Light-weight Contexts: An OS Abstraction for Safety and Performance

James Litton[1,2], Anjo Vahldiek-Oberwagner[2], Eslam Elnikety[2], Deepak Garg[2], Bobby Bhattacharjee[1], and Peter Druschel[2]

[1]University of Maryland, College Park
[2]Max Planck Institute for Software Systems (MPI-SWS), Saarland Informatics Campus

Abstract

We introduce a new OS abstraction—light-weight contexts (*lwC*s)—that provides independent units of protection, privilege, and execution state within a process. A process may include several *lwC*s, each with possibly different views of memory, file descriptors, and access capabilities. *lwC*s can be used to efficiently implement roll-back (process can return to a prior recorded state), isolated address spaces (*lwC*s within the process may have different views of memory, e.g., isolating sensitive data from network-facing components or isolating different user sessions), and privilege separation (in-process reference monitors can arbitrate and control access).

*lwC*s can be implemented efficiently: the overhead of a *lwC* is proportional to the amount of memory exclusive to the *lwC*; switching *lwC*s is quicker than switching kernel threads within the same process. We describe the *lwC* abstraction and API, and an implementation of *lwC*s within the FreeBSD 11.0 kernel. Finally, we present an evaluation of common usage patterns, including fast roll-back, session isolation, sensitive data isolation, and in-process reference monitoring, using Apache, nginx, PHP, and OpenSSL.

1 Introduction

Processes abstract the unit of isolation, privilege, and execution state in general-purpose operating systems. Computations that require memory isolation, privilege separation, or continuations at the OS level must be run in separate processes[1]. Unfortunately, switching and communicating between processes incurs the cost of invoking the kernel scheduler, resource accounting, context-switching, and IPC. The actual hardware-imposed cost of isolation and privilege separation, however, is much smaller: if the TLB is tagged with an address space identifier, then switching context requires as little as a system call and loading a CPU register.

Just as threads separate the unit of execution from a process, we assert that there is benefit to decoupling memory isolation, execution state, and privilege separation from processes. We show that it is possible to isolate memory and privileges, and maintain multiple execution states *within* a process with low overhead, thus streamlining common computation patterns and enabling more efficient and safe code.

We introduce a new first-class OS abstraction: the *light-weight context* (*lwC*). A process may contain multiple *lwC*s, each with their own virtual memory mappings, file descriptor bindings, and credentials. Optionally and selectively, *lwC*s may share virtual memory regions, file descriptors and credentials.

*lwC*s are not schedulable entities: they are completely orthogonal to threads that may execute within a process. Thus, a thread may start in *lwC a*, and then invoke a system call to *switch* to *lwC b*. Such a switch atomically changes the VM mappings, file table entries, permissions, instruction and stack pointers of the thread. Indeed multiple threads may execute simultaneously within the same *lwC*. *lwC*s maintain per-thread state to ensure a thread that enters a *lwC* resumes at the point where it was created or last switched out of the *lwC*.

*lwC*s enable a range of new in-process capabilities, including fast roll-back, protection rings (by credential restriction), session isolation, and protected compartments (using VM and resource mappings). These can be used to implement efficient in-process reference monitors to check security invariants, to isolate components of an app that deal with encryption keys or other private information, or to efficiently roll back the process state.

We have implemented *lwC*s within the FreeBSD 11.0 kernel. The prototype shows that it is possible to implement *lwC*s in a production OS efficiently. Our experience with implementing and retrofitting large applications such as Apache and nginx with *lwC*s has taught us that it is possible to introduce many new capabilities, such as rollback and secure data compartments, to existing production code with minimal overhead. This paper's contributions are:

- We introduce *lwC*s, a first-class OS abstraction that extends the POSIX API, and present common coding patterns demonstrating its different uses.

- We describe an implementation of *lwC*s within FreeBSD, and show how *lwC*s can be used to implement efficient session isolation in production web servers, both process-oriented (Apache, via roll-back) and event-driven (nginx, via memory isolation). We show how efficient snapshotting can provide session isolation while

[1]Language runtimes can provide these properties at the expense of additional overhead, language dependence, and an increased trusted computing base.

improving performance on web-based applications using a PHP-based MVC application on nginx. We show how cryptographic libraries such as OpenSSL can efficiently create isolated data compartments *within a process* to render sensitive data (such as private keys) immune to external attacks (e.g., buffer overruns a la Heartbleed [7]). Finally, we show how *lwC*s can efficiently implement in-process reference monitors, again for industrial-scale servers such as Apache and nginx, that can introspect on system calls and memory.

• We evaluate *lwC*s using a range of micro-benchmarks and application scenarios. Our results show that existing methods for session isolation are often slower than *lwC*s. Other common uses such as *lwC*-supported sensitive data compartments and reference monitoring have little to negligible overhead on production servers. Finally, we show that using the *lwC* snapshot capability to quickly launch an initialized PHP runtime can improve the performance of a production server.

The rest of this paper is organized as follows: we discuss related work in Section 2 and describe the *lwC* abstraction, API, and design in Section 3. We present common *lwC* coding patterns in Section 4. We describe our FreeBSD implementation of *lwC*s in Section 5, and present an experimental evaluation in Section 6. We conclude in Section 7.

2 Related work

Wedge [5] provides privilege separation and isolation among *sthreads*, which otherwise share an address space. Sthreads are implemented using Linux processes. *lwC*s are orthogonal to threads and therefore avoid the cost of scheduling when switching contexts. Moreover, *lwC*s can snapshot and resume an execution in any state, while a sthread can only revert to its initial state. Wedge provides a software analysis tool that helps refactor existing applications into multiple isolated compartments. *lwC*s could benefit from a such a tool as well.

Shreds [9] builds on architectural support for *memory domains* in ARM CPUs, a compiler toolchain, and kernel support to provide isolated compartments of code and data within a process. Like *lwC*s, shreds provide isolated contexts within a process. *lwC*s, however, are fully independent of threads, require no compiler support, and rely on page-based hardware protection only. *lwC*s also provide protection rings and snapshots, which shreds do not.

In SpaceJMP [12], address spaces are first-class objects separate from processes. While both systems can switch address spaces within a process, SpaceJMP and *lwC*s provide different abstractions, capabilities, and are motivated by entirely different applications. SpaceJMP supports applications that wish to use memory larger than the available virtual address bits allow, wish to maintain pointer-based data structures beyond process lifetime, and share large memory objects among processes. A SpaceJMP context switch is not associated with a mandatory control transfer and, therefore, SpaceJMP does not support applications that require isolation or privilege separation within a process. *lwC*s, on the other hand, provide in-process isolated contexts, privilege separation, and snapshots.

Dune [4] provides a kernel module and API that export the Intel VT-x architectural virtualization support safely to Linux processes. Privilege separation, reference monitors, and isolated compartments can be implemented within a process using Dune. *lwC*s instead provide a unified abstraction and API for these capabilities, and their implementation does not rely on virtualization hardware, the use of which could interfere with execution on a virtualized platform. While the *lwC* implementation incurs a higher cost for system call redirection, it avoids Dune's overhead on TLB misses and kernel calls.

In Trellis [20], code annotations, a compiler, run time, and OS kernel module provide privilege separation within an application. The kernel and runtime ensure that functions can be called and data accessed only by code with the same or higher privilege level. *lwC*s provide privilege separation without language/compiler support, and can switch domains at lower cost. Moreover, *lwC*s additionally support snapshots.

Switching among *lwC*s is similar to migrating threads in Mach [13], where they were implemented to optimize local RPCs. Migration of threads across address spaces is also an element of the model described by Lindström et al. [18] and the COMPOSITE OS [24]. In single address space operating systems (SASOS) like Opal [8] and Mungi [15], all processes as well as persistent storage share a single large (64-bit) address space. Unlike *lwC*s, these systems do not provide privilege separation, isolation, or snapshots *within* a process.

Mondrian Memory Protection (MMP) [32] and Mondrix [33] use hardware extensions to provide protection at fine granularity within processes. The CHERI [31, 34] architecture, compiler, and operating system provides hybrid hardware-software object capabilities for fine-grained compartmentalization within a process. *lwC*s provide in-process isolation at page granularity without specialized hardware or language support.

Resource containers [3] separate the unit of resource accounting from a process and account for all resources associated with an application activity, even if the activity requires processing in multiple processes and the kernel. *lwC*s are orthogonal to resource containers.

The Corey [6] OS provides fine-grained control over the sharing of memory regions and kernel resources among CPU cores to minimize contention. *lwC*s provide the orthogonal capability of in-process isolation, privi-

lege separation, and snapshots.

Light-weight isolation, privilege separation, and snapshots can be provided also within a programming language. Functional languages like Scheme and ML provide closures through the primitive call/cc, which can be used to record a program state and revert to it later, and to implement co-routines. Typed object-oriented languages like C++ and Java provide *static* isolation and privilege separation through private and protected class fields but do not isolate objects of the same class from each other. *Dynamic* language-based protection, often implemented as object capabilities [14, 22, 23], provides fine-grained isolation and privilege separation but has considerable runtime overhead. *lwC*s instead provide in-process isolation, privilege separation, and snapshots at the OS level, independent of a programming language.

