu, "LXD," https://help.ubuntu.com/lts/serverguide/lxd.htmllxd-seccomp.
[19] Google, "gVisor: Container Runtime Sandbox," https://github.com/google/gvisor/blob/master/runsc/boot/filter/config.go.
[20] AWS, "Firecracker Design," https://github.com/firecracker-microvm/firecracker/blob/master/docs/design.md.
[21] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D.-M. Popa, "Firecracker: Lightweight virtualization for serverless applications," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. Santa Clara, CA: USENIX Association, 2020, pp. 419–434.
[22] Rtk Documentation, "Seccomp Isolators Guide," https://coreos.com/rkt/docs/latest/seccomp-guide.html.
[23] Singularity, "Security Options in Singularity," https://sylabs.io/guides/3.0/user-guide/security_options.html.
[24] Kubernetes Documentation, "Configure a Security Context for a Pod or Container," https://kubernetes.io/docs/tasks/configure-pod-container/security-context/, Jul. 2019.
[25] Mesos, "Linux Seccomp Support in Mesos Containerizer," http://mesos.apache.org/documentation/latest/isolators/linux-seccomp/.
[26] "Sandboxed API," https://github.com/google/sandboxed-api, 2018.
[27] T. Hromatka, "Seccomp and Libseccomp performance improvements," in *Linux Plumbers Conference 2018*, Vancouver, BC, Canada, Nov. 2018.
[28] ——, "Using a cBPF Binary Tree in Libseccomp to Improve Performance," in *Linux Plumbers Conference 2018*, Vancouver, BC, Canada, Nov. 2018.
[29] M. Kerrisk, "Using seccomp to Limit the Kernel Attack Surface," in *Linux Plumbers Conference 2015*, Seattle, WA, USA, Aug. 2015.
[30] A. Grattafiori, "Understanding and Hardening Linux Containers," NCC Group, Tech. Rep., Jun. 2016.
[31] T. Kim and N. Zeldovich, "Practical and Effective Sandboxing for Non-root Users," in *Proceedings of the 2013 USENIX Conference on Annual Technical Conference (USENIX ATC'13)*, San Jose, CA, Jun. 2013.
[32] C. Kruegel, D. Mutz, F. Valeur, and G. Vigna, "On the Detection of Anomalous System Call Arguments," in *Proceedings of the 8th European Symposium on Research in Computer Security*, Gjøvik, Norway, Oct. 2003.
[33] D. Mutz, F. Valeur, G. Vigna, and C. Kruegel, "Anomalous System Call Detection," *ACM Transactions on Information and System Security (TISSEC)*, vol. 9, no. 1, pp. 61–93, Feb. 2006.
[34] F. Maggi, M. Matteucci, and S. Zanero, "Detecting Intrusions through System Call Sequence and Argument Analysis," *IEEE Transactions on Dependable and Secure Computing (TDSC)*, vol. 7, no. 4, pp. 381–395, Oct. 2010.
[35] Intel, "Intel 64 and IA-32 Architectures Software Developer's Manual," 2019.
[36] Linux, "Secure Computing with filters," https://www.kernel.org/doc/Documentation/prctl/seccomp_filter.txt.
[37] T. Garfinkel, "Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools," in *Proceedings of the 2004 Network and Distributed System Security Symposium (NDSS'04)*, San Diego, California, Feb. 2003.
[38] R. N. M. Watson, "Exploiting Concurrency Vulnerabilities in System Call Wrappers," in *Proceedings of the 1st USENIX Workshop on Offensive Technologies (WOOT'07)*, Boston, MA, USA, Aug. 2007.
[39] AWS, "Firecracker microVMs," https://github.com/firecracker-microvm/firecracker/blob/master/vmm/src/default_syscalls/filters.rs.
