

Seminar Report on  
**Light-weight Contexts**

An OS Abstraction for Safety and Performance

James Litton<sup>1,2</sup> Anjo Vahldiek-Oberwagner<sup>2</sup> Eslam Elnikety<sup>2</sup> Deepak Garg<sup>2</sup> Bobby Bhattacharjee<sup>1</sup> Peter Druschel<sup>2</sup>

<sup>1</sup>University of Maryland, College Park <sup>2</sup>Max Planck Institute for Software Systems

Report by Christian Schwarz

### Abstract

Contemporary operating systems define processes as an execution environment for threads, consisting of a shared address space, file descriptor table, and system-level privilege. The low overhead of threads compared to processes enables more efficient server applications, but prohibits meaningful memory isolation and privilege separation between potentially malicious requests.

Light-weight contexts (lwCs) provide an OS abstraction for privilege separation within a process by decoupling execution environment & logical control flow from process & threads: an lwC is comprised of an address space, file descriptor table and system credential, as well as one logical control flow per thread. By allowing threads to rapidly switch between lwCs, pass arguments, and selectively share resources, compartmentalizing requests or code modules becomes possible.

In this report, we summarize the original lwC design and perform a security analysis, including a threat model and example use case. We proceed with a discussion of the authors' evaluation of lwCs on FreeBSD 11.0 in PHP-FPM and the Apache & nginx web servers. We conclude with a critique of the design & evaluation, pointing out substantial flaws and open questions with regards to real-world performance.

## 1 Introduction

Modern application architecture emphasizes modularization and information hiding to achieve testability, maintainability, exchangeability, and reusability. The extreme of reusability are large libraries that are used by multiple independent applications. Type systems, packaging hierarchies, and visibility features in many programming languages can help to enforce said principles in large applications.

However, there is a chasm between the way we architect software and the way it is executed it at runtime: whereas the programming language enforces

strict separation between modules on the type level at compile time, the program binary or interpreted script executes in a single protection domain — the process — with shared address space, file descriptor table and system-level privilege for all modules. Consequently, an exploitable vulnerability in a single logical module can be used to compromise the entire application, extract user data from application memory or serve as a basis for further privilege escalation.

Language technology provides holistic solutions to this problem through memory-safe languages, runtime ACL systems, etc. However, performance concerns, lack of interoperability between different source languages within a process, and legacy code bases shift the search for better runtime safety guarantees toward the next lower layer of the software stack: Can the operating system provide an efficient abstraction to maintain some of the aforementioned compile-time isolation at runtime?

Light-weight contexts (lwCs) are such an OS-based solution that allows for multiple protection domains (*contexts*) within the same process: An lwC is comprised of an address space, file descriptor table and system credential (privilege level). In contrast to the implicit execution environment provided by a process, lwCs are a first-class OS abstraction and explicitly tangible from user space as file descriptors. [Lit+16]

In the proposed design, threads no longer execute a single logical control flow, but have one *per lwC*. A new system call allows for rapid voluntary switching to a different lwC: apart from resuming the thread's target lwC logical control flow, the system call also installs the target lwC's address space, file descriptor table and system credential for the current thread. Argument passing on switch then enables the decomposition of an application's functionality into multiple lwCs that can provide independent security guarantees. [Lit+16]

Light-weight contexts can be created as snapshots of the current lwCs address space layout, file descriptors and privilege level. Dynamic sharing of these resources is also supported through a capability system built on top of lwC file descriptors. Privilege escalation through out-of-process channels can be prevented using syscall interposition, which enables efficient program-defined policy enforcement for syscalls made by an lwC. [Lit+16]

The authors provide an evaluation of their implementation in FreeBSD 11.0, focusing on the enhancement of inter-request isolation and TLS private key protection in web servers. The presented results are impressive at first glance, but we are going to show that the evaluation lacks key metrics for the presented use cases and does not sufficiently investigate obvious performance-constraints of the design.

## 1.1 Structure of this Report

This report provides a summary of the original light-weight context paper enriched with several insights drawn from the authors’ open source implementation. Section 2 provides an overview of the design and system API exposed by lwCs, followed by a brief security analysis in section 3, which also includes an example use case for lwCs in application security hardening. We give an overview of the changes made to FreeBSD 11.0 in order to support lwCs in section 4 and proceed with a discussion of the evaluation results of the original paper (section 5). Subsequently, we survey other approaches to application compartmentalization in section 6. Finally, we provide an elaborate critique of the design & evaluation in section 7.

## 2 Design

We find it most helpful to develop the general idea of *light-weight Contexts* (lwCs) by starting from the canonical abstraction of processes & threads, as visualized in figure 1a: Conventionally, processes define an execution environment which is shared by one or more threads. The execution environment consists of an address space, a file descriptor table and a representation of the process’s system-wide privileges (*credential*). A thread has **two closely related but separate roles**: **first**, it represents a **single unit of logical control flow** within the process-defined environment. Control flow has associated state, e.g., instruction pointer, stack pointer, general purpose register contents, FPU state, etc. That state resides in a CPU’s registers while the

thread is executing, or in the thread control block (TCB). The **second** role of a thread is that of a **scheduling entity**: The scheduler time-multiplexes threads onto CPU cores and implements the concept of blocking & waiting between threads. The scheduler state required for this task is stored in the TCB.

The authors introduce light-weight contexts as a new OS abstraction and restructure the roles of canonical processes & threads, as visualized in figure 1b:

- An lwC represents a single protection domain (address space, file descriptor table and system credential) and all logical control flows within that protection domain.
- Threads always execute within one lwC at any given time and always execute the same logical control flow while within that lwC.
- Multiple threads can execute simultaneously within an lwCs, sharing a protection domain but executing different logical flows.
- lwCs are represented as file descriptors, and thus explicitly tangible from user space.
- Threads can switch protection domains and logical control flow by switching between lwCs.

[Lit+16; LWck16]

In the following subsections, we provide an outline of the system API for managing and using lwCs in an application.

### 2.1 lwC Switching

We start our survey of the lwC API surface with the most central functionality: switching between lwCs. We accept the existence of multiple lwCs for now and come back to lwC creation in the next subsection.

Threads can switch between lwCs by invoking the `lwcSwitch` system call:

```
caller, carg := lwcSwitch(target, arg)
```

The first argument `target` specifies the file descriptor of the lwC into which the calling thread wants to switch. When invoking the system call, the kernel

- saves the current control flow state into the current thread’s lwC,
- atomically switches to the new protection domain by installing the target lwC address space, file descriptor table and credential for the current thread,
- and restores the control flow state saved for the current in thread in the target lwC.