In low-level languages like C, isolation and privilege separation can be attained using binary rewriting and compiler-inserted checks as in CFI [1], CPI [17] and secure compilation [25]. All these techniques rely on dynamic checks that have runtime overhead. Techniques such as CPI and secure compilation rely on OS support for the isolation of a reference monitor, which *lwC*s can provide at low cost.

Software fault isolation (SFI) [29] and NaCl [35] rely on static checking and instrumentation of binaries to isolate memory within applications running on unmodified operating systems. SFI and NaCl do not selectively protect system calls and file descriptors. *lwC*s instead allow fine-grained control over memory, file descriptors and other process credentials, and provide snapshots as part of an OS abstraction.

Process checkpoint facilities create a linearized snapshot of a process's state [10,19,26,38]. The snapshot can be stored persistently and subsequently used to reconstitute the process and resume its execution on the same or a different machine. Checkpoint facilities are used for fault-tolerance and load balancing. *lwC*s instead provide very fast in-memory snapshots of a process's state.

The Determinator OS [2] relies on a private workspace model for concurrency control, which enables deterministic execution on multi-core platforms. To support the model, Determinator provides *spaces*, in which programs mutate private copies of shared objects. Like *lwC*s, spaces provide isolated address spaces. Unlike a *lwC*, however, a space is tied to one thread, does not have access to I/O or shared memory, and can interact only with its parent and only in limited ways.

Intel's Software Guard Extensions (SGX) [16] provide ISA support to isolate code and data in *enclaves* within a process. By mapping contexts to enclaves, SGX could be used to harden *lwC*s against a stronger threat model (untrusted OS) and to provide hardware attestation of contexts. However, enclaves have no access to OS services, so some *lwC* applications would need considerable re-architecting to run on SGX.

NOVA [27] provides protection domains (separate address spaces) and execution contexts (an abstraction similar to threads) in a micro hypervisor. NOVA's goal is to isolate VMMs and VMs from the core hypervisor, which is different from *lwC*'s goal of providing isolation, privilege separation, and snapshots within processes.

3 *lwC* design

*lwC*s are separate units of isolation, privilege, and execution state within a process. Each *lwC* has its own virtual address space, set of page mappings, file descriptor bindings, and credentials. Threads and *lwC*s are independent. Within a process, a thread executes within one *lwC* at a time and can switch between *lwC*s. *lwC*s are named using file descriptors. Each process starts with one *root lwC*, which has a well-known file descriptor number.

Table 1 shows the *lwC* API. A *lwC* may create a new (child) *lwC* using the `lwCreate` operation and receive the child's file descriptor. If a context *a* has a valid descriptor for *lwC c*, a thread executing inside *a* may switch to *c* using the `lwSwitch` operation. A *lwC c* is terminated (and its resources released) when the last *lwC* with a descriptor for *c* closes the descriptor. Common usage patterns of the *lwC* API will be shown in Section 4.

3.1 Creating *lwC*s

The `lwCreate` call creates a new (child) *lwC* in the current process. The operation's default semantics are similar to that of a POSIX *fork*, in that the child *lwC*'s initial state is an identical copy of the calling (parent) *lwC*'s state, except for its descriptor. Unlike with fork, however, child and parent *lwC* share the same process id, and no new thread is created. No execution takes place in the new *lwC* until an existing thread switches to it.

`lwCreate` returns the descriptor of the new child *lwC* *new* to the parent *lwC* with the *caller* descriptor set to -1. When a thread switches to the new *lwC* (*new*) for the first time, the `lwCreate` call returns with the caller's *lwC* descriptor in *caller* and the parent's *lwC* descriptor in *new*, along with any arguments from the caller in args.

By default, the new *lwC* gets a private copy of the calling *lwC*'s state at the time of the call, including per-thread register values, virtual memory, file descriptors, and credentials. Shared memory regions in the calling *lwC* are shared with the new *lwC*. The parent *lwC* may modify the visibility of its resources to the child *lwC* using the resource-spec argument, described in Section 3.3.

The implementation does not stop other threads executing in the parent *lwC* during an `lwCreate`. To ensure that the child *lwC* reflects a consistent snapshot of the parent's state, all threads that are active in the parent at the time of the `lwCreate` therefore should be in a consis-

Function	Return Value		System Call
Create *lwC*	{*new*, *caller*, args}	←	`lwCreate`(resource-spec, options)
Switch to *lwC*	{*caller*, args}	←	`lwSwitch`(*target*, args)
Resource access	status	←	`lwRestrict`(*l*, resource-spec)
	status	←	`lwOverlay`(*l*, resource-spec)
	status	←	`lwSyscall`(*target*, mask, syscall, syscall-args)

Table 1: API for interacting with *lwC*s. Parameters in italics *new, caller, ...* are *lwC* descriptors. Arguments args are passed during *lwC* switches; resource-spec denotes resources (e.g. memory pages, file descriptors) that can be shared or narrowed.

tent state. The application may achieve this, for instance, by barrier synchronizing such threads with the thread that calls `lwCreate`. A thread that does not exist in the parent *lwC* at the time of the `lwCreate` may not switch to the child in the future.

The `lwCreate` call takes several option flags. `LWC_SHAREDSIGNALS` controls signal handling in the child *lwC*, as described in Section 3.7. `LWC_SYSTRAP` indicates that any system calls for which the child does not hold the required OS capability should be redirected to its parent. This feature enables a parent to interpose and mediate its child's system call activity, as described in Section 3.6.

The fork semantics of `lwCreate` enable the convenient, language independent creation of *lwC*s based on the current state of the calling *lwC*. No additional APIs are required to initialize a new *lwC*. The new *lwC* can be viewed also as a snapshot of the state of the caller at the time of invoking `lwCreate`, enabling the caller to revert to this state in the future.

3.2 Switching between *lwC*s

The `lwSwitch` operation switches the calling thread to the *lwC* with descriptor *target*, passing args as parameters. `lwSwitch` retains the state of the calling thread in the present *lwC*. When this *lwC* is later switched back into by the same thread, the call returns with the switching *lwC* available as *caller* and arguments passed in args.

Note that returns from a `lwSwitch` and `lwCreate`, any signal handlers that were installed, and the instruction pointer locations of threads in a parent *lwC* at the time of a `lwCreate` define the only possible entry points into a *lwC*. (The root *lwC* has an additional one-time entry point when the process is launched.)

`lwSwitch` is semantically equivalent to a coroutine `yield`. In fact, as far as control transfer is concerned, *lwC*s can be viewed as isolated and privilege separated coroutines. Recall that a procedure is a special case of a coroutine. To achieve a (remote) procedure call among *lwC*s, the called procedure, when done, simply switches to its caller and then loops back to its beginning. This functionality can be provided easily as part of a library.

3.3 Static resource sharing

When a *lwC* is created using `lwCreate`, the child *lwC* receives a copy-on-write snapshot of all its parent's resources by default. The parent can modify this behavior using the *resource-spec* argument in the `lwCreate` operation. The *resource-spec* is an array of C unions: each array element specifies either a range of file descriptors, virtual memory addresses, or credentials. For each range, one of the following sharing options can be specified. `LWC_COW`: the child receives a logical copy of the range of resource (the default). `LWC_SHARED`: the range of resources is shared among parent and child. `LWC_UNMAP`: the range of resources is not mapped from the parent into the child. (The child may subsequently map different resources in the address range.)

When restricting the resources inherited by the child, care must be taken to minimally pass on the stacks, code, synchronization variables, and other dependencies of all threads in the parent *lwC*, to ensure predictable behavior if these threads switch to the child in the future.

3.4 Dynamic resource sharing

A *lwC* may dynamically map (overlay) resources from another *lwC* into its address space using the `lwOverlay` operation. The caller specifies which regions of a given resource type (file descriptor or memory) are to be overlayed, and whether the specified region should be copied or shared, in the *resource-spec* parameter. The `lwOverlay` call will only succeed if the caller *lwC* holds *access capabilities* (described below in Section 3.5) for the requested resources. A successful `lwOverlay` operation unmaps any existing resources at the affected addresses in the caller's address space.

3.5 Access capabilities

Access capabilities are associated with *lwC* file descriptors. Each *lwC* holds a descriptor with a universal access capability for itself. When a *lwC* is created, its parent receives a descriptor with a universal access capability for the child. A parent *lwC* may grant a child *lwC* access

capabilities for the parent *lwC* selectively by marking resource ranges as `LWC_MAY_ACCESS` in the *resource-spec* argument passed to the `lwCreate` call.

