SysPart: Automated Temporal System Call Filtering for Binaries

Vidya Lakshmi Rajagopalan
Stevens Institute of Technology
Hoboken, NJ, USA

Konstantinos Kleftogiorgos
Stevens Institute of Technology
Hoboken, NJ, USA

Enes Göktaş
Stevens Institute of Technology
Hoboken, NJ, USA

Jun Xu
University of Utah
Salt Lake City,UT, USA

Georgios Portokalidis
Stevens Institute of Technology
Hoboken, NJ, USA
IMDEA Software Institute
Madrid, Spain

Abstract

Restricting the system calls available to applications reduces the attack surface of the kernel and limits the functionality available to compromised applications. Recent approaches automatically identify the system calls required by programs to block unneeded ones. For servers, they even consider different phases of execution to tighten restrictions after initialization completes. However, they require access to the source code for applications and libraries, depend on users identifying when the server transitions from initialization to serving clients, or do not account for dynamically-loaded libraries. This paper introduces SysPart, a semi-automatic system-call filtering system designed for binary-only server programs that addresses the above limitations. Using a novel algorithm that combines static and dynamic analysis, SysPart identifies the serving phases of all working threads of a server. Static analysis is used to compute the system calls required during the various serving phases in a sound manner, and dynamic observations are only used to complement static resolution of dynamically-loaded libraries when necessary. We evaluated SysPart using six popular servers on x86-64 Linux to demonstrate its effectiveness in automatically identifying serving phases, generating accurate system-call filters, and mitigating attacks. Our results show that SysPart outperforms prior binary-only approaches and performs comparably to source-code approaches.

CCS Concepts

• **Security and privacy** → **Systems security**.

Keywords

System-call filtering, temporal, binary analysis, attack-surface reduction, exploit mitigation.

ACM Reference Format:
Vidya Lakshmi Rajagopalan, Konstantinos Kleftogiorgos, Enes Göktaş, Jun Xu, and Georgios Portokalidis. 2023. SysPart: Automated Temporal System Call Filtering for Binaries. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS '23), November 26–30, 2023, Copenhagen, Denmark.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3576915.3623207

1 Introduction

Applications interact with operating systems (OSs) through system calls (syscalls). Over time, the number of syscalls has increased to accommodate the growing size and complexity of application software. Indicatively, the latest Linux kernel (v6.3.1) provides 451 syscalls in x86-64 architectures, compared to just 347 in the previous version (v5.5). Astoundingly, more than 100 new syscalls have been added since then. However, this increase is not without risks. Unprivileged applications, whether malicious or compromised [11], can exploit vulnerabilities in system-call code, which runs with higher privileges, to elevate their own privileges [28–30, 33, 45, 46]. System calls are also the only means by which malicious applications can execute dangerous actions, such as downloading and running malware, communicating over the network, and so on.

The Literature: To mitigate this issue, recent approaches attempt to limit the number of system calls available to applications. One such approach is sysfilter [13] (SF), which utilizes binary analysis to statically approximate the minimum set of system calls required by an entire binary application over its lifetime. A filter is then installed to block unnecessary calls at load time. In contrast, temporal system call specialization [20] (TSP) focuses on server applications, and partitions the application lifetime into initialization and serving phases. Static analysis at the compiler-level is employed to further restrict the system calls available during the serving phase. While TSP offers superior security, it has several limitations: ❶ TSP works only on source code and cannot be applied to applications with binary components; ❷ it depends on the manual identification of the serving phase; and ❸ it ignores dynamically-loaded libraries (DLL), potentially resulting in false positives when syscalls are needed for those libraries.

Our Approach: In this paper, we present SysPart, a binary-only, automatic system-call filtering system for server programs. We introduce two new techniques to overcome TSP's limitations.
► Similar to TSP, we divide a server's lifetime into two phases: initialization and serving. However, unlike TSP, SysPart automatically identifies the beginning of the serving phase. Our approach employs a novel algorithm that combines static and dynamic analysis with simple workloads, freeing the user from the burden of

manual identification. We observe that a server's serving phase typically employs a loop structure, and we identify a loop that *can only be entered once* and also *dominates execution time* as the *main loop*. SysPart applies this algorithm to each thread of execution independently, which enables automatic identification of all serving phases in applications that use different types of work threads (e.g., multiple services). This can mitigate attacks where a compromised serving thread uses another one as a confused deputy [24] to perform syscalls on its behalf.

► In order to accurately determine the system calls required during the serving phase, it is necessary to calculate the code partition that is accessible from the beginning of the main loop. To accomplish this, SysPart utilizes static analysis of the server binary and all of its library dependencies to construct safe, albeit conservative, versions of the program's function-call graph (FCG) and control-flow graph (CFG). These graphs are then used to compute the serving partition and the system calls made from it. Although reverse-engineering arbitrary binary software can be challenging, recent research [13, 56] has demonstrated that it can be accomplished in a sound manner for modern x86-64/ARM Linux binaries. Moreover, SysPart introduces a combination of value-flow analysis (VFA) and heuristics, among other static analyses, to increase the precision of the FCG and to resolve the names of libraries and functions loaded at run time (e.g., via `dlopen()` and `dlsym()`).

Evaluation: SysPart was implemented for x86-64 Linux, using static and dynamic analyses built as passes over the Egalito framework [56] and run-time tools over Intel's Pin framework [35], respectively. The FCG was further pruned using TypeArmor [55], and application binaries were rewritten using Egalito to install a Linux Seccomp-BPF filter just before the main loop. We evaluated SysPart using six popular servers (Nginx, Apache Httpd, Lighttpd, Bind, Memcached, and Redis), demonstrating that it outperforms sysfilter and performs closely to TSP, even without source code access. On average, SysPart only allows 8.33% more syscalls than TSP, and filters as many security-critical syscalls as TSP in 88.23% of cases. In fact, SysPart outperforms TSP in 2% of cases. SysPart is effective in thwarting exploit payloads, with a success rate of 53.83% - 78.5% and a success rate of 36.11% - 77.77% in blocking kernel vulnerabilities, for the ones tested, depending on the application. When considering libraries loaded at runtime, SysPart outperforms sysfilter in resolving instances of `dlopen()` and `dlsym()` by 58.33% and 18.75%, respectively, and resolves all related calls in Redis server entirely with static analysis. We also evaluated SysPart on Abyss Web Server [1], which is a closed-source server. Last, on average, the static analysis component of SysPart runs 80% faster than TSP.

Contributions: Our main contributions are as follows:

- We introduce SysPart, a system that can significantly limit the number of available syscalls during the serving phase of a server application, even without access to the application's source code.
- We design a novel algorithm for automatically detecting the main loop corresponding to the serving phases of a server application's threads and processes.
- We propose a novel static analysis that combines VFA, TypeArmor, and heuristics to refine the FCG and resolve dynamically-loaded libraries and functions.
- We evaluate SysPart using six server applications.
  - In terms of security benefits, SysPart surpasses the state-of-the-art binary-only solution sysfilter by tightly restricting the available syscalls during the serving phase of server applications. It also performs closely to TSP (the state-of-the-art source-code solution).
  - SysPart beats sysfilter in resolving DLL-related calls and, to the best of our knowledge, is the first to soundly handle a server application loading libraries at run time.
- We make SysPart available at https://github.com/vidyalakshmir/SysPartArtifact.git.

2 Threat Model

Binary server applications may contain vulnerabilities in the application or the used libraries, which can be exploited [54] during the serving of requests. These vulnerabilities can allow attackers to execute arbitrary code using techniques like code injection and code reuse [14, 53, 57], bypassing popular defenses such as stack-canaries [10], ASLR [44], DEP [7], and others [4] using known methods [21, 22, 52].

This work focuses on what a compromised process can do after this point. If the process is unprivileged, attackers frequently exploit a vulnerability reachable through a system call to elevate their privileges [28–30, 33, 45, 46]. Compromised programs, also use system calls to perform malicious actions, which are part of their payload, such as downloading and executing malicious software, attacking other servers over the network, and more. In multi-threaded/process applications, where the compromised thread is restricted and cannot perform a vulnerable system call, attackers may attempt to confuse another (unrestricted) thread or process into performing the syscall on their behalf [24]. This can be achieved, for example, by corrupting the data used by the other thread or sending malformed data to another process through inter-process communication (IPC) channels.

3 Background and Motivation

3.1 Filtering System Calls using Seccomp-BPF

Filtering the unused system calls of a process is one way to limit the attack surface of the kernel and what a process can do in adherence to the principle of least privilege. Seccomp-BPF [31] is a Linux kernel facility which allows filtering of system calls using Berkeley Packet filter rules. The process of filtering system calls using Seccomp-BPF is one-way, which means the filtered system calls cannot be re-allowed later as this would open a window for attackers to reactivate them once an application is compromised. Seccomp-BPF filters are manually defined which is an error-prone process as determining the system calls that are actually needed by an entire application is often complex.

3.2 Automating Filter Generation in Binaries

Previous works, such as sysfilter [13], aim to automate the process of installing Seccomp-BPF filters on binaries. This is achieved by disabling syscalls that are not required during the binary program's lifetime. Using static analysis, sysfilter first constructs the function call graph of the binary and extracts the syscalls reachable from

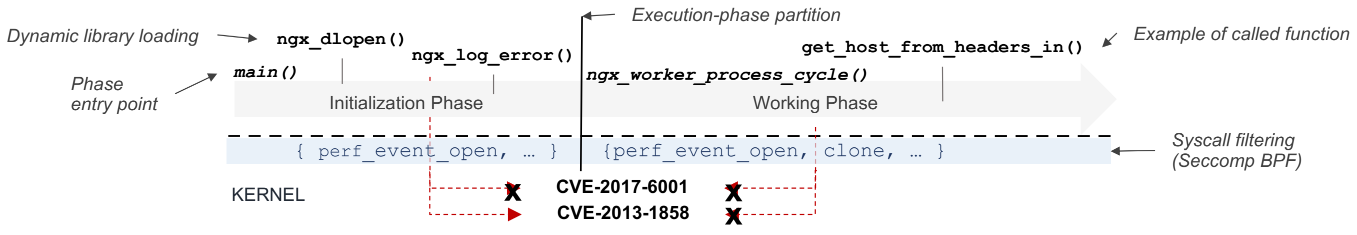

Dynamic library loading
ngx_dlopen()
Execution-phase partition
Example of called function
get_host_from_headers_in()
ngx_log_error()
Phase entry point
main()
ngx_worker_process_cycle()
Initialization Phase
Working Phase
{ perf_event_open, … }
{perf_event_open, clone, … }
Syscall filtering (Seccomp BPF)
KERNEL
CVE-2017-6001
CVE-2013-1858

Figure 1: *Motivating example*–System call filtering prevents compromised processes from performing system calls to interact with the OS or exploit kernel vulnerabilities. Installing more restrictive filters for the working phase of processes, like the Nginx web server's serving phase, further eliminates attacks and limits attacker capabilities. Existing approaches depend on manual defining the execution-phase partition, ignore dynamic library loading, and require source code and recompilation.

its code. It then injects code in the binary to install the generated Seccomp-BPF filters at load time.

3.3 Temporal System-Call Filtering

Compared to limiting syscalls for the whole execution, a more effective approach is temporal system-call filtering: *installing progressively more restrictive system-call filters as an application executes.* For example, most server programs work in two phases. ❶ The **initialization** phase performs tasks like initializing configuration parameters, forking service processes, spawning worker threads, etc.❷ Upon completion of the initialization phase, it enters the **serving** phase, which is usually designed as a loop—or the **main loop**— to continuously handle client requests. We define the point of transition from the initialization phase to the serving phase as the **transition point**. Many syscalls are no longer needed after this transition point, which can be filtered out to further constraint the application.