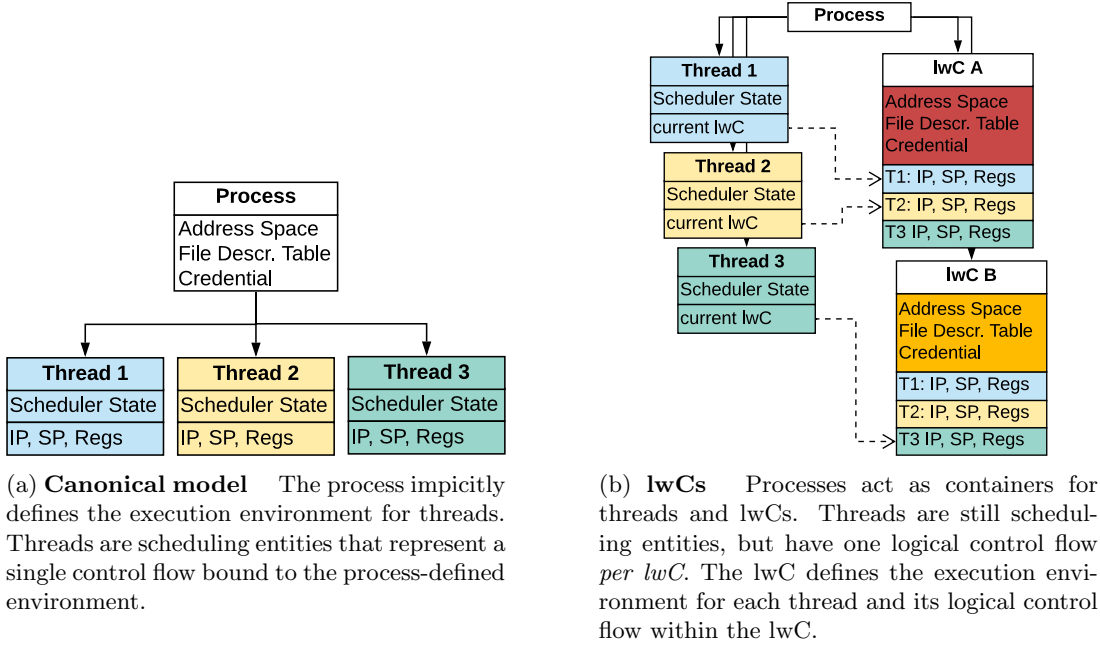


Figure 1: Canonical processes and threads vs. the lwC design.

Pseudo code for this procedure is provided in figure 2. [LWck16; Lit+16]

It is crucial to understand that **execution after a switch always resumes at an lwC call site**, except for the very first switch into a newly created lwC (see next section). Note that this behavior is analogous to a voluntary context switch, e.g., with `pthread_yield`. However, in contrast to the canonical model of processes & threads, **lwCSwitch does not switch to another scheduling entity**: from the scheduler’s perspective, it is still the same scheduling entity (formerly thread) that is executing on the CPU. Regular context saving and restoration after interrupts, exceptions or when a thread blocks, is left unchanged. Only the memory location for off-cpu control flow state changes from the thread’s control block to the thread’s slot in the current lwC. [LWck16]

```

syscall
SP, IP, ... → curLWC->tcb[tid]
curThd->vmospace = targetLWC->vmospace
curThd->fdt = targetLWC->fdt
curThd->cred = targetLWC->cred
curLWC = targetLWC
SP, IP, ... ← targetLWC->tcb[tid]
(arg kept in register)
sysret

```

Figure 2: Pseudo code for lwCSwitch.

The second argument to `lwCSwitch` is an opaque value called **arg** which is passed through to the code that starts executing in the target lwC after the switch is completed. **arg** is made available in the target lwC as the return value **carg** (we remember that execution always resumes at an lwC call site). The second return value **caller** is the file descriptor of the lwC from where the switch was initiated. [Lit+16]

Note that these three properties of `lwCSwitch` (kernel-moderated control flow switching, address space switching, and argument passing) enable the construction of lwCs that fulfill a server-like role within an application, providing independent security properties: as will be discussed in section 3.1, one lwC can provide some high-assurance service to other lwCs because it has a private address space and can enforce specific entry-points (`lwCSwitch` call sites) where argument validation can be performed.

## 2.2 lwC Creation & Destruction

A process in an lwC system starts with a single thread that runs within a root lwC created by the OS. This design enables backwards binary-compatibility with the exception that the root lwC file descriptor is a well-known number analogous to those for `stdio`. [Lit+16]

New lwCs can be created from any thread with the

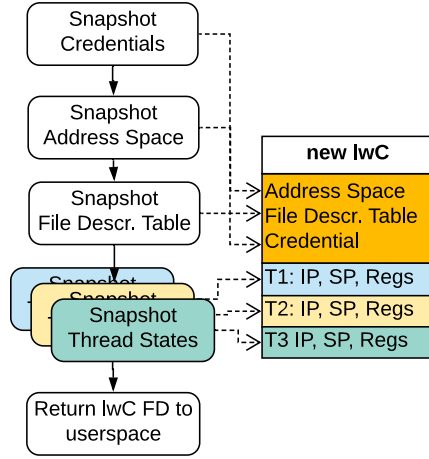


Figure 3: Steps involved in lwc creation. By default, a new lwc is a copy (snapshot) of the current lwc’s resources. All threads that exist at the time of lwc creation can enter the new lwc.

`lwcCreate` system call. By default, a new lwc is a snapshot of the calling thread’s current lwc, as depicted in figure 3: The kernel first creates copies of the resources *address space*, *file descriptor table* and *credentials* and stores them in the new lwc. It then temporarily preempts all threads that execute in the current lwc and stores their control flow state in the new lwc. The execution state of other threads that can switch to the creating lwCs is copied to the new lwc. Finally, the file descriptor referring to the new lwc is returned to user space in the return value `new`. [Lit+16]

Note that in contrast to the original paper, we avoid the term *fork* when describing the lwc creation procedure: forking has the same resource snapshotting semantics as *lwcCreate*, but also implies the creation of a new process and scheduling entity. *lwcCreate* only creates a copy of the protection domain and a handle for existing threads to switch to it later — no independent scheduling entity is created.

However, one problem of forking translates to lwc creation: what happens when we first switch into a newly created lwc? How does snapshotting interact with multi-threading? After all, we preempt and snapshot all threads’ control flow states on lwc creation, and the `lwcSwitch` implementation restores that state on first switch, expecting that it was an `lwcSwitch` call site. We distinguish two cases: the `lwcCreate`ing thread returns “a second time” from that syscall and populates `lwcCreate`’s `caller` and `carg` return values with those of `lwcSwitch`. Additionally, the `new` return value contains the lwc

descriptor of the creator lwc. The behavior in the creating thread is thus well-defined. The second case covers other threads, which could be at random points in their logical control flow when being snapshotted. Switching to such threads results in undefined behavior — we refer to introductory literature about multi-threading and forking for details. The authors recommend usage of barrier-synchronization to established well-defined initial entrypoints for all threads in the new lwc. However, it is unclear to us how the `caller` and `carg` return values are accessible without inline-assembly or language support for non-creator threads. [Lit+16]

Note that a new lwc only stores control flow states for the threads that existed at the time of its creation: threads created after the lwc cannot switch into the lwc.

new, caller, carg := lwcCreate(rspece)			
	new	caller	carg
creator	new lwc fd	$\perp$	$\perp$
new lwc	creator lwc fd	like lwcSwitch	

Figure 4: `lwcCreate` return values are different in creator and new lwc to ensure well-defined behavior when the creator switches to the new lwc for the first time.

