`sysfilter`: Automated System Call Filtering for Commodity Software

Nicholas DeMarinis Kent Williams-King Di Jin
Rodrigo Fonseca Vasileios P. Kemerlis

Department of Computer Science
Brown University

Abstract

Modern OSes provide a rich set of services to applications, primarily accessible via the system call API, to support the ever growing functionality of contemporary software. However, despite the fact that applications require access to part of the system call API (to function properly), OS kernels allow full and unrestricted use of the entire system call set. This not only violates the principle of least privilege, but also enables attackers to utilize extra OS services, after seizing control of vulnerable applications, or escalate privileges further via exploiting vulnerabilities in less-stressed kernel interfaces.

To tackle this problem, we present `sysfilter`: a binary analysis-based framework that automatically (1) limits what OS services attackers can (ab)use, by enforcing the principle of least privilege with respect to the system call API, and (2) reduces the attack surface of the kernel, by restricting the system call set available to userland processes. We implement `sysfilter` for x86-64 Linux, and present a set of program analyses for constructing system call sets statically, and in a scalable, precise, and complete (safe over-approximation) manner. In addition, we evaluate our prototype in terms of correctness using 411 binaries (real-world C/C++ applications) and ≈38.5K tests to assert their functionality. Furthermore, we measure the impact of our enforcement mechanism(s), demonstrating minimal, or negligible, run-time slowdown. Lastly, we conclude with a large scale study of the system call profile of ≈30K C/C++ applications (from Debian `sid`), reporting insights that justify our design and can aid that of future (system call-based) policing mechanisms.

1 Introduction

Software is continuously growing in complexity and size. `/bin/true`, the "tiny" command typically used as aid in shell scripts, was first introduced in the 7th edition of the Unix distribution (Bell Labs) and consisted of *zero* lines of code (LOC); by 2012, in Ubuntu, `true` has grown up to 2.3 KLOC [28]. Likewise, the `bash` binary has gone from 11.3 KB (Unix V5, 1974) up to 2.1 MB (Ubuntu, 2014) [28].

This constant stream of additional functionality integrated into modern applications, i.e., *feature creep*, not only has dire effects in terms of security and protection [1, 71], but also necessitates a rich set of OS services: applications need to interact with the OS kernel—and, primarily, they do so via the system call (syscall) API [52]—in order to perform useful tasks, such as acquiring or releasing memory, spawning and terminating additional processes and execution threads, communicating with other programs on the same or remote hosts, interacting with the filesystem, and performing I/O and process introspection.

Indicatively, at the time of writing, the Linux kernel (v5.5) provides support for 347 syscalls in x86-64. However, not every application requires access to the complete syscall set; e.g., ≈45%/65% of all (C/C++) applications in Debian "`sid`" (development distribution) [86] do not make use of `execve`/`listen` in x86-64 Linux. In other words, roughly one-half of these applications do not require the ability (and should not be permitted) to invoke other programs or accept network connections. Alas, the OS kernel provides *full* and *unrestricted* access to the entirety set of syscalls. This is not only a violation of the *principle of least privilege* [76], but also allows attackers to: (a) utilize additional OS services after seizing control of vulnerable applications [69], and (b) escalate their privileges further via exploiting vulnerabilities in unused, or less-stressed, OS kernel interfaces [33–35, 43, 65, 66].

To mitigate the effects of (a) and (b) above, we present `sysfilter`: a framework to (automatically) limit what OS services attackers can (ab)use, by enforcing the principle of least privilege with respect to the syscall API [69], and reduce the *attack surface* of the OS kernel, by restricting the syscall set available to userland processes [43]. `sysfilter` consists of two parts: system call set extraction and system call set enforcement. The former receives as input a target application, in binary form, automatically resolves dependencies to dynamic shared libraries, constructs a *safe*—but *tight*—over-approximation of the program's function-call graph (FCG), and performs a set of program analyses atop the FCG, in order to extract the set of developer-intended syscalls.