Running Example: In Figure 1, we show a running example on Nginx to demonstrate the security benefits of temporal system-call filtering. The serving phase of Nginx starts at the function `ngx_worker_process_cycle()`. By exploiting vulnerabilities like CVE-2013-2028 [39], adversaries can control the execution of the serving phase through ROP. Temporal syscall filtering can help prevent such adversaries from moving deeper by elevating the privileges of the controlled process. For instance, it can filter out the `clone` syscall during the serving phase, disallowing the adversaries to compromise the kernel via vulnerability CVE-2013-1858 [38]. In contrast, whole-binary syscall filtering like sysfilter cannot achieve this as `clone` is needed in the initialization phase.

Existing Solutions: TSP [20] is the state-of-the-art solution for temporal syscall filtering. TSP runs semi-automatic analyses on the source code to determine the serving phase and the required syscalls. It further mounts a syscall filter at the transition point to disable the unneeded syscalls. While insightful, the design of TSP has several limitations that can restrict its practicality.

► TSP requires source code. It cannot work on closed-source applications that are common in various domains: (i) Commercial servers like Aprelium Abyss Web Server, SAP NetWeaver, Oracle Database, etc., are only provided in the format of binary; (ii) Many closed-source desktop applications, such as Zoom and Skype, act as both clients and servers; (iii) Governments often acquire and run specialized proprietary software, including Linux servers, as attested by their interest in securing them [3]; (iv) Cloud users can run binary servers without sharing the source code for cloud vendors to provide protection. Even for open-source applications, gathering the correct version of source code and all its dependent libraries is cumbersome and can be infeasible (e.g., open-source servers can use proprietary binary libraries). In addition, inline assembly can arise in source code (like those in GNU libc) and fail source-code-based approaches.

► TSP requires users to identify the transition point manually. This is more complicated and time-consuming than it appears to be. For instance, Memcached offers a diverse set of services (client request handling, LRU maintenance, slab rebalancing, etc.) via different worker threads. These services use different serving code and have different transition points. It is important to identify all of them and place a syscall filter on each of them. Otherwise, confused deputy attacks are possible: a compromised but filter-restricted thread can hijack another unrestricted thread to invoke an unavailable syscall. The manual effort needed and the possibility of errors dramatically increase in such cases.

► TSP currently does not handle dynamically-loaded libraries, which can cause critical functionality issues. For instance, Httpd may use `mod_ssl.so` to enable SSL and TLS connections, requiring 14 additional system calls. Handling dynamically-loaded libraries presents a non-trivial challenge for server applications, which often follow a modularized design and load these libraries at run time using interfaces like `dlopen()` and `dlsym()` based on a configuration file. However, resolving the libraries and functions imported by an application through these interfaces is necessary to avoid errors.

4 Design and Implementation

4.1 Overview

The objective of SysPart is to overcome the limitations of previous works [13, 20] and generate an automatic system-call filter for server binaries. To achieve this, our approach uses both static and dynamic analyses, as illustrated in Figure 2. All analyses operate on binary code. Static analyses are based on the Egalito [56] framework, while dynamic analyses use Intel's Pin [35] framework. To automatically identify the serving phase of server binaries, we first use static analysis to locate loops, and then employ dynamic analysis to determine the dominant loop for each thread, which corresponds to its main loop. Before generating system-call filters,

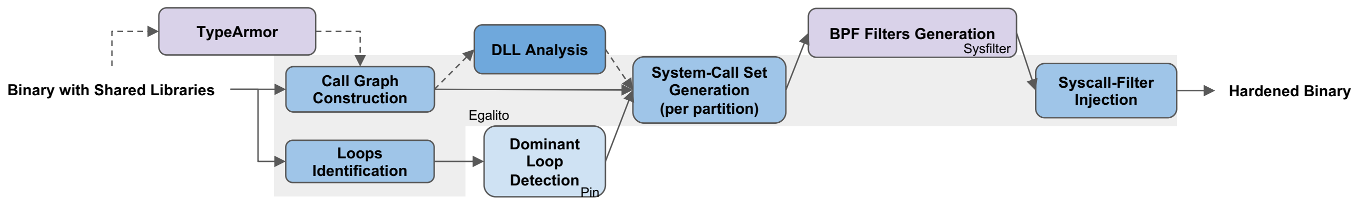


Figure 2: Workflow of SysPart.

we construct the function call graph of the application statically, and then refine it using value-flow analyses (VFA) and heuristics. We also use TypeArmor [55], a third-party tool, to further refine the graph. To determine the use of dynamically-loaded libraries (DLL) by the application, we combine both static and dynamic analysis, with the former providing soundness. The above information is then combined to generate a list of required system calls for each serving phase. Next, SysPart generates a BPF program that will install the appropriate Seccomp-BPF filters, which is inspired from sysfilter. Finally, using Egalito, we rewrite the application and inject these functions at the phase transition points. All extensions that we developed over Egalito is described in section A.5.

4.2 Serving Phase Detection

To define the beginning of the serving phase of a server program, we build on the following observations:

- Server applications utilize a loop to serve clients.
- A system-call filter should be installed outside a loop to avoid repeated installation.

Hence, we define the serving phase of a program as the beginning of a top-level loop, where it spends most time executing. Here, top-level means the loop is not enclosed in another loop. Our algorithm for detecting the serving phase, therefore, focuses on identifying this *main loop*. Preceding it is the transition point, where a system-call filter can be installed. Serving phase detection consists of a static-analysis phase (§4.2.1), which finds all loops in the application and its libraries, and a dynamic-analysis phase (§4.2.2), where this information is used to find the dominant or main loop. By design, this algorithm can find the main loop used by different worker threads (or processes) of the server, enabling SysPart to generate system-call filters for all of them. We describe them below.

4.2.1 Loops Identification: To identify all loops, we employ the Egalito [56] binary-analysis tool to statically disassemble applications and libraries, and extract the control-flow graph (CFG) of each function of the application and its libraries. We focus on loops produced using: `for() {}`, `while() {}`, `do{} while()`, and `goto` statements. For each function, we employ the worklist algorithm [12, 17, 34], which builds on the concept of dominance, to identify all loops present in the function. Briefly, the algorithm is based upon finding *dominator* nodes and *back edges*, as defined below.

Dominator: A node N (which represents a basic block in a CFG) dominates node M if all paths to M must pass through N. A node dominates itself. If there is an edge from node N to M in the CFG, then N is a predecessor of M and M is a successor of N. The set of dominators of node Z is defined as:

$dom(Z) = Z \cup (\cap(dom(Y)),$
where Y = set of all predecessors of Z.

Back Edge: An edge from node N to node H is a back edge if H dominates N. Node H is the "header" of the loop (the place where the loop is entered).

Using the above, the algorithm identifies the code blocks comprising the *body* of loops, as well as the *exit nodes* of each loop:

Loop Body: For a loop with back edge from node N to node H (header node), the body of the loop is obtained by starting from node N and recursively finding all predecessor nodes until the header node is reached.

Exit Nodes: The exit nodes of a loop are determined by identifying those nodes within the loop body who have outgoing edges to nodes outside of the loop.

Algorithm 1 Dynamically compute the executed top-level loops

```
Input: A set of loops S with a start_address and a set E of exit_address
Output: Set executed_top_loops which contains executed top-level loops
 1: function COMPUTE_OUTER_LOOP(S)
 2:     cur_loop ← []
 3:     for Each instruction with address i do
 4:         for Each loop L in S whose exit_address is i do
 5:             if cur_loop is L then
 6:                 endtime(cur_loop) ← get_current_time()
 7:                 duration(cur_loop) ← endtime(cur_loop) - starttime(cur_loop)
 8:                 executed_top_loops ← executed_top_loops ∪ cur_loop
 9:                 cur_loop ← NULL
10:                 break
11:         if i is the entry_address of loop L then
12:             if cur_loop is NULL then
13:                 cur_loop ← L
14:                 starttime(L) ← get_current_time()
15:                 iterations(L) ← 0
16:             else if cur_loop is L then
17:                 iterations(L) ← iterations(L) + 1
```

4.2.2 Dominant Loop Detection: We implement a component using Intel's Pin [35] to determine the dominant loop when running servers with simple workloads . Intel Pin is a dynamic binary instrumentation tool for developing and applying tools, known as pintools, on binary programs at run-time. We develop a pintool that utilizes the data obtained statically in the previous step, including the start address of the loop and the addresses of its exit nodes, to calculate the amount of time each top-level loop encompasses execution for each process and thread, according to Algorithm 1.

The tool takes a set of loops S as input, where each loop has a starting address and a set of exit addresses. The algorithm iterates through each executed instruction in the program and, for each instruction, checks whether it is the entry address of a loop in S. If it is, and if the currently executing loop (`cur_loop`) is NULL, then it sets `cur_loop` to this loop and sets its start time and iteration count to zero. If `cur_loop` is not NULL, the algorithm increments the iteration count for `cur_loop`. If the instruction is an exit address of `cur_loop`, the algorithm calculates the duration of the loop and adds it to the output set of executed loops. This process is repeated for each instruction in the program until the program exits. At the end of the algorithm, the set of executed loops contains all the top-level loops that were executed, along with their start time, end time, duration, and iteration count. The information is organized per thread/process. We consider the loop with the largest execution time as the main loop of each execution thread, and its start address the transition point where filter installation should be placed.

4.3 Call Graph Construction

In order to compute the system calls needed by each main loop, we need to determine the code that is reachable after the transition point. As a first step, we generate the function-call graph (FCG) of the program rooted at `main()` and across shared libraries. Although, we do not include any code running before `main()`, such as initialization routines, we do account for how it may affect the FCG (discussed in §4.3.2). The method described here is based on Egalito's VacuumFCG pass, which is also used by sysfilter and demonstrated to be sound. Below, we summarize the methodology.

4.3.1 Computing Direct Edges: FCG construction starts from the functions that the Linux program loader will call, which includes initialization functions in `.init`, `.init_array`, and `.preinit_array`, cleanup functions in `.fini` and `.fini_array`, and the program's `main()`. Egalito disassembles each function's instructions and follows direct branches, such as calls, jumps, and conditional branches, to construct an initial CFG. When there are calls to functions in shared libraries, those edges initially point to the binary's procedure linkage table (PLT). Egalito emulates the dynamic linker/loader to resolve them and point them to the actual function, extending (this way) the graph across shared libraries. Egalito is robust enough to also handle GNU libc. Also, we resolve calls to functions over the Name Service Switch (NSS) scheme using the approach mentioned in sysfilter.

4.3.2 Computing Indirect Edges: The FCG includes indirect control-flow edges that correspond to calls made through function pointers, which are represented by indirect call or jump instructions in the basic blocks of the disassembled binaries. However, it is difficult to resolve the targets of these indirect branches [51]. Therefore, SysPart over-approximates the set of functions that these branches can target. Specifically, it includes all address taken (AT) functions, which are functions whose start address is referenced or loaded in the program either directly or by taking the address of a PLT entry. To identify AT functions, VacuumFCG takes advantage of the fact that modern Linux binaries are built as position-independent code (PIC) and include metadata in the form of relocation entries for every function address taken in the program [15].

Many function pointers in programs and libraries are part of the initialization of constant function-pointer arrays. To reduce the number of AT functions, VacuumFCG prunes [5, 6] those that are included in such arrays but are never used or taken by live (i.e., actually used) program code. When symbols are available, the analysis can calculate the boundaries of data objects using symbol information and consider only the AT functions that are in objects used by the application. This process is performed iteratively to find objects and AT functions actually used by the program. If symbols are not available, any program address pointing to a section of the binary leads to the inclusion of all AT functions found in it.