### 2.2.1 lwc Resource Specifiers

The previous section presented the default behavior of `lwcCreate` for an empty `rspece` argument. By default, a new lwc is a snapshot copy of the current lwc’s resources *address space*, *file descriptor table* and *credential*.

*Resource specifiers* provide fine-grained control of the resource sharing on `lwcCreate`: callers can specify per resource-kind whether the resource shall be *copied to*, *unmapped from* or *shared* with the new lwc. Address space and file descriptor sharing can be specified on a per-page or per-descriptor basis. Figure 5 summarizes the different combinations of resources and sharing behavior. [Lit+16]

## 2.3 Dynamic Resource Sharing

Apart from sharing at lwc creation time, it is also possible to dynamically copy or share mem-

language,  
prettier  
table

`rspecs := [{Resource , [Start, End], How}]`

What How	Address Space	File Descriptor Table	Credential
COPY	copy on write sharing	dup open FDs once	copy credential kobject
UNMAP	do not map	do not dup FDs	$\perp$
SHARE	shared memory	share kobject	share kobject

Figure 5: Resource specifiers change the default snapshot semantics of `lwcCreate`: each resource can be copied, shared or unmapped from the new lwC. Address space and file descriptor sharing can be configured per page / per descriptor. *copy* sharing is roughly equivalent to what happens on `fork`, *share* to what happens on thread creation.

ory and file descriptors between lwCs. using the `lwcOverlay(src, rspecs)` system call. The `src` argument is the lwC descriptor of the lwC from which the subset of resources specified in `rspecs` should be mapped (*overlaid*) into the current lwC. The target address, or file descriptor number, respectively, is guaranteed to be the same as in `src`. To unmap an existing overlay, `src` must be set to the current lwC and `rspecs` must be filled with `UNMAP` resource specifiers. Overlays do not support stacking: after unmapping an overlay that overlaid an existing memory mapping, that original memory mapping is not automatically remapped. System credential overlays enable temporary privilege escalation to the credential of the `src` lwC. [Lit+16]

Overlays are subject to an **access capability system**: the lwC descriptor kernel representation (`struct filedescent`) carries an rspec-like mask that represents permitted overlays. `lwcOverlay` checks that the requested overlay `rspec` is a subset of the permitted-overlay mask. The rules for modifying the permitted-overlay mask are as follows:

- After lwC creation, the creator receives the new lwC descriptor with a universal access capability to the child lwC.
- After lwC creation, the child receives the lwC descriptor of its creator in `new`, equipped with access capabilities to ranges flagged with `LWC_MAY_ACCESS` on `lwcCreate`.
- Access capabilities cannot be extended.
- Access capabilities can be reduced with the `lwcRestrict(target, rspecs)` system call by any lwC that holds the lwC descriptor `target`: after a successful `lwcRestrict`, the resource ranges specified in `rspecs` can no longer be overlaid using `lwcOverlay(target, ...)`.
- Access capabilities are invariant across overlays: to the file descriptor entry in the kernel, and is therefore invariant across overlays.

Note that revocation of existing overlays is not possible, i.e., `lwcRestrict` only affects future overlays. [LWck16; Lit+16]

## 2.4 Syscall Interposition

It can be desirable to limit the system calls an lwC may perform, e.g., to limit file system access to a subdirectory, prohibit network communication, or avoid privilege escalation through other system calls. The lwC design builds onto existing syscall filtering and mechanisms, e.g. Capsicum capability mode on FreeBSD or seccomp on Linux: when an lwC is created with the `LWC_TRAP_SYSCALL` flag, system calls made by the new lwC or any of its children that would normally trap due to the syscall filtering mechanism are redirected to the creator as an `lwcSwitch`. The `caller` return value of `lwcSwitch` is set to the trapping lwC and `cargs` contains the syscall arguments for inspection by the trap-handling lwC. [Lit+16]

The trap-handling lwC can use the `lwcSyscall` syscall execute a system call **in the context of the trapping lwC**: the `mask` argument allows the trap-handling lwC to choose per resource type<sup>1</sup> whether its own resource mapping or that of the trapping lwC should be used while the syscall is executed. It is further possible to make `lwcSyscall` return to the trapping lwC on completion of the syscall. Both options are optimizations that avoid temporary overlays and additional context switches. [Lit+16; LWck16]

```
lwcSyscall(trappingLwc, mask,
           syscall, syscall-args)
```

## 2.5 Signal Delivery

UNIX signal delivery with lwCs faces similar design questions as signal delivery in multi-threaded

<sup>1</sup>granularity: all memory, all file descriptors, the credential

cite

applications: To which lwCs should a given signal be delivered? The authors' solution is classification of signals as either *attributable* or *non-attributable*: attributable signals such as for segmentation faults are always delivered to the thread in the lwC that caused the signal to occur. Non-attributable signals are delivered to *all* lwCs created with a corresponding flag. Consequently, each thread must have a separate signal mask per lwC. [Lit+16]

## 2.6 Forking & Exit

A forked process inherits the parent's lwCs. However, only the forking thread exists in the child process, presumably due to POSIX compliance[OG04]. Shared memory established through `lwcCreate`, `lwcOverlay`, or `mmap(..., MAP_SHARED)`, stays shared across forks. The authors do not specify the behavior for file descriptor tables and credential; we assume unmodified fork semantics per lwC. [Lit+16]

The `exit` system call terminates the entire process, regardless of the lwC in which it was called. Syscall interposition can be used to avoid this specific problem, but process-wide denial of service from within a compromised lwC cannot be fully avoided (see section 3). [Lit+16]

## 2.7 API Summary

<code>lwcCreate</code>	
<code>lwcClose</code>	
<code>lwcGetLwc</code>	
<code>lwcSwitch</code>	
<code>lwcOverlay</code>	
<code>lwcRestrict</code>	
<code>lwcSyscall</code>	

Figure 6: lwc API summary table

## 3 Security Analysis

In this section, we define a threat model for lwCs and assess how the design meets the canonical information security properties of *confidentiality*, *integrity* and *availability*[ST04].