Access capabilities may be restricted on a *lwC* descriptor with the `lwRestrict` call. The *resource-spec* parameter restricts the set of resources that may be overlayed or accessed by any context that holds the *lwC* descriptor *l*. The valid resource types are file descriptors, virtual memory addresses, and syscall numbers. Subsequent to the call, the descriptor will allow `lwOverlay` to succeed for any file descriptors and memory addresses, and `lwSyscall` for any syscalls, respectively, that are within the intersection of the resource-spec set and whatever capabilities *l* had previous to the call.

3.6 System call interposition/emulation

Consider an *lwC C* that was created with the `LWC_SYSTRAP` flag. If a thread in *C* invokes a system call for which *C* does not hold a capability according to the OS's sandboxing mechanism, the thread is switched to its parent *lwC* instead, if the thread exists in the parent (if the thread does not exist in the parent, the call fails with an error). When the thread is resumed in the parent *lwC* as a result of a faulting syscall by the child, the arguments in the switch contain the system call number attempted and the arguments passed to it. The parent can choose to decline the syscall and return an error to the child, or perform a syscall on behalf of the child, possibly with different arguments (see below). To signal the completion of the child's system call, the thread executing in the parent *lwC* switches back to the child with the return value and any error code as arguments to the switch call.

An authorized *lwC* may perform a syscall on behalf of another *lwC target* using the `lwSyscall` operation. The `lwSyscall` succeeds if the *lwC* calling the operation holds an access capability (see Section 3.5) for the *target* and syscall, and holds the OS credentials required to perform the requested syscall. The effects of a successful execution of `lwSyscall` are as if the *target* had executed the requested syscall, except that it returns to the calling context. The mask parameter allows the caller to modify this behavior by specifying aspects of its own context that are to be put in place for the duration of the system call. Specifically, the caller may specify that the target's file table, memory space, credentials, or any combination be replaced by the caller's equivalent for the duration of the call. This allows the efficient implementation of useful patterns, such as enabling a untrusted *lwC* to read (or append) a fixed number of bytes from (to) a protected file *without* having access to the file descriptor.

3.7 Signal handling

*lwC*s modify the standard POSIX signal handling semantics in the following way. We distinguish between *attributable* signals, which can be attributed to the execution of a particular instruction in a *lwC*, and *non-attributable* signals, which cannot. Attributable signals, such as `SIGSEGV` or `SIGFPE`, are delivered to the *lwC* that caused the signal immediately. Non-attributable signals, such as `SIGKILL` or `SIGUSR1`, are delivered to the root *lwC* and any *lwC*s in the process that were created with the LWC_SHARESIGNALS option by a parent *lwC* that is able to receive such signals. A non-attributable signal is delivered to a *lwC* upon the next switch to the *lwC*.

3.8 System call semantics

*lwC*s modify the behavior of some existing POSIX system calls. During a `fork`, all *lwC*s in the calling process are duplicated in the child process. Any memory regions that were `mmap`'ed as MAP_SHARED in some *lwC*s of the calling process are shared with the corresponding *lwC*s in the new child process, *within and across* the two processes. Any memory regions that are shared among *lwC*s in the parent process using the LWC_SHARED option in `lwCreate` are shared among the corresponding *lwC*s *within* the child process only. An *exit* system call in any *lwC* of a process terminates the entire process.

3.9 *lwC* isolation

Because *lwC*s do not have access to the state of each others' memory, file descriptors, and capabilities unless explicitly shared, they can provide strong isolation and privilege separation within a process. Since *lwC*s share executable threads, however, an application needs to make certain assumptions about the behavior of other *lwC*s in the same process, even if they don't share resources and don't have overlay capabilities for each other. Specifically, a *lwC* can block or execute a thread indefinitely or terminate the process prematurely by invoking `exit`.

We believe these assumptions are reasonable in practice because the *lwC*s of a process are part of the same application program. Denial-of-service within a process is self-defeating. On the other hand, *lwC*s can reliably prevent accidental leakage of private information across user sessions, isolate authentication credentials and other secrets, and ensure the integrity of a reference monitor.

A *lwC* can learn about certain activities of other *lwC*s by registering for non-attributable signals. An application that wishes to limit information flow across *lwC*s should create *lwC*s without the LWC_SHARESIGNALS option (the default).

3.10 *lwC* security

*lwC*s provide isolation and privilege separation within a process, but include powerful mechanisms for sharing and control among the *lwC*s of a process. Therefore, it is important to understand the threat model and the security properties provided by the *lwC* abstraction.

Threat model We assume that the kernel is trustworthy and uncompromised, and that the tool chain used to build, link, and load the application does not have exploitable vulnerabilities that can be used to hijack control before main() starts. When a *lwC* is created, its parent has universal privileges on the *lwC*. Consequently, the security of a *lwC* assumes that its parent (and, by transitivity, all its ancestors) cannot be hijacked to abuse these privileges. In practice, the parent should drop all unnecessary privileges on the child immediately after the child is created, so this assumption is needed only with respect to the remaining privileges. When an application uses dynamic sharing, the same assumption must be extended to all *lwC*s that obtain privileges indirectly. The *lwC* API does not enable any inter-process communication or sharing beyond the standard POSIX API. Consequently, no new assumptions regarding *lwC*s in other processes are needed.

Security properties The properties of a *lwC* are constrained by the properties of the process in which it exists. A *lwC* cannot attain privileges that exceed those of its process, and the confidentiality and integrity properties of any *lwC* cannot be weaker than those of its process. The properties of the root *lwC* are those of the process. In applications that do not use dynamic sharing, the privileges of a non-root *lwC* are bounded by those of its parent and, transitively, by those of its ancestors; its integrity and confidentiality cannot be weaker than those of any of its ancestors. In applications that use dynamic sharing through the exchange of access capabilities via a common ancestor, the integrity (confidentiality) of a *lwC* depends on all siblings and descendants that have write (read) rights to it. For this reason, dynamic sharing should be used with caution.

In typical patterns of privilege separation, the root *lwC* should run a high-assurance component, i.e., one that is simple, heavily scrutinized, and exports a narrow interface. A component that protects sensitive state is at or near the root, to minimize its dependencies. More complex, less stable, network or user-facing components should be encapsulated in de-privileged *lwC*s at the leaves of a process's *lwC* tree and should execute with the least privileges required.

4 Common *lwC* usage patterns

In this section, we illustrate *lwC* use patterns for snapshots, isolation and protection rings. For some of the patterns, we use a web server as an illustrative setting. However, all the patterns are broadly applicable.

Snapshot and rollback A common *lwC* use pattern is snapshot and rollback, where a service process (such as a server worker process) initializes its state to the point where it is ready to serve requests (or sessions), snapshots this state, serves a request and rolls its state back to the snapshot before serving the next request. As compared to a setup where the process manually cleans up request-specific state after each request, the snapshot and rollback can improve performance by efficiently discarding the request-specific state with a single call, and also improves security by isolating *sequential* requests served by the same task from each other.

Algorithm 1 shows the pseudocode of a small library containing two functions—snapshot() and rollback()—and a main() server function illustrating their use. The server initializes its state and calls snapshot() on line 12 to create a snapshot. snapshot() duplicates the current *lwC* (copy-on-write) using `lwCreate` on line 2. The descriptor of the duplicated snapshot, called new, is returned at line 4 and stored in the variable snap. The program serves the request and then, to reset its state, calls rollback(). Control transfers to line 2 *in the snap* (the child) and then immediately to line 6 where the original *lwC* is closed (its resources are reclaimed). The snap recursively calls snapshot() (line 7). At line 2, it creates a duplicate of itself and returns that duplicate to main() at line 12. The cycle then repeats, with snap and its duplicate having taken the roles of the original *lwC* and the snap, respectively.

Algorithm 1 Snapshot and rollback

```
 1: function SNAPSHOT()
 2:     new,caller,arg = lwCreate(default_spec, ...)
 3:     if caller = -1 then                          ▷ parent
 4:         return new
 5:     else
 6:         close(caller)
 7:         return snapshot()
 8: function ROLLBACK(snap)                  ▷ never returns
 9:     lwSwitch(snap, 0)
10: function MAIN()
11:     ...                                  ▷ initialize state
12:     snap = snapshot()
13:     ...                                  ▷ serve request
14:     rollback(snap)
            ▷ kills current lwC, continues at line 12 in snap
```

In our evaluation, we use this pattern to roll back the state of pre-forked worker processes after each session in the Apache web server.

Isolating sessions in an event-driven server High throughput servers like nginx handle several sessions in single-threaded processes using event-driven multiplexing. However, they provide no isolation among sessions within a process. This shortcoming can be addressed using *lwC*s. Algorithm 2 illustrates the usage pattern.