4.4 Refining the FCG

Prior works [5, 6, 13] operating on binaries had to rely on the above method of over-approximating the FCG, by assuming that all indirect calls can target any AT function, to ensure soundness. To increase the precision of the FCG, SysPart implements data-flow analysis for constants (i.e., value-flow analysis or VFA), and incorporates TypeArmor, a binary-analysis system that aims to reduce the number of possible targets for indirect call sites.

4.4.1 Forward Value-Flow Analysis: SysPart employs forward VFA to determine where each function pointer flows and eliminate it, if possible, from the list of AT functions obtained in § 4.3.2. If the pointer is only used as the target of an indirect call, it is removed from the list of AT functions and an edge between the indirect call and pointed function is also added to the FCG. If the pointer is not passed as an argument to a function, nor stored in memory nor returned as the value of a function, then it is removed from the list of AT functions. For a function to be completely eliminated from the list, all of its pointers need to be resolved in this manner.

The example in Listing 1 shows such a case from Nginx, where a pointer to the `ngx_http_upstream_init_round_robin()` (located at `0x7e1d(%rip)`) is taken on line 6 and is later used by the indirect call in line 3. The only use of the function pointer is as target of the indirect call and hence can be removed from the AT list.

```
1 bbl2:
2     movq        %r12, %rdi
3     call        *%rax
4     ...
5 bbl1:
6     leaq        0x7e1d(%rip), %rax ; <ngx_http_upstream_init_round_robin>
7     jmp         0x6b2d2            ; <bbl2>
```

Listing 1: Flow of function pointer in the Nginx server.

Likewise, the pointer to `RedisModuleCommandDispatcher()` in Listing 2 is only used in a compare instruction in line 3. We can safely remove it from the AT list, because it cannot be the target of any indirect call.

```
1 bbl1:
2     leaq        -0x21c9(%rip), %rbp    <RedisModuleCommandDispatcher>
3     cmpq        %rbp, 8(%rax)
4     (JUMP jne)  0xb1948
```

Listing 2: Flow of function pointer in the Redis server.

We implement VFA using Egalito's use-def chains. Use-def analysis, tracks the uses and definitions of a register or memory location. A register or memory location is defined when a value is written to it, and it is considered used whenever its value is read. The use-def chains, provided by Egalito, also maintain the locations in successor basic blocks where a register or memory location is later used, and the locations in predecessor blocks where it was previously defined. For example in Listing 3, use-def analysis determines that the instruction at address `0x0002a7aa`, defines register `rdx` and uses `rcx`. Use-def chains also inform us that `rcx` was previously defined by the instruction at `0x0002a72c` and `rdx` is later used by the instruction at `0x0002a7b1`. Even though Egalito does not maintain use-def chains across function calls, our VFA is inter-procedural as it tracks values passed (forward) to functions through registers (e.g., as arguments).

```
0x0002a72c:        movl        8(%rdi), %rcx
0x0002a7aa:        movl        %rcx, %rdx
0x0002a7b1:        addl        %rdx, %rdi
```

Listing 3: Use-def information provided by Egalito.

4.4.2 *Backward Value-Flow Analysis:*

SysPart leverages use-def chains to perform backward value-flow analysis starting from indirect calls to determine the possible values of the operand of the indirect call instruction. If a path points to a value that is a function pointer, an edge to it is added from the function containing the indirect call. If all paths lead to values from function pointers, the indirect call is no longer assumed to target all the functions in the AT list, considerably improving the precision of the FCG. However, even if one path leads to a memory load the analysis for that indirect call terminates. This analysis is also partially inter-procedural, as the forward one.

The example in Listing 4 shows such a case, where two functions, lines 1 and 5, call `ngx_sort`, passing a function pointer as an argument, through the `rcx` register. This call now has exactly two possible targets that it can call into. Our backward VFA starts from line 11 to determine the value of register `r15`. Register `r15` is referenced in line 10 and its value is defined to be loaded from `rcx`. Since `rcx` is an argument register, inter-procedural analysis is performed to find all invokations of `ngx_sort()`. From the usedef information at the call sites of `ngx_sort()` (lines 3 and 7), the analysis determines that register `rcx` is referenced at line 2 and line 6, respectively. The analysis terminates at lines 2 and 6, where a function address is loaded into `rcx`. Hence, the analysis is able to completely resolve the indirect call target at line 11.

```
1  ngx_resolver_process_response:
2    leaq    -0x59a5(%rip), %rcx    ; <ngx_resolver_cmp_srvs>
3    call    0x25a33                ; <ngx_sort>
4    ...
5  ngx_http_block/bb+5407:
6    leaq    -0x1d43(%rip), %rcx    ; <ngx_http_cmp_conf_addrs>
7    call    0x25a33                ; <ngx_sort>
8    ...
9  ngx_sort:
10   movq    %rcx, %r15
11   (CALL*) *%r15
12   ...
```

Listing 4: Resolving all possible values of an indirect call in the Nginx server.

4.4.3 *TypeArmor:*

SysPart uses TypeArmor [55] to further prune the number of targets for each indirect-call site. TypeArmor is a static analysis tool for binaries that aims to refine the set of functions that can be targeted by indirect function calls. It does so by detecting the signature of call sites and functions. For each indirect call, it calculates the number of arguments prepared and whether it expects a return value. For each function, it calculates the maximum number of arguments it expects and whether it returns a value. Call sites with n arguments are matched to AT functions that expect $\leqslant n$ arguments and similar return behaviors (value vs. no value returned).

4.5 Dynamically-Loaded Libraries Analysis

In many servers, optional, non-core functionality is sometimes activated at run time, depending on the presence of a library on the system or configuration options. When such functionality requires additional libraries, those are loaded using `dlopen()`, which loads the library with `filename` into the address space. Interfaces to DLLs are obtained through `dlsym()`, that takes as input the handle returned by `dlopen()` and the name of a symbol, which could be an exported function or global variable. Function pointers returned by `dlsym()` can be called through an indirect call and may perform system calls, hence, it is necessary to resolve the libraries loaded and symbols queried through these two functions.

SysPart employs static analysis to recover this information. To handle cases where the static analysis fails to resolve all possible values, we use dynamic training by running the application with a desired configuration and common workloads.

4.5.1 *Static Analysis:*

SysPart uses two approaches to statically discover this information: backward VFA and a heuristic.

Backward VFA: We leverage this type of analysis again to find the `filename` and `symbol` arguments used in `dlopen()` and `dlsym()`, respectively. If the application is using constant strings, we can find pointers to those strings flowing to the first and second argument, respectively. In x86-64 Linux, these correspond to registers `rdi` and `rsi`.

Heuristic: The interfaces queried using `dlsym()` are frequently hard-coded in applications, while the names of the libraries to be loaded are provided in configuration. For example, the Bind server includes a plugin for accessing Samba Active Directory (AD) databases. The interface to this plugin includes a set of functions (`dlz_*`) which are constant. However, depending on the AD database in use, the user can load a different plugin version with a configuration like the following.

```
dlz "AD DNS Zone" {
    # For BIND 9.16.x
    # database "dlopen /usr/local/samba/lib/bind9/dlz_bind9_16.so";
};
```

Listing 5: Bind configuration file specifying plugin.

Based on this observation, when we are able to resolve all possible values flowing into `dlsym()` call sites, but not to `dlopen()`, we use the first to search the system for libraries exporting any of resolved symbols. We consider all matching libraries as potential inputs to `dlopen()` and include them in our analysis.

4.5.2 *Dynamic Analysis:*

To discover the libraries loaded at run time, we run applications with desired configuration options and common workloads, and intercept calls to `dlopen()` and `dlsym()` to record their arguments. We use function interposition to redirect calls to our functions, which record their arguments and proceed to call the original functions. This is done by creating a shared library defining these two functions and pre-loading it through the

LD_PRELOAD environment variable on Linux, when launching the application.

4.5.3 Incorporating Results: SysPart considers any libraries found in this step as additional dependencies and resolved symbols are marked as taken at the call site of the corresponding dlsym(). Finally, the static analysis described in §4.4.1 is reapplied to again prune the list of AT functions and indirect-call targets.

4.6 Handling execve

The execve syscall allows a process to load and execute a new program, so it needs special handling. Similar to how we deal with DLLs, SysPart combines static analysis and dynamic analysis to collect the arguments passed to execve—in particular the path of the new program to be executed. First, SysPart runs static, backward VFA to find the arguments of execve. Second, SysPart traces the execution of the application under desired configurations and common workloads to learn the arguments passed to execve, with the help of Pin. SysPart also offers users the option to add the list of programs that can be executed through execve.

Based on the above analyses and user input, any new programs that may be launched by execve during the serving phase will be further analyzed to gather the syscalls they require. The analysis results can be applied in two ways. First, the newly identified syscalls will be added to the allowed list and the entire list will be propagated to the new program. Second, we start with the extended allowed list for the initial program but further reduce the list every time execve is invoked to only keep those syscalls needed by the new program. This approach is inspired by sysfilter [13].

4.7 System-Call Set Generation

Computing the complete set of system calls reachable from the transition point is crucial to determine the correct system-call filter for the serving phase, as erroneous filters can lead to program termination. In order to determine the system calls of each serving phase, first we compute the system calls which are invoked by each function within the FCG. Next, the code which is reachable from the transition point of that serving phase is determined. Finally, we collect all system calls which are invoked from the reachable code.

4.7.1 Finding System Calls Reachable from Each Function of the FCG: A system call is represented by a system-call number and may or may not have arguments. Applications can invoke system calls by invoking libc wrapper functions, which in turn invoke the system call (for example, open() invokes the corresponding open syscall), use the syscall() libc function, or use inline assembly and the syscall instruction.

SysPart uses Egalito's FindSyscalls() which is run on all the functions in the server and its dependent libraries to find all system calls that are invoked directly from these functions. The pass parses each instruction within a function and searches for syscall instructions. If one is found, it employs backward VFA to determine the value of the register rax that contains the system-call number to be executed. Similarly, backward VFA is used for calls to syscall() to find the value of the argument specifying the syscall number (e.g., the register rdi in x86-64 Linux systems).

Algorithm 2 An algorithm to find all system calls reachable from all functions in FCG rooted at main()

```
Input: FCG; Map M(S, L) where L is a list of functions which directly invoke syscall S
Output: Map fsyscalls(F, L) where L is the list of syscalls reachable from function F
 1: function SYSCALLS_FUNCTION(FCG,S)
 2:     for Each (s, flist) in M do
 3:         for Each f in flist do
 4:             Push (f,f) onto stack S
 5:             processed ← []
 6:             while stack S is not empty do
 7:                 (cur,cur_child) ← top of stack S
 8:                 Pop from stack S
 9:                 if cur is in processed then
10:                     continue
11:                 processed ← processed ∪ cur
12:                 pp ← Parents of cur in FCG
13:                 for Each parent in pp do
14:                     Push (parent,cur) onto stack S
15:                 if cur == cur_child then
16:                     continue
17:                 Insert s into fsyscalls[cur]
    return func_syscalls
```

The set of system calls **reachable** from each function is computed using algorithm 2, which works by combining the set of system calls directly invoked by the function with those that are reachable from its child functions in the FCG. It is a graph traversal algorithm that begins at each function that invokes system calls directly, and iteratively propagates the value of the system call to its parents.