**Threat Model** The run-time trusted computing base of a process that uses light-weight contexts is the hardware, firmware, monolithic OS kernel, any user space processes able to influence the execution of the lwC-process, and any user space code that

runs before `main` starts executing. Once `main` starts executing, we assume an attacker who is able to hijack control flow and execute arbitrary code in user-mode in the currently established execution context, i.e., the current lwC. Specifically, an attacker may access any mapped memory through unprivileged instructions and invoke any system call, including lwC management calls, and attempt privilege escalation directly or indirectly by invoking said system calls. [Lit+16]

**Resource Mappings** An lwC consists of resource mappings for the three resource kinds *address space*, *file descriptor table* and *system credential*. The kernel implementation of `lwcSwitch` must guarantee that all kernel subsystems as well as the memory management unit will always perform their respective access permission checks against the resource mappings of the lwC that was last switched to, i.e., the current thread's current lwC. Under that assumption, the lwC security guarantees rely solely on the soundness of the rules by which resource mappings can be manipulated from user space through `lwcCreate`, `lwcRestrict` and `lwcOverlay`. With regards to static resource sharing, the rules are sound if access to a parent lwCs can only be reduced along an lwC creation chain. For dynamic resource sharing, the semantics defining soundness depend on the use case and thus cannot be proven generically. Neither the original paper's authors nor us provide a proof of soundness of the resource mapping manipulation.

The `lwcSyscall` API must be considered separately: its `mask` argument allows the trap-handling lwC to specify a *subset* of resource mappings to be established for the duration of the syscall. This differs from `lwcSwitch`, which only allows switching *all three resource mappings* atomically before executing code in the context of the target lwC. Therefore, it must not be possible to extend the set of *passing* permission checks in kernel or memory management unit by invoking `lwcSyscall` with a `mask` that combines resource mappings from trapping and trap-handling lwC, compared to invoking the syscall directly from either the trapping or the trap-handling lwC.

**Achieving Protection Goals** Applications must create lwCs with correct resource mappings and access capabilities in order to achieve their desired protection goals. For example, full confidentiality and integrity of a region of memory in the root lwC can be achieved by asserting that all child lwCs are created with a resource specifier that marks that



respective memory region as `UNMAPed`. "Loop holes" such as attaching a debugger to the parent lwC from the child lwC must also be addressed, e.g., through system call interposition or by dropping privileges in the child lwC.

As mentioned in the previous subsection, purely static resource sharing guarantees monotonically decreasing access rights along the lwC creation chain — application developers must only verify that no unintended implicit sharing occurs. Dynamic resource sharing should be used carefully because the dynamic flow of access rights can only and must be verified by the application developer. [Lit+16]

Availability cannot be guaranteed by lwCs: an attacker in control of an lwC can invoke the `exit` system call to terminate the process unless syscall interposition is used to prevent that syscall. An attacker may also modify the code in a compromised lwC to block indefinitely or never invoke `lwcSwitch`, effectively trapping all threads that switch to the compromised lwC. The authors dismiss both problems by describing denial-of-service (DoS) within a process as "self-defeating". We remark that the original paper does not address per-lwC resource usage (limits), which could serve as another DoS vector. [Lit+16]

### 3.1 Example Use Case: Application Compartmentalization

Application developers can leverage lwCs to achieve application protection goals. We restrict ourselves to an example use case for lwCs of a fictional application called *cnvrt*. We are going to use lwCs for *compartmentalization* to increase confidentiality of user data and the server's TLS private key. *cnvrt* is a network service which

1. starts a TLS server which accepts connections from arbitrary clients on the Internet,
2. reads a JPEG images over established connection,
3. uses an image processing library to convert the image into the client-requested output format and
4. sends a response with the converted image back to the client.

We consider an attacker who exploits a vulnerability in the image processing library's image codec parser to achieve remote code execution: in an un-compartmentalized design, a successful attacker can read the entire process's memory to extract the private key of the TLS key pair or access other users' images from

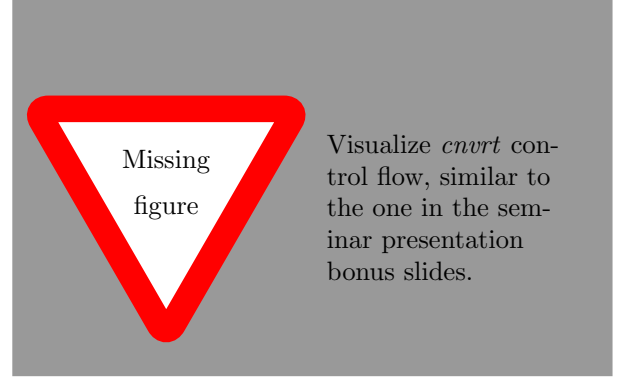


Figure 7: The compartmentalized *cnvrt* application uses a separate lwC for the initial TLS handshake to protect the private key and per-request lwCs to isolate concurrent requests from each other.

other requests. Both *private key confidentiality* as well as *client session isolation* are compromised.

In contrast, an lwC-enabled design loads the TLS private key into the root lwC and waits for connections. For each new connection, it creates a worker lwC with private copy of the root lwC's address space and file descriptor table and switches into it. After performing the TLS hand-shake, it erases the private key from the worker address space and, only then, starts executing the riskier image processing code. A successful attacker has no longer access to the private key and is limited to the socket representing their own connection, as well as their own request data. A visualization of the compartmentalized design is given in figure 7.

Note that we have constructed a simplified example where requests can be handled in full isolation. In practice, some sharing will be required, e.g., for global statistics, a database connection pool, logging services, etc. In an lwC-compartmentalized design, these components would execute in separate lwCs and be made accessible to the worker lwCs via shared memory, `lwcSwitch`, `lwcOverlay` or traditional IPC mechanisms.

Note further that we have not discussed the concurrency model of *cnvrt*: both threaded and event-driven models are possible, as will be explained briefly in section 5.3.

## 4 Implementation

The authors implement light-weight contexts in the FreeBSD 11.0 operating system and make the source code publicly available. [LWCK16; LWCu16] The

major changes to the kernel consist of:

- An implementation of syscall handlers and resource specifier logic (+2388 lines in `sys/sys/kern_snapp.c`).
- Support for resource specifiers and overlays in file descriptor management (`struct filedesc`, +251 lines in `sys/kern/kern_descrip.c`) and memory management subsystem (+693 lines in `sys/vm/vm_map.c`, +178 lines in `sys/amd64/amd64/pmap.c`).
- Platform-dependent code for `lwcSwitch` (+250 lines in `sys/amd64/amd64/cpu_switch.S`, +171 lines in `sys/amd64/amd64/vm_machdep.c`).

User-space support for lwCs is provided by a C library containing syscall wrappers and a synchronized hash-table that enables key-value sharing across lwCs. The authors also provide a PHP extension that makes the lwC API accessible from scripts, which is required for the evaluation.

