HCSS 2012

Static Previrtualization[1]

N. Shankar

Computer Science Laboratory
SRI International
Menlo Park, CA

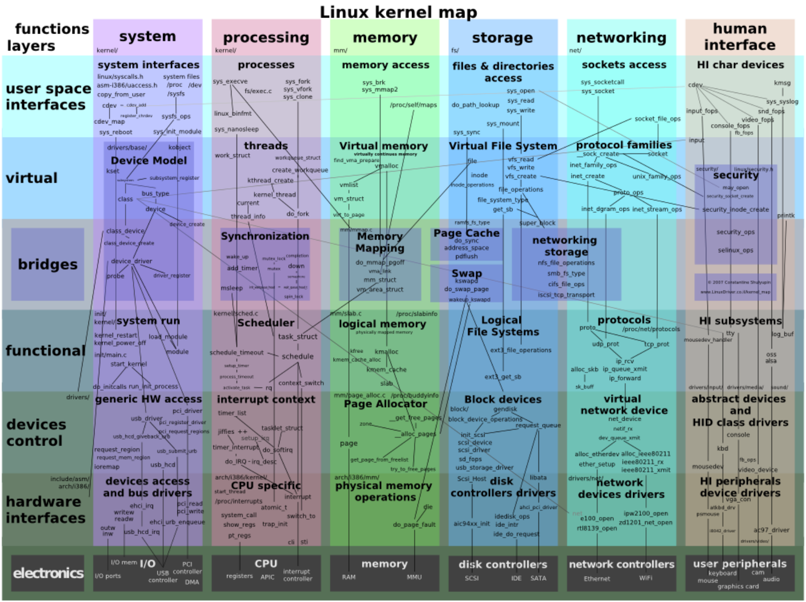
May 9, 2012

[1] Funded by the Maryland Procurement Office Contract No. R21.002.10 and NASA Cooperative Agreement NNX08AY53A. Joint work with Drew Dean, Bruno Dutertre, Ashish Gehani, Gregory Malecha, Ian Mason, Sam Owre, Hassen Saïdi.

The View from the Top

- The modern hardware/software/communication stack is an unprecedented success.
- Complex applications are now easier to write.
- But this simplicity comes at a cost
  - **MiniBlog** – "Simple" PHP blogging application
    - 683 lines of PHP code
    - Depends on PHP & MySQL
  - **PHP** – Programming language interpreter
    - 625,000 lines of C
    - Depends on LibC
  - **LibC** – C standard runtime library
    - 650,000 lines of C
    - Depends on Linux kernel
  - **Linux Kernel** – Operating System
    - 15 million lines of code!

A House of Cards? [Wikipedia]



Linux kernel map

functions layers | system kernel/ | processing kernel/ | memory mm/ | storage fs/ | networking net/ | human interface

user space interfaces
- system interfaces: linux/syscalls.h, asm-i386/uaccess.h, copy_from_user, system files /proc /dev /sysfs, cdev, cdev_add, cdev_map, register_chrdev, sysfs_ops, sys_reboot, sys_init_module
- processes: sys_execve, fs/exec.c, sys_fork, sys_vfork, sys_clone, linux_binfmt, sys_nanosleep
- memory access: sys_brk, sys_mmap2, /proc/self/maps
- files & directories access: sys_open, sys_read, sys_write, do_path_lookup, sys_mount, sys_sync
- sockets access: sys_socketcall, sys_socket, socket_file_ops
- HI char devices: cdev, kmsg, sys_syslog, input_fops, snd_fops, video_fops, console_fops, fb_fops, input

virtual
- Device Model: drivers/base/, kobject, kset, subsystem_register, bus_type, class, device, device_create
- threads: work_struct, workqueue_struct, create_workqueue, kthread_create, kernel_thread, current, thread_info, do_fork
- Virtual memory: virtually continuous memory, find_vma_prepare, vmalloc, vmlist, vm_struct, virt_to_page
- Virtual File System: file, vfs_read, vfs_write, inode, vfs_create, inode_operations, file_operations, file_system_type, get_sb
- protocol families: sock_create, socket, inet_family_ops, inet_create, unix_family_ops, proto_ops, inet_dgram_ops, inet_stream_ops
- security: security/, linux/security.h, may_open, security_socket_create, security_inode_create, printk

bridges
- class_device, class_device_create, device_driver, probe, driver_register
- Synchronization: wake_up, mutex_lock, completion, add_timer, mutex, down, semaphore, msleep, spin_lock
- Memory Mapping: mm/mmap.c, do_mmap_pgoff, vma_link, mm_struct, vm_area_struct
- Page Cache: do_sync, address_space, pdflush
- Swap: kswapd, do_swap_page, wakeup_kswapd
- ramfs_fs_type, super_block
- networking storage: nfs_file_operations, smb_fs_type, cifs_file_ops, iscsi_tcp_transport
- security_ops, selinux_ops
- © 2007 Constantine Shulyupin www.LinuxDriver.co.il/kernel_map