Algorithm 3 An Algorithm to find system calls reachable from the transition point located at address entry_addr within function f

```
Input: Address entry_addr, Function f, FCG rooted at main()
Output: Set SC, which contains system calls reachable from entry_addr
 1: function SYSCALLS_PARTITION(entry_addr,f,FCG)
 2:     noreturnFns = Find all noreturn functions
 3:     threadFns = Find all thread start functions
 4:     Push (entry_addr, f) to stack S
 5:     SC ← []
 6:     while stack S is not empty do
 7:         (addr,fun) ← top of stack S
 8:         B ← basic block at address addr
 9:         Pop from stack S
10:         result ← syscalls_invoked_at_instruction(addr, fun)
11:         CFG ← CFG of fun
12:         visited ← []
13:         Insert B into visited
14:         for Each successor succ of B in CFG do
15:             Push succ to stack SS
16:         result ← []
17:         while Stack SS is not empty do
18:             T ← top of stack SS
19:             Pop from stack SS
20:             Insert T into visited
21:             for Each instruction i with address iaddr in T do
22:                 cur ← []
23:                 if i is a call instruction then
24:                     for c in call target set CT of i do
25:                         cur ← cur ∪ reachable_syscalls(c)
26:                 else
27:                     cur ← syscalls_invoked_at_instruction(iaddr, fun)
28:                 result ← result ∪ cur
29:             for Each successor succ of T do
30:                 if succ is not in visited then
31:                     Push succ to stack SS
32:         SC ← SC ∪ result
33:         if fun is in noreturnFns then
34:             continue
35:         if fun is in threadFns then
36:             continue
37:         P ← Parents of fun in FCG
38:         for Each p in P do
39:             calladdr ← address at which p invokes fun
40:             Push (calladdr, p) to the stack S
    return SC
```

4.7.2 Finding System Calls Reachable from Transition Point: The transition point of the serving phase is the start address of the main loop which can be represented by a tuple `(f, addr)`, where `f` is the function containing the main loop and `addr` is the start address of the main loop. Algorithm 3 describes the algorithm which determines the code reachable from `(f, addr)` and the system calls invoked from the reachable code. The algorithm starts with non-returning functions analysis and computing thread start functions which are described below.

Non-returning Functions Analysis: We improved the no-return analysis pass in Egalito to find all non-returning functions. A function `f` starting from basic block B is non-returning, only if all paths from B in the `CFG` of `f` is non-returning.

Computing Thread Start Functions: Threads are created by invoking the `pthread_create()` function, with the thread start function specified as the third argument, which is the function executed as soon as the thread is spawned. SysPart checks if `pthread_create()` function is a part of the FCG, and all invocations of `pthread_create()` are determined. Then, backward VFA (4.4) is employed at these callsites to find the third argument passed to the function (register `rdx`).

Syscall-Set Computation: In order to find all system calls reachable from instruc- tion i with address addr of function `f`, all code reachable from `i` has to be determined. To start with, we find all basic blocks within `f` which are reachable from instruction `i`. Next, we consider all functions which are invoked from within these basic blocks and get all system calls which are reachable from all these functions (computed in section 4.7.1). Next, if f returns, then all callsites `(f', addr')` which invoke function `f()` are found. Next, the analysis recursively proceeds to find all the system calls which are reachable from `(f',addr')`. The algorithm stops when main() is reached or a thread start function is reached or if function is non-returning. In this way SysPart computes all code which are reachable from the serving phase which also includes the code that the function returns back to, which is not considered by TSP.

4.8 Enforcing System-Call Filters at the Partition Boundary

4.8.1 Filter Policy: We enforce a policy that aims to only allow syscalls used in the application's main loop. This policy has significant and immediate security benefits: (i) attacking payloads are unable to utilize dangerous but blocked syscalls and (ii) compromised applications cannot attack the operating system kernel using blocked syscalls. While further restricting syscall arguments could additionally enhance security, it poses challenges with Seccomp-BPF and is susceptible to TOCTOU attacks [36]. Note that previous work [37] has explored argument restriction for library APIs.

4.8.2 Seccomp-BPF Filter Generation: The list of allowed system calls of the serving phase obtained in §4.7.2 is used to create a C function which uses Seccomp-BPF, as in sysfilter [13], to install a filter that will be enforced by the kernel. The generated function, `install_filter()`, is compiled into a shared library (`libsyspart.so`).

4.8.3 Filter Insertion: We developed a tool using Egalito that links against the generated library and inserts call to `install_filter` at the transition point and generates a hardened binary. The server binary, the transition point represented by the tuple `(f, addr)`, and `libsyspart.so` is fed as input to the tool. `libsyspart.so` is parsed using Egalito. The CFG of `f` is generated and traversed to determine a basic block `B` such that B precedes the basic block with address `addr` and B is not a part of the main loop. A new Egalito pass `SyspartPass` is then used to insert a function call to `install_filter()` after basic block `B`. A new hardened binary with the installed filter is generated using Egalito `mirrorgen` output generation mode.

5 Evaluation

To evaluate SysPart and compare it with prior work, we apply it to the same benchmark applications used by TSP [20]: Bind 9.15.8 (incl. libuv-1.34.0 and OpenSSL-1.1.1f), Httpd 2.4.39 (incl. apr-1.7.0 and apr-util-1.6.1), Lighttpd 1.4.54, Memcached 1.5.21 (incl. SASL-2.0.25 and libevent-2.1.11-stable) and Nginx 1.17.1 (incl. OpenSSL-1.1.1f), and Redis 5.0.7. All the dependent libraries used by the applications are from Ubuntu 18.04.6 LTS, which is also adopted in the evaluation of TSP. We note that Ubuntu 18.04.6 uses glibc-2.27 by default while TSP runs its static analysis on glibc-2.24. To stay aligned, we also use glibc-2.24 for static analysis but glibc-2.27 for dynamic library profiling (the older version cannot run). All the experiments were performed on a 4-core Intel Core i7 8550U 1.80GHz CPU with 16GB of RAM, running Ubuntu 18.04.6 LTS (kernel version 5.4.0-150).

5.1 Serving Phase Detection

We first evaluate SysPart's ability to automatically detect the beginning of the serving phase. To dynamically profile each server, we used its default configuration. The algorithm is designed to identify the dominant top-level loop in each thread, excluding the loops used in initialization and cleanup, so even with some configuration or workload changes, the main loop remains the same.

Simply launching the servers and letting them wait was sufficient to correctly identify the serving phase with all servers, as it is common behavior for servers to wait within their main loop. However, to ensure the main loop is entered we assume that the desired workload is sent to the tested server. For the evaluation, we used the following workloads: 10k HTTP requests for Httpd, Lighttpd, and Nginx; 10k store/set requests of randomly generated key-value pairs, followed by a retrieve/get request for each pair for Redis and Memcached; and 10k IP address queries using the utility `dig` against Bind.

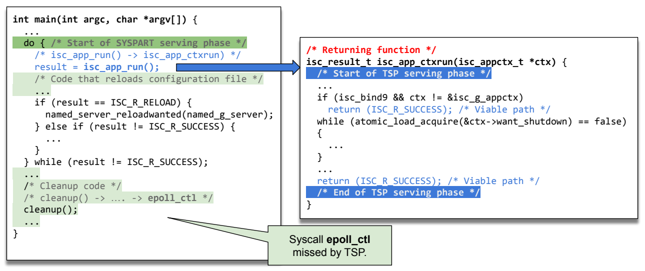
In certain cases, the server can be set up to run in different modes, which correspond to different main loops. For instance, Nginx can be configured as a proxy or cache server instead of a web server (the default configuration). To capture the main loops of these different servers within Nginx, the user only needs to profile it with the appropriate configuration. Once properly configured, our approach automates the detection of different main loops, relieving the user from the burden of code review. In our evaluation, we did not test Nginx as a proxy or cache server.

Table 1 summarizes the results of the experiment. SysPart is able to automatically detect the main loop corresponding to the serving

Table 1: Results of serving phase detection in SysPart compared with TSP. In red, we highlight the cases where TSP demonstrably creates wrong filters. "⟻" indicates that the returning functions lead to errors in TSP. "Size Δ" stands for the number of more instructions in SysPart's main loop than that in TSP's serving-phase function. "# of Other Loops" is the number of extra loops detected by SysPart for auxiliary server threads. "Concurrency" shows the number of processes (P) / threads (T) used by each server application, where "*" means one or more.

Application		TSP	SysPart		
Name	Concurrency	Serving Function	Main Loop	Size Δ	# of Other Loops
Bind	1/*	⟻`isc_app_ctxrun`	`main+0xe01`	+2.5K	4
Httpd	*/*	`child_main`	`child_main+0x598`	+0.3K	5
Lighttpd	1/1	⟻`server_main_loop`	`main+0x84`	+83K	0
Memcached	1/*	`worker_libevent`	`event_base_loop+0xbb`	+1K	6
Nginx	*/1	`ngx_worker_process_cycle`	`ngx_worker_process_cycle+0xbb`	+0.3K	1
Redis	1/*	⟻`aeMain`	`aeMain+0x10`	+0.2K	1

process/thread of each server, whose results are summarized in Table 11. We further compared the main loops with the serving-phase function manually identified by TSP developers. In three applications (Httpd, Nginx, Redis), SysPart identifies the main loop in the TSP's serving-phase function. In the case of Memcached, the main loop runs inside a child function of TSP's serving-phase function (`worker_libevent()` → `event_base_loop()`). In the other two applications (Bind and Lighttpd), the situation is reversed. TSP's serving-phase function is invoked inside the main loop identified by SysPart.



```
int main(int argc, char *argv[]) {
  ...
  do { /* Start of SYSPART serving phase */
    /* isc_app_run() -> isc_app_ctxrun) */
    result = isc_app_run();
    /* Code that reloads configuration file */
    ...
    if (result == ISC_R_RELOAD) {
      named_server_reloadwanted(named_g_server);
    } else if (result != ISC_R_SUCCESS) {
      ...
    }
  } while (result != ISC_R_SUCCESS);
  ...
  /* Cleanup code */
  /* cleanup() -> .... -> epoll_ctl */
  cleanup();
  ...
}
```

```
/* Returning function */
isc_result_t isc_app_ctxrun(isc_appctx_t *ctx) {
  /* Start of TSP serving phase */
  ...
  if (isc_bind9 && ctx != &isc_g_appctx)
    return (ISC_R_SUCCESS); /* Viable path */
  while (atomic_load_acquire(&ctx->want_shutdown) == false)
  {
    ...
  }
  ...
  return (ISC_R_SUCCESS); /* Viable path */
  /* End of TSP serving phase */
}
```

Figure 3: Advantage of SysPart's automatic detection of serving phase over TSP's manual approach.

Quantitatively, the size of the serving phase detected by SysPart is only slightly larger than TSP except for Lighttpd. The increase of size—including the case of Lighttpd—is mainly because we consider the code following the return of the serving phase function (if it returns) and the finalization functions while TSP does not consider these. Our choice is desired as ignoring the post-serving code can omit system calls needed for reloading configuration data or cleanup before termination. This can potentially lead to issues like corrupting server data and leaking operating systems resources. One such example from Bind is shown in Figure 3. TSP identifies `isc_app_ctxrun()` as the serving-phase function. When it returns, the cleanup code attempts to performs an `epoll_ctl` system call, which is filtered by TSP since it doesn't include it within its serving phase and leads to abrupt server termination.

Another advantage of SysPart is its ability to identify the serving phase for auxiliary server threads(Table 11). For five applications, SysPart detects at least one auxiliary serving phase. In the case of Memcached, the number of auxiliary serving phases increases to six. This enables us to filter out a safe but also tight set of system calls for each server thread.