The program defines a set of network socket descriptors to poll, one for each client connection, on line 10

Algorithm 2 Event-driven server with session isolation

```
 1: function SERVE_REQUEST(retlwc, client)
 2:   loop
 3:     if would_block(client) then
 4:       lwSwitch(retlwc, 0);
 5:     else if finished(client) then
 6:       lwSwitch(retlwc, 1);
 7:     else
 8:       serve(client)
 9: function MAIN
10:   descriptors = { accept_ descriptor }
11:   file2lwc_map = { accept_descriptor => root }
12:   loop
13:     next = descriptors.ready()
14:     if next = accept_descriptor then
15:       fd = accept(next)
16:       descriptors.insert(fd)
17:       specs = { ... }        ▷ Share fd descriptor only
18:       new,caller,arg = lwCreate(specs, ...)
19:       if caller = -1 then              ▷ context created
20:         file2lwc_map[fd] = new
21:       else
22:         serve_request(root, fd)
23:     else
24:       lwc = file2lwc_map[next]
25:       from, done = lwSwitch(lwc, ...)
26:       if done = 1 then
27:         close(next);close(from)
28:         descriptors.remove(next)
29:         file2lwc_map.unset(next)
```

and sets a mapping of the listening socket descriptor to the current *lwC* on line 11.

Once a descriptor is ready the program moves past line 13 and either accepts and encapsulates a new descriptor in a worker *lwC* or resumes execution of a previous one that is now ready. In the former case, the worker's *lwC* is created on line 18 such that no descriptor other than `fd` is passed to it (line 17), the created *lwC* descriptor is mapped on line 20 and the loop resumes. In the latter case, the previously mapped worker *lwC* is retrieved on line 24. This *lwC* is now immediately switched into on the subsequent line. At this point execution resumes on line 18 *in the worker*. As a result, it enters the `serve_request` function on line 22.

When the worker is done executing it switches back into the root *lwC*. It uses the `lwSwitch` argument to indicate whether it is done with its work (arg = 1) or not (arg = 0). When it switches back to the root, control flow resumes at line 25. Depending on the argument passed in from the worker, the root *lwC* either closes the socket and the worker or leaves them intact for later service.

Since all worker *lwC*s obtain a private copy of the root's state, no worker sees session-specific state of other workers. This isolates the sessions from each other.

Sensitive data isolation A third common use pattern isolates sensitive data within a process by limiting access to a single *lwC* that exposes only a narrow interface. As an illustration, Algorithm 3 shows how to isolate a private signature key that is available to a signing function, but kept hidden from the rest of the (large and network-facing) program.

Algorithm 3 Sensitive Data Isolation

```
 1: function SIGN(key, data, out_buffer)
 2: function SIGN_SSTUB(caller,arg)
 3:   loop
 4:     lwOverlay(caller,{VM,arg,sizeof(arg),SHARE})
 5:     sign(privkey, arg.in, arg.out)
 6:     lwOverlay(caller,{VM,arg,sizeof(arg),UNMAP})
 7:     caller,arg = lwSwitch(caller, 0)
 8: function SIGN_CSTUB(buf)
 9:   caller,res = lwSwitch(child, buf)
10: function MAIN
11:   ...                          ▷ initialization, load privkey
12:   child,caller,arg =
13:   lwCreate({VM,0,MAX,MAY_OVERLAY}, 0)
14:   if caller != -1 then
15:     sign_sstub(caller,arg)
16:   privkey = 0                                  ▷ erase key
17:   lwRestrict(child, {VM,0,MAX,NO_ACCESS})
18:   loop
19:     ...
20:     sign_cstub(buf)
21:     ...
```

The main function initializes the program and loads the private signing key into the variable `privkey` (line 11). Next, it calls `lwCreate` to create a second *lwC* with the same initial state (line 13). The child *lwC*, which will become the isolated compartment with access to the `privkey`, is granted the privilege to overlay any part of the parent's virtual memory.

The parent *lwC* continues executing on line 16, where it deletes its copy of the private signing key and then revokes its privilege to overlay any part of the child *lwC*'s memory. Any code executed in the parent after this point (line 17) has no way to access the private key. When this code wishes to sign data, it calls `SIGN_CSTUB` passing as argument a structure that contains the data to sign and a large enough buffer to hold the returned signature.

The `SIGN_CSTUB` function performs a `lwSwitch` to the child *lwC*, passing a pointer to the buffer as the argument. The first time the child is switched to, it returns from `lwCreate` with caller != -1 and calls `SIGN_SSTUB` (line 15), from which it does not return.

SIGN_SSTUB now uses lwOverlay to map the buffer from the parent *lwC* as a shared region into its own address space (line 4), calls the SIGN function with the private key, and then unmaps the buffer from its address space. Finally, the function calls lwSwitch to return control to the parent *lwC*, which resumes by returning from the lwSwitch in line 9. Upon future invocations of SIGN_CSTUB, the child *lwC* returns from the lwSwitch in line 7 and loops back.

In our evaluation with web servers, we use this pattern to isolate parts of the OpenSSL library that handle long-term private keys, thus protecting the keys from vulnerabilities like the widespread Heartbleed bug [7]. (Heartbleed remains a threat even after global key revocations and reissues [11, 37].)

Protected reference monitor Next, we describe a pattern that allows a parent *lwC* to intercept any subset of system calls made by its child and monitor those calls. In our evaluation, we use this pattern to implement a reference monitor for system calls made by the web server.

Algorithm 4 Reference Monitor

```
 1: function MONITOR(child)
 2:     _,call = lwSwitch(child, NULL)
 3:     loop
 4:         if is_allowed(call) then
 5:             spec = { type = CRED, SANDBOX }
 6:             rv = lwSyscall(child, spec,
                              call.num, call.params)
 7:             out.err,out.rv = errno, rv;
 8:         else
 9:             out.err,out.rv = EPERM, -1;
10:         _,call = lwSwitch(child, out)
11: function MAIN
12:     specs = { ... } ▷ Share (COW) all but private data
13:     child,c,_ = lwCreate(specs, LWC_SYSTRAP)
14:     if c = -1 then                ▷ parent becomes refmon
15:         monitor(child)                      ▷ Never returns
16:     privdrop() && run()                   ▷ Child starts here
```

Algorithm 4 shows the pseudocode of the pattern for the case where the monitoring parent is the root *lwC*. On line 13, the root creates a child *lwC* but reserves a private region, which may contain secrets (e.g., encryption keys) of which the child is not allowed to get a copy. The child is created with the flag LWC_SYSTRAP, so any system calls that the child lacks the capability for trap to the root *lwC*. Once the child *lwC* is created, the root *lwC* enters the monitoring function, which never returns.

Within the monitoring function, the root, now acting as the reference monitor, yields to the child immediately (line 2). The reference monitor regains control when the child makes a system call that it does not have the capabilities for. The reference monitor checks whether the call should be allowed (line 4) and, if so, makes the call *in the context of the child* (line 6). It yields to the child with the system call's result and error code. If the system call should be disallowed, the reference monitor yields to the child with error code EPERM. The reference monitor loops to handle the next system call.

The child starts execution on line 16 where it immediately drops privileges for all system calls that should be monitored. This causes all these system calls to trap to the reference monitor, which handles them as described above.

For simplicity, our example reference monitor merely filters system calls, a capability already provided by many operating systems. A more interesting monitor could inspect the system call arguments or other parts of the child's state by overlaying in the appropriate regions, or perform arbitrary actions and system calls on behalf of the child.

5 Implementation

We have implemented *lwC*s in the FreeBSD 11.0. We begin with a brief background of the FreeBSD kernel structures used in implementing *lwC*s.

5.1 FreeBSD Background

In implementing *lwC*s, we had to modify FreeBSD kernel data structures corresponding to process memory, file tables and credentials.

Memory In FreeBSD, the address space of a process is organized under a vmspace structure (described fully in [21]). Within the address space, there are virtual memory regions that correspond to a contiguous interval of memory mapped into the process's virtual address space. These memory regions are represented as vm_map_entry structures. Attempting to access any memory that is not within a memory region results in a segmentation fault.

Two memory regions that are contiguous and have the same protection bits can be merged into a single vm_map_entry. The number of memory regions within a process is typically small (few tens), though for some processes (notably Apache, that maps modules into different regions) it can be larger. Work performed during fork and lwCreate is proportional to the number of vm_map_entry structures.

Switching the virtual address space map of a process during a context switch (*lwC* or otherwise) can be a relatively efficient operation on modern processors. Previous generations of processors required a TLB flush whenever the address space had to be changed, as is the case during process context switches, or *lwC* switches. Modern processors include a "process context identifier" (PCID) that can be used to distinguish pages that belong to differ-

ent page tables. (On current Intel processors, the PCID is 12-bits, enabling 4096 different page tables to be distinguished.) TLB entries are tagged with the PCID that was active when they were resolved. Whenever the active page table is ready to be changed, the kernel sets the CR3 register to a value containing the PCID and the address of the first page directory entry. Any cached TLB entries that share this PCID are considered valid and may be used. Importantly, the entire TLB does not have to be flushed upon a context switch since entries belonging to other PCIDs are simply considered invalid by the hardware. This facility reduces the cost of context switches by reducing the frequency of TLB flushes. FreeBSD 11.0 supports PCIDs and each *lwC* is assigned a unique one for every core it is activated on.