The latter enforces the extracted set of syscalls, effectively sandboxing the input binary. We implemented `sysfilter` in x86-64 Linux, atop the Egalito framework [95], while our program analyses, crafted as "passes" over Egalito's intermediate representation, are *scalable*, *precise*, and *complete*. `sysfilter` can extract a tight over-approximation of the set of developer-intended syscalls for ≈90% of all C/C++ applications in Debian `sid`, in less than 200s (or for ≈50% of these apps in less than 30s; § 5). Moreover, `sysfilter` requires no source code (i.e., it operates on stripped binaries, compiled using modern toolchains [26, 67]), and can sandbox programs that consist of components written in different languages (e.g., C, C++) or compiled-by different frameworks (GCC, LLVM). Importantly, `sysfilter` does not rely (on any form of) dynamic testing, as the results of this approach are usually both unsound and incomplete [62].

Further, we evaluate `sysfilter` across three dimensions: (1) correctness, (2) performance overhead, and (3) effectiveness. As far as (1) is concerned, we used 411 binaries from various packages/projects, including the GNU Coreutils (100, 672), SPEC CINT2006 (12, 12), SQLite (7, 31190), Redis (6, 81), Vim (3, 255), Nginx (1, 356), GNU M4 (1, 236), GNU Wget (1, 130). MariaDB (156, 2059), and FFmpeg (124, 3756), to extract and enforce their corresponding syscall sets; once sandboxed, we stress-tested them with ≈38.5K tests. In all cases, `sysfilter` managed to extract a complete and tight over-approximation of the respective syscall sets, demonstrating that our prototype can successfully handle complex, real-world software. (The numbers A, B in parentheses denote the number of binaries analyzed/enforced and the number of tests used to stress-test them, respectively.)

Regarding (2), we used SPEC CINT2006, Nginx, and Redis—i.e., 19 program binaries in total. In all cases, the sandboxed versions exhibited minimal, or negligible, run-time slowdown due to syscall filtering; we explored a plethora of different settings and configurations, including interpreted vs. JIT-compiled filters, and filter code that implements sandboxing using a linear search (within the respective syscall set) vs. filter code that utilizes a skip list-based approach. Lastly, with respect to (3), we investigated how `sysfilter` can reduce the attack surface of the OS, by inquiring what percentage of all C/C++ applications in Debian `sid` (≈30K binaries in total) can exploit 23 Linux kernel vulnerabilities after hardened with `sysfilter`. Although `sysfilter` does not defend against control- or data-flow hijacking [87] our results demonstrate that it can mitigate attacks by means of least privilege enforcement and (OS) attack surface reduction.

We conclude our work with a large scale study of the syscall sets of ≈30K C/C++ applications (Debian `sid`), reporting insights regarding the syscall set sizes (i.e., the number of syscalls per binary), most- and least-frequently used syscalls, syscall site distribution (libraries vs. main binary), and more. The results of this analysis not only guide our design, but can also aid that of future syscall policing mechanisms.

2 Background and Threat Model

Adversarial Capabilities In this work, we consider userland applications that are written in memory-unsafe languages, such as C/C++ and assembly (ASM). The attacker can trigger vulnerabilities, either in the main binaries of the applications or in the various libraries the latter are using, resulting in memory corruption [87]. Note that we do not restrict ourselves to specific kinds of vulnerabilities (e.g., stack- or heap-based memory errors, or, more generally, spatial or temporal memory safety bugs) [59, 60] or exploitation techniques (e.g., code injection, code reuse) [13, 78, 79, 87, 97].

More specifically, the attacker can: (a) trigger memory safety-related vulnerabilities in the target application, multiple times if needed, and construct and utilize exploitation primitives, such as arbitrary memory writes [16] and reads [81]; and (b) use, or combine, such primitives to tamper-with critical data (e.g., function and `vtable` pointers, return addresses) for hijacking the control flow of the target application and achieve arbitrary code execution [83] via means of code injection [97] or code reuse [10, 13, 24, 29, 32, 78, 79]. In terms of adversarial capabilities, our threat model is on par with the state of the art in C/C++/ASM exploitation [41, 87]. Lastly, we assume that the target applications consist of *benign* code: i.e., they do not contain malicious components.

Hardening Assumptions The primary focus of this work is modern, x86-64 Linux applications, written in C, C++, or ASM (or any combination thereof), and compiled in a *position-independent* [64] manner via toolchains that (by default) *do not mix code and data* [3, 4], such as GCC and LLVM.[1] In addition to the above, we assume the presence of *stack unwinding information* (`.eh_frame` section) in the ELF [12] files that constitute the target applications.