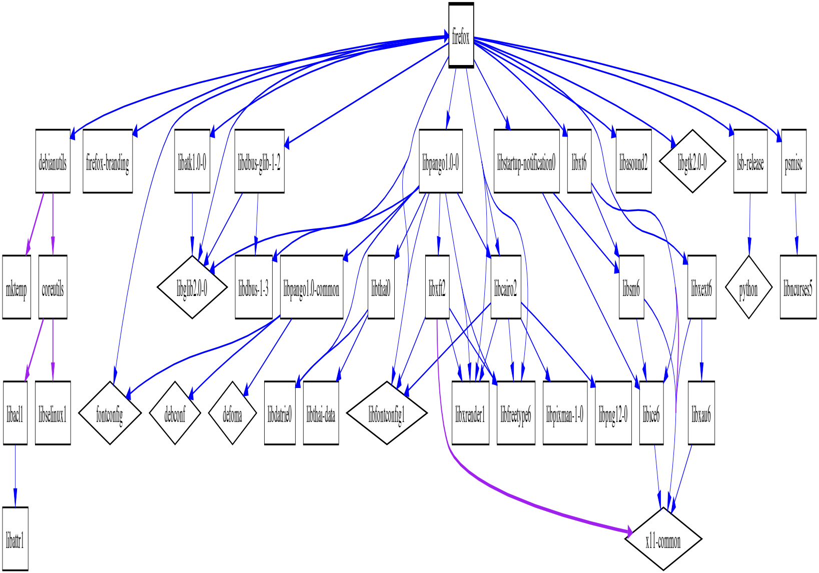
functional
- system run: init/, kernel/, kernel_restart, kernel_power_off, load_module, module, init/main.c, start_kernel, do_initcalls, run_init_process
- Scheduler: kernel/sched.c, task_struct, schedule_timeout, schedule, setup_timer, process_timeout, activate_task, rq, context_switch
- logical memory: mm/slab.c, /proc/slabinfo, physically mapped memory, kfree, kmalloc, kmem_cache_alloc, kmem_cache, slab
- Logical File Systems: ext3_file_operations, ext3_get_sb
- protocols: proto, /proc/net/protocols, udp_prot, tcp_prot, ip_rcv, alloc_skb, ip_queue_xmit, ip_forward, sk_buff
- HI subsystems: tty, log_buf, mousedev_handler, oss, alsa

devices control
- drivers/, generic HW access: pci_driver, usb_driver, pci_register_driver, pci_request_regions, usb_hcd_giveback_urb, request_region, request_mem_region, usb_submit_urb, ioremap, usb_hcd
- interrupt context: timer_list, tasklet_struct, jiffies ++, setup_irq, timer_interrupt, do_softirq, do_IRQ, irq_desc
- Page Allocator: mm/page_alloc.c, /proc/buddyinfo, get_free_pages, zone, alloc_pages, page, get_page_from_freelist, try_to_free_pages
- Block devices: block/, gendisk, block_device_operations, request_queue, init_scsi, scsi_device, scsi_driver, sd_fops, usb_storage_driver
- virtual network device: net_device, netif_rx, dev_queue_xmit, alloc_etherdev, alloc_ieee80211, ether_setup, ieee80211_rx, ieee80211_xmit
- abstract devices and HID class drivers: drivers/input/, drivers/media/, sound/, console, kbd, fb_ops, mousedev, video_device

hardware interfaces
- devices access and bus drivers: include/asm/, arch/i386/, ehci_irq, pci_read, writew, pci_write, readw, ehci_urb_enqueue, outw, inw, usb_hcd_irq
- CPU specific: arch/i386/kernel/, start_thread, /proc/interrupts, interrupt, atomic_t, system_call, switch_to, show_regs, trap_init, pt_regs, cli, sti
- physical memory operations: arch/i386/mm/, die, do_page_fault
- disk controllers drivers: Scsi_Host, libata, ahci_pci_driver, idedisk_ops, aic94xx_init, ide_intr, ide_do_request
- network devices drivers: drivers/net/, e100_open, ipw2100_open, zd1201_net_open, rtl8139_open
- HI peripherals device drivers: vga_con, atkbd_drv, psmouse, i8042_driver, ac97_driver, drivers/video/

electronics
- I/O: I/O mem, I/O ports, PCI controller, USB controller, DMA
- CPU: registers, APIC, interrupt controller
- memory: RAM, MMU
- disk controllers: SCSI, IDE, SATA
- network controllers: Ethernet, WiFi
- user peripherals: keyboard, mouse, cam, audio, graphics card