File Table In FreeBSD, all files, sockets, devices, etc. open in a process are accessible via the process's file table, which is held as a reference in the process structure. Each entry contains a cursor, per-process flags, and access capabilities. In our implementation, *lwC*s are also accessed via file-table entries. Upon `fork`, the file table is copied from the parent to the child process.

Credentials Process credentials determine capabilities and privileges, and include process user identifiers (uid, gid), limits (cpu time, maximum number of file descriptors, stack size, etc.), the current FreeBSD jail (a restrictive chroot-like environment) the process is operating in, and other accounting information.

The credentials of a process are attached to the process structure via a `struct ucred` pointer. Upon a `fork`, a reference to the parent structure is given to the child; system calls that modify the credential structure allocate a new `struct ucred` for the process, and copy unmodified fields from the parent.

5.2 *lwC* Implementation

Like a process, each *lwC* has a file table, virtual memory space, and credentials associated with it.

Memory Unless otherwise specified, `lwCreate` replicates the `vmspace` associated with the parent *lwC* in exactly the same manner as `fork`. However, any memory regions that are specified as `LWC_UNMAP` during the `lwCreate` call are not mapped into the new *lwC*'s address space. Any memory regions that are marked as `LWC_SHARE` are mapped into the *lwC* as memory that differs from shared memory in only one respect: a subsequent `fork` will not share this region with its parent. During a `lwSwitch`, the calling thread saves its CPU registers, releases its reference to the current vmspace structure, and acquires a reference from the address space of the switched to *lwC*.

File Table By default, during a call to `lwCreate` all file descriptors are copied into the *lwC* file table in the same manner as `fork` except that any associated file descriptor overlay rights are copied as well, as described in section 5.2. If the user specifies an interval in the resource specifier as `LWC_UNMAP`, the corresponding descriptors are not copied into the file table. The user may specify that the entire file table is to be shared; in this scenario, as an optimization, we store a reference to the parent *lwC*'s file table.

lwC **descriptors** With one exception, *lwC* descriptors have the same visibility as regular file descriptors. Upon `lwCreate`, if the file table or a *lwC* descriptor is not shared, then the child *lwC* is not able to access the parent's *lwC*s. *lwC*s closed with the `close` syscall results in their removal from the calling *lwC*'s file table. Upon a `lwCreate` or `lwSwitch`, if a *caller* parameter is specified, then the newly created (or switched to) *lwC a* inherits a reference to the caller *lwC b* as a file descriptor. This descriptor, corresponding to *b*, is inserted into *a*'s file table when *a* is switched to next. (If *a*'s file table already had a descriptor for *b*, then that descriptor is reused, and *a*'s file table is not modified.)

Credentials We copy credentials the same way that they are copied during a `fork` call. Restoring previous credentials (using a *lwC* switch) may reverse calls that dropped privileges/put the process into a sandbox. Our reference monitor example (Section 4) shows how this mechanism can be used. Credentials are treated similarly to file descriptors and `vmspace` structures. The calling thread's credential structure is replaced with a reference to the target *lwC*'s reference structure.

Permissions and Overlays An executing *lwC* interacts with another *lwC* within a process by either switching to it or by overlaying (some of) that *lwC*'s resources.

A *lwC a* may switch to a *lwC b* only if *b*'s descriptor is present in *a*'s file table. Overlay permissions are more fine-grained: upon creating a new *lwC c*, the parent *p* passes a set of resource specifiers. Some of these may have `LWC_MAY_OVERLAY` flag set, which allows *c* to overlay specified resources from *p*.

The `lwCreate` call (*p* creating *c*) results in two file descriptors. One refers to *c* and has full overlay rights, and is inserted into *p*'s file table. Thus the creator (parent) *lwC* obtains all rights to the child.

The second descriptor, given to *c*, refers to the *p lwC* and only allows overlays on the descriptor as specified by *p* in the `lwCreate` call. File descriptors duplicated via the `dup` or similar calls create a new descriptor with a copy of the overlay rights. These rights can be narrowed using the `lwRestrict` call.

The `lwOverlay` call imports resources from one *lwC* into the calling *lwC*, assuming permissions are not violated. File table entries that are masked by an overlay are closed prior to inserting new entries. Similarly, mem-

ory region overlays unmap existing regions in the calling *lwC* that are within the overlay interval prior to importing overlaid regions. If the `LWC_SHARE` flag is set, the memory will be shared with the target *lwC* (i.e., writes will be visible to both *lwC*s). This sharing does not persist past a `fork`.

Multi-Threaded Support Our implementation supports *lwC*s in multithreaded programs. In addition to necessary synchronization, *lwC*-specific state that used to be associated with a process (and shared amongst all threads) must instead be associated with each *lwC*. This does not affect the existing semantics of processes because in normal operation each thread has a reference counted pointer to shared objects (e.g., memory spaces). Once *lwC* system calls are invoked it is possible for two threads to reference separate address spaces (i.e., *lwC*s). The modifications to the existing kernel were largely superficial outside of process creation and destruction.

6 Evaluation

In this section, we evaluate *lwC*s using microbenchmarks, and when applying the usage patterns discussed in Section 4 in the context of the Apache and nginx web servers. Our experiments were performed on Dell R410 servers, each with 2x Intel Xeon X5650 2.66 GHz 6 core CPUs with both hyperthreading and SpeedStep disabled, 48GB main memory, running FreeBSD 11.0 (amd64) and OpenSSL 1.0.2. The servers were connected via Cisco Nexus 7018 switches with 1Gbit Ethernet links. Each server has a 1TB Seagate ST31000424SS disk formatted under UFS.

6.1 *lwC* switch

Table 2 compares the time to execute a `lwSwitch` call compared to context switching between processes (using a semaphore), between kernel threads (using a semaphore, which we found to be faster than a mutex), and user threads. The user threads use the `getcontext` and `setcontext` calls specified by POSIX.1-2001. A *lwC* switch takes less than half the time of a process or kernel thread switch. The reason is that a *lwC* switch avoids the synchronization and scheduling required for a process or thread context switch, instead requiring only a switch of the vm mapping. Somewhat surprisingly, a kernel thread switch is on par with a process context switch when both use the same form of synchronization. The reason is that the kernel code executed during a switch between two kernel threads in the same process or in different processes is largely the same.

User threads are only moderately faster than *lwC* switches, because in FreeBSD 11, the user context switch is implemented by a system call. In Linux glibc, it is instead implemented in userspace assembly. In an experiment with Linux 3.11.10 on the same hardware, user thread switches run in 6% of the time required by semaphore-based kernel thread switches.

lwC	process	k-thread	u-thread
2.01 (0.03)	4.25 (0.86)	4.12 (0.98)	1.71 (0.06)

Table 2: Median switch time (in microseconds) and standard deviation over ten trials.

6.2 *lwC* creation

Next, we measured the total cost of creating, switching to, and destroying *lwC*s with default arguments (all resources shared COW with the parent) within a single process. When no pages are written in either the parent or child *lwC* during the lifetime of the child, the system is able to create, switch into once, and destroy an *lwC* in 87.7 microseconds on average, with standard deviation below 1%. This result is independent of the amount of memory allocated to the process. Each page written in either parent or child, however, causes a COW fault, which requires a page frame allocation and copy. When 100, 1000, 10000, and 100000 pages are written in the child during the experiment described above, the average total time taken per *lwC* increases to 397, 3054, 35563, and 34182 microseconds, respectively. Standard deviation was below 7% in all cases. The cost of maintaining a separate *lwC* is approximately linearly dependent on the number of unique pages it creates, and is lowest when *lwC*s in a process share most of their pages.

The results of our microbenchmarks can be used to estimate the cost of using *lwC*s in an application, given an estimate of the rate of *lwC* creations and switches, and the number of unique pages in each *lwC*. Later in this section, we evaluate the overhead of *lwC*s in the context of specific applications: Apache and nginx.

6.3 Reference monitoring

Following the pattern described in Section 4, we have implemented an in-process reference monitor using *lwC*s. When a process starts, the reference monitor gains control first and creates a child *lwC*, which executes the server application. The child *lwC* is sandboxed using FreeBSD Capsicum and disallowed from using certain system calls, which are instead redirected to the parent *lwC* using the LWC_SYSTRAP option. Our reference monitor restricts access to the filesystem, though other policies that restrict any system call or inspect memory (using `lwOverlay`) can readily be implemented within our basic schema. We compare the *lwC* reference monitor (**lwc-mon**) to two other techniques:

Inline Monitoring (inline) This is a baseline scheme where the reference monitor checks are inlined with the application code. The monitored process is

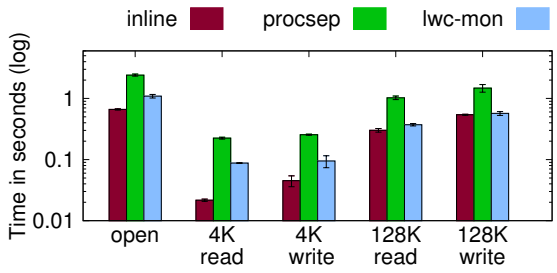


Figure 1: Cost of 10,000 monitored system calls in seconds (log scale). Error bars show standard deviation.