In § 3, we explain in detail the reasons for these two requirements—i.e., position-independent code (`PIC`) and `.eh_frame` sections. However, note that (a) `PIC` is enabled by default in modern Linux distributions [14, 95], while (b) `.eh_frame` sections are present in modern GCC- and LLVM-compiled code [3, 95]. The main reason for (a) is full ASLR (Address Space Layout Randomization): in position-dependent executables ASLR will only randomize the process stack and `mmap`- and `brk`-based heap [7]. Moreover, `PIC` in x86-64 incurs negligible performance overhead due to the existence of `PC`-relative data transfer instructions (`%rip`-relative `mov`) and extra general-purpose registers (16 vs. 8 in x86). As far as (b) is concerned, stack unwinding information is mandated by C++ code for exception handling [49], while both GCC and LLVM emit `.eh_frame` sections even for C code to support interoperability with C++ [95] and various features of certain `libc` (C library) implementations—e.g., `backtrace` in `glibc` (GNU C Library).

[1] Andriesse et al. [4], and Alves-Foss and Song [3], have independently verified that modern versions of both GCC and LLVM *do not* mix code and data. `icc` (Intel C++ Compiler) still embeds data in code [3].

Lastly, we assume a Linux kernel with support for seccomp-BPF (SECure COMPuting with filters) [36]. (All versions $\geq$ v3.5 provide support for seccomp-BPF in x86-64.) Every other standard userland hardening feature (e.g., NX, ASLR, stack-smashing protection) is orthogonal to sysfilter; our proposed scheme does not require nor preclude any such feature. The same is also true for less-widespread mitigations, like CFI [9,96], CPI [40,53], code randomization/diversification [38,41,94], and protection against data-only attacks [68].

3 Design and Implementation

Approach sysfilter aims at mitigating the effects of application compromise by *restricting access to the syscall API* [52]. The benefits of this approach are twofold: (1) it limits post-exploitation capabilities [69], and (2) it prevents compromised applications from escalating their privileges further via exploiting vulnerabilities in unused, or less-stressed, kernel interfaces [33–35, 43, 65, 66].

The main idea behind (1) is that applications need to interact with the kernel—and they primarily do so via the syscall API—in order to perform useful tasks. Indicatively, at the time of writing, the Linux kernel (v5.5) provides support for 347 syscalls in x86-64. (This number does *not* include the syscalls needed for executing 32-bit x86 applications atop a 64-bit kernel, or 64-bit processes that adhere to the x32 ABI [50].) However, despite the fact that applications only require access to part of the aforementioned API to function properly (e.g., non-networked applications do not need access to the socket-related syscalls), the OS kernel provides full and unrestricted access to the entirety set of syscalls.

This approach violates the *principle of least privilege* [76] and enables attackers to utilize additional OS services after seizing control of vulnerable applications. By restricting access to certain syscalls, sysfilter naturally limits what OS services attackers can (ab)use and enforces the principle of least privilege with respect to the syscall API: i.e., programs are allowed to issue only developer-intended syscalls.

As far as (2) is concerned, multiple studies have repeatedly divulged that the exploitation of vulnerabilities in kernel (or in even lower-level, more-privileged [19,99]) code is an essential part of privilege escalation attacks [33–35, 43, 65, 66]. To this end, sysfilter reduces the *attack surface* of the OS kernel, by restricting the syscall set available to userland processes, effectively providing *defense-in-depth* protection.

Overview sysfilter consists of two parts (see Figure 1): (1) a syscall-set *extraction* component; and (2) a syscall-set *enforcement* component. The former receives as input the target application in binary form (ELF file), automatically resolves all dependencies to dynamic shared libraries (.so ELF objects), and constructs a *safe* over-approximation of the program's FCG—across all objects in scope. Finally, it performs a set of program analyses atop FCG, in order to make the over-approximation as tight as possible and construct the syscall set in question. Note that the tasks above are performed *statically* and the syscall set returned by the extraction tool is *complete*: i.e., under any given input, the syscalls performed by the corresponding process are guaranteed to exist in the syscall set—this includes syscalls that originate from the binary itself, libc, or any other dynamically-loaded shared library. The latter part enforces the extracted set of syscalls, effectively *sandboxing* the input binary. Specifically, given a set of syscall numbers, the enforcement tool converts them to a BPF program [54] to be used with seccomp-BPF [36].