Firefox Package Dependencies



The Problem

- Software stack growing in size
  - More functionality
  - More hardware supported
  - Diverse application needs
  - Complex and numerous device drivers
  - Less robust to change
- Adverse security implications
  - More code to analyze
  - Increasingly complex interactions
  - Same time-to-market

    →

    - Reduced relative code coverage
    - Greater diversity of exploits
    - Wider attack surface
    - Larger consequences

Static Previrtualization: A Principled Approach

- Shrink the software by specializing it to a specific deployment
- Target full software stack and address all code growth.
- Specialize software stack to
    - Specific applications
    - Specific hardware/deployment configuration
        - Kernel modules trimmed
        - Device drivers trimmed
        - Libraries trimmed
- Applicable to
    - Specialized application servers
    - Custom libraries

Comparison

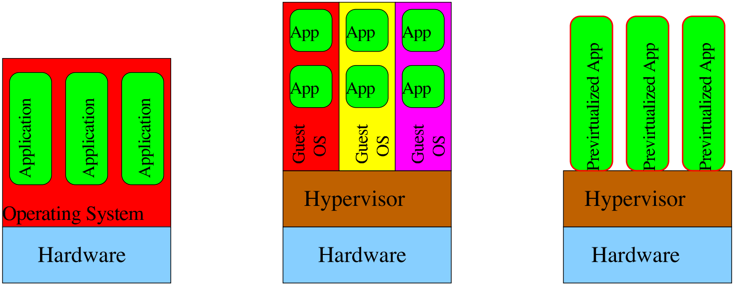


Figure 1: Single OS systems, Virtualized Systems, and Previrtualized Systems

Partial Evaluation

- A practical application of Kleene's s-m-n theorem for optimizating code by specializing it to specific arguments

-
  ```
  int atoi_strcat(char *str1, char *str2){
    return atoi(strcat(str1, str2));
  }
    ...
    char *pad = "000";
    i = atoi_strcat(mystr, pad);
  ```

- The `atoi_strcat` invocation can be replaced by

  ```
    atoi_strcat_1000(argv[1])
  ```

  where

  ```
  int atoi_strcat_1000(char *str1){
    return 1000 * atoi(str1);
  }
  ```

Partial Evaluation

- Fixes specific parameters
    - Interpreter + program → object code
    - Specializer + interpreter → compiler
    - Specializer + specializer → compiler-compiler
- Partial evaluation of common system calls
    - Improved performance (*McNamee, 2001*)
- Customized runtimes via partial evaluation
    - Application developer / user knows needs
    - Operating system designer does not (*Howell, 1998*)

Static Previrtualization

- Application system calls partially evaluated
- Resulting code is
  - Compact
  - Efficient
  - Portable
  - Isolatable
  - Less vulnerable to attack
  - More amenable to static analysis
- LiveCD or virtual appliance, but without redundant software

Project Background

- Tried package minimization: Reduces footprint but granularity is too coarse
- Evaluated existing partial evaluation technology:
  - CMix-II (Henning Makholm)
  - Tempo (Charles Consel)
- The technology is impressive but not current with language/architecture changes (C99 or 64-bit machines)
- More recently, we switched to LLVM
- We have developed a previrtualization/monitoring toolchain called **Occam** based on LLVM
- It has been applied to web servers and PHP/MiniBlog

Reducing Functionality with **Occam**

- **Program**: `thttpd`
- **Size**: 11,322 lines
- **Problems**
  - Uses potentially dangerous functions like listen , connect, etc.
  - Reads configuration data from the command line.
- **Solutions**
  - Limit the ways that dangerous functions can be called.
  - Compile configuration data into the program.