5.2 Filtered System Calls

In this experiment, we compare the system calls allowed by SysPart with the ones generated by TSP and sysfilter. As TSP and sysfilter do not resolve dynamically-loaded libraries, we ignore them in this experiment. We consider four different code partitions. ① `main()` includes all the code after the `main` function is entered. ② *Serving* refers to TSP's serving phase. ③ *Main loop* corresponds to SysPart's serving phase. ④ *All* includes all code from the time a process is spawned (used by sysfilter) not including NSS.

During the experiment, we discovered a variety of suspicious differences with TSP. Upon thorough investigation, we found a series of issues in TSP that are rooted in coding bugs, incomplete algorithms, and its inherent inability to handle inline assembly. We confirmed all the problems by consulting with the authors of TSP [20] and list all of them below. Because of them, TSP can miss required system calls or include unused ones. To obtain a more representative comparison, we fixed all problems aside #11, which requires modifications to TSP's algorithm. We use $\text{TSP}_{\text{fixed}}$ to refer to the version of TSP with all our fixes incorporated.

TSP Issues Leading to Missing Required System Calls (refer to Appendix A.1 for more details):

1. Cannot handle certain inline assembly constructs.
2. Does not handle `syscall` function.
3. Does not handle certain function aliases in glibc.
4. Ignores AT functions in initialization and dynamic-linker code.
5. Does not handle function pointers passed to library functions.
6. Does not handle glibc NSS libraries [41].
7. Library not considered due to missing CFG.

TSP Issues Leading to Additional Allowed System Calls (Refer to Appendix A.2 for more details):

8. Source-code analysis (Egypt) cannot differentiate between multiple libraries in the same directory and merges all of them in the resulting FCG.

Table 2: Number of system calls allowed with SysPart (binary), compared to TSP (source code) and sysfilter (binary). "TSP" refers to the publicly available prototype, and the numbers in parentheses correspond to the results in the paper. $\text{TSP}_{\text{fixed}}$ stands for the version of TSP with our fixes incorporated (the fixes used are indicated by the subscript on each application). The entries highlighted in red indicate a confirmed error occurred in TSP's definition of the serving phase.

Application	sysfilter	TSP		$\text{TSP}_{\text{fixed}}$		SysPart			
	All	`main()`	Serving	`main()`	Serving	`main()`	Serving	Main loop	AT Functions
Bind $_{1, 2, 7, 8, 9, 10, 11}$	119	100 (99)	86 (85)	100	83	112	103	103	102
Httpd $_{1, 3, 4, 8, 9, 11}$	98	94 (94)	79 (79)	94	80	92	92	92	92
Lighttpd $_{1, 3, 6, 8, 9}$	99	95 (95)	76 (76)	95	76	95	80	93	74
Memcached $_{1, 3, 4, 8, 9, 11}$	104	99 (99)	84 (84)	100	85	98	82	85	82
Nginx $_{1, 2, 3, 5, 8, 9, 11}$	115	106 (104)	97 (97)	109	100	108	104	104	104
Redis $_{1, 3, 8, 9}$	104	90 (90)	82 (82)	91	83	92	85	85	85

(9) Implementation erroneously considers certain null nodes in glibc callgraph as actual functions and uses them during analysis.
(10) TSP includes all exported functions as AT functions in cases where callgraph of libraries are not present.
(11) Algorithm fails to prune certain inaccessible AT functions.

Table 2 summarizes the comparison results. In summary, SysPart performs closely to TSP. On average, we allow 8.33% more system calls compared to TSP. We manually inspected the differences and found that they are mainly attributed to the availability of source code to TSP. With source code, TSP can perform point-to analysis to resolve indirect calls more accurately (more details in section A.3). In contrast, SysPart only assumes binary code and our VFA powered FCG refining still results in an inflated function-call graph. In the last column of Table 2, we list the number of system calls that are reachable from AT functions through direct edges. It indicates that our approach represents the best efforts unless better methods that can track values through global and heap memory are available to resolve indirect calls in binary code.

5.3 Security Benefits

5.3.1 Kernel Attack Surface Reduction: This experiment evaluates the effectiveness of SysPart in reducing kernel attack surface using the 36 kernel vulnerabilities also used by prior work [13, 20]. We count the number of applications where sysfilter, TSP, and SysPart filter the system calls required to trigger each vulnerability. The evaluation results are summarized in Table 3. SysPart performs either better or the same as sysfilter, regardless of which vulnerability we consider. Compared to TSP, SysPart performs better or the same for 24 vulnerabilities but worse in 12 `execve`-based ones. Fundamentally, SysPart allows `execve` in those cases because it is reachable from AT functions stored in global data. Our VFA cannot track their propagation and, thus, cannot rule them out from indirect call targets.

5.3.2 Exploit Mitigation and Hindrance: We also evaluate the effectiveness of SysPart in thwarting exploit shellcode. Specifically, we reuse the 535 shellcodes involved in the evaluation of TSP and consider a shellcode to be stopped if at least one needed system call is filtered. Considering that certain Linux system calls provide interchangeable functionalities and adversaries could easily adapt the shellcode to use alternative system calls, we deem a payload to be stopped only when all equivalent system calls are filtered. The list is shown in table 4. To identify equivalent system calls, we follow the same rules as proposed in the TSP paper [20].

We summarize the results in Table 5. Unsurprisingly, SysPart outperforms sysfilter by a large margin but stops fewer shellcodes than TSP for most applications (again because of the `execve` system call). In the case of Memcached, SysPart outperforms TSP. The reason is that SysPart can filter out `getsockopt` and `setsockopt`, while TSP cannot due to issue 11 discussed in §5.2.

Security-Sensitive System Calls We further narrow down our attention to security-sensitive systems calls (or in TSP's definition, system calls frequently used in payloads because of their usefulness to attackers). Table 6 presents our findings. Out of 102 cases (17 system calls in 6 applications), TSP outperforms SysPart in 10 cases (9.8%). This is due to the over-approximation of indirect call targets by SysPart and the gains delivered to TSP by its points-to analysis. In two cases (2%), SysPart outperforms TSP. Specifically, `listen` in Nginx and `recvfrom` in Bind are *not* filtered by TSP due to issue 11. In all other cases, SysPart and TSP are similar. Compared to sysfilter, SysPart performs better in 24 cases (23.5%).

5.4 Dynamically Loaded Libraries

We propose dynamic library profiling in §4.5 to resolve the libraries loaded at runtime. In this evaluation, we measure the accuracy and the necessity of this approach. We run the servers with both the default configuration and customized configurations where different modules are enabled. For Redis and Lighttpd, we use the test cases shipped with the respective packages. For Nginx and Httpd, we use `nginx-tests` [42] and `Apache-Test-1.4.3` [8] as test cases, respectively.

By analyzing the outcomes of the evaluation above, we unveil that the arguments to `dlopen()` and `dlsym()` are either hard-coded, read from a configuration file, or constructed dynamically by concatenating strings. In Table 7, we show how well we can resolve those arguments. Given Redis and Memcached, our static analysis—combing VFA and heuristics—can resolve all `dlopen()` and `dlsym()`. For Bind and Nginx, we can resolve all but one `dlopen()` and two `dlsym()`. The unresolved cases are due to limitations of VFA when handling dynamically generated arguments in the `libcrypto` library. These callsites are not observed during runtime too. In the cases of Httpd and Lighttpd, our analysis cannot

Table 3: Mitigated kernel vulnerabilities. SF and SP stand for sysfilter and SysPart, respectively.

CVE	System Calls	SF	SP	TSP/TSP$_{\text{fixed}}$
2018-18281	execve(at), mremap	0	0	4/4
2016-3672 2015-3339 2015-1593 2014-9585 2013-0914 2012-4530 2010-4346 2010-3858 2008-3527 2018-14634	execve(at)	0	2	4/4
2012-3375	epoll_ctl	0	0	1/1
2011-1082	epoll_ctl, epoll_pwait, epoll_wait	0	0	1/1
2010-4243	uselib, execve(at)	0	2	4/4
2019-11815	clone unshare	0	2	1/2
2013-1959	write	0	0	0/0
2015-8543	socket	0	0	0/0
2017-17712	sendto sendmsg	0	0	0/0
2013-1979	recvfrom recvmsg	0	0	0/0
2016-4998 2016-4997 2016-3134	setsockopt	0	1	0/0
2017-18509	setsockopt, getsockopt	0	1	0/0
2017-14954	waitid	6	6	6/6
2014-5207	mount	6	6	6/6
2018-12233	setxattr	6	6	6/6
2016-0728 2014-9529	keyctl	6	6	6/6
2019-13272 2018-1000199	ptrace	6	6	6/6
2014-4699	fork, clone, ptrace	0	2	1/2
2014-7970	pivot_root	6	6	6/6
2019-10125	io_submit	6	6	6/6
2017-6001	perf_event_open	6	6	6/6
2016-2383	bpf	6	6	6/6
2018-11508	adjtimex	6	6	6/6

resolve any case because the arguments are loaded from configuration files. SysPart presents a better accuracy in resolving dlopen and dlsym arguments when compared with sysfilter. Out of 12 `dlopen()` cases, SysPart can resolve 8 while sysfilter resolves none. Out of 16 `dlsym()` cases, SysPart can handle 9, but sysfilter only tackles 6.

We further count the system calls used by the dynamically loaded libraries and explore their impacts. In four applications (Httpd, Lighttpd, Memcached, and Nginx), the dynamic libraries require additional system calls. In those cases, it is essential to resolve the dynamic libraries. Otherwise, the system-call filter can break the

Table 4: List of equivalent system calls.

`execve`	`execveat`
`accept`	`accept4`
`dup`	`dup2`, `dup3`
`eventfd`	`eventfd2`
`chmod`	`fchmodat`
`recv`	`recvfrom`, `read`
`send`	`sendto`, `write`
`open`	`openat`
`select`	`pselect6`, `epoll_wait`, `epoll_wait_old`, `poll`, `ppoll`, `epoll_pwait`

normal functionality. As expected, taking the dynamically loaded libraries into account can reduce the security benefits. By further allowing the system calls listed in Table 7, we observe a 2.77% drop in kernel vulnerability mitigation on all configurations of Httpd and Lighttpd and a 5.47% drop when considering Memcached with the default configuration.

5.5 `execve` System Call

During the evaluation, SysPart statically identifies that `execve` is used to launch "/bin/sh" by Httpd and Lighttpd. The binary or shell script further executed by "/bin/sh" cannot be resolved statically. SysPart also finds that `execve` is used in Nginx and Redis while the arguments cannot be determined statically. Further running dynamic analysis under the default configurations, however, SysPart observes no use of `execve` by those four programs.

After a closer look via manual analysis, we find that our static analysis is not wrong. The four programs only invoke `execve` under special configurations. Specifically:

- Lighttpd: when configured to load CGI modules via `mod_cgi`, it invokes `execve` to run CGI scripts.
- Httpd: it can also be configured to launch CGI scripts via `execve`.
- Nginx: when requested to upgrade the server binary upon special signals, it uses `execve` to launch a new version of the server. This occurs outside the main loop[1] and does not impact our filter.
- Redis: when configured to run in the debug mode, it uses `execve` to restart the server upon request. Also, when configured to run in the sentinel mode, it uses `execve` to run pending scripts. These occur outside of the main loop[1] and does not happen in the default release mode. Thus, it has no impact on our filter.

To sum up, servers can use `execve` under certain configurations. To determine the arguments of `execve` in those cases, SysPart will need the users' help with setting up the desired configurations for its dynamic analysis.