[40] Moby Project, "A collaborative project for the container ecosystem to assemble container-based systems," https://github.com/moby/moby/blob/master/profiles/seccomp/default.json.
[41] Singularity, "Singularity: Application Containers for Linux," https://github.com/sylabs/singularity/blob/master/etc/seccomp-profiles/default.json.
[42] Subgraph, "Repository of maintained OZ profiles and seccomp filters," https://github.com/subgraph/subgraph-oz-profiles.
[43] "QEMU," https://github.com/qemu/qemu/blob/master/qemu-seccomp.c.
[44] Julien Tinnes, "A safer playground for your Linux and Chrome OS renderers," https://blog.cr0.org/2012/09/introducing-chromes-next-generation.html, Nov. 2012.
[45] ——, "Introducing Chrome's next-generation Linux sandbox," https://blog.cr0.org/2012/09/introducing-chromes-next-generation.html, Sep. 2012.
[46] "OZ: a sandboxing system targeting everyday workstation applications," https://github.com/subgraph/oz/blob/master/oz-seccomp/tracer.go, 2018.
[47] "binctr: Fully static, unprivileged, self-contained, containers as executable binaries," https://github.com/genuinetools/binctr.
[48] "go2seccomp: Generate seccomp profiles from go binaries," https://github.com/xfernando/go2seccomp.
[49] oci-seccomp-bpf-hook, "OCI hook to trace syscalls and generate a seccomp profile," https://github.com/containers/oci-seccomp-bpf-hook.
[50] "Kubernetes Seccomp Operator," https://github.com/kubernetes-sigs/seccomp-operator.
[51] S. Kerner, "The future of Docker containers," https://lwn.net/Articles/788282/.
[52] Red Hat, "Configuring OpenShift Container Platform for a Custom Seccomp Profile," https://docs.openshift.com/container-platform/3.5/admin_guide/seccomp.htmlseccomp-configuring-openshift-with-custom-seccomp.
[53] Rtk Documentation, "Overriding Seccomp Filters," https://coreos.com/rkt/docs/latest/seccomp-guide.htmloverriding-seccomp-filters.
[54] Q. Zeng, Z. Xin, D. Wu, P. Liu, and B. Mao, "Tailored Application-specific System Call Tables," The Pennsylvania State University, Tech. Rep., 2014.
[55] C.-C. Tsai, B. Jain, N. A. Abdul, and D. E. Porter, "A Study of Modern Linux API Usage and Compatibility: What to Support When You're Supporting," in *Proceedings of the 11th European Conference on Computer Systems (EuroSys'16)*, London, United Kingdom, Apr. 2016.
[56] Z. Wan, D. Lo, X. Xia, L. Cai, and S. Li, "Mining Sandboxes for Linux Containers," in *Proceedings of 2017 IEEE International Conference on Software Testing, Verification and Validation (ICST'17)*, Tokyo, Japan, Mar. 2017.
[57] Moby Project, "An open framework to assemble specialized container systems without reinventing the wheel," https://mobyproject.org/, 2019.
[58] K. McAllister, "Attacking hardened Linux systems with kernel JIT spraying," https://mainisusuallyafunction.blogspot.com/2012/11/attacking-hardened-linux-systems-with.html, Nov. 2012.
[59] E. Reshetova, F. Bonazzi, and N. Asokan, "Randomization can't stop BPF JIT spray," https://www.blackhat.com/docs/eu-16/materials/eu-16-Reshetova-Randomization-Can't-Stop-BPF-JIT-Spray-wp.pdf, Nov. 2016.
[60] R. Gawlik and T. Holz, "SoK: Make JIT-Spray Great Again," in *Proceedings of the 12th USENIX Workshop on Offensive Technologies (WOOT'18)*, 2018.
[61] R. Pagh and F. F. Rodler, "Cuckoo Hashing," *Journal of Algorithms*, vol. 51, no. 2, pp. 122–144, May 2004.