Partial Evaluation of LLVM Bitcode

- Low Level Virtual Machine (LLVM) is a typed, machine-independent intermediate format (IF, called bitcode) due to Adve and Lattner.
- The IF uses static single assignment on typed registers.
- Many languages have front-ends to generate LLVM, e.g., Ada, C, C++, Objective C, Haskell.
- Analyzers and code-generators can be driven from LLVM.
- `glibc` could not be directly converted to bitcode, but `uClibc` was adequate for this purpose.
- Simple forms of partial evaluation on LLVM have been explored:
  - Fujita uses cloning and LLVM's optimizer
  - Smowton and Hand inline file data

Workflow

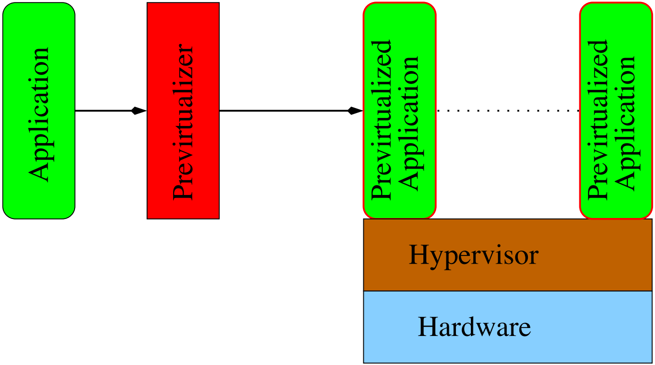
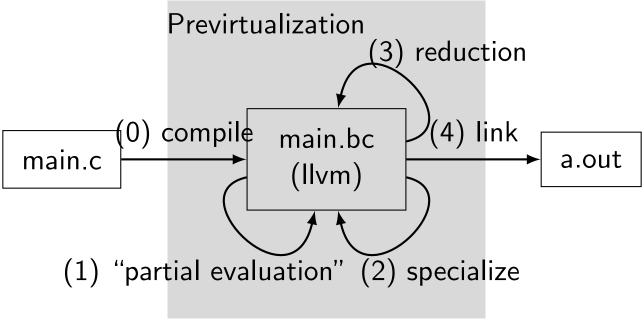


Figure 2: Previrtualizing Applications

Outline of Talk

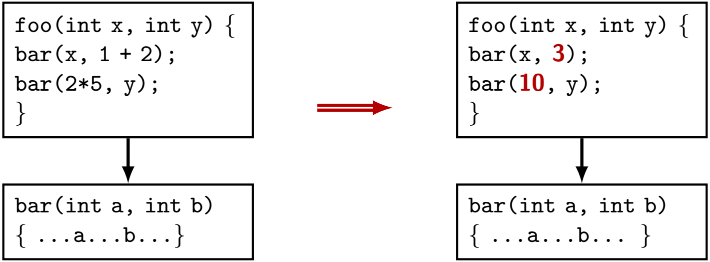
1. Reduce the “functionality” of a system
    - `nweb` is a simple webserver.
    - It doesn’t need to be able to listen on arbitrary ports.
    - Make configuration options static.
2. Overcome static analysis
    - Miniblog should never send email, so that functionality should not be in the system.
    - We need to cut it out, since mail is in the PHP standard library (compiled into the interpreter!).
3. Monitor systems and enforce dynamic policies
    - Log function calls as the program runs.
    - Check security properties.

Module Previrtualization Overview



Partial Evaluation

- Simplify the program as much as possible, want to expose constants.



```
foo(int x, int y) {
bar(x, 1 + 2);
bar(2*5, y);
}
```

```
bar(int a, int b)
{ ...a...b...}
```

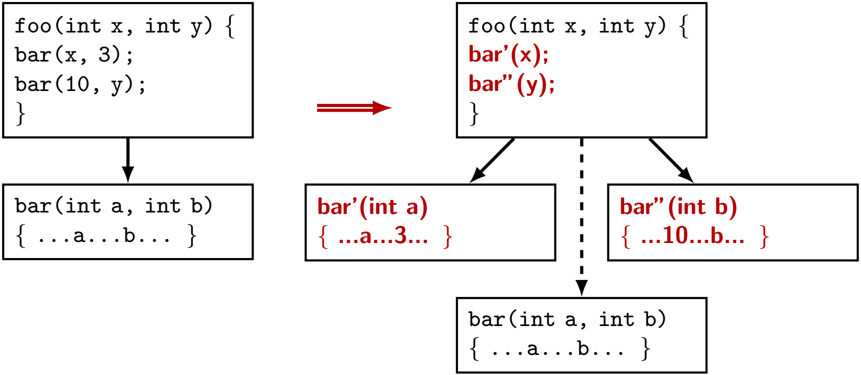
```
foo(int x, int y) {
bar(x, 3);
bar(10, y);
}
```

```
bar(int a, int b)
{ ...a...b... }
```

- Use LLVM's -O3.

