Temporal System Call Specialization for Attack Surface Reduction

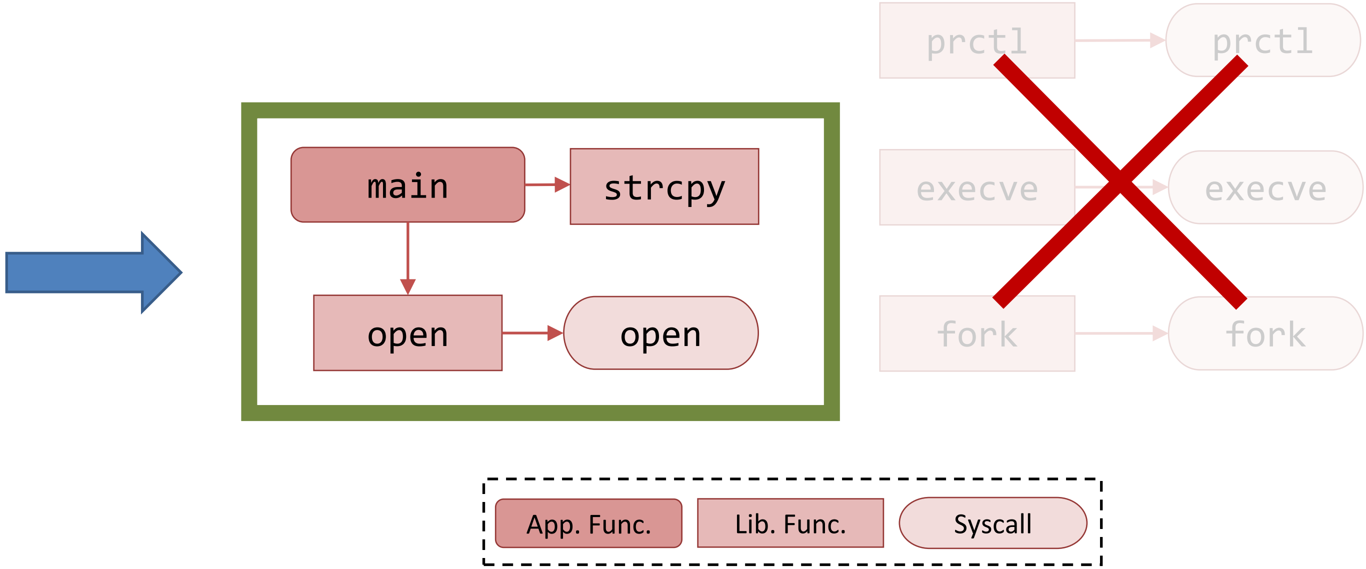
Seyedhamed Ghavamnia, Tapti Palit, Shachee Mishra, Michalis Polychronakis
{sghavamnia, tpalit, shmishra, mikepo}@cs.stonybrook.edu
Stony Brook University

Software Debloating and Specialization

- Applications typically include code they don't use and have access to features they don't need
  - Some modules/plugins are not needed by a given configuration
  - Some library functions are not imported at all
  - Some system calls are never used

```
#include <stdlib.h>
#include <fcntl.h>

int main(int argc, char** argv){
   char dest[1024];
   if ( argc == 2 ){
       strcpy(dest, argv[1]);
       int fd = open(dest,O_CREAT);
   }
   return 0;
}
```



Software Debloating and Specialization

- This "code bloat" has security implications
  - Unneeded code: more ROP gadgets for writing code reuse exploits
  - Unused (dangerous) system calls: exploit code can still invoke them to perform harmful operations (e.g., `execve()`)
  - Unused system calls: entry points for exploiting kernel vulnerabilities that can lead to privilege escalation
- **Our focus: reduce the attack surface by disabling system calls**
  - Break exploit payloads (shellcode, ROP)
  - Neutralize kernel vulnerabilities associated with certain system calls