3.1 System Call Set Extraction

3.1.1 Analysis Scope

The input to the syscall-set extraction component of sysfilter is an (x86-64) ELF file that corresponds to the main binary of the application (see Figure 1A). sysfilter requires PIC as input, which is the default setting in modern Linux distributions [14, 95]. Once sysfilter verifies that the main binary is indeed PIC, adds it to the *analysis scope*, and proceeds to resolve dependencies regarding dynamic shared libraries. This is accomplished by first checking the PT_INTERP header, which conventionally contains the path of the respective dynamic linker/loader (e.g., /lib64/-ld-linux-x86-64.so), followed by iterating the .dynamic ELF section for DT_NEEDED entries that correspond to the names of the required shared libraries. Each of these libraries is added to the analysis scope and their .dynamic section is also scanned, recursively, to recover additional (library) dependencies. The process stops when all *explicit* dynamic library dependencies are resolved and the related ELF files have been added to the analysis scope.

In addition to the above, it is also possible to provide as input a set of *implicit* dynamic shared object dependencies to sysfilter: i.e., a list of additional .so ELF files that need to be added to the analysis scope, irrespectively of whether they exist in any of the loaded objects' .dynamic section (see Figure 1B). This functionality is important in order to support the analysis of binaries that have run-time dependencies to shared objects (e.g., via dlopen) or use LD_PRELOAD.

3.1.2 Function-Call Graph Construction

Once every ELF object is added to the analysis scope, sysfilter proceeds with the construction of the *function-call graph* (FCG) of the *whole* program. The FCG contains parts of the code (functions) that are reachable, under any possible input to the corresponding process. Note that for every included .so ELF object in the analysis scope, not all of their code is used: e.g., applications that link with libc, libpthread, libdl, *etc.*, do not make full use of the latter; usually, only part of library functionality is utilized [1, 71].

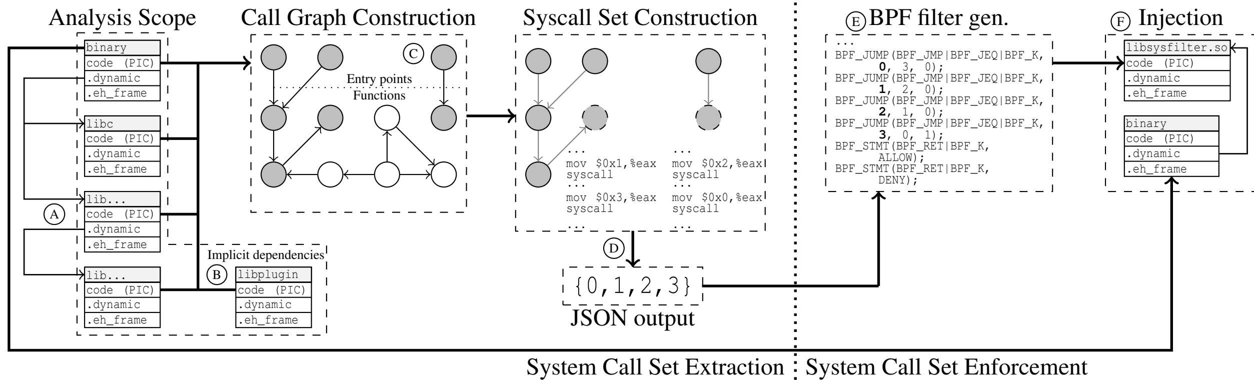

Figure 1: **The `sysfilter` Architecture.** The tool consists of two parts: the system call set extraction part (left) and the system call set enforcement part (right). The former receives as input the target application in binary form, automatically resolves all dependencies to dynamic shared libraries, constructs a safe over-approximation of the program's FCG—across all objects in scope—, and, finally, performs a set of program analyses atop the FCG, in order to extract the set of developer-intended syscalls. The latter enforces the extracted set of syscalls, effectively sandboxing the input binary, using `seccomp-BPF`.