Specialization

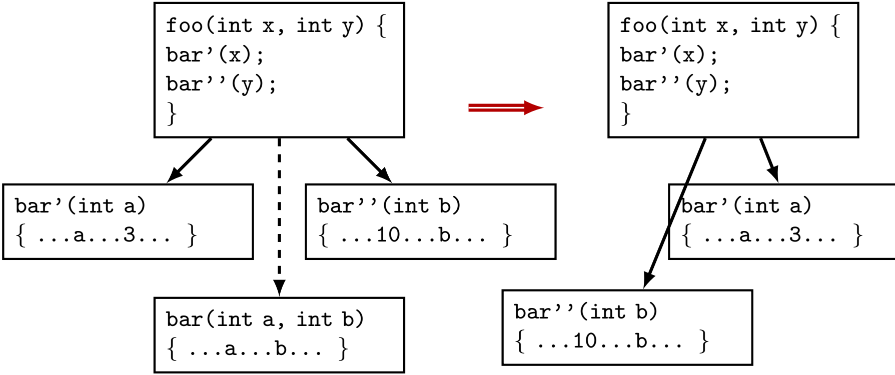
- Specialize functions when they take constant arguments.



- Clone functions and inline constants in a custom LLVM pass.

Reduction

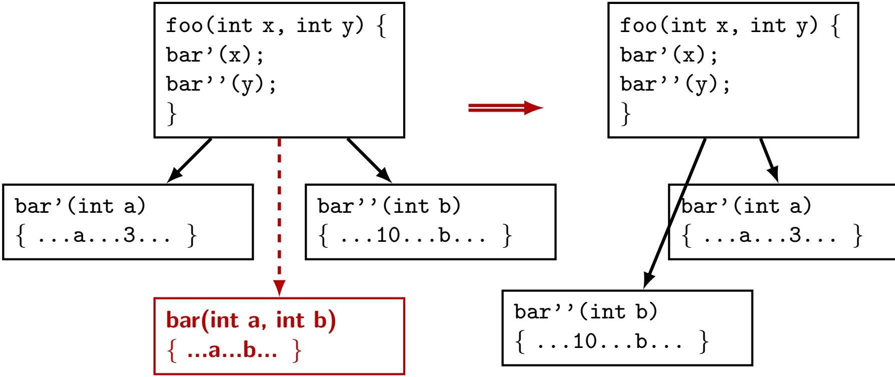
- Eliminate unused code.



- LLVM dead-code/global elimination pass.

Reduction

- Eliminate unused code.



```
foo(int x, int y) {
bar'(x);
bar''(y);
}
```

```
bar'(int a)
{ ...a...3... }
```

```
bar''(int b)
{ ...10...b... }
```

bar(int a, int b)
{ ...a...b... }

```
foo(int x, int y) {
bar'(x);
bar''(y);
}
```

```
bar'(int a)
{ ...a...3... }
```

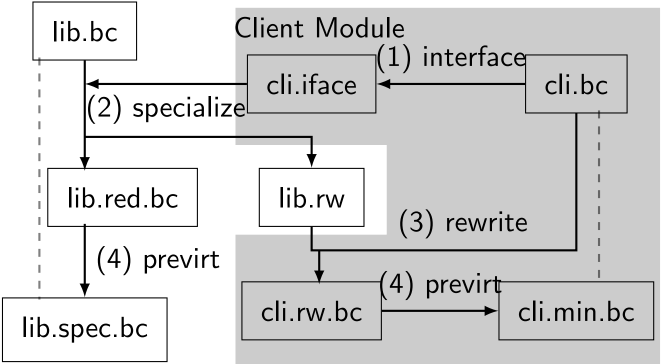
```
bar''(int b)
{ ...10...b... }
```

- LLVM dead-code/global elimination pass.

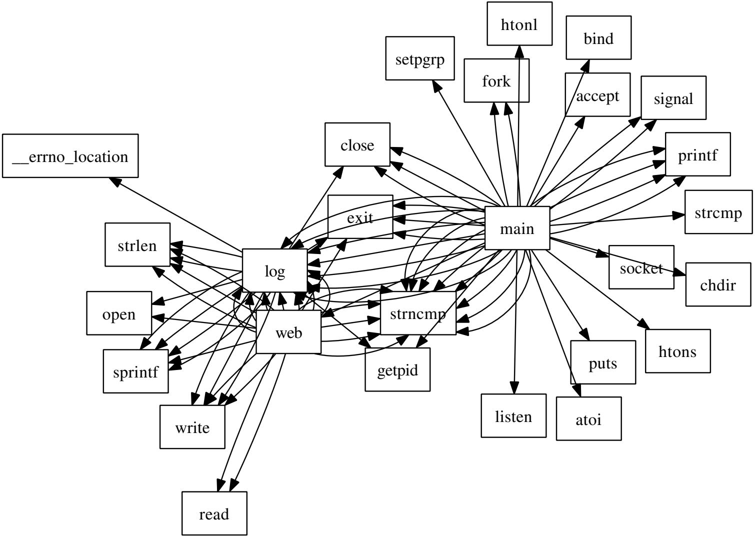
thttpd Previrtualization

1. Specialization of command line parameters
2. Partial evaluation by optimization
3. Aggressive specialization of dangerous functions
4. Dead-code elimination
5. Goto 2!