5.6 Robustness and Efficiency

5.6.1 Compilers and Optimizations: The compiler and optimizations used to build an application affect the resulting binary. We evaluate the effects those on SysPart, by building our benchmarks using both GCC-7.5.0 and CLANG-6.0.0 under varying optimization settings. We apply SysPart on the resulting binaries and count the

[1]SysPart cannot filter `execve` in this case because the function calling `execve` is determined as an AT function. Due to approximation, SysPart considers that the AT function can be called by indirect calls inside the main loop.

Table 5: Percentage of shellcodes (total 535) stopped by SysPart (SP), sysfilter (SF), and TSP (original and after fixes). Subscripts *With* and *Without* indicate whether we consider equivalent system calls or not, respectively.

Application	SF_{With}	SP_{With}	TSP_{With}	$\text{TSP}_{\textbf{fixed}\ With}$	$\text{SF}_{Without}$	$\text{SP}_{Without}$	$\text{TSP}_{Without}$	$\text{TSP}_{\textbf{fixed}\ Without}$
Bind	20.37	67.85	67.47	71.96	33.45	71.40	76.82	73.45
Httpd	36.82	41.68	78.13	78.13	46.16	52.89	83.92	83.92
Lighttpd	32.14	58.69	60.37	60.37	34.01	60.37	62.05	62.05
Memcached	16.63	77.1	72.89	73.08	38.50	78.50	74.39	74.57
Nginx	12.89	30.28	45.42	47.85	34.95	53.83	68.22	68.41
Redis	33.08	49.53	49.15	49.53	38.31	65.23	65.98	65.23

Table 6: Security-sensitive system calls filtered by SysPart, $\text{TSP}_{\text{fixed}}$, and sysfilter.

Syscall	Bind	Httpd	Lighttpd	Memcached	Nginx	Redis
accept	✗	✓	✗	✗	✗	✗
accept4	✗	✗	✗	✗	✗	✓
bind	✗	○	○	✗	✗	✗
chmod	○	✓	◑	✓	✗	◑
clone	◑	✗	✗	✗	◑	✗
connect	✗	✗	✗	✗	✗	✗
execve	◑	○	✗	◑	○	✗
execveat	✓	✓	✓	✓	✓	✓
fork	✓	✓	✓	✓	✓	✓
listen	✗	○	◑	✗	●	◑
mprotect	✗	✗	✗	✗	✗	✗
ptrace	✓	✓	✓	✓	✓	✓
recvfrom	●	◑	✗	✗	✗	◑
setgid	○	○	◑	◑	✗	◑
setreuid	◑	◑	◑	◑	◑	◑
setuid	○	○	◑	◑	✗	◑
socket	✗	✗	✗	✗	✗	✗

✗: Filtered by none; ✓: Filtered by all; ◑: Filtered by all but sysfilter; ○: Filtered only by $\text{TSP}_{\text{fixed}}$; ●: Filtered only by SysPart.

number of allowed syscalls in the main loop. The evaluation results are presented in Table 8.

Overall, GCC and CLANG lead to nearly identical results. Only in the case of Redis, SysPart identifies one more syscall given GCC-compiled binaries. Throughout further analysis, we find that this is because GCC (since version 4.0) enforces `-D_FORTIFY_SOURCE=1` at optimization level O1 and above [2]. This will transform `longjmp` to a checked version `__longjmp_chk`, and the latter additionally needs the `sigaltstack` syscall. In contrast, CLANG does not set up `-D_FORTIFY_SOURCE` by default, avoiding the use of `sigaltstack`. We want to note that Redis compiled by GCC under O0 does not include `sigaltstack`. However, it presents one extra dummy AT function, which leads to the inclusion of the `creat` syscall. This is why that binary under O0 has the same amount of syscalls.

The optimization levels present no impact except for the cases of Memcached and Redis. Given Memcached compiled under optimization level O0 (with both GCC and CLANG), SysPart detects 13 more syscalls that are needed for the main loop. The reason—based on our inspection—is that the O1-O3 optimization levels more aggressively eliminate unreachable code, inline functions, and unroll loops, etc. These lead to a more accurate analysis by SysPart (in particular fewer AT functions) and thus, reduce the number of dummy syscalls. The detail about Redis has been discussed above.

5.6.2 *Binary-only Software:* To evaluate SysPart, we use open-source software so that we can verify the correctness of the results. To further assess its ability to handle binaries, we apply SysPart on a proprietary, closed-source web server, the Abyss Web Server v2.16 from Aprelium [1]. Unlike modern binaries, Abyss is not compiled as position-independent code (PIC)—which is necessary for benefiting from basic defenses like ASLR. This forces further over-approximation of AT functions and possibly inflates the number of required syscalls. This also breaks Egalito's functionality to correctly rewrite Abyss and inject the code setting up the Seccomp-BPF filter. To inject the filter, we use another static binary rewriter e9patch [16]. e9patch minimally alters the Abyss binary to add a trampoline to the Seccomp-BPF filter program before the main loop entrance. To make it work, we include the definition of `prctl()` to the libc tailored by e9patch. To run dynamic analysis, we profile Abyss with the default configurations and 10k HTTPD requests.

The evaluation finds that Abyss spawns six threads at startup for different services. SysPart detects a main loop for each thread, and we summarize the details in Table 9. SysPart reports that the `main()` of the server only requires 86 syscalls for correct operation, filtering security-sensitive syscalls including `accept4`, `chmod`, `execveat`, `ptrace`, `setreuid`, and `fork`. Narrowing down to the serving phase (i.e., the main loop), SysPart further filters out 4 more syscalls (`pipe`, `dup2`, `setsid`, and `execve`) for two threads and one more syscall (`setsid`) for the main thread. To further validate the stability of the server after the Seccomp-BPF filter is inserted, we rerun the patched server with the aforementioned 10k HTTP requests. It shows that the server works without exceptions.

We also inspect the CGI modules of Abyss. When enabled, similar to Lighttpd and Httpd, Abyss calls `execve` to launch "/bin/sh" for executing the CGI scripts. This explains why SysPart cannot filter `execve` for Abyss' request-processing thread.

5.6.3 *Analysis Speed:* We compare the execution times of the static analysis phase (call-graph construction and system-call set generation) between SysPart and TSP in Table 10. In all cases, SysPart runs at least 81% faster than TSP.

Table 7: Results of dynamic library analysis. For each application, we list the configuration tested and the number of `dlopen()`/`dlsym()` call sites that we statically resolved *fully*, *partially*, or *not at all*. The numbers in parentheses indicate the call sites that were also observed dynamically. We also list the number of additional syscalls added by the analysis for each configuration. The footnotes annotate the reasons why full resolution is not achieved.

Application	Configuration	dlopen()			dlsym()			+Syscalls
		Full	Partial	Unres.	Full	Partial	Unres.	
Bind	Default	4 (3)‡	0 (0)	1 (0)$^{\diamond}$	3 (3)	0 (0)	2 (0)$^{\diamond}$	0
	`+DLZ module`	4 (3)‡	0 (0)	1 (0)$^{\diamond}$	3 (3)	0 (0)	2 (0)$^{\diamond}$	0
Httpd	Default	0 (0)	0 (0)	1 (1)†	0 (0)	0 (0)	1 (1)†	3
	`+mod_ssl`	0 (0)	0 (0)	1 (1)†	0 (0)	0 (0)	1 (1)†	14
Lighttpd	Default	0 (0)	0 (0)	1 (1)†	0 (0)	0 (0)	1 (1)†	1
	`+mod_(cgi)`	0 (0)	0 (0)	1 (1)†	0 (0)	0 (0)	1 (1)†	1
Memcached	Default (w/`SASL`)	0 (0)	1 (1)$^{\diamond\ddagger}$	0 (0)	0 (0)	1 (1)$^{\dagger\diamond}$	0 (0)	13
Nginx	Default	2 (1)‡	0 (0)	1 (0) $^{\diamond}$	3 (3)	0 (0)	2 (0) $^{\diamond}$	0
	`+ngx_http_image_filter_module`	2 (1)‡	0 (0)	1 (0) $^{\diamond}$	3 (3)	0 (0)	2 (0) $^{\diamond}$	20
Redis	Default	1 (1)‡	0 (0)	0 (0)	3 (1)	0 (0)	0 (0)	0
	`+redis_cell`	1 (1)‡	0 (0)	0 (0)	3 (1)	0 (0)	0 (0)	0

†Read from configuration file. $^{\diamond}$Limitations of VFA. ‡Through heuristic based on `dlsym()` resolved symbols.

Table 8: Allowed system calls when building the benchmarks with different compilers and optimization levels. The underlined values correspond to the configurations tested in §5.2.

		Bind	Httpd	Lighttpd	Memcached	Nginx	Redis
GCC	O0	103	92	93	98	104	85
	O1	103	92	93	85	104	85
	O2	103	92	93	85	104	85
	O3	103	92	93	85	104	85
CLANG	O0	103	92	93	98	104	84
	O1	103	92	93	85	104	84
	O2	103	92	93	85	104	84
	O3	103	92	93	85	104	84

Table 9: Main loops detected in the Abyss web server and the number of syscalls required by them. Partition 0 is the main process and Partitions 1 to 5 are threads of child process. Thread in partition 3 spawns new threads to handle requests. `main()` requires 86 syscalls.

	Partition Address		Main Loop	Filtered
ID	Function	Main Loop	syscalls	syscalls
0	0x443da0	0x443dca	85	setsid
1	0x42edd0	0x42efea	82	pipe,dup2,execve, setsid
2	0x493330	0x4933e8	82	pipe,dup2,execve, setsid
3	0x466740	0x466771	86	–
4	0x405960	0x405970	86	–
5	0x459000	0x45906a	86	–

Table 10: Analysis time (in seconds) of SysPart and TSP.

Application	TSP	SysPart
Bind	562	4.06 (-99.27%)
Httpd	17	0.73 (-95.68%)
Lighttpd	3	0.56 (-81.10%)
Memcached	3	0.45 (-85.00%)
Nginx	83	5.01 (-93.95%)
Redis	23	1.23 (-94.63%)

5.6.4 FCG Improvements over Egalito: Comparing with Egalito, our FCG refinement using VFA and TypeArmor achieved a reduction in FCG edges of 8.34%, 6.68%, 2.36%, 3.93%, 6.74% and 10.36% in Bind, Httpd, Lighttpd, Memcached, Nginx and Redis, respectively.

6 Related Work

6.1 System-Call Interposition

System-call interposition [18, 23, 26] is an early research direction to restrict system calls. It interposes at interfaces between the application and the OS kernel to enforce security policies. Janus [23] leverages `ptrace` to dynamically monitor and restrict system calls that an application can perform. In contrast, Ostia [18] sandboxes an application and runs a delegate program to make system calls on behalf of the application according to a user-specified security policy. Besides intercepting system calls through a pair of kernel database and a user-space daemon, Systrace [48] further automates the generation of security policies using dynamic analysis.

6.2 System-Call Filtering

System-call filtering represents a more recent method to limit system calls. The idea is to identify the set of system calls required by the application and filter the unneeded ones at runtime through mechanisms like Seccomp-BPF. Constructing and mounting the filters is trivial. Thus, research in this line primarily focuses on determining the list of required system calls. Abhaya [43], Chestnut [9], Confine [19], and SYSFILTER [13] all rely on static analysis to over-approximate the set of system calls. Notably, Abhaya and SYSFILTER both consider the dependent libraries, with SYSFILTER being binary only. SYSFILTER also employs dynamic library profiling using static VFA, but our VFA with heuristics performs better in resolving `dlopen` and `dlsym` callsites as mentioned in section 5.4. In contrast, SIT [58] runs best-effort static analysis to get an initial allow-list of system calls and incorporates runtime monitoring to compensate for false negatives. BASTION [27] introduces the concept of "integrity" to system-call filtering. With BASTION, syscall invocations are bound by call type integrity, control-flow integrity and argument integrity. Adopting a similar principle, Shredder [37] derives the expected argument values of system APIs and specializes the APIs to only allow those values.