## 5 Evaluation

The authors evaluate their implementation using micro-benchmarks to determine the primitive operations' latency and integrate light-weight contexts into production applications, demonstrating its applicability and real-world performance impact. The benchmarking setup consists of two machines with dual Intel Xeon X5650 2.66GHz six-core CPUs with disabled hyperthreading and dynamic frequency scaling. The machines are connected by a 1Gbit Ethernet switch. [Lit+16]

### 5.1 Micro-Benchmarks

The micro-benchmarks show a 2x speedup for `lwcSwitch` latency compared to regular context switching induced by a kernel semaphore. Regular context switching is a valid micro-benchmark baseline because it would be necessary for process-based privilege separation. [Lit+16]

The latency of lwC creation and destruction is only explored cumulatively: creating and immediately destroying a single lwC takes  $87.7\mu s \approx 233282$  cycles on the evaluation machine. However, the authors do not provide a meaningful baseline, e.g., the latency of a forking and immediately exiting in the child. An extended version of this micro-benchmark is also used to argue that the direct run-time overhead *within* an lwC is limited to CoW faults. However, the indirect cost of multiple address spaces (e.g. increased TLB pressure) is neither considered nor

evaluated. [Lit+16]

The syscall interposition feature is evaluated by example: the authors implement a reference monitor that intercepts the `open`, `read` and `write` system calls, execute a dummy application that performs a fixed number of those system calls, and measure total execution time. For comparison, they also measure a variant with inlined policy checks, as well as a variant that uses FreeBSD's Capsicum with multiple processes. The result is the *total execution time* per variant and system call, which is only useful to compare *throughput*, *not latency*. Expectably, inlined policy checks exhibit the lowest overhead in all cases. Capsicum and syscall interposition are penalized by the required context in the case of short syscalls like `open`, small `read` and `write`. For longer syscalls (large `read`, `write`), syscall interposition benefits from the `mask` argument because — unlike Capsicum — copying the buffer to the reference monitor is not necessary. [Lit+16]

language

figure helpful?

### 5.2 PHP-FPM

The first application benchmark uses light-weight contexts within the PHP-FPM FastCGI server to reduce interpreter and application initialization time, using a technique called *snapshot-and-rollback*: on first startup, before handling any request-specific data, the PHP script creates a snapshot of its initialized state using `lwcCreate`. This snapshot is then used as the basis for handling subsequent requests in a separate isolated lwCs which start execution at the pre-initialized state of the original snapshot. Pseudo code for this operation is provided in figure 8. [Lit+16]

The authors adapt a Zend-Framework web application template to use snapshot-and-rollback and compare the **throughput** of against upstream PHP-FPM with the unmodified app, both with and without PHP opcode cache: snapshot-and-reload achieves **2.7x (1.3x) throughput** with disabled (enabled) opcode cache. Notably, this performance gain also comes with the security benefit of handling each request in a separate address space. [Inc19; Lit+16]

### 5.3 Session Isolation in Web Servers

The authors also integrate lwCs into the popular **Apache** and **nginx** web servers to protect TLS private keys and per-session data from attackers (*session isolation*). The Apache variant extends the Apache pre-fork concurrency mode, which uses ded-



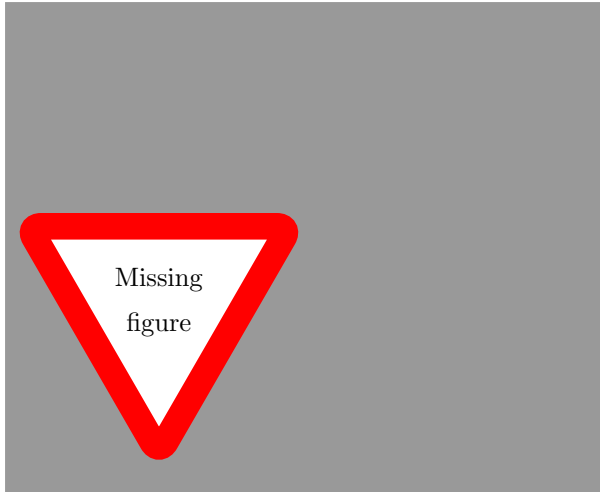


Figure 8: Pseudo-Code for Snapshot& Rollback as used for the PHP-FPM benchmark.

icated threads per active client connection: before a thread starts handling a new connection, it creates an lwC with private address space and file descriptor table and switches into it, thereby isolating potential attackers to a single thread in a dedicated lwC. Nginx implements an event-loop using non-blocking I/O with a single worker thread per core: an lwC is created per connection and the descriptor is tracked in a hash map index by the socket file descriptor. When the kernel notifies the worker thread about pending I/O on a socket, the worker looks up the corresponding lwC and switches to it before resuming regular nginx request handling. When the kernel reports that socket I/O would block, the worker switches back to the main event loop to handle other pending I/O. Pseudo code for both Apache and nginx modifications is provided in figure 9. [Lit+16]

The first set of benchmarks measures *throughput* of GET requests for a *single 45 byte document* at a constant number of concurrent clients. The authors perform the experiment for different *session lengths*, i.e., the number of requests sent over a single connection using HTTP keep-alive. For Apache, the lwC modifications exhibit significantly worse throughput for short sessions than stock pre-fork mode ( $\geq 80\%$  lower throughput). 16-request sessions, which we consider generous for highly interactive web applications, still exhibit  $\sim 16\%$  lower throughput. For nginx, performance implications are much less severe, with at most 22% lower throughput for four-request sessions and  $\sim 6\%$  lower throughput at 16-request sessions. Our interpretation of the throughput improvements for Apache with growing session

length is that the one-time additional latency for lwC creation and destruction is amortized for longer sessions, as can be observed at 256 reqs/session. However, this theory does not explain why nginx throughput is less affected, because nginx also creates an lwC per connection. [Lit+16]

The authors also conduct a *scalability* experiment for nginx: at a fixed session length of 256 requests, for 45 byte and 900 byte documents, the total throughput is measured for different counts of concurrent clients. For up to 6500 concurrent clients, there is no significant difference between upstream nginx and lwC nginx. For the 45 byte experiment, higher concurrency correlates with higher standard deviation and at most 19% lower mean throughput. 900 byte requests degrade more gracefully, with up to 10% lower mean throughput at 19500 concurrent clients. The authors explain the sudden drop in performance at 6500 clients with CPU-bound behavior of an interrupt handler thread, but do not provide details on the network hardware or driver which likely caused this problem. It is conceivable that the additional CPU time required by lwCSwitch causes the steeper drop in throughput for lwC-nginx. [Lit+16]

Apart from session isolation, the authors also modify the OpenSSL library to isolate the TLS private key used for HTTPS in a dedicated lwC. This refactoring analogous to our example in section 3.1 is possible because the private key is only required for TLS session establishment to negotiate the symmetric session key. The authors claim that the OpenSSL modifications were made such that they are **not visible to the application code** consuming the library.<sup>2</sup> The evaluation in nginx shows **only 0.6% lower throughput** for 10000 TLS handshakes with 24 concurrent clients compared to upstream OpenSSL. [Lit+16]

## 6 Related Work

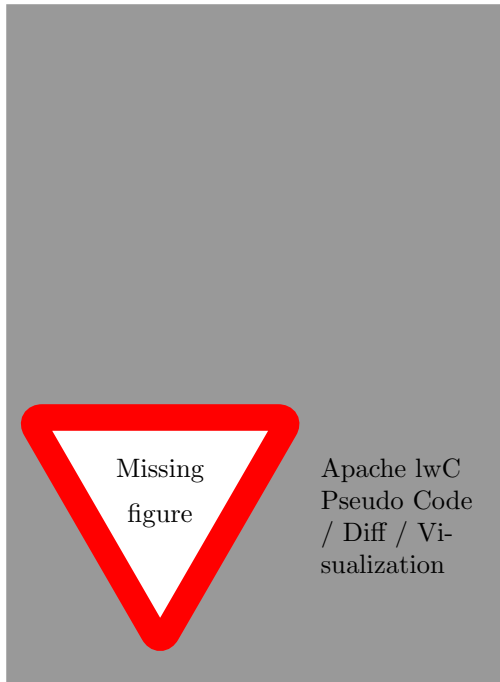
The evaluation demonstrates two use-cases for lightweight contexts: *application compartmentalization* and process *snapshot-and-rollback*. For the sake of brevity, we are going to limit ourselves to an investigation of related work in the former category.