Authorized licensed use limited to: University of Illinois. Downloaded on January 22,2021 at 22:12:22 UTC from IEEE Xplore. Restrictions apply.

[62] D. Fotakis, R. Pagh, P. Sanders, and P. Spirakis, "Space Efficient Hash Tables with Worst Case Constant Access Time," *Theory of Computing Systems*, vol. 38, no. 2, pp. 229–248, Feb 2005.

[63] ECMA International, "Standard ECMA-182," https://www.ecma-international.org/publications/standards/Ecma-182.htm.

[64] Microsoft, "Windows System Call Disable Policy," https://docs.microsoft.com/en-us/windows/win32/api/winnt/ns-winnt-process_mitigation_system_call_disable_policy.

[65] Akamai, "Akamai Online Retail Performance Report: Milliseconds Are Critical," https://www.akamai.com/uk/en/about/news/press/2017-press/akamai-releases-spring-2017-state-of-online-retail-performance-report.jsp.

[66] Google, "Find out how you stack up to new industry benchmarks for mobile page speed," https://www.thinkwithgoogle.com/marketing-resources/data-measurement/mobile-page-speed-new-industry-benchmarks/.

[67] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. W. Fletcher, and J. Torrellas, "InvisiSpec: Making Speculative Execution Invisible in the Cache Hierarchy," in *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-51)*, 2018.

[68] M. Taram, A. Venkat, and D. Tullsen, "Context-Sensitive Fencing: Securing Speculative Execution via Microcode Customization," in *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'19)*, 2019.

[69] C. Sakalis, S. Kaxiras, A. Ros, A. Jimborean, and M. Själander, "Efficient Invisible Speculative Execution Through Selective Delay and Value Prediction," in *Proceedings of the 46th International Symposium on Computer Architecture (ISCA'19)*, 2019.

[70] K. N. Khasawneh, E. M. Koruyeh, C. Song, D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, "SafeSpec: Banishing the Spectre of a Meltdown with Leakage-Free Speculation," in *Proceedings of the 56th ACM/IEEE Design Automation Conference (DAC'19)*, 2019.

[71] G. Saileshwar and M. K. Qureshi, "CleanupSpec: An "Undo" Approach to Safe Speculation," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-52)*, 2019.

[72] P. Li, L. Zhao, R. Hou, L. Zhang, and D. Meng, "Conditional Speculation: An Effective Approach to Safeguard Out-of-Order Execution Against Spectre Attacks," in *International Symposium on High Performance Computer Architecture*, 2019.

[73] F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, and R. B. Lee, "Catalyst: Defeating last-level cache side channel attacks in cloud computing," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016, pp. 406–418.

[74] Y. Wang, A. Ferraiuolo, D. Zhang, A. C. Myers, and G. E. Suh, "SecDCP: Secure Dynamic Cache Partitioning for Efficient Timing Channel Protection," in *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2016.

[75] Elastic, "Elasticsearch: A Distributed RESTful Search Engine," https://github.com/elastic/elasticsearch, 2019.

[76] Yahoo!, "Yahoo! Cloud Serving Benchmark," https://github.com/brianfrankcooper/YCSB, 2019.

[77] Apache, "ab - Apache HTTP server benchmarking tool," https://httpd.apache.org/docs/2.4/programs/ab.html, 2019.

[78] SysBench, "Scriptable database and system performance benchmark," https://github.com/akopytov/sysbench.

[79] Redis, "How fast is Redis?" https://redis.io/topics/benchmarks, 2019.

[80] OpenFaaS, "OpenFaaS Sample Functions," https://github.com/openfaas/faas/tree/master/sample-functions.

[81] HPCC, "HPC Challenge Benchmark," https://icl.utk.edu/hpcc/.

[82] UnixBench, "BYTE UNIX benchmark suite," https://github.com/kdlucas/byte-unixbench.

[83] IPCBench, "Benchmarks for Inter-Process-Communication Techniques," https://github.com/goldsborough/ipc-bench.

[84] L. Lei, J. Sun, K. Sun, C. Shenefiel, R. Ma, Y. Wang, and Q. Li, "SPEAKER: Split-Phase Execution of Application Containers," in *Proceedings of the 14th Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA'17)*, Bonn, Germany, 2017.