Existing Work: Library Debloating

- Applications typically use only a fraction of library functions
- *Library debloating:* remove non-imported functions from memory

  [Mulliner and Neugschwandtner ’15] [Quach et al. ’18] [Agadakos et al. ’19] [Porter et al. ’20]
  - Generate the call graph of imported shared libraries
  - Identify library function dependencies
- Caveat: the entire lifetime of the program is considered
  - If a function/system call is used *even only once*, it cannot be disabled

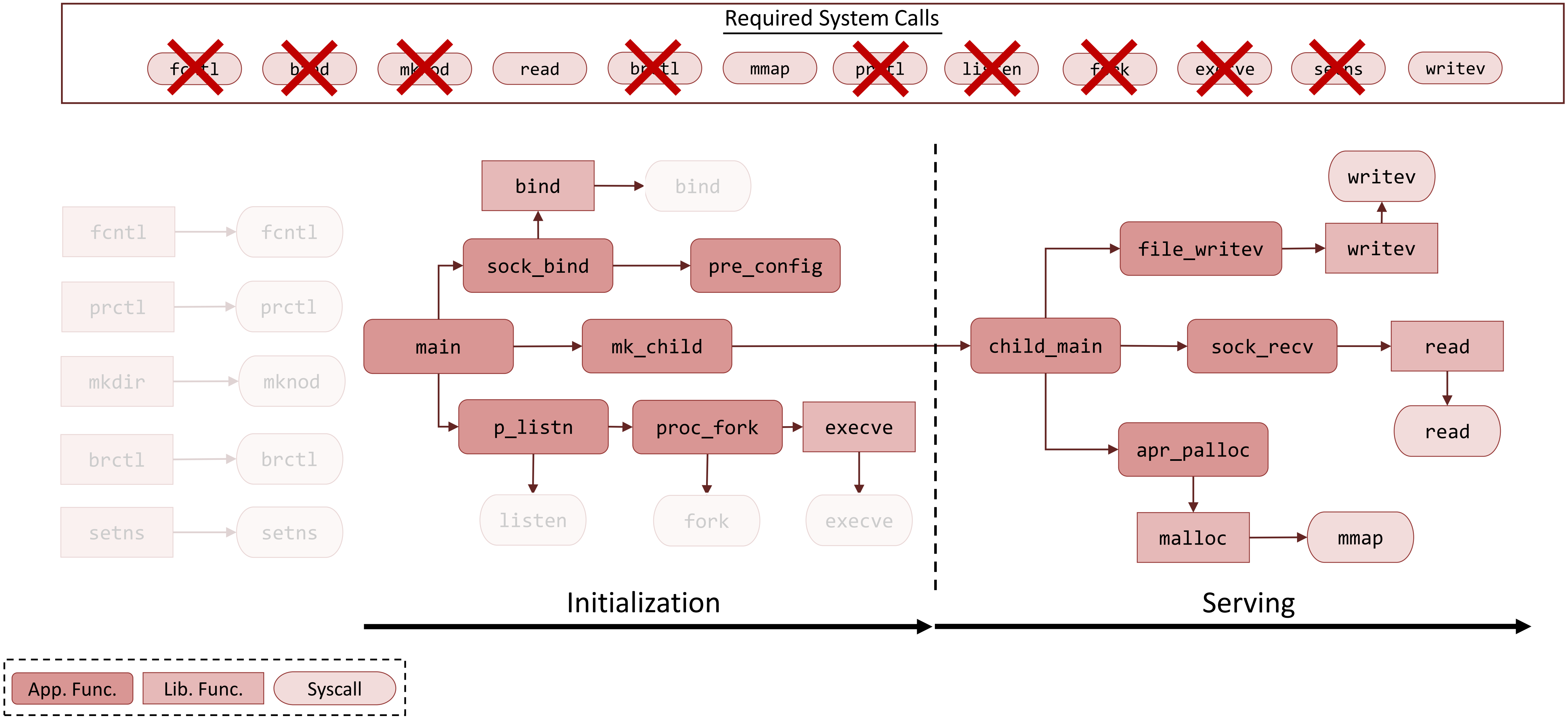
Can we disable ***more*** system calls by differentiating between a process’ different ***phases of execution***?