Precise Disassembly Obtaining the complete and precise disassembly of arbitrary binary programs is an undecidable task [91]. The problem stems from two main reasons: (a) not being able to decisively differentiate code from data [91]; and (b) not being able to precisely identify function boundaries [5, 6]. Fortunately, modern toolchains, like GCC and LLVM, (1) do not mix code and data [3, 4] and (2) embed stack unwinding information to (x86-64) C/C++ binaries [95]. `sysfilter` takes advantage of (1) and (2) in order to precisely disassemble the executable code from all ELF files in the analysis scope, *without requiring symbols or debugging information* (If such information is available, `sysfilter` will use it, but our techniques are designed for stripped binaries.)

More specifically, for each `.so` object in the analysis scope, `sysfilter` uses the stack unwinding information (`.eh_frame` section) to get the exact boundaries of all functions in executable sections (e.g., `.text`, `.plt`). Moreover, special care is taken to correctly identify functions of `crtstuff.c` (`libgcc`), which are compiled into the sections `.init` and `.fini`, as well as into `crtbegin.o`. Armed with precise information about function boundaries, and given the strict separation of code and data, `sysfilter` performs a *linear sweep*, in all code regions that correspond to identified functions, to disassemble their executable code. The resulting disassembly does not contain any invalid instruction, due to data treated as code or incorrect function boundary detection, nor it misses instructions due to unidentified code—the resulting disassembly is complete, precise, and accurate.

Direct Call Graph Building upon the precise disassembly obtained during the previous step, `sysfilter` proceeds to construct the FCG of the input program. First, it puts together the *direct call graph* (DCG): i.e., the part of the FCG that corresponds to directly-invoked functions/code. This is achieved by first adding to the DCG the entry point of the main binary, followed by all the functions whose addresses are stored at the subsequent (ELF) sections: `.preinit_array`, `.init_array`, and `.fini_array`; the code/function in `.init` and `.fini` is also added to the DCG. Subsequently, the same process is repeated for every other `.so` ELF object in the analysis scope. At the end of this step, a set of *initial* functions are added to the DCG, which correspond to the entry points of the code that is executed during the initialization/finalization of the respective process by the dynamic linker/loader (`ld.so`). It is also possible to provide as input a set of implicit function dependencies (see Figure 1B). Again, this is required to aid the analysis of binaries that have run-time dependencies to certain functions (e.g., via `dlsym`) or use `LD_PRELOAD`.

Next, the code of each such function in the DCG is scanned, linearly, to identify direct `call` instructions that target other functions in the respective ELF objects. Branch instructions, like (un)conditional `jmp`, which cross function boundaries are also taken into consideration as they are typically used for implementing *tail-call elimination* [84]. Each identified target function (callee) is also added to the DCG, and the process is repeated until no additional functions can be added. Cross-shared library calls, via the Procedure Linkage Table (PLT), are handled by inspecting the `.dynsym`, and `.dynstr`, sections of the ELF files in scope and "emulating" the binding (symbol resolution) rules of `ld.so`.[2] (Direct cross-`.so` object calls via PLT are treated as direct intra-`.so` function calls.)

The net result of the above is the construction of the part of the FCG that contains the entry point(s) and the initialization/finalization functions of the ELF objects in scope (plus the implicitly-added functions, if provided), followed by every other function that is *directly-reachable* from them (i.e., reachable by following the targets of direct `call`/`jmp` instructions and resolving PLT entries).

[2] This is analogous to executing `ld.so` with '`LD_BIND_NOW=1`'.

Address-taken Call Graph The aforementioned process does not take into consideration functions targeted-by indirect `call`/`jmp` instructions. Such instructions have as operand a (general-purpose) register, or a memory location, which stores the target address (i.e., address of the callee), and are typically used for dereferencing function pointers (C/C++) and implementing dynamic dispatch (C++) [77]. Resolving the target addresses of indirect `call`/`jmp` instructions, statically, is a hard problem [72], mostly due to the imprecision of *points-to* analysis [44]. (Note that resolving target addresses using dynamic testing is even more problematic, as the results of this approach usually lack *soundness* [62].) Starting with DCG, `sysfilter` proceeds to *over-approximate* the FCG by constructing, what we refer to as, the *address-taken call graph* (ACG). The process of constructing the ACG is *complete*: i.e., it never excludes functions that can be executed by the program (under any possible input).