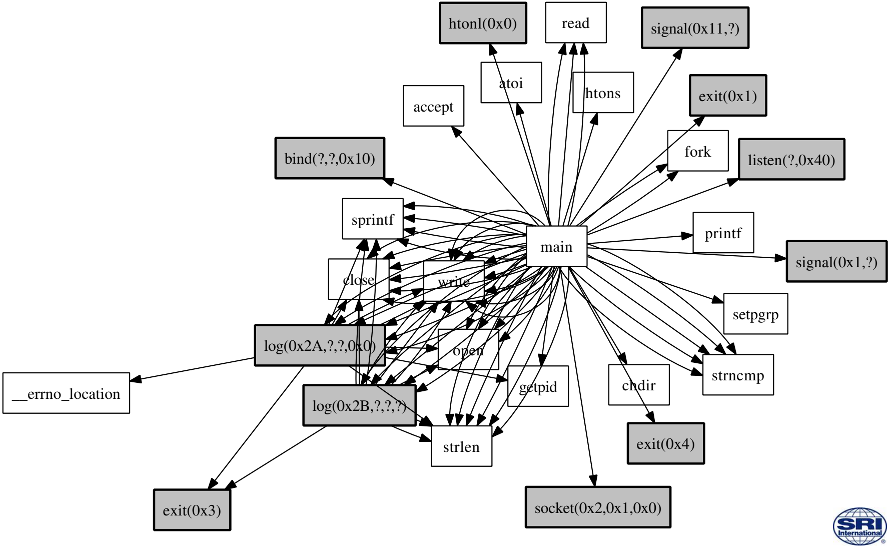
Cross-module Previrtualization



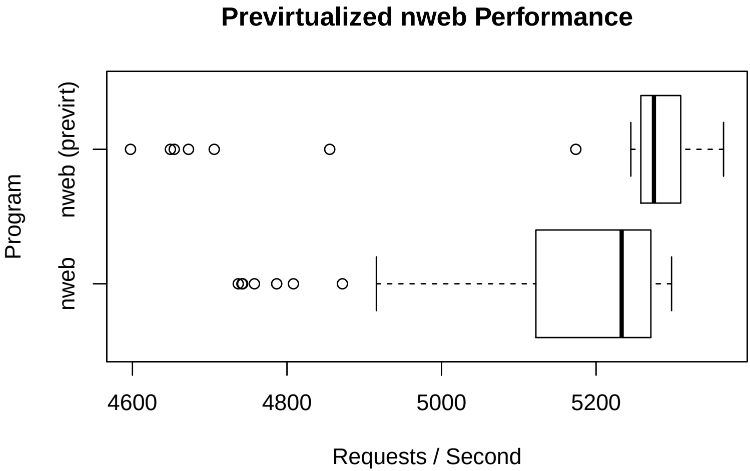
Callgraphs: Before



Callgraphs: Before & After

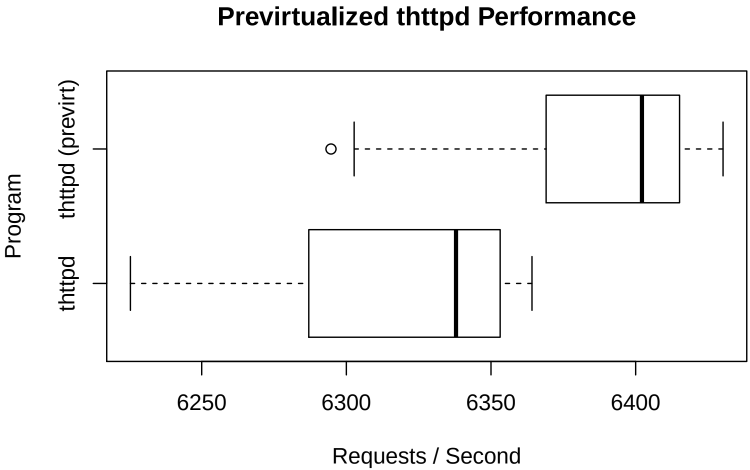


Performance of Previrtualized Software



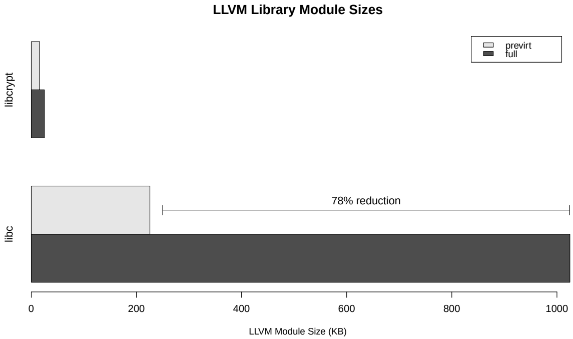
nweb performance before previrtualization.

Performance of Previrtualized Software (`thttpd`)

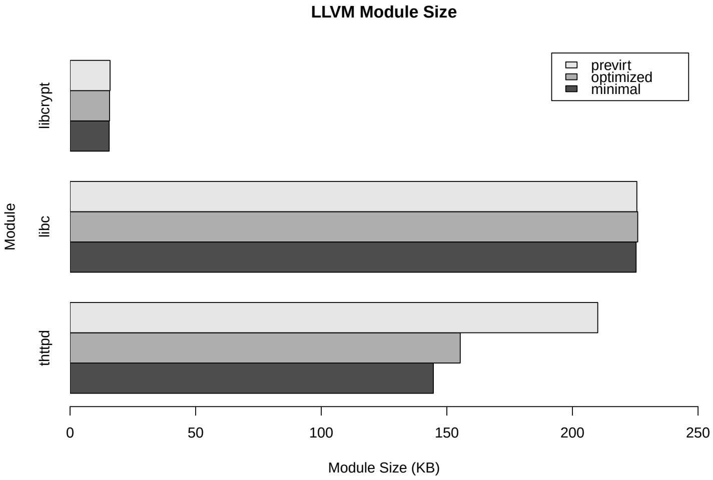


`thttpd` performance after previrtualization.

Size of Previrtualized Software



Size of Previrtualized Software



Limits of Compile-time Analysis

- We can't automatically determine reachable code in all cases
  - Cross-language or cross-binary calls
  - Indirect function calls & static approximations (PHP problem)
  - Function pointers and binary compatibility
- **PHP built-in functions are implemented as a large function table.**
  - Static analysis has to say that all of these are reachable.
  - Lets the bad in with the good.

PHP Snippet

```
const zend_function_entry basic_functions[] =