Motivation

- Server applications typically perform ***initialization*** operations at the beginning of their execution
  - Read configuration files
  - Fork worker processes
  - Execute other programs
  - Create files and set their permissions
- Afterwards, they enter their main ***serving*** phase
  - Handle client requests
  - Establish connections

  …

Example: Apache Web Server



Temporal System Call Specialization

- Disable additional system calls that are needed only during the *initialization* phase, after entering the *serving* phase

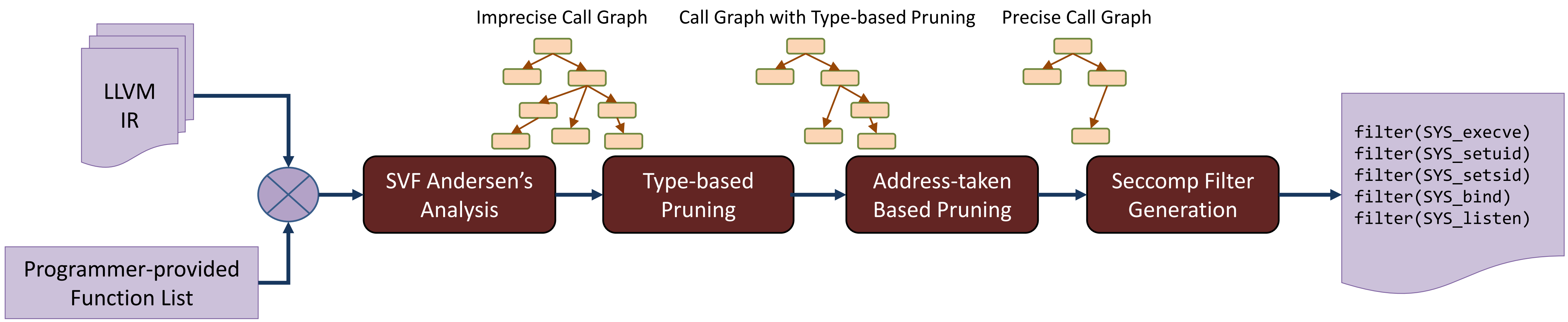
- Disables **51%** more security-critical system calls, breaking **218** more shellcodes and ROP payloads
- Mitigates **13** more Linux kernel CVEs

- Examples
  - Apache Httpd and Nginx invoke `execve` **only** during initialization
  - Lighttpd, Bind, and Redis invoke `chmod` **only** during initialization

Outline

- Introduction
- Generating the call graph
  - Pruning based on argument types
  - Pruning based on taken addresses
- Identifying the required system calls for each phase
- Enforcing system call filters after the initialization phase
- Experimental evaluation
- Conclusion

System Overview



Call Graph Generation

- A complete and sound call graph is required to identify unnecessary system calls
  - The use of function pointers necessitates points-to analysis
  - While sound, points-to analysis comes with severe over-approximation
- Over-approximation prevents the precise differentiation of the system call requirements between the two phases
  - No security benefit if both initialization and serving phases use the same set of system calls
- *Goal: improve precision without losing soundness*

Pruning based on Argument Types

- Match arguments passed to callsite with function argument types
- Consider only `struct` types (no primitives, no `void*`)
- Consider only non-variadic functions

Callsite

```
(*gic)(int r, void *d, apr_wait_t s)
```

Target Functions

```
piped_log_mnt (int p, void *m, apr_wait_t a)
```

```
event_run (apr_pool * p, apr_pool * pl, serv_rec * s)
```