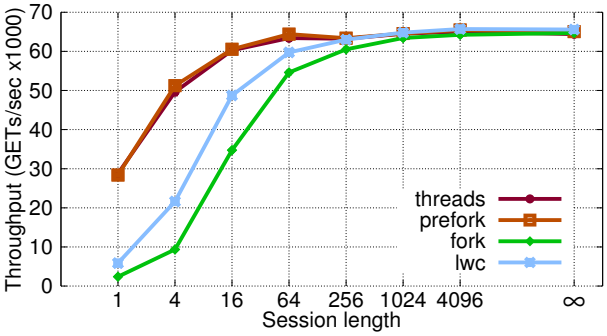
LD_PRELOADed with a library that intercepts each system call and checks arguments. Inlining provides a lower bound on overhead, but does not provide security since the monitored process can overwrite the checks or otherwise bypass the interception library.

Process Separation (procsep) This method provides a secure reference monitor in a separate process. The monitored process runs in a sandbox based on FreeBSD Capsicum [30]: the sandbox ensures that the monitored process is unable to issue prohibited system calls (e.g. **open**). At initialization, but prior to entering the sandbox, the monitored process connects to the reference monitor process over a Unix domain socket, which it can subsequently use to communicate with the reference monitor, even while sandboxed. All **open** calls (which the sandbox restricts) must be vectored through this socket, which allows the reference monitor to inspect and restrict the access as necessary. If the access is to be granted to the sandboxed application, the reference monitor shares a file descriptor over the socket.

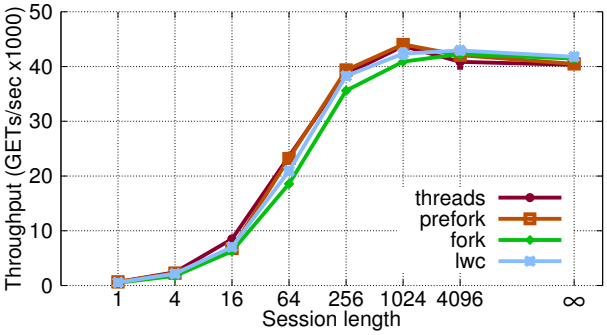
Figure 1 shows the overhead of monitoring open, read and write system calls, while an application is accessing a file stored in an in-memory file system. The application calls each system call 10,000 times and we report the average of 5 runs. Faster system calls have higher relative overhead since the fixed cost of redirecting the system call has to be paid. **lwc-mon** does not require data copying or IPC and hence outperforms **procsep** by a factor of two or more.

6.4 Apache

Modern web servers are designed to efficiently map user sessions to available processing cores. For instance, the popular Apache HTTP server provides multi-threading using kernel threads (**threads**) in one configuration and pre-forked processes that map to different cores (**prefork**) in another. Higher performance servers, such as nginx, use an event loop (based on kqueue or epoll) within a process, and have the option of spawning multiple processes that map to cores, each with their own



(a) HTTP



(b) HTTPS

Figure 2: Apache throughput in (GETs/sec) of 128 concurrent clients, 45 byte docs. Error bars show standard deviation, which was below 3.7%.

event loop.

Consider the problem of isolating individual user sessions to separate the privileges of different user sessions or to implement per-user information flow control. None of the above mentioned server configurations provide such isolation: multi-threaded and event-driven configurations serve different sessions concurrently in the same process; pre-forked processes sequentially share among different sessions. Apache can be configured to fork a new process for each user session (**fork**), which provides memory isolation and privilege separation. As our results demonstrate, however, this configuration has low performance for small session lengths, due to the overhead of forking processes[2].

*lwC*s can provide memory isolation, privilege separation, and high performance. We have augmented the prefork mode in Apache (version 2.4.18) to provide session isolation using the snapshot and rollback pattern from Section 4. Within each Apache process, we create a *lwC* that serves a user session; when the session ends, the

[2]In fact, we had to patch Apache (in server/mpm_common.c) to continuously check the status of child processes (rather than at 1s intervals) to get this configuration to perform at all at small to modest session lengths.

lwC switches (reverts) to its initial (untainted) state before serving the next user session, thereby ensuring the isolation property.

In the following set of experiments, we use ApacheBench (`ab`) to issue HTTP and HTTPS requests to our Apache server. We modified `ab` to support varying client session lengths by using HTTP Keepalive and terminating a session after a certain number of requests. We launch a single ApacheBench instance which repeatedly makes up to 128 concurrent requests for a small 45 byte document. We chose small document requests to make sure the results are not I/O-bound. Figure 2 shows the number of GET requests served per second by the different Apache configurations at different session lengths, and for HTTP and HTTPS. For HTTPS, the server uses TLSv1.2, ECDHE-RSA-AES256-GCM-SHA384 with 4096 bit keys. The results were averaged over five runs of 60 seconds each.

At session length ∞, each client maintains a session for the duration of the experiment. The **threads** and **prefork** configurations, which provide no isolation, perform comparably for all session lengths and protocols. **fork** and **lwc** configurations provide isolation: **lwc** has better throughput in all cases, and has a significant advantage for short sessions (256 and below), particularly for HTTP. (In HTTPS, the high CPU overhead for session establishment dominates overall cost; however, emerging hardware support for crypto will diminish these costs, exposing once again the costs of isolation.) Moreover, **lwc** achieves performance comparable to the best configuration *without isolation* for sessions lengths of 256 and larger.

We also repeated the experiment with GET requests for 900 byte documents. These documents are 20x larger but still small enough not to saturate the network link. The trends and relative throughput between the different configuration were very close to those in Figure 2, with the absolute peak throughput within 10%.

We have integrated reference monitoring within Apache (and nginx). Figure 3 shows the throughput of Apache **prefork** in different reference monitor configurations when used to serve short (45 byte) documents. The results were averaged over five runs of 20 seconds each. In this experiment, the **open** and **stat** system calls are monitored and checked against a whitelist of allowed directories. These results show that a reference monitor implementation based on in-process *lwC* incurs lower overhead than an implementation based on process separation even for large applications where the monitored system calls constitute only part of what the applications do. The overhead of reference monitoring increases with session length due to the increase in relative number of reference monitored system calls (open and stat) compared to other system calls (accept, read, send, close).

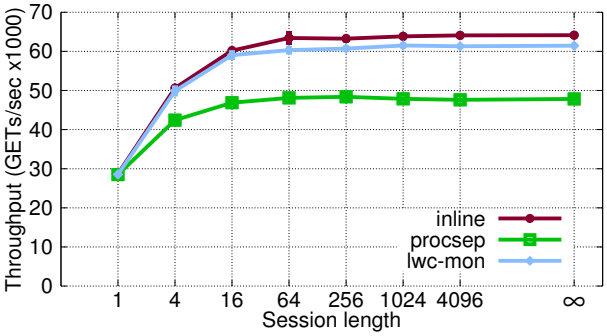


Figure 3: Throughput of different Apache reference monitoring configurations in (GETs/sec) of 128 concurrent clients, 45 byte docs. Error bars show standard deviation, which was below 2%.

6.5 Nginx

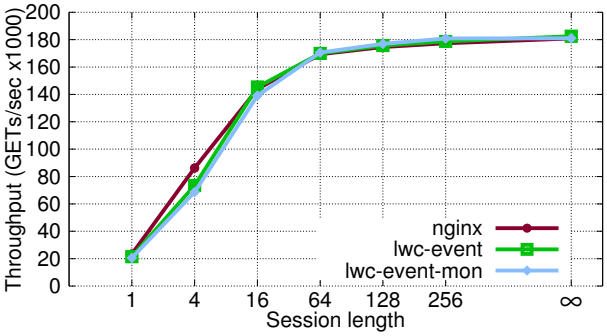
To enable session isolation in nginx (version 1.9.15), we allocate a *lwC* for each new connection: each event for a single connection is isolated within the *lwC*, following the session isolation pattern from Section 4. Note that in the nginx case, each process may serve many different connections simultaneously, and our implementation creates a *lwC* per active connection within the process. We have also integrated a reference monitor with nginx.

We experiment with different nginx configurations: the stock **nginx**, **lwc-event** augments nginx's event loop to create a new *lwC* per connection, and **lwc-event-mon** combines a reference monitor with the per-connection *lwC*. In each case we configured nginx to use 10 worker processes, as we found that this had the best performance. We launch four ApacheBench instances, each of which repeatedly makes up to 75 concurrent requests for a small 45 byte document.