{ ... PHP_FE(system, arginfo_system) , ... };
PHP_FUNCTION(system)
{ php_exec_ex(INTERNAL_FUNCTION_PARAM_PASSTHRU, 1); }
```

Solution

1. “Statically analyze” the PHP code and determine the functions that it will call.
   - For relatively static applications this can be done with a `grep`-like static analysis.
   - Miniblog requires about 46 PHP functions out of the 1028 functions that a minimal PHP install would have.
2. Implement a transformation that will replace these unused functions with a simple exit(1).
   - Previrtualize the result to remove all the unnecessary code.

Specifying Rewrites

- We can specify subs the same way that we refer to specializations.

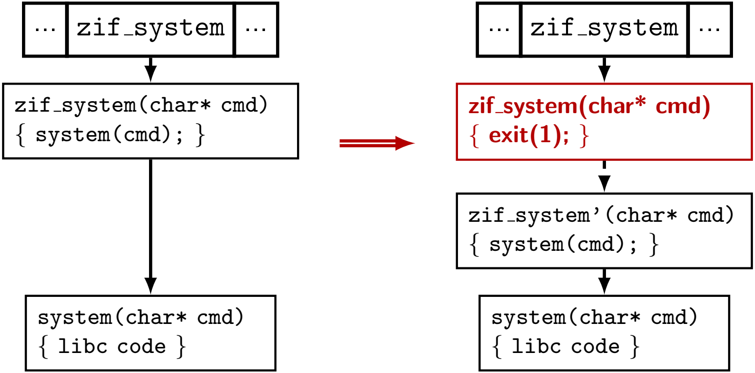
Remove `system` Function

```
zif_system(?) => fail
```

- **fail** is a keyword meaning call `exit(1)`.
- Question marks specify wildcard arguments; here we stub *all* calls to `zif_system`.
  - Also support integer constants, so we can reject some calls but not others.