Pruning based on Argument Types

- Match arguments passed to callsite with function argument types
- Consider only `struct` types (no primitives, no `void*`)
- Consider only non-variadic functions

Callsite

```
(*gic)(int r, void *d, apr_wait_t s)
```

Target Functions

```
piped_log_mnt (int p, void *m, apr_wait_t a)
```

```
event_run (apr_pool * p, apr_pool * pl, serv_rec * s)
```

Pruning based on Argument Types

- Match arguments passed to callsite with function argument types
- Consider only `struct` types (no primitives, no `void*`)
- Consider only non-variadic functions

Callsite

```
(*gic)(int r, void *d, apr_wait_t s)
```

Target Functions

```
piped_log_mnt (int p, void *m, apr_wait_t a)
```

```
event_run (apr_pool * p, apr_pool * pl, serv_rec * s)
```

Pruning based on Argument Types

- Match arguments passed to callsite with function argument types
- Consider only `struct` types (no primitives, no `void*`)
- Consider only non-variadic functions

Callsite

```
(*gic)(int r, void *d, apr_wait_t s)
```

Target Functions

```
piped_log_mnt (int p, void *m, apr_wait_t a)
```

```
event_run (apr_pool * p, apr_pool * pl, serv_rec * s)
```

Pruning based on Argument Types

- Match arguments passed to callsite with function argument types
- Consider only `struct` types (no primitives, no `void*`)
- Consider only non-variadic functions

Callsite

```
(*gic)(int r, void *d, apr_wait_t s)
```

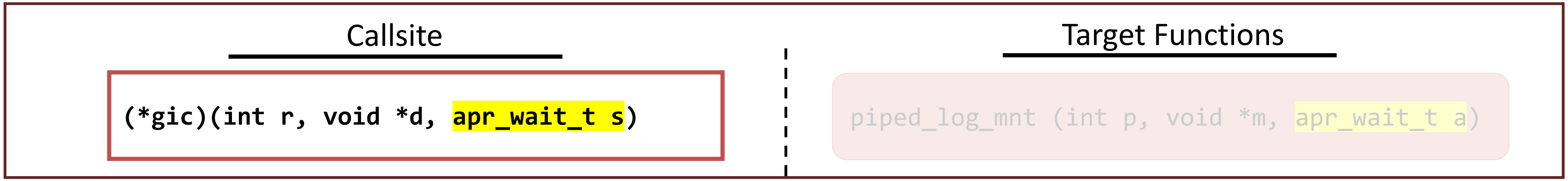
Target Functions

```
piped_log_mnt (int p, void *m, apr_wait_t a)
```

```
event_run (apr_pool * p, apr_pool * pl, serv_rec * s)
```

Pruning based on Taken Addresses

- Identify where a function address is being taken (global and local)
- Check if those locations (local) are accessible from `main()`
- Prune edges to functions that are:
  - Not accessed directly *and*
  - The location where the address is being taken is not accessible
- Example: address of `piped_log_mnt` is only taken in `start_module`
  - `start_module` is not accessible from `main`