The first step in the construction of the ACG is the identification of all *address-taken* functions: i.e., functions whose address appears in `rvalue` expressions, function arguments, `struct`/`union` initializers, and C++ object initializers, or functions that correspond to *virtual* methods (C++). The set of all address-taken (AT) functions is a superset of the possible targets of every indirect call site in scope. This is because indirect `call`/`jmp` instructions take as operands (general-purpose) registers, or memory locations, which can only hold absolute addresses; therefore, in order for a function to end-up being invoked via an indirect `call`/`jmp` instruction, its address much first be "taken", and then loaded in the respective operand (be it a register or memory location).

`sysfilter` leverages the fact that every ELF object in the analysis scope is compiled as `PIC`, in order to identify all AT functions. More specifically, locations in code, or data, ELF regions that correspond to absolute function addresses must always have accompanying relocation entries (`relocs`), if `PIC` is enabled [14]. `sysfilter` begins with identifying all the relocation sections (i.e., sections of type `SHT_REL` or `SHT_RELA`) in the ELF objects in scope. Next, it processes all the `relocs`, searching for cases where the computation of the relocation involves the starting address of a function (recall that we have already identified the boundaries of every function in scope, during the construction of the DCG). Every such function, whose starting address is used in `relocs` computation, is effectively an AT function. The same function can have its address taken multiple times in different locations (e.g., function arguments, `rvalue` expressions in function bodies, or as part of global `struct`/`union`/C++ object initializers). Relocations that are applied to special sections (e.g., `.plt`, `.dynamic`) are ignored, as they are only related to dynamic binding.

Armed with the set of all AT functions, `sysfilter` proceeds with computing the reachable functions from (each one of) them using the same approach we employed for constructing the DCG. ACG effectively contains as "entry points" the discovered AT functions, followed by every other function that is directly-reachable from them. The combined set of functions in DCG and ACG is a superset of the set of functions in the program's FCG: i.e., $V[FCG] \subseteq (V[DCG] \cup V[ACG])$.

Vacuumed Call Graph Although $DCG \cup ACG$ is a safe over-approximation of FCG, it is not a tight one, as every AT function included in the considered call graph is (potentially) "polluting" it in a considerable manner by bringing in scope every other function that is reachable from itself. In order to keep the over-approximation as tight as possible, `sysfilter` *prunes* the ACG using a technique for software debloating [1, 71]. In particular, we begin with the observation that each time the address of a function is taken, a code pointer is created. By taking into account the location (ELF section) that such code pointers are created, `sysfilter` further separates those found in code (e.g., `.text`) and data (e.g., `.(ro)data`) regions. For the former, it iterates every function that has been deemed as unreachable, and checks if the address of an AT function is only taken within functions that are (strictly) not part of the call graph. If this condition is true, it removes the respective AT function from ACG, which may result in additional removals (e.g., everything directly-reachable from the removed AT function); `sysfilter` iteratively performs the above until no additional functions can be pruned.