Compartmentalization describes the decomposition of an application into isolated compartments that can provide independent security guarantees, as demonstrated in section 3.1 and the web server benchmarks (section 5.3). The guiding principle is

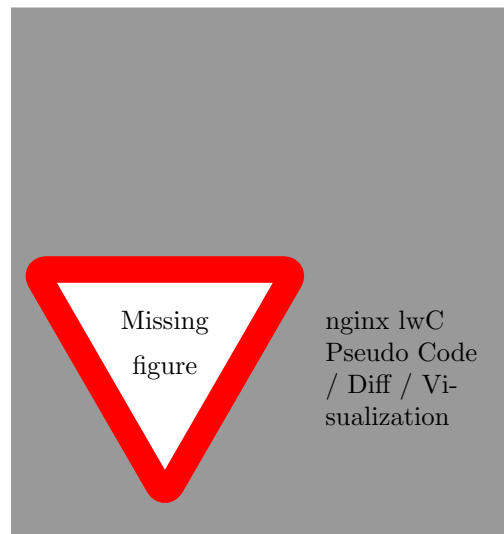
<sup>2</sup>The OpenSSL modifications were made in an intermediate build stage of the FreeBSD port system.

need explain?

pending I/O comprehensible?



(a) **Apache** uses thread-based concurrency: the handler thread switches to a new lwC once and performs blocking I/O within the lwC. lwc-Switch is only invoked at the start and end of the connection.



(b) **Nginx** uses an event loop. The lwC descriptor for each active connection is stored in a hash map. Before resuming work on a connection, the event loop worker looks up the corresponding lwC in the hash map and switches to it.

Figure 9: Session-Isolation in the Apache and nginx web servers.

that of *least privilege*, as formulated in during the development of Multics in 1974:

”Every program and every privileged user of the system should operate using the least amount of privilege necessary to complete the job.” [Sal74]

This principle also applies within an application: it should be possible that different code modules and subsystems operate with the lowest system privilege possible and only have access to memory and global system state that is necessary to provide their functionality.

**Programming language technology** provides the most constructive but least flexible approach to application compartmentalization. Memory safe languages systematically eliminate the need for memory isolation and unintended remote code execution. Formal analysis can be used to prove information flow rules between different components of an application. Runtime systems with sufficiently comprehensive access control systems eliminate the need for syscall filtering, private file descriptors, etc. Java is a good example for a memory safe language with a runtime mandatory access control mechanism (Java `SecurityManager`). The downside of language-specific solutions is the implicit requirement to use that specific programming language and hence inapplicability to existing projects in other languages. Further, the trusted computing base (TCB) then encompasses the entire language runtime implementation which, in the case of Java, has been shown to have countless vulnerabilities and sandbox escapes over its lifetime. Light-weight contexts are language- and runtime-system-independent and have a significantly smaller TCB. [Java19; BD18]

**Byte-code virtual machines** are often used to execute untrusted code in a sandboxed environment, which is established and enforced either through one-time verification and JIT compilation or per-instruction interpretation with security checks. Web Assembly or Linux kernel eBPF implement variants of this concept. Light-weight contexts allow native code execution without interpreter or verifier overhead. [Haa+17; Fle17].

**Operating system** designers have long pursued privilege separation and compartmentalization models: even very basic established techniques like virtual memory, multi user systems, file system permissions and `chroot` enable effective process-based privilege separation, e.g., in the OpenSSH server

and the Chromium web browser. Whereas these approaches creatively re-combine existing kernel subsystems to achieve isolated address spaces or different privileges, light-weight contexts make the protection domain a first-class kernel abstraction. [PFH03; Bar+08]

Microkernels such as Fiasco-OC and seL4 have pursued the more holistic concept of **object capability systems**: capabilities are unforgeable tokens that represent the right to access (invoke) a resource (system call). Starting from a root process with capabilities to all system resources, new processes must be constructed with an explicit list of access capabilities, encompassing address space, file system access rights, system call gates. Like lwCs, the level of (static) privilege only decreases along the capability creation chain, but we believe that the requirement to explicitly list all requirements of a new process facilitates audits of the system architecture, compared to the default CoW sharing between lwCs parent and child. [EH13]

**FreeBSD Capsicum** is a technology described as ”practical capabilities for UNIX”: after entering *capability mode*, a process is limited to file-descriptor-relative system calls (`openat` instead of `open`), and inter-process communication through already open sockets and IPC handles. Access to any kind of global namespace (file system, inet- and unix sockets, ...) is prohibited, which gives file descriptors the role of capabilities. Compartmentalization is achieved by decomposing an application into multiple processes in capability mode and connecting them through pipes. This technique enables private address spaces, file descriptor tables and system credential per logical application component and limits communication to well-defined entry-points via IPC. Arbitrary policy checks on system calls require proxying the call through a separate processes, which introduces latency and CPU overhead avoided by lwC system call interposition. However, Capsicum retains some advantages over lwCs, such as the use of established IPC mechanisms and compatible libraries for communication between components, as opposed to `lwcSwitch` semantics. [Wat+10]

The **Wedge** system implements Linux kernel support for creating *stthreads*, which behave similarly to lwCs with regards to static resource sharing. Wedge avoids the forking semantics of `lwcCreate`, following a ”default-deny model” where all resources to be shared with a child must be explicitly listed. Notably, memory is not a resource in the lwC sense, but handled through *tags*: all memory allocations are as-

sociated with a tag per compartment. Permissions to access data on the compartment-private heap is thus also expressed through tags being passed between *sthreads*. Wedge requires modification of all memory allocations in a program if sources are available or the use of linker-based compatibility modes for binary-only libraries. Light-weight contexts are less invasive with regards to program modification and do not treat memory differently from other resources. [Bit+08]

preposition

The **Shreds** system combines a variant of Control-Flow Integrity and ARM Memory Domains to implement private memory and hijacking protection for small code blocks. Developers mark such code blocks with runtime calls and use Shreds memory allocation functions within it. A customized compiler then generates control-flow integrity checks for the marked code blocks. A kernel module then verifies the integrity of a shred on entry and maps the shred-private memory pool for the duration of execution within the shred. ARM Memory Domains are the vehicle to implement Shreds on ARM with acceptable performance costs, but make the solution less portable than lwCs, which only require standard memory virtualization (see our critique in section 7). The mandatory use of a custom compiler and mandatory availability of source code for all Shreds is also more inhibiting to adoption than lwCs. [Che+16; Aba+09; ARMDom]

## 7 Critique

The light-weight context design is appealing on first glance due to

- its direct applicability within an existing production-grade OS on amd64 hardware,
- its relatively small implementation footprint in terms of code size and locality as well as
- the original paper's evaluation results (section 5), demonstrating applicability in real-world applications, promising either performance gains or moderate losses in exchange for convincing security benefits.