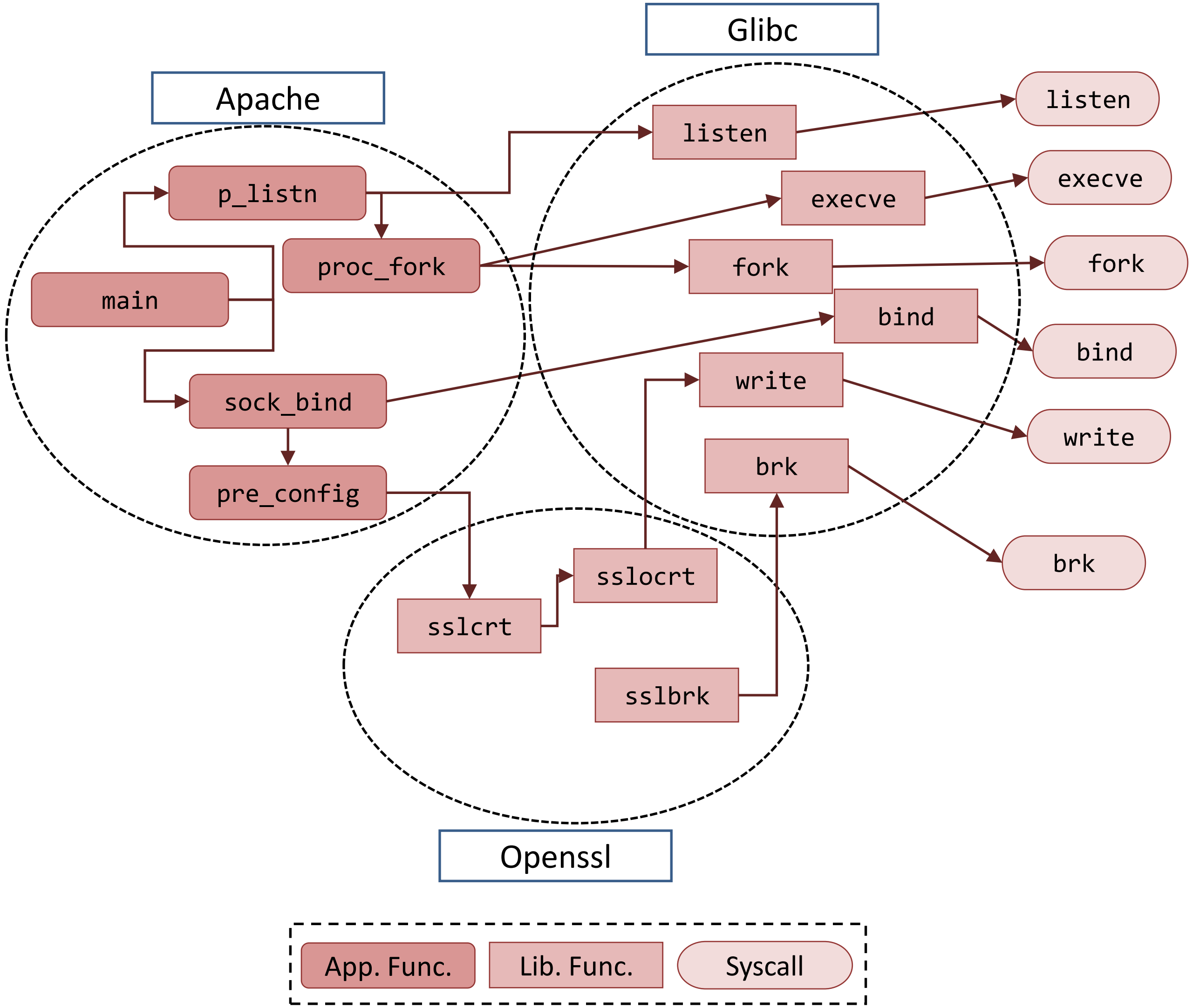


Outline

- Introduction
- Generating the call graph
  - Pruning based on argument types
  - Pruning based on taken addresses
- Identifying the required system calls for each phase
- Enforcing system call filters after the initialization phase
- Experimental evaluation
- Conclusion