[85] Heroku Engineering Blog, "Applying Seccomp Filters at Runtime for Go Binaries," https://blog.heroku.com/applying-seccomp-filters-on-go-binaries, Aug. 2018.

[86] Adobe Security, "Better Security Hygiene for Containers," https://blogs.adobe.com/security/2018/08/better-security-hygiene-for-containers.html, 2018.

[87] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A Full System Simulation Platform," *IEEE Computer*, vol. 35, no. 2, pp. 50–58, Feb. 2002.

[88] A. F. Rodrigues, J. Cook, E. Cooper-Balis, K. S. Hemmert, C. Kersey, R. Riesen, P. Rosenfeld, R. Oldfield, and M. Weston, "The Structural Simulation Toolkit," in *Proceedings of the 2006 ACM/IEEE conference on Supercomputing (SC'10)*, Tampa, Florida, Nov. 2006.

[89] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "DRAMSim2: A Cycle Accurate Memory System Simulator," *IEEE Computer Architecture Letters*, vol. 10, no. 1, pp. 16–19, Jan 2011.

[90] N. Chachmon, D. Richins, R. Cohn, M. Christensson, W. Cui, and V. J. Reddi, "Simulation and Analysis Engine for Scale-Out Workloads," in *Proceedings of the 2016 International Conference on Supercomputing (ICS'16)*, 2016.

[91] R. Balasubramonian, A. B. Kahng, N. Muralimanohar, A. Shafiee, and V. Srinivas, "CACTI 7: New Tools for Interconnect Exploration in Innovative Off-Chip Memories," *ACM Transactions on Architecture and Code Optimization*, vol. 14, no. 2, pp. 14:1–14:25, Jun. 2017.

[92] Synopsys, "Synopsys Design Compiler," https://www.synopsys.com/.

[93] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer, "A Secure Environment for Untrusted Helper Applications Confining the Wily Hacker," in *Proceedings of the 6th USENIX Security Symposium*, San Jose, California, Jul. 1996.

[94] D. A. Wagner, "Janus: an Approach for Confinement of Untrusted Applications," EECS Department, University of California, Berkeley, Tech. Rep. UCB/CSD-99-1056, 2016.

[95] N. Provos, "Improving Host Security with System Call Policies," in *Proceedings of the 12th USENIX Security Symposium*, Washington, DC, USA, Aug. 2003.

[96] K. Jain and R. Sekar, "User-Level Infrastructure for System Call Interposition: A Platform for Intrusion Detection and Confinement," in *Proceedings of the 2000 Network and Distributed System Security Symposium (NDSS'00)*, San Diego, California, USA, Feb. 2000.

[97] T. Garfinkel, B. Pfaff, and M. Rosenblum, "Ostia: A Delegating Architecture for Secure System Call Interposition," in *Proceedings of the 2004 Network and Distributed System Security Symposium (NDSS'04)*, San Diego, California, Feb. 2004.

[98] A. Acharya and M. Raje, "MAPbox: Using Parameterized Behavior Classes to Confine Untrusted Applications," in *Proceedings of the 9th USENIX Security Symposium*, Denver, Colorado, USA, Aug. 2000.

[99] A. Alexandrov, P. Kmiec, and K. Schauser, "Consh: Confined Execution Environment for Internet Computations," The University of California, Santa Barbara, Tech. Rep., 1999.

[100] D. S. Peterson, M. Bishop, and R. Pandey, "A Flexible Containment Mechanism for Executing Untrusted Code," in *Proceedings of the 11th USENIX Security Symposium*, San Francisco, CA, USA, Aug. 2002.

[101] T. Fraser, L. Badger, and M. Feldman, "Hardening COTS Software with Generic Software Wrappers," in *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, 1999.

[102] C. M. Linn, M. Rajagopalan, S. Baker, C. Collberg, S. K. Debray, and J. H. Hartman, "Protecting Against Unexpected System Calls," in *Proceedings of the 14th USENIX Security Symposium*, Baltimore, MD, USA, Jul. 2005.