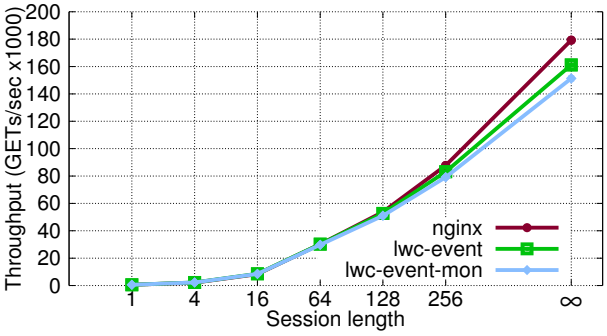
Figure 4 shows the average number of queries served by each of the configurations over five runs of 60 seconds each. The standard deviation did not exceed 0.9%.

nginx is considered the state of the art high-performance server. It uses a highly optimized event loop and is about 2.88x quicker than Apache. Introducing *lwC*s in this base configuration (named **lwc-event** in the results) has no significant impact on the throughput of this high-performance configuration. Similarly, reference monitoring adds only minimal overhead. For both HTTP and HTTPS, with isolation and reference monitoring, *lwC*-augmented nginx performs comparably to native nginx.

Large scale servers may need to maintain tens of thousands of concurrent user sessions. Using *lwC*s for session isolation increases the amount of per-session state. Therefore, our next experiment explores how using *lwC*s for session isolation affects nginx's performance under a



(a) HTTP



(b) HTTPS

Figure 4: Nginx throughput in GETs/sec with 10 workers, 45B documents, 300 concurrent requests. Error bars show standard deviation, which was below 0.9%.

large number of concurrent client connections. We experimented with two configurations: in the first, we use between 6 and 76 ApacheBench instances, and each instance issues 250 concurrent requests for a 45 byte document. The session length was 256 and we used 10 nginx workers. The second configuration is identical except the ApacheBench instances request 900 byte documents.

Figure 5 shows the average number of requests served, over 5 runs of the experiment, as a function of the number of client sessions for stock nginx and **lwc-event**for both file sizes.

For small documents, **lwc-event** matches the performance of native nginx up to 6500 clients. Beyond, the performance of both configurations declines following the same trend, but the absolute throughput of **lwc-event** falls below that of nginx by up to 19% at 19,500 concurrent clients. In investigating this result further, we find that FreeBSD kernel threads, in particular, the interrupt handler thread, gets CPU bound after 6500 clients, and the CPU consumption of the nginx worker threads *reduces* with higher numbers of clients as the nginx worker threads block waiting for the kernel to demultiplex packets. The **lwc-event** configuration further pays an extra cost of *lwC* switches, which reduces performance compared to stock nginx. However, given that **lwc-event** provides session isolation, this is a still a strong result.

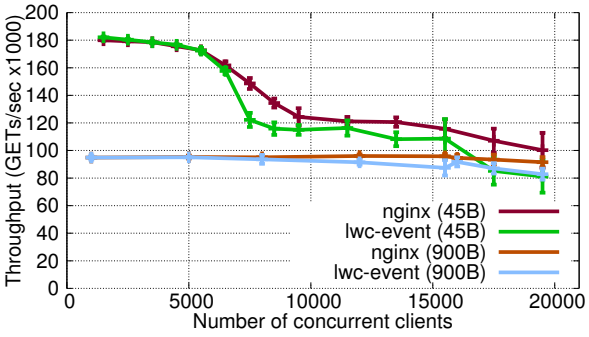


Figure 5: Nginx cumulative throughput in GETs/sec with 10 workers, session length 256, 45B and 900B documents, increasing number of concurrent clients. Error bars show standard deviation.

For 900 byte documents, the performance of stock nginx and **lwc-event** remain similar until ~12000 simultaneous clients. Performance of stock nginx is not affected by increasing numbers of clients: this is because the rate of incoming requests is lower, which means the kernel threads do not saturate the CPU. With increasing numbers of clients, eventually the cost of *lwC* switches, which were amortized over serving a larger document, become a measurable factor.

Overall, our results show that using *lwC*s, it is possible to implement features such as session isolation and reference monitoring at low cost for both HTTPS and HTTP sessions, and even in a high-performance server under a challenging workload.

6.6 Isolating OpenSSL keys

*lwC*s provide a particularly effective way to isolate sensitive data from network-based attacks such as buffer overflows or overreads. The sensitive data is stored in a *lwC*, within the process, such that the network-facing code has no visibility into pages that store the sensitive data. In this way, unless the kernel is compromised, the data is guaranteed safe, but access to functions that require the data can be rapid, using a safe *lwC*-crossing interface.

As an example, we have isolated parts of the OpenSSL library that manipulate secret information within Apache and nginx. In our case, the web server certificate private keys are isolated; note that such a scheme would have rendered attacks such as Heartbleed completely ineffective since the buffer overread that Heartbleed relied on would not have visibility into the memory storing the private keys. We evaluate this scheme using the following configurations:

In-process LwC Sensitive data is stored in a *lwC* within the process, following the pattern from Algo-

rithm 3 in Section 4. The network-facing code within the process has no visibility into the sensitive data; access is through a narrow interface exported via *lwC* switch entry points. The isolated *lwC* has a copy of the original process at the time of creation and may call whatever functions are available within its address space. Our encapsulated OpenSSL library takes advantage of this fact because the isolated *lwC* hosts a COW copy of the OpenSSL code and global state and need not be aware that it is running in a restricted environment. None of the changes in the sensitive *lwC* are visible to the network facing code.

We evaluate the cost of providing this isolation by performing SSL handshakes (TLSv1.2,ECDHE-RSA-AES256-GCM-SHA384 with 4096 bit keys) with the nginx web server. The server was configured to spawn four worker processes. We used ApacheBench with concurrency level 24 and a session length of 1. In our experiments, native nginx required 99.7 seconds to complete ten thousand SSL handshakes, whereas the configuration with a *lwC* isolated SSL library required 100.4 seconds. With *lwC*s, isolating SSL private keys is essentially free.

Our prototype isolates only the server certificate private key, but not session keys or other sensitive information. More fine-grained isolation of the OpenSSL state, such as that described in [5], can be implemented readily using *lwC*s.

6.7 FCGI fast launch

We demonstrate the utility of *lwC* snapshotting by adding a "fast launch" capability to a PHP application. When a PHP request is served, a PHP script is read from disk, compiled by the interpreter, and then executed. During execution, other PHP files may be included and executed. We modified the PHP 7.0.11 programming language to add a `pagecache` call that allows the script to "fast-forward" using previous snapshots. Our implementation augments PHP-FPM [28], which functions as a FCGI server for nginx. Our test application is based on the MVC skeleton application that is included with the Zend PHP framework [36], which provides the core functionality for creating database-backed web-based applications such as blogs.

Before a PHP script performs any computation that depends on request-specific parameters (e.g., cookie information), the script may invoke the `pagecache` call, which implements the snapshot pattern (Algorithm 1). The first time a `pagecache` is invoked, we take a snapshot and then revert to it on subsequent requests to the same URL, effectively jumping execution forward in time. We use a shared memory segment to store data that must survive a snapshot rollback, including request-specific data and network connection information.

Our experiments run PHP-FPM with 11 workers. PHP itself includes an opcode cache (which caches the compilation of each script in memory) and our results include configurations where the PHP opcode cache is enabled and not. When combining the opcode cache and the *lwC* snapshot, we warm up the opcode cache before taking the snapshot. The results in Table 3 are an average of five runs and overall standard deviation was less than 2%.

stock php no cache	*lwC* php no cache	stock php cache	*lwC* php cache
226.1	615.8	1287.5	1701.4

Table 3: Average requests per second over 60 seconds with 24 concurrent requests.

With or without the opcode cache, the *lwC* snapshot is able to skip over much of the initialization of the runtime and whatever PHP execution would otherwise occur before the `pagecache` call. This result is remarkable in that it shows *lwC*s can provide significant performance benefit to highly optimized end-to-end applications such as web frameworks, *while adding isolation between user requests.*

7 Conclusions

We have introduced and evaluated light-weight contexts (*lwC*s), a new first-class OS abstraction that provides units of isolation, privilege, and execution state independent of processes and threads. *lwC*s provide isolation and privilege separation among program components within a process, as well as fast OS-level snapshots and coroutine style control transfer among contexts, with a single abstraction that naturally extends the familiar POSIX API. Our results show that fast roll-back of FCGI runtimes, compartmentalization of crypto secrets, isolation and monitoring of user sessions can be implemented in the production Apache and nginx web server platforms with performance close to or better than the original configurations in most cases.

8 Acknowledgments

We would like to thank the anonymous reviewers, Paarijaat Aditya, Björn Brandenburg, Mike Hicks, Pete Keleher, Matthew Lentz, Dave Levin, Neil Spring, and our shepherd KyoungSoo Park for their helpful feedback. This research was supported in part by US National Science Foundation Awards (TWC 1314857 and NeTS 1526635), the European Research Council (ERC Synergy imPACT 610150), and the German Science Foundation (DFG CRC 1223).