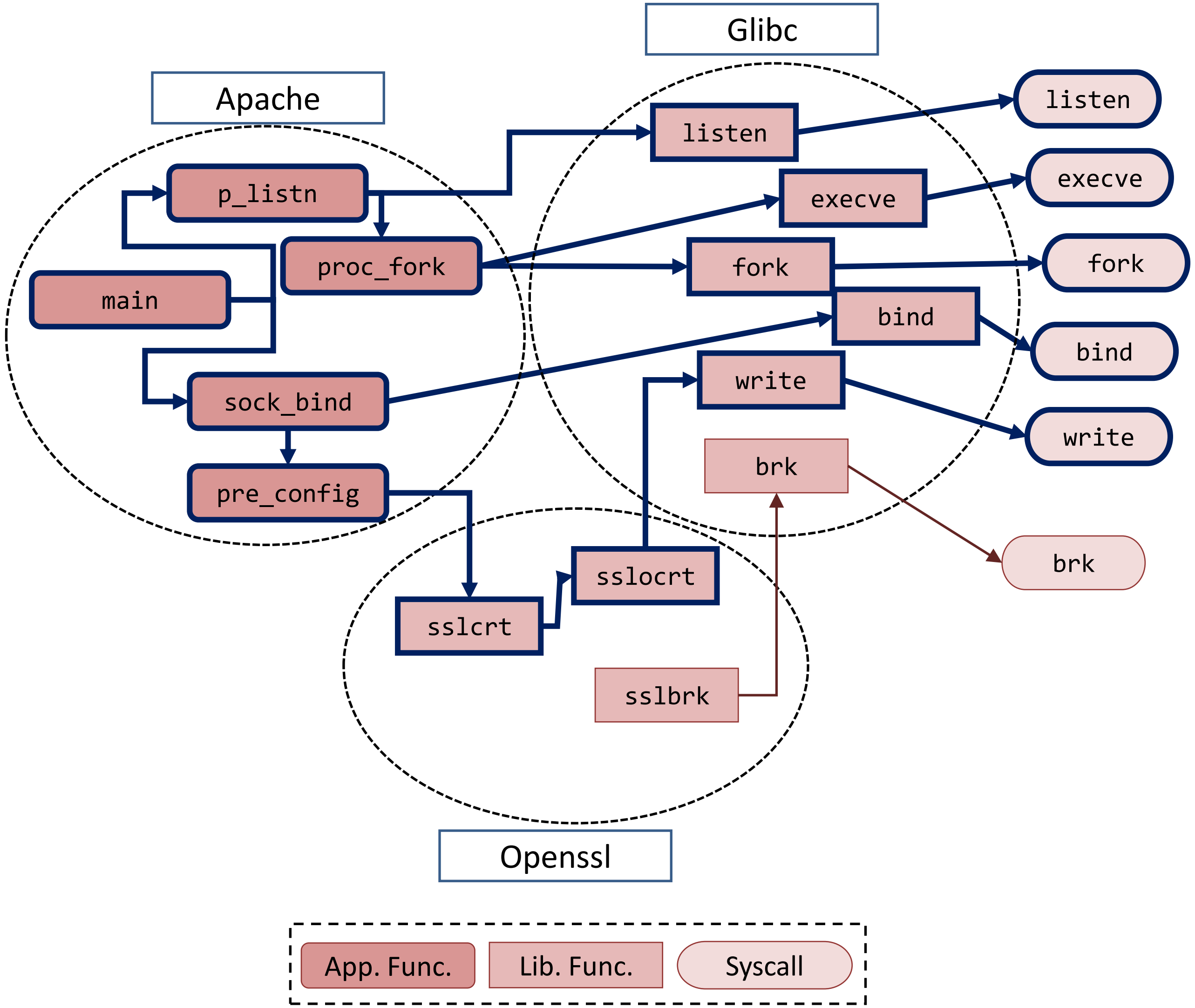
System Call Mapping

- Glibc call graph generation
  - Map all exported functions to the system calls they use
- Generate call graph for all libraries
  - Leaves are either system calls *or*
  - Functions from other libraries
- Combine all call graphs to create a complete graph



System Call Mapping

- Glibc call graph generation
  - Map all exported functions to the system calls they use
- Generate call graph for all libraries
  - Leaves are either system calls *or*
  - Functions from other libraries
- Combine all call graphs to create a complete graph