However, we believe that the evaluation does not provide a substantiated argument for the practicality of light-weight contexts.

First, the **lwcSwitch** micro-benchmark does not account for the indirect costs of address-space switching, specifically, the additional TLB pressure. These effects have been well-known for decades, e.g., in micro-kernel research and implementation. [Lie+97] An evaluation should compare the effects of this indi-

rect overhead against other sandboxing and privilege separation approaches.

Second, we are dissatisfied with the chosen web-server benchmark metrics: the authors only measure throughput and not latency, which is of equal or even higher importance for web applications. In particular, the scalability experiment results are almost worthless for web-app use cases without a comparison of latency distributions.

Third, the web-server benchmarks are not representative because they actually contain very few lwC and address space switches: the clients only ever request a single 45B or 900B document from a RAM disk, which is to high certainty available in the buffer cache. Thus, any file system I/O performed by Apache or nginx is not going to block. Further, both the HTTP requests and responses for those documents fit within a single TCP frame transferred over a low-latency dedicated ethernet link. Therefore, socket I/O is also not likely to block if the evaluation setup is not network-bandwidth bound (see below). Given the low probability of blocking, we suspect that *both* the nginx and Apache benchmarks only perform the following lwC and scheduling-related operations:

consistent hyphens

1. For a new connection, create the lwC once and switch to it,
2. handle requests on that connection until the TCP tx buffer is full (no context switching required)
3. *if no more requests*: goto 6,
4. *if nginx* (non-blocking I/O): switch out of lwC, when returning, goto 2,
5. *if Apache* (blocking I/O): block, OS context-switches to another thread in other lwC, original thread is woken up eventually, goto 2,
6. leave lwC and destroy it.

Our own experiments<sup>9.1</sup> show 1146B (284B) TCP response payloads for 900B (40B) requests. Thus, 28 (115) responses fit into the standard initial 32KiB tx buffer of a default FreeBSD TCP socket [FB19] Assuming non-blocking reads for all request payloads<sup>3</sup>, this implies that that session lengths of up to 28 (115) requests per session can be handled without any intermediate context switching. In reality though, both file and socket I/O are going to block, dramatically increasing the number of lwC switches compared to the presented application benchmarks.

Fourth, we believe that the nginx scalability benchmark was operating close to the maximum network

<sup>3</sup>Requests are very short and likely pipelined.

bandwidth of 1Gbit/s, which might have needlessly disrupted or distorted the measurements: With a peak throughput of  $\sim 95k$  req/sec for 900B and 180k req/sec for 40B, our experiments show 0.92 (0.5) Gbit/s response Ethernet traffic (see 9.1). To exclude issues with driver performance, flow control or congestion control, we recommend repeating the experiment with a 10 Gbit/s link.

Fifth, the evaluation is performed on a machine with a tagged TLB — untagged TLB systems are not measured. Given the envisioned use-case for lwCs is in server applications running in data centers, and the dominance of Intel in the server market, and the availability of tagged TLBs on Intel CPUs since 2010, requiring tagged TLBs for performance might be acceptable. However, such a requirement should be made explicit and justified through appropriate benchmarks.[Mon19]

Sixth, the evaluation does not address the issue of TLB-tag exhaustion: since lwCs encourage the creation of many address spaces and rapid switching between them (see section 9), a system might have fewer TLB tags available than active lwCs. For example, the Intel system used for the evaluation uses 12-bit wide TLB tags. This allows for at most  $2^{12} = 4096$  concurrent lwCs per CPU core before TLB tags need to be time-multiplexed on a single core or space-multiplexed between different cores. Time-multiplexing necessarily implies TLB shoot-downs and thus indirect performance overheads. Space-multiplexing requires tighter integration into the scheduler and necessarily comes with trade-offs in work-conservation, fairness, etc. The *mechanism* for TLB tag exhaustion must be addressed by the virtual memory (VM) system. But the lwC design likely breaks implicit assumptions of the VM system designers with regards to frequency of AS creation and switching. Thus, the authors should provide an appropriate multiplexing and scheduling *policy* and measure its effects in application-level benchmarks, also addressing fairness in multi-tenant systems (multi-user, jails, containers). Note that the benchmarking setup might not have been affected by TLB tag exhaustion because, if our aforementioned theory about never-blocking file and socket I/O holds, each lwC only lived for a very short time.

## 8 Conclusion

We have presented the design of light-weight contexts and their evaluation in FreeBSD 11.0. From an OS architecture perspective, the fundamental change introduced with lwCs is a decoupling of exe-

cution environment & control flow from processes & threads: threads now have a logical control flow *per lwC*, each of which can execute in a separate address space, with separate file descriptor table and system credential. Argument passing and dynamic resource sharing then allow for RPC-like communication between lwCs, enabling application compartmentalization.

It is debatable whether the fork-like lwC creation is a favorable programming paradigm or just a convenient way to implement isolation through copy-on-write in unsafe legacy languages. Either way, we would like to point the reader to the well-founded arguments against any further proliferation of fork semantics due to its composability problems, thread-unsafety and security implications (implicit instead of explicit sharing).[Bau+19]

The authors' evaluation has shown that, with regards to throughput, light-weight context indeed have lower overhead than process-based privilege separation. Further, the application-level benchmarks demonstrate that non-invasive usage of lwCs in web application stacks is also possible: each client request can be handled in an isolated address space, protecting client sessions from each other as well as the server's TLS private key.

However, we have found the the benchmarks for the modified applications to be incomplete and unrepresentative of real-world work loads. In particular, the availability of a tagged TLB seems critical for performance, but neither the design nor the evaluation address the issue of TLB tag exhaustion.