TSP [20] advances system-call filtering to be temporal. Instead of viewing an application as a whole, TSP separates the execution of the application into an initialization phase and a serving phase. It tailors different filter rules for different phases. As described in §3.3, TSP presents several major limitations to achieve practical temporal system-call filtering. Our system SYSPART, overcoming those limitations, offers the first solution to provide automated, binary-only, and robust temporal system-call filtering.

6.3 Code Debloating

Code debloating removes code not required by an application or its libraries during runtime, which can help reduce the attack surface. RAZOR [49] utilizes a set of test cases and control-flow-based heuristics to perform code reduction for deployed binaries, preserving only the essential code needed to support user-expected functionalities. Chisel [25] also reduces unneeded code from the application, but using machine learning with a high-level specification of the desired functionality. In contrast, CodeFreeze [40], Nibbler [6], BlankIt [47], and configuration-driven software debloating [32] focus on debloating library code, using static analysis and/or dynamic analysis. Specifically, CodeFreeze and Nibbler remove unused code from dynamic libraries at loading time, while configuration-driven software debloating [32] identifies the required libraries based on the configurations and disable the remaining by not loading them. BlankIt [47] uses a more fine-grained strategy by only loading the library functions that will be used at each call site at runtime. Different from these works, Piece-wise [50] combines compilation-time analysis and loading-time enforcement to only load code needed by the program, offering support for both applications and libraries. It combines a static and training based approach to resolve arguments to `dlopen` and `dlsym`, although no evaluation results are provided for this analysis.

7 Conclusion

We presented SYSPART, a semi-automatic system-call filtering system for binary-only server programs. SYSPART identifies the serving phases of applications threads, computes the set of system calls required by each, and installs an efficient Seccomp-BPF filter that disallows other system calls. We implemented SYSPART on x86-64 Linux and evaluated it on six popular server applications. The results demonstrate that SYSPART accurately locates the point where serving phase begins and performs comparably to prior source-code-based work [20], but without errors. Moreover, SYSPART only allows 8.33% more syscalls overall, while it filters as many security-critical syscalls in 88.23% of cases. Unsurprisingly, it also outperforms prior work [13] on binaries that does not consider execution phases. In terms of security, SYSPART is successful in blocking exploit payloads and preventing kernel vulnerabilities with success rates ranging from 53.83% to 78.5% and 36.11% to 77.77%, respectively, for the ones tested. Finally, SYSPART surpasses prior work [13] in soundly resolving instances of `dlopen()` and `dlsym()` by 58.33% and 18.75%, respectively.

Acknowledgments

We thank the anonymous reviewers for their valuable comments and time. This work was supported by the Office of Naval Research (ONR) awards N00014-17-1-2788, N00014-22-1-2643, and N00014-17-1-2787, DARPA award D21AP10116-00, and National Science Foundation (NSF) awards CNS-2213727 and OAC-2319880. Any opinions, findings, and conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the US government, ONR, DARPA, or NSF.

References

[1] [n. d.]. Aprelium. https://aprelium.com/.
[2] [n. d.]. FORTIFY_SOURCE Semantics. https://hockeyinjune.medium.com/fortify-source-semantics-de54ca4bbe12.
[3] 2018. Navy - Protocol Feature Identification and Removal. https://www.navysbir.com/n18_A/N18A-T018.htm.
[4] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2005. Control-Flow Integrity. In *Proceedings of the ACM Conference on Computer and Communications Security*. 340–353.
[5] Ioannis Agadakos, Nicholas Demarinis, Di Jin, Kent Williams-King, Jearson Alfajardo, Benjamin Shteinfeld, David Williams-King, Vasileios P. Kemerlis, and Georgios Portokalidis. 2020. Large-Scale Debloating of Binary Shared Libraries. *Digital Threats: Research and Practice (DTRAP)* 1, 4, Article 19 (Dec. 2020), 28 pages. https://dl.acm.org/doi/pdf/10.1145/3414997
[6] Ioannis Agadakos, Di Jin, David Williams-King, Vasileios P. Kemerlis, and Georgios Portokalidis. 2019. Nibbler: Debloating Binary Shared Libraries. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)* (San Juan, Puerto Rico).
[7] Starr Andersen and Vincent Abella. 2004. Changes to Functionality in Microsoft Windows XP Service Pack 2, Part 3: Memory Protection Technologies, Data Execution Prevention. Microsoft TechNet Library–http://technet.microsoft.com/en-us/library/bb457155.aspx.
[8] Apache-Test. [n. d.]. Test suite for Apache. https://metacpan.org/dist/Apache-Test.
[9] Claudio Canella, Mario Werner, Daniel Gruss, and Michael Schwarz. 2021. Automating Seccomp Filter Generation for Linux Applications. In *Proceedings of the 2021 on Cloud Computing Security Workshop* (Virtual Event, Republic of Korea) *(CCSW '21)*. Association for Computing Machinery, New York, NY, USA, 139–151. https://doi.org/10.1145/3474123.3486762
[10] Crispin Cowan, Calton Pu, Dave Maier, Heather Hinton, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, et al. 1998. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Symposium*, Vol. 81. 346–355.
[11] National Vulnerability Database. 2019. BlueKeep Vulnerability (CVE-2019-0708). NIST. https://nvd.nist.gov/vuln/detail/CVE-2019-0708
[12] Bjorn De Sutter, Ludo Van Put, and Koen De Bosschere. 2007. A Practical Interprocedural Dominance Algorithm. *ACM Trans. Program. Lang. Syst.* 29, 4 (aug 2007), 19–es. https://doi.org/10.1145/1255450.1255452
[13] Nicholas DeMarinis, Kent Williams-King, Di Jin, Rodrigo Fonseca, and Vasileios P. Kemerlis. 2020. sysfilter: Automated System Call Filtering for Commodity Software. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*. USENIX Association, San Sebastian, 459–474. https://www.usenix.org/conference/raid2020/presentation/demarinis

[14] Solar Designer. [n. d.]. Getting around non-executable stack (and fix). https://seclists.org/bugtraq/1997/Aug/63.

[15] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. 2020. RetroWrite: Statically Instrumenting COTS Binaries for Fuzzing and Sanitization. In *IEEE Symposium on Security and Privacy (S&P)*. 128–142.

[16] Gregory J. Duck, Xiang Gao, and Abhik Roychoudhury. 2020. Binary Rewriting without Control Flow Recovery. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 151–163.

[17] Charles N. Fischer. [n. d.]. Finding Loops in Control Flow Graphs. https://pages.cs.wisc.edu/~fischer/cs701.f14/finding.loops.html.

[18] Tal Garfinkel, Ben Pfaff, Mendel Rosenblum, et al. 2004. Ostia: A Delegating Architecture for Secure System Call Interposition.. In *NDSS*.

[19] Seyedhamed Ghavamnia, Tapti Palit, Azzedine Benameur, and Michalis Polychronakis. 2020. Confine: Automated System Call Policy Generation for Container Attack Surface Reduction. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*. USENIX Association, San Sebastian, 443–458.

[20] Seyedhamed Ghavamnia, Tapti Palit, Shachee Mishra, and Michalis Polychronakis. 2020. Temporal System Call Specialization for Attack Surface Reduction. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 1749–1766. https://www.usenix.org/conference/usenixsecurity20/presentation/ghavamnia

[21] Enes Göktaş, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. 2014. Out Of Control: Overcoming Control-Flow Integrity. In *Proceedings of the IEEE Symposium on Security and Privacy* (San Jose, CA, USA). 575–589.

[22] Enes Göktaş, Robert Gawlik, Benjamin Kollenda, Elias Athanasopoulos, Georgios Portokalidis, Cristiano Giuffrida, and Herbert Bos. 2016. Undermining Entropy-based Information Hiding (And What to do About it). In *Proceedings of the USENIX Security Symposium* (Austin, TX, USA). 105–119.

[23] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. 1996. A Secure Environment for Untrusted Helper Applications Confining the Wily Hacker. In *Proceedings of the 6th Conference on USENIX Security Symposium, Focusing on Applications of Cryptography - Volume 6* (San Jose, California) *(SSYM'96)*. USENIX Association, USA, 1.

[24] Norm Hardy. 1988. The Confused Deputy: (Or Why Capabilities Might Have Been Invented). *SIGOPS Oper. Syst. Rev.* 22, 4 (oct 1988), 36–38.

[25] Kihong Heo, Woosuk Lee, Pardis Pashakhanloo, and Mayur Naik. 2018. Effective Program Debloating via Reinforcement Learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 380–394. https://doi.org/10.1145/3243734.3243838

[26] K. Jain and R. C. Sekar. 2000. User-Level Infrastructure for System Call Interposition: A Platform for Intrusion Detection and Confinement. In *Network and Distributed System Security Symposium*.

[27] Christopher Jelesnianski, Mohannad Ismail, Yeongjin Jang, Dan Williams, and Changwoo Min. 2023. Protect the System Call, Protect (Most of) the World with BASTION. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 528–541. https://doi.org/10.1145/3582016.3582066

[28] Vasileios P. Kemerlis. 2015. *Protecting Commodity Operating Systems through Strong Kernel Isolation.* Ph. D. Dissertation. Columbia University.

[29] Vasileios P. Kemerlis, Michalis Polychronakis, and Angelos D. Keromytis. 2014. ret2dir: Rethinking Kernel Isolation. In *USENIX Security Symposium (SEC)*. 957–972.

[30] Vasileios P. Kemerlis, Georgios Portokalidis, and Angelos D. Keromytis. 2012. kGuard: Lightweight Kernel Protection against Return-to-user attacks. In *USENIX Security Symposium (SEC)*. 459–474.

[31] The kernel development community. [n. d.]. Seccomp BPF (SECure COMPuting with filters). https://www.kernel.org/doc/html/v4.19/userspace-api/seccomp_filter.html. ([n. d.]).

[32] Hyungjoon Koo, Seyedhamed Ghavamnia, and Michalis Polychronakis. 2019. Configuration-Driven Software Debloating. In *Proceedings of the 12th European Workshop on Systems Security* (Dresden, Germany) *(EuroSec '19)*. Association for Computing Machinery, New York, NY, USA, Article 9, 6 pages. https://doi.org/10.1145/3301417.3312501

[33] Yiwen Li, Brendan Dolan-Gavitt, Sam Weber, and Justin Cappos. 2017. Lock-in-Pop: Securing Privileged Operating System Kernels by Keeping on the Beaten Path. In *USENIX Annual Technical Conference (ATC)*. 1–13.

[34] Edward S. Lowry and C. W. Medlock. 1969. Object Code Optimization. *Commun. ACM* 12, 1 (jan 1969), 13–22. https://doi.org/10.1145/362835.362838

[35] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proc of the Conference on Programming Language Design and Implementation (PLDI)*. 190–200.

[36] LWN.net. 2020. Seccomp and deep argument inspection. https://lwn.net/Articles/822256/.

[37] Shachee Mishra and Michalis Polychronakis. 2018. Shredder: Breaking Exploits through API Specialization. In *Annual Computer Security Applications Conference (ACSAC)*. 1–16.

[38] MITRE. 2013. CVE-2013-1858. https://nvd.nist.gov/vuln/detail/CVE-2013-1858.