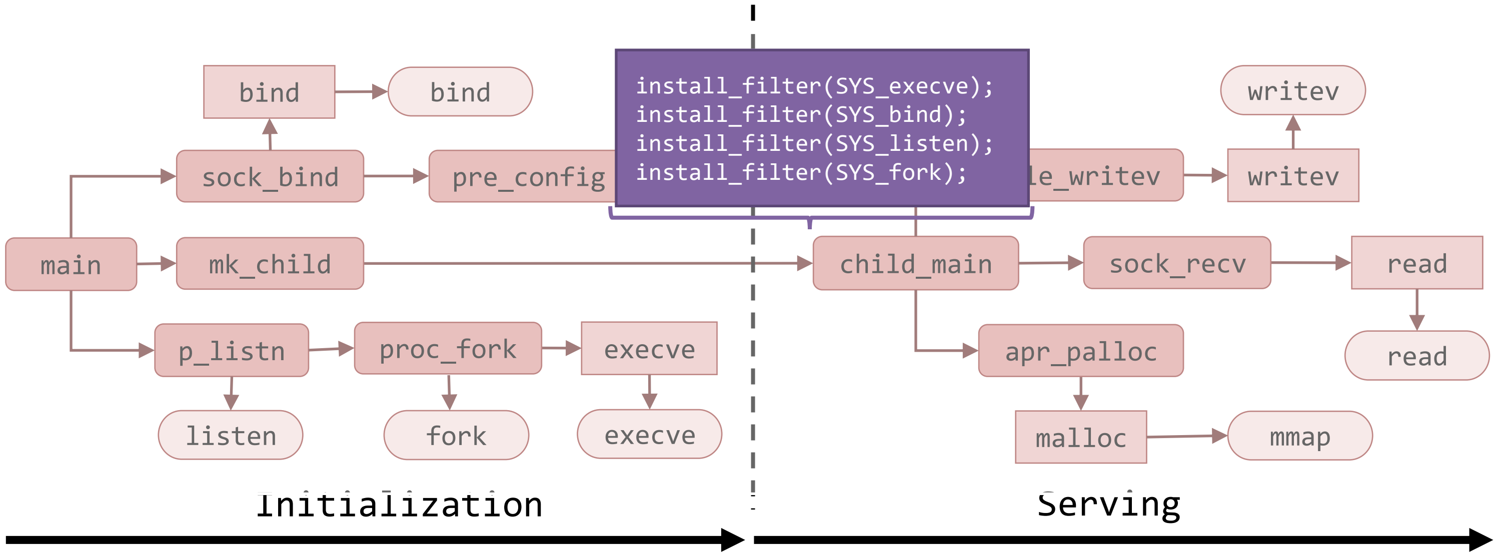


Outline

- Introduction
- Generating the call graph
  - Pruning based on argument types
  - Pruning based on taken addresses
- Identifying the required system calls for each phase
- Enforcing system call filters after the initialization phase
- Experimental evaluation
- Conclusion

System Call Filtering Enforcement

- Seccomp BPF
  - Standard Linux kernel facility for system call filtering
  - Filters installed by invoking the `seccomp` or `prctl` system call
  - In case of filter conflicts, the least privileged ones are considered
- Install more restrictive filters after entering the *serving phase*



Outline

- Introduction
- Generating the call graph
  - Pruning based on argument types
  - Pruning based on taken addresses
- Identifying the required system calls for each phase
- Enforcing system call filters after the initialization phase
- Experimental evaluation
- Conclusion

Lib. vs. Temporal Specialization: Retained System Calls

Number of system calls retained (out of 333 available) after applying library debloating and temporal specialization

Application	Library Debloating	Temporal Specialization	
		Initialization	Serving
Nginx	104	104	**97**
Apache Httpd	105	94	**79**
Lighttpd	95	95	**76**
Bind	127	99	**85**
Memcached	99	99	**84**
Redis	90	90	**82**