We are not convinced that the presented design of light-weight contexts is an appropriate general-purpose OS abstraction. Practical usefulness in the domain of web application hardening depends on the unreported latency implications of lwCs as well as real-world TLB miss behavior. We would like to see a revision of the evaluation that addresses the listed flaws, providing a more complete picture of real-world lwC performance.

ausschliessen

cite  
wikipedia  
link

should give  
detailed  
hardware  
info in eval  
section

## 9 Appendix

### 9.1 HTTP/TCP 40B/900B Experiment

No.	Time	Source	Destination	Protocol	Length	Info
4	0.014841	XXX.XXX.XXX.XX	YYY.Y.YY.YYY	HTTP	157	GET /40B HTTP/1.1

Frame 4: 157 bytes on wire (1256 bits), 157 bytes captured (1256 bits)

Transmission Control Protocol, Src Port: 44084, Dst Port: 80, Seq: 1, Ack: 1, Len: 91

Hypertext Transfer Protocol

No.	Time	Source	Destination	Protocol	Length	Info
5	0.029361	YYY.Y.YY.YYY	XXX.XXX.XXX.XX	HTTP	350	HTTP/1.1 200 OK

Frame 5: 350 bytes on wire (2800 bits), 350 bytes captured (2800 bits)

Transmission Control Protocol, Src Port: 80, Dst Port: 44084, Seq: 1, Ack: 92, Len: 284

Hypertext Transfer Protocol

Data (40 bytes)

No.	Time	Source	Destination	Protocol	Length	Info
14	4.606617	XXX.XXX.XXX.XX	YYY.Y.YY.YYY	HTTP	158	GET /900B HTTP/1.1

Frame 14: 158 bytes on wire (1264 bits), 158 bytes captured (1264 bits)

Transmission Control Protocol, Src Port: 44086, Dst Port: 80, Seq: 1, Ack: 1, Len: 92

Hypertext Transfer Protocol

No.	Time	Source	Destination	Protocol	Length	Info
15	4.621203	YYY.Y.YY.YYY	XXX.XXX.XXX.XX	HTTP	1212	HTTP/1.1 200 OK

Frame 15: 1212 bytes on wire (9696 bits), 1212 bytes captured (9696 bits)

Transmission Control Protocol, Src Port: 80, Dst Port: 44086, Seq: 1, Ack: 93, Len: 1146

Hypertext Transfer Protocol

Data (900 bytes)



## References

- [Aba+09] M. Abadi et al. “Control-flow integrity principles, implementations, and applications”. In: *ACM Transactions on Information and System Security (TISSEC)* 13.1 (2009), p. 4.
- [ARMDom] *ARM11 Processor Manual: Memory Access Control: Domains*. URL: <http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0211k/Babjdffh.html> (visited on 07/22/2019).
- [Bar+08] Adam Barth et al. “The security architecture of the chromium browser”. In: *Technical report*. Stanford University, 2008.
- [Bau+19] Andrew Baumann et al. “A fork() in the road”. In: *17th Workshop on Hot Topics in Operating Systems*. ACM, May 2019. URL: <https://www.microsoft.com/en-us/research/publication/a-fork-in-the-road/>.
- [BD18] Alexandre Bartel and John Doe. “Twenty years of Escaping the Java Sandbox”. In: *Phrack* (2018).
- [Bit+08] Andrea Bittau et al. “Wedge: Splitting applications into reduced-privilege compartments”. In: *USENIX Association*. 2008.
- [Che+16] Yaohui Chen et al. “Shreds: Fine-grained execution units with private memory”. In: *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2016, pp. 56–71.
- [EH13] Kevin Elphinstone and Gernot Heiser. “From L3 to seL4 what have we learnt in 20 years of L4 microkernels?”. In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM. 2013, pp. 133–150.
- [FB19] The FreeBSD Project. *FreeBSD 11.0 src - tcp\_output.c*. URL: [https://svnweb.freebsd.org/base/release/11.0.0/sys/netinet/tcp%5C\\_output.c?revision=306280&view=markup#l1110](https://svnweb.freebsd.org/base/release/11.0.0/sys/netinet/tcp%5C_output.c?revision=306280&view=markup#l1110) (visited on 07/21/2019).
- [Fle17] Matt Fleming. *A thorough introduction to eBPF*. Dec. 2, 2017. URL: <https://lwn.net/Articles/740157> (visited on 07/21/2019).
- [Haa+17] Andreas Haas et al. “Bringing the web up to speed with WebAssembly”. In: *ACM SIGPLAN Notices*. Vol. 52. 6. ACM. 2017, pp. 185–200.
- [Inc19] Perforce Software Inc. *Zend the PHP Company*. 2019. URL: <https://www.zend.com> (visited on 07/25/2019).
- [Java19] Oracle and/or its affiliates. *Java Security Overview*. 2019. URL: <https://docs.oracle.com/javase/8/docs/technotes/guides/security/overview/jsoverview.html> (visited on 07/21/2019).
- [Lie+97] Jochen Liedtke et al. “Achieved IPC performance (still the foundation for extensibility)”. In: *Proceedings. The Sixth Workshop on Hot Topics in Operating Systems (Cat. No. 97TB100133)*. IEEE. 1997, pp. 28–31.
- [Lit+16] James Litton et al. “Light-Weight Contexts: An {OS} Abstraction for Safety and Performance”. In: *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*. 2016, pp. 49–64.
- [LWCh18] James Litton et al. *Light-Weight Contexts Homepage*. 2018. URL: <http://www.cs.umd.edu/projects/lwc> (visited on 07/26/2019).
- [LWck16] James Litton et al. *lwC kernel Git Repository - git diff 816162fb8..origin/OSDI16*. 2016. URL: [git://jameslitton.net/lwckernel.git](https://github.com/jameslitton/net/lwckernel.git) (visited on 05/12/2019).
- [LWCu16] James Litton et al. *lwC user-space libraries*. 2016. URL: [git://jameslitton.net/lwclibs.git](https://github.com/jameslitton/net/lwclibs.git) (visited on 05/12/2019).
- [Mon19] Joseph Tsai Monica Chen. *AMD to challenge Intel server processor market dominance (90% Intel market share)*. Mar. 28, 2019. URL: <https://www.digitimes.com/news/a20190328PD200.html> (visited on 07/25/2019).
- [OG04] The Open Group. *The Open Group Base Specifications Issue 6 IEEE Std 1003.1. fork - create a new process*. 2004. URL: <http://pubs.opengroup.org/onlinepubs/000095399/functions/fork.html> (visited on 07/19/2019).

- [PFH03] Niels Provos, Markus Friedl, and Peter Honeyman. “Preventing Privilege Escalation.” In: *USENIX Security Symposium*. 2003.
- [Sal74] Jerome H. Saltzer. “Protection and the Control of Information Sharing in Multics”. In: *Commun. ACM* 17.7 (July 1974), pp. 388–402. ISSN: 0001-0782. DOI: 10.1145/361011.361067. URL: <http://doi.acm.org/10.1145/361011.361067>.
- [ST04] Alex Summers and Chris Tickner. *The CIA principle*. 2004. URL: <https://www.doc.ic.ac.uk/~ajs300/security/CIA.htm> (visited on 07/01/2019).
- [Wat+10] Robert NM Watson et al. “Capsicum: Practical Capabilities for UNIX.” In: *USENIX Security Symposium*. Vol. 46. 2010, p. 2.