References

[1] ABADI, M., BUDIU, M., ERLINGSSON, U., AND LIGATTI, J. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS)* (2005), pp. 340–353.

[2] AVIRAM, A., WENG, S.-C., HU, S., AND FORD, B. Efficient system-enforced deterministic parallelism. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2010), OSDI'10, USENIX Association, pp. 193–206.

[3] BANGA, G., DRUSCHEL, P., AND MOGUL, J. C. Resource containers: A new facility for resource management in server systems. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation* (Berkeley, CA, USA, 1999), OSDI '99, USENIX Association, pp. 45–58.

[4] BELAY, A., BITTAU, A., MASHTIZADEH, A., TEREI, D., MAZIÈRES, D., AND KOZYRAKIS, C. Dune: Safe user-level access to privileged CPU features. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)* (Hollywood, CA, 2012), USENIX, pp. 335–348.

[5] BITTAU, A., MARCHENKO, P., HANDLEY, M., AND KARP, B. Wedge: Splitting applications into reduced-privilege compartments. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2008), NSDI'08, USENIX Association, pp. 309–322.

[6] BOYD-WICKIZER, S., CHEN, H., CHEN, R., MAO, Y., KAASHOEK, F., MORRIS, R., PESTEREV, A., STEIN, L., WU, M., DAI, Y., ZHANG, Y., AND ZHANG, Z. Corey: An operating system for many cores. In *8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2008).

[7] CERT Vulnerability Note VU#720951: OpenSSL TLS heartbeat extension read overflow discloses sensitive information. `http://www.kb.cert.org/vuls/id/720951`.

[8] CHASE, J. S., LEVY, H. M., FEELEY, M. J., AND LAZOWSKA, E. D. Sharing and protection in a single-address-space operating system. *ACM Trans. Comput. Syst. 12*, 4 (Nov. 1994), 271–307.

[9] CHEN, Y., REYMONDJOHNSON, S., SUN, Z., AND LU, L. Shreds: Fine-grained execution units with private memory. *2016 IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 23-25, 2015* (2016), 20–37.

[10] DIETER, W. R., AND LUMPP, JR., J. E. User-level checkpointing for LinuxThreads programs. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference* (Berkeley, CA, USA, 2001), USENIX Association, pp. 81–92.

[11] DURUMERIC, Z., KASTEN, J., LI, F., AMANN, J., BEEKMAN, J., PAYER, M., WEAVER, N., HALDERMAN, J. A., PAXSON, V., AND BAILEY, M. The matter of Heartbleed. In *ACM Internet Measurement Conference (IMC)* (2014).

[12] EL HAJJ, I., MERRITT, A., ZELLWEGER, G., MILOJICIC, D., ACHERMANN, R., FARABOSCHI, P., HWU, W.-M., ROSCOE, T., AND SCHWAN, K. SpaceJMP: programming with multiple virtual address spaces. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2016), ASPLOS '16, ACM, pp. 353–368.

[13] FORD, B., AND LEPREAU, J. Evolving Mach 3.0 to a migrating thread model. In *Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference* (Berkeley, CA, USA, 1994), WTEC'94, USENIX Association.

[14] GOOGLE CAJA TEAM. Google-Caja: A source-to-source translator for securing javascript-based web.

[15] HEISER, G., ELPHINSTONE, K., VOCHTELOO, J., RUSSELL, S., AND LIEDTKE, J. The Mungi single-address-space operating system. *Softw. Pract. Exper. 28*, 9 (July 1998), 901–928.

[16] INTEL CORP. *Intel 64 and IA-32 Architectures Software Developer's Manual: Vol. 3D*, June 2016.

[17] KUZNETSOV, V., SZEKERES, L., PAYER, M., CANDEA, G., SEKAR, R., AND SONG, D. Code-pointer integrity. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2014), pp. 147–163.

[18] LINDSTROM, A., ROSENBERG, J., AND DEARLE, A. The grand unified theory of address spaces. In *Proceedings of the Fifth Workshop on Hot Topics in Operating Systems (HotOS-V)* (Washington, DC, USA, 1995), HOTOS '95, IEEE Computer Society.

[19] LITZKOW, M., TANNENBAUM, T., BASNEY, J., AND LIVNY, M. Checkpoint and migration of UNIX processes in the Condor distributed processing system. Tech. Rep. UW-CS-TR-1346, University of Wisconsin—Madison CS Department, April 1997.

[20] MAMBRETTI, A., ONARLIOGLU, K., MULLINER, C., ROBERTSON, W., KIRDA, E., MAGGI, F., AND ZANERO, S. Trellis: Privilege Separation for Multi-User Applications Made Easy. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)* (Sept. 2016).

[21] MCKUSICK, M. K., AND NEVILLE-NEIL, G. V. *The Design and Implementation of the FreeBSD Operating System*. Pearson Education, 2004.

[22] METTLER, A., WAGNER, D., AND CLOSE, T. Joe-e: A security-oriented subset of java. In *NDSS* (2010), vol. 10, pp. 357–374.

[23] MILLER, M. *Robust composition: Towards a unified approach to access control and concurrency control*. PhD thesis, Johns Hopkins University, 2006.

[24] PALMER, G. The case for thread migration: Predictable IPC in a customizable and reliable OS. In *Proceedings of the Workshop on Operating Systems Platforms for Embedded Real-Time applications (OSPERT '10)* (2010).

[25] PATRIGNANI, M., AGTEN, P., STRACKX, R., JACOBS, B., CLARKE, D., AND PIESSENS, F. Secure compilation to protected module architectures. *ACM Transactions on Programming Languages and Systems 37*, 2 (Apr. 2015).

[26] PLANK, J. S., BECK, M., KINGSLEY, G., AND LI, K. Libckpt: Transparent checkpointing under Unix. In *Usenix Winter Technical Conference* (January 1995), pp. 213–223.

[27] STEINBERG, U., AND KAUER, B. Nova: A microhypervisor-based secure virtualization architecture. In *Proceedings of the 5th European Conference on Computer Systems* (2010), EuroSys '10, pp. 209–222.

[28] THE PHP GROUP. FastCGI Process Manager (FPM). `http://php.net/manual/en/install.fpm.php`, 2016.

[29] WAHBE, R., LUCCO, S., ANDERSON, T. E., AND GRAHAM, S. L. Efficient software-based fault isolation. *SIGOPS Oper. Syst. Rev. 27*, 5 (Dec. 1993), 203–216.

[30] WATSON, R. N. M., ANDERSON, J., LAURIE, B., AND KENNAWAY, K. A taste of Capsicum: Practical capabilities for unix. *Communnications of the ACM 55*, 3 (Mar. 2012).

[31] WATSON, R. N. M., WOODRUFF, J., NEUMANN, P. G., MOORE, S. W., ANDERSON, J., CHISNALL, D., DAVE, N. H., DAVIS, B., GUDKA, K., LAURIE, B., MURDOCH, S. J., NORTON, R., ROE, M., SON, S., AND VADERA, M. CHERI: A hybrid capability-system architecture for scalable software compartmentalization. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015* (2015), pp. 20–37.

[32] WITCHEL, E., CATES, J., AND ASANOVIĆ, K. Mondrian memory protection. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2002), ASPLOS X, ACM, pp. 304–316.

[33] WITCHEL, E., RHEE, J., AND ASANOVIC, K. Mondrix: Memory isolation for Linux using Mondriaan memory protection. In *Proceedings of the 20th Symposium on Operating Systems Principles (SOSP '05)* (Brighton, UK, October 2005).

[34] WOODRUFF, J., WATSON, R. N., CHISNALL, D., MOORE, S. W., ANDERSON, J., DAVIS, B., LAURIE, B., NEUMANN, P. G., NORTON, R., AND ROE, M. The CHERI capability model: Revisiting RISC in an age of risk. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture* (Piscataway, NJ, USA, 2014), ISCA '14, IEEE Press, pp. 457–468.

[35] YEE, B., SEHR, D., DARDYK, G., CHEN, J. B., MUTH, R., ORMANDY, T., OKASAKA, S., NARULA, N., AND FULLAGER,

N. Native Client: A sandbox for portable, untrusted x86 native code. *2009 IEEE Symposium on Security and Privacy, SP 2016, Berkeley, CA, USA, May 17-20, 2009* (2016), 79–93.

[36] ZEND. MVC Skeleton Application. `https://framework.zend.com/downloads/skeleton-app`, 2016.

[37] ZHANG, L., CHOFFNES, D., DUMITRAŞ, T., LEVIN, D., MISLOVE, A., SCHULMAN, A., AND WILSON, C. Analysis of SSL Certificate Reissues and Revocations in the Wake of Heartbleed. In *ACM Internet Measurement Conference (IMC)* (2014).

[38] ZHONG, H., AND NIEH, J. CRAK: Linux checkpoint/restart as a kernel module. Tech. Rep. CUCS-014-01, Columbia University CS Department, November 2001.