Security-critical System Calls Disabled

	Syscall	Nginx		Apache Httpd		Lighttpd		Bind		Memcached		Redis	
		Lib.	Temp.	Lib.	Temp.	Lib.	Temp.	Lib.	Temp.	Lib.	Temp.	Lib.	Temp.
Cmd Execution	clone	✘	✓	✘	✘*	✘	✘*	✘	✘*	✘	✘*	✘	✘*
	execveat	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	execve	✘	✓	✘	✓	✘	❖	✓	✓	✓	✓	✘	❖
	fork	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	ptrace	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Permissions	chmod	✘	✘	✓	✓	✘	✓	✘	✓	✓	✓	✘	✓
	mprotect	✘	✘	✘	✘	✘	✘	✘	✘	✘	✘	✘	✘
	setgid	✘	✘	✘	✓	✘	✓	✘	✓	✘	✓	✓	✓
	setreuid	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	setuid	✘	✘	✘	✓	✘	✓	✘	✓	✘	✓	✓	✓

✓: Syscall is removed ✘: Syscall is not removed

❖: Can be removed by applying configuration-driven specialization *: Can be removed by applying API specialization

Broken Shellcodes and ROP Payloads

- Collected 567 shellcodes and 17 ROP payloads
- Increased set of shellcodes to 1,726 by generating shellcode variations based on equivalent system calls
  - Example: `accept` and `accept4` are equivalent

Average number (%) of payloads broken by library and temporal specialization across applications

Payload Category	Count	Library Debloating	Temporal Specialization
Shellcode	1726	1103 (63%)	**1321 (76%)**
ROP Payloads	17	9 (52%)	**11.6 (68%)**

Neutralized Linux Kernel CVEs

Kernel CVEs mitigated by filtering unneeded system calls

CVE	System Call(s)	Description	Library	Temporal
CVE-2018-18281	`execve(at), remap`	Allows user to gain access to a physical page after it has been released	0	4
CVE-2016-3672	`execve(at)`	Allows user to bypass ASLR by disabling stack consumption resource limits	2	4
CVE-2015-3339	`execve(at)`	Race condition allows privilege escalation by executing program	2	4
CVE-2015-1593	`execve(at)`	Bug in stack randomization allows attackers to bypass ASLR by predicting top of stack	2	4
CVE-2014-9585	`execve(at)`	ASLR protection can be bypassed du to bug in choosing memory locations	2	4
CVE-2013-0914	`execve(at)`	Allows local user to bypass ASLR by executing a crafted application	2	4
CVE-2012-4530	`execve(at)`	Sensitive information from the kernel can be leaked via a crafted application	2	4
CVE-2012-3375	`epoll_ctl`	Denial of service can be caused due to improper checks in epoll operations	0	1
CVE-2011-1082	`epoll_(ctl, pwait, wait)`	Local user can cause denial of service due to improper checks in epoll data structures	0	1
CVE-2010-4346	`execve(at)`	Allows attacker to conduct NULL pointer dereference attack via a crafted application	2	4
CVE-2010-4243	`uselib, execve(at)`	Denial of service can be caused via a crafted exec system call	2	4
CVE-2010-3858	`execve(at)`	Denial of service can be caused due to bug in restricting stack memory consumption	2	4
CVE-2008-3527	`execve(at)`	Allows a local user to escalate privileges or cause DoS due to improper boundary checks	2	4

Conclusion

- Temporal specialization *removes* security-critical system calls by differentiating between the execution phases of a process
  - Proposed two novel call graph pruning techniques
- Filters **51%** more *security-critical system calls* than previous library debloating techniques
- Mitigates **13** more *Linux kernel CVEs* compared to previous library debloating techniques

Source code: https://github.com/shamedgh/temporal-specialization

Seyedhamed Ghavamnia, Tapti Palit, Shachee Mishra, Michalis Polychronakis
{sghavamnia, tpalit, shmishra, mikepo}@cs.stonybrook.edu
Stony Brook University