Rewriting Code

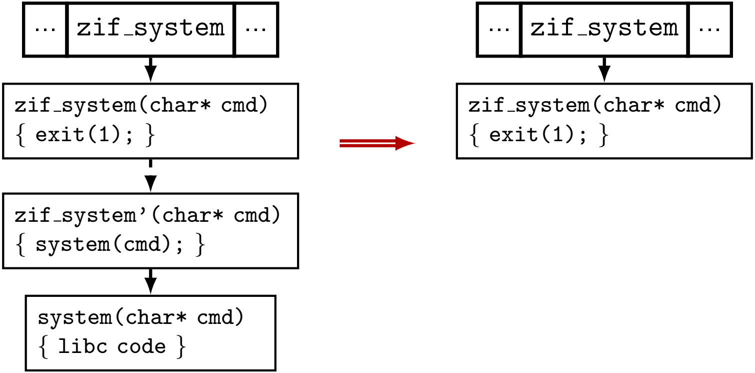
- Small transformation pass to replace function bodies.



- Implemented as a custom LLVM transformation pass.

Reusing the Previtualization Hammer

- Remove dead code using previrtualization.



```
··· zif_system ···

zif_system(char* cmd)
{ exit(1); }

zif_system'(char* cmd)
{ system(cmd); }

system(char* cmd)
{ libc code }
```

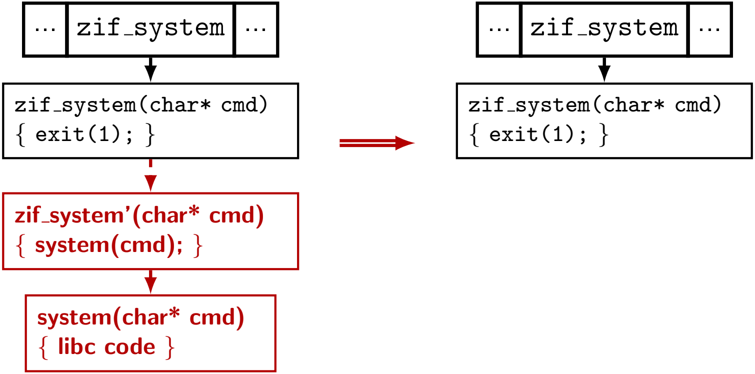
```
··· zif_system ···

zif_system(char* cmd)
{ exit(1); }
```

- Reduce to an already solved problem!

Reusing the Previrtualization Hammer

- Remove dead code using previrtualization.



- Reduce to an already solved problem!

PHP/Miniblog Previrtualization

- Remove dangerous PHP functions:
  - system
  - mail
  - etc.
- Previrtualization removes unused dependencies.

Approach

- Extend the enforcement mechanism.
- An implementation of *Aspect-Oriented Programming* for LLVM.
    - Monitor when execution enters/exits a function.
    - Support access to function arguments and return values.
    - Support conditional monitoring.
        - Allowing exit (1) on certain parameters.
    - Monitored binaries can be run *without* monitors.

Does Previrtualization Increase Security?

- Many attacks exploit buffer overflow and format string vulnerabilities to inject code or invoke existing functionality.
- Protections like *StackGuard* have a runtime cost.
- Attacks like `return-to-libc` and return-oriented programming (ROP) can be defeated by address space layout randomization (ASLR) on 64-bit machines.
- Even through previrtualization introduces more potential attack sequences for ROP, this is dwarfed by the entropy introduced by 64 bits.
- The bigger gain is that many potential vulnerabilities are pruned by previrtualization.
- Specifically, vulnerabilities in start-up code are unavailable in the previrtualized application.

Related Work

- Massalin's *Synthesis* kernel used partial evaluation for a form of run-time code generation for efficiency.
- McNamee, *et al.* optimized frequently used system calls with PE.
- Fujita exploited the LLVM optimizer for intra-module partial evaluation
- Smowton and Hand used this technique for inlining file data.
- Turnkey Linux distributes coarsely pruned appliances for several applications, including a JeOS (Just enough Operating System) stack.

Future Work

- Improve the previrtualization toolchain
- Deeper partial evaluation
- Kernel previrtualization: Compiled FreeBSD with `clang`
- Inter-application previrtualization
- Adding security checks and monitoring during previrtualization
- VM previrtualization
- Other platforms: Android
- Previrtualization as a service.

Conclusions

- **Occam** is a tool for *previrtualization*
  - Program specialization to reduce *functionality*.
  - Partial evaluation through optimization.
  - Works well for generic platforms, e.g., languages with large libraries like PHP.
- Monitoring program execution
  - Monitors can be placed around functions and modify both inputs and outputs.
  - Monitors can be arbitrary C++ code and can maintain state between calls.
  - Aspect-oriented programming
- These techniques can be used for shrinking and wrapping the software stack for each deployment