[103] A. Dan, A. Mohindra, R. Ramaswami, and D. Sitara, "ChakraVyuha (CV): A Sandbox Operating System Environment for Controlled Execution of Alien Code," IBM Research Division, T.J. Watson Research Center, Tech. Rep. RC 20742 (2/20/97), Feb. 1997.

[104] D. P. Ghormley, D. Petrou, S. H. Rodrigues, and T. E. Anderson, "SLIC: An Extensibility System for Commodity Operating Systems," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference (USENIX ATC'98)*, New Orleans, Louisiana, Jun. 1998.

[105] E. B. Nightingale, D. Peek, P. M. Chen, and J. Flinn, "Parallelizing Security Checks on Commodity Hardware," in *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIII)*, Seattle, WA, USA, Mar. 2008.

[106] Y. Oyama, K. Onoue, and A. Yonezawa, "Speculative Security Checks in Sandboxing Systems," in *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05)*, Denver, CO, USA, Apr. 2005.

Authorized licensed use limited to: University of Illinois. Downloaded on January 22,2021 at 22:12:22 UTC from IEEE Xplore. Restrictions apply.

[107] H. Chen, X. Wu, L. Yuan, B. Zang, P.-c. Yew, and F. T. Chong, "From Speculation to Security: Practical and Efficient Information Flow Tracking Using Speculative Hardware," in *Proceedings of the 35th Annual International Symposium on Computer Architecture (ISCA'08)*, Beijing, China, Jun. 2008.

[108] J. Woodruff, R. N. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe, "The CHERI Capability Model: Revisiting RISC in an Age of Risk," in *Proceeding of the 41st Annual International Symposium on Computer Architecture (ISCA'14)*, 2014.

[109] H. Xia, J. Woodruff, S. Ainsworth, N. W. Filardo, M. Roe, A. Richardson, P. Rugg, P. G. Neumann, S. W. Moore, R. N. M. Watson, and T. M. Jones, "CHERIvoke: Characterising Pointer Revocation Using CHERI Capabilities for Temporal Memory Safety," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-52)*, 2019.

[110] R. N. M. Watson, R. M. Norton, J. Woodruff, S. W. Moore, P. G. Neumann, J. Anderson, D. Chisnall, B. Davis, B. Laurie, M. Roe, N. H. Dave, K. Gudka, A. Joannou, A. T. Markettos, E. Maste, S. J. Murdoch, C. Rothwell, S. D. Son, and M. Vadera, "Fast Protection-Domain Crossing in the CHERI Capability-System Architecture," *IEEE Micro*, vol. 36, no. 5, pp. 38–49, Jan. 2016.

[111] L. Vilanova, M. Ben-Yehuda, N. Navarro, Y. Etsion, and M. Valero, "CODOMs: Protecting Software with Code-Centric Memory Domains," in *Proceeding of the 41st Annual International Symposium on Computer Architecture (ISCA'14)*, 2014.

[112] L. Vilanova, M. Jordà, N. Navarro, Y. Etsion, and M. Valero, "Direct Inter-Process Communication (DIPC): Repurposing the CODOMs Architecture to Accelerate IPC," in *Proceedings of the 12th European Conference on Computer Systems (EuroSys'17)*, 2017.

[113] U. Dhawan, C. Hritcu, R. Rubin, N. Vasilakis, S. Chiricescu, J. M. Smith, T. F. Knight, B. C. Pierce, and A. DeHon, "Architectural Support for Software-Defined Metadata Processing," in *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'15)*, 2015.

[114] K. Sinha and S. Sethumadhavan, "Practical Memory Safety with REST," in *Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA'18)*, 2018.

[115] H. Sasaki, M. A. Arroyo, M. T. I. Ziad, K. Bhat, K. Sinha, and S. Sethumadhavan, "Practical Byte-Granular Memory Blacklisting Using Califorms," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-52)*, 2019.

Authorized licensed use limited to: University of Illinois. Downloaded on January 22,2021 at 22:12:22 UTC from IEEE Xplore. Restrictions apply.