[39] MITRE. 2013. CVE-2013-2028. https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2013-2028.

[40] Collin Mulliner and Matthias Neugschwandtner. 2015. Breaking Payloads with Runtime Code Stripping and Image Freezing. https://www.blackhat.com/us-15/briefings.html#breaking-payloads-with-runtime-code-stripping-and-image-freezing.

[41] Network Service Switch. [n. d.]. Linux manual page. https://man7.org/linux/man-pages/man5/nss.5.html.

[42] nginx-tests. [n. d.]. Test suite for Nginx. https://github.com/nginx/nginx-tests.

[43] Shankara Pailoor, Xinyu Wang, Hovav Shacham, and Isil Dillig. 2020. Automated Policy Synthesis for System Call Sandboxing. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 135 (nov 2020), 26 pages. https://doi.org/10.1145/3428203

[44] PaX Team. 2003. Address Space Layout Randomization (ASLR). https://pax.grsecurity.net/docs/aslr.txt.

[45] Marios Pomonis, Theofilos Petsios, Angelos D. Keromytis, Michalis Polychronakis, and Vasileios P. Kemerlis. 2017. kR^X: Comprehensive Kernel Protection against Just-In-Time Code Reuse. In *European Conference on Computer Systems (EuroSys)*. 420–436.

[46] Marios Pomonis, Theofilos Petsios, Angelos D. Keromytis, Michalis Polychronakis, and Vasileios P. Kemerlis. 2019. Kernel Protection against Just-In-Time Code Reuse. *ACM Transactions on Privacy and Security (TOPS)* 22, 1 (2019), 1–28.

[47] Chris Porter, Girish Mururu, Prithayan Barua, and Santosh Pande. 2020. BlankIt Library Debloating: Getting What You Want Instead of Cutting What You Don't. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 164–180. https://doi.org/10.1145/3385412.3386017

[48] Niels Provos. 2003. Improving Host Security with System Call Policies.. In *USENIX Security Symposium*. 257–272.

[49] Chenxiong Qian, Hong Hu, Mansour Alharthi, Pak Ho Chung, Taesoo Kim, and Wenke Lee. 2019. RAZOR: A Framework for Post-deployment Software Debloating. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 1733–1750. https://www.usenix.org/conference/usenixsecurity19/presentation/qian

[50] Anh Quach, Aravind Prakash, and Lok Yan. 2018. Debloating Software through Piece-Wise Compilation and Loading. In *Proceedings of the 27th USENIX Conference on Security Symposium* (Baltimore, MD, USA) *(SEC'18)*. USENIX Association, USA, 869–886.

[51] Ganesan Ramalingam. 1994. The Undecidability of Aliasing. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 16, 5 (1994), 1467–1471.

[52] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. 2015. Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications. In *IEEE Symposium on Security and Privacy (S&P)*. 745–762.

[53] Hovav Shacham. 2007. The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86). In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 552–561.

[54] László Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. SoK: Eternal War in Memory. In *IEEE Symposium on Security and Privacy (S&P)*. 48–62.

[55] Victor van der Veen, Enes Göktaş, Moritz Contag, Andre Pawoloski, Xi Chen, Sanjay Rawat, Herbert Bos, Thorsten Holz, Elias Athanasopoulos, and Cristiano Giuffrida. 2016. A Tough Call: Mitigating Advanced Code-Reuse Attacks at the Binary Level. In *2016 IEEE Symposium on Security and Privacy (SP)*. 934–953. https://doi.org/10.1109/SP.2016.60

[56] David Williams-King, Hidenori Kobayashi, Kent Williams-King, Graham Patterson, Frank Spano, Yu Jian Wu, Junfeng Yang, and Vasileios P. Kemerlis. 2020. Egalito: Layout-Agnostic Binary Recompilation. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 133–147. https://doi.org/10.1145/3373376.3378470

[57] Yves Younan, Wouter Joosen, and Frank Piessens. 2012. Runtime Countermeasures for Code Injection Attacks against C and C++ Programs. *ACM Computing Surveys (CSUR)* 44, 3 (2012), 1–28.

[58] Qiang Zeng, Zhi Xin, Dinghao Wu, Peng Liu, and Bing Mao. 2014. *Tailored Application-specific System Call Tables.* Technical Report.

A Appendix

A.1 TSP Issues Leading to Missing Required System Calls

A.1.1 Cannot handle certain inline assembly constructs : In glibc, inline assembly code is used to invoke system calls using `syscall` instruction. In most cases, macro `INLINE_SYSCALL` is used to invoke the system calls which is recognized by TSP. But there are some patterns which are not recognized by TSP due to which the system calls are missed by TSP.

A.1.2 Does not handle `syscall()` function : System calls can be invoked using the `syscall()` function. The first argument to the `syscall()` function is the system call number that identifies the system call. TSP detects `syscall()` but cannot determine the system call number and hence misses the corresponding system call.

A.1.3 Does not handle certain function aliases in glibc : Glibc uses internal alias functions. The glibc callgraph produced by TSP using Egypt tool has callpaths which are incomplete. This causes TSP to miss the following system calls.

For example, in the glibc callgraph generated by TSP, `luaL_loadfile()` invokes `freopen()`, which invokes `*__GI__dup3()`. `dup3()` invokes `__dup3()` which invokes `dup3` system call. But there is no outgoing edge for *__GI__dup3() in TSP glibc callgraph and hence the path to `dup3` system call is missed by TSP in Redis.

A.1.4 Ignores AT functions in initialization and dynamic-linker code : TSP doesn't consider the functions which are AT in the initialization section of the application or libraries as well as in the dynamic linker. This causes it to miss certain system calls that are reachable from these AT functions.

A.1.5 Does not handle function pointers passed to library functions : In TSP, the callgraphs of the application and the shared libraries are generated separately. Hence, in cases where there are AT functions which are passed as arguments to shared libraries, the callgraph generation algorithm of TSP doesn't generate any outgoing edges for those AT functions as a result of which the functions that are invoked from those AT functions are missed by TSP. An example is signal handler functions that are passed as argument to `sigaction()`. In Nginx, `ngx_signal_handler` is a signal handler function that is passed as argument to `sigaction()`. `ngx_signal_handler` has a callpath leading to the system call wait4, which is missed by TSP.

A.1.6 Does not handle glibc NSS libraries : TSP doesn't resolve NSS functions and hence misses out on the system calls reachable from these functions. An example is `sendmmsg` in Lighttpd.

A.1.7 Library not considered due to missing CFG : In certain cases when CFG of a library is not already generated, TSP tries to find the exported symbols of the library. But if the library is not present in the system at all, then it misses out on all the system calls which are reachable from functions in that library.

A.2 TSP Issues Leading to Additional Allowed System Calls

A.2.1 Source-code analysis (Egypt) cannot differentiate between multiple libraries in the same directory and merges all of them in the resulting glibc FCG : The glibc callgraph produced by Egypt in TSP includes functions from unused modules. This results in including callpaths which are not necessarily accessible. For example, glibc contains malloc/memusage.c, within which function interposition is used to invoke `malloc(), calloc(), free()` etc for memory profiling. This is compiled into `libmemusage.so`, and the functions within this shared library can be invoked only if this library is preloaded by setting the environmental variable `LD_PRELOAD`. TSP callgraph contains this callpath,
`malloc()->me()->creat64()->creat()-> syscall(85)`
This causes the system call `creat` to be included by TSP whenever `malloc()` is used by an application or shared library, while this callpath will only be invoked if libmemusage.so is enabled, which is not used by any of the servers.

A.2.2 Implementation erroneously considers certain null nodes as actual functions and uses them during analysis : In the glibc callgraph used by TSP there are outgoing NULL edges and incoming NULL edges. This cause their traversal algorithm to traverse parts of glibc callgraph which are not actually reachable.

System calls `arch_prctl` and `set_tid_address` are included by TSP in all servers due to the following paths from the glibc callgraph.

```
  _pthread_cleanup_push -> NULL
NULL -> __pthread_initialize_minimal_internal
__pthread_initialize_minimal_internal -> __libc_setup_-
tls
__libc_setup_tls -> arch_prctl (syscall)

  _pthread_cleanup_push -> NULL
NULL -> __pthread_initialize_minimal_internal
__pthread_initialize_minimal_internal -> set_tid_address
(syscall)
```

A.2.3 TSP including all exported functions as AT functions in cases where callgraph of libraries are not present : If the callgraph of libraries are not present, TSP includes all exported functions within the library as AT functions.

A.2.4 Inefficient Pruning Algorithm : TSP employs a pruning algorithm to prune out function pointers which are address taken in functions which are inaccessible from main. But the list of AT functions that are used as input for this algorithm is incomplete and doesn't include all the AT functions used to generate the callgraph, as a result of which the algorithm misses out on pruning certain AT functions which are not reachable from main, and thereby including the system calls from these AT functions.

A.3 Pruning of Indirect-Call Targets in TSP :

TSP uses SVF implementation of Andersen's algorithm to produce callgraphs of applications. SVF Andersen's considers the number of arguments while resolving indirect call targets. TSP further refines the callgraph by matching the indirect callsites with functions based on argument types, more specifically struct argument types. Also, it prunes those indirect call edges to those AT functions which are address taken in paths which are inaccessible from main. These pruned edges causes TSP to remove some system calls which are accessible using these edges.

Table 11: All main loops detected by SysPart, including the ones for auxiliary threads/processes.

Application	**Server in Default config**	**Partitions detected by SysPart (No: of Syscalls)**
Bind	Single-process Multi-threaded	`main + 0x13701` (103)
		`netthread + 0x105` (112)
		`nm_thread + 0x99` (112)
		`run + 0x358`[†] (112)
		`run + 0x119`[*] (112)
Httpd	Multi-process Multi-threaded	`ap_build_config + 0x402` (92)
		`ap_run_mpm + 0x103` (92)
		`child_main + 0x5170` (92)
		`listener_thread + 0x274` (92)
		`start_threads + 0x360` (92)
		`worker_thread + 0x388` (92)
Lighttpd	Single-process Single-threaded	`main + 0x306` (93)
Memcached	Single-process Multi-threaded	`main + 0x26742` (85)
		`logger_thread + 0x100` (82)
		`event_base_loop + 0x391` (85)
		`assoc_maintenance_thread + 0x50` (82)
		`item_crawler_thread + 0x276` (82)
		`lru_maintainer_thread + 0x579` (82)
		`slab_rebalance_thread + 0x100` (82)
Nginx	Multiple-process Single-threaded	`ngx_master_process_cycle + 0x5456` (106)
		`ngx_worker_process_cycle + 0x391` (104)
Redis	Single-process Multi-threaded	`aeMain + 0x22` (85)
		`bioProcessBackgroundJobs + 0x581` (85)

[†]: `isc/timer.c`, [*]: `isc/task.c`

A.4 All Main Loops Detected by SysPart

Table 11 lists all the main loops detected by SysPart.

A.5 Extensions to Egalito

Besides identifying and fixing bugs, we incorporated the following extensions into Egalito to support our needs.

(1) Added loop identification (§4.2).
(2) Added VFA analysis for the following :
  (a) To resolve indirect-branch targets (§4.4) for callgraphs. This completely resolved an additional 2.84% of indirect branches in the evaluated servers (a 4.24% reduction in FCG edges).
  (b) To resolve arguments for `dlopen`, `dlsym`, `execve` (§4.5).
  (c) To identify thread start functions (§4.7.2).
(3) Improved non-returning function detection (§4.7.2), discovering an average of 41% more such functions.
(4) Added a new pass to insert the syscall filter before the main loop's start (§4.8.3).