For the latter case, `sysfilter` cannot prune AT functions using the same approach, as the resulting code pointers can be part of encapsulating data structures whose usage cannot be tracked without access to access to symbol (or debugging) information. However, if such information is indeed available—note that modern toolchains (GCC, LLVM) include symbols in the resulting ELF objects (`.symtab` section) by default, while popular Linux distributions provide symbols for their packaged binaries—, `sysfilter` can (more aggressively) eliminate AT functions from data sections as follows. First, it leverages symbol information to identify the bounds (`[OBJ_BEGIN - OBJ_END]`) of global data objects (i.e., symbols of type `OBJECT`/`GLOBAL`). Next, it checks for `relocs` that: (a) correspond to AT functions; and (b) fall within the bounds of any global object. The net result of this approach is the identification of statically-initialized arrays of code pointers or data structures that contain code pointers. Lastly, `sysfilter` iterates every function that has been classified as unreachable, and checks if `OBJ_BEGIN` is taken only within such functions. Again, if this condition is true, it marks the AT functions that correspond to the object beginning at `OBJ_BEGIN` as unreachable, and iteratively performs the purging process until no additional functions can be classified as unreachable. We refer to the pruned ACG, combined with DCG, as *vacuumed call graph* (VCG). Specifically, $VCG = DCG \cup ACG'$, where ACG' denotes the pruned ACG using the approach outlined above (see Figure 1C). Again, VCG is a complete, tight over-approximation of the true FCG: i.e., `sysfilter` only excludes functions that can never be executed by the program (under any possible input); more formally: $V[FCG] \subseteq V[VCG] \subseteq (V[DCG] \cup V[ACG])$.

```
1  #define ctor __attribute__((constructor))
2  typedef void (*fptr)(void);
3
4  void f10(void) { ... }
5  ctor void f9(void) { ... f10(); ... }
6  void f8(void) { ... }
7  void f7(void) { ... f8(); ... }
8  void f6(void) { ... }
9  void f5(void) { ... fp_arr[n](); ... }
10 void f4(void) { ... f5(); ... }
11 void f3(void) { ... }
12
13 fptr f2(void) { ... return &f4; }
14 fptr f1(void) { ... return &f3; }
15
16 fptr fp;
17 fptr fp_arr[] = {&f6, &f7};
18
19 int main(void)
20 {
21   ...
22   fp = f1();
23   ...
24   fp();
25   return EXIT_SUCCESS;
26 }
```

Figure 2: **VCG Construction Example.** Without symbol information $V[VCG] = \{\texttt{main}, \texttt{f1}, \texttt{f3}, \texttt{f6}, \texttt{f7}, \texttt{f8}, \texttt{f9}, \texttt{f10}\}$, whereas with symbols (or debugging information) available, $V[VCG] = \{\texttt{main}, \texttt{f1}, \texttt{f3}, \texttt{f9}, \texttt{f10}\}$.

Figure 2 illustrates a C-like program, which we will be using as an example to demonstrate the VCG construction. `sysfilter` will initially include `main` (ln. 19) and `f9` (ln. 5). DCG will also include all the directly-reachable functions from the above initial set: `f1` (reachable from `main`, ln. 22) and `f10` (reachable from `f9`, ln. 22). Hence, $V[DCG] = \{\texttt{main}, \texttt{f1}, \texttt{f9}, \texttt{f10}\}$. Next, `sysfilter` will proceed with the construction of the ACG, which, initially, will include all the address-taken functions: `f3` (ln. 14), `f4` (ln. 13), and `f6` and `f7` (ln. 17). ACG will also include all the directly-reachable functions from set of AT functions: `f5` (reachable from `f4`, ln. 10) and `f8` (reachable from `f7`, ln. 7). Thus, $V[ACG] = \{\texttt{f3}, \texttt{f4}, \texttt{f5}, \texttt{f6}, \texttt{f7}, \texttt{f8}\}$.

`sysfilter` will then continue with pruning the ACG as follows. First, it will remove `f4`, as its address is only taken in function `f2` (ln. 13), which is unreachable. This will also result in removing `f5`, as it is only directly-reachable from `f4` (ln. 10). If the respective ELF object is stripped, the pruning process will terminate at this point, resulting in the following set of functions: $V[ACG'] = \{\texttt{f3}, \texttt{f6}, \texttt{f7}, \texttt{f8}\}$. If symbol (or debugging) information is available, then `sysfilter` can perform more aggressive pruning by identifying that `fp_arr` is not referenced by any function in scope. Therefore, the AT functions `f6` and `f7` can also be removed, as well as `f8` that is directly-reachable only from `f7` (ln. 7). The net result of the above is the following set of functions: $V[ACG''] = \{\texttt{f3}\}$.

To summarize, without symbol information, $V[VCG] = V[DCG] \cup V[ACG'] = \{\texttt{main}, \texttt{f1}, \texttt{f3}, \texttt{f6}, \texttt{f7}, \texttt{f8}, \texttt{f9}, \texttt{f10}\}$, whereas with symbols (or debugging information) available, $V[VCG] = V[DCG] \cup V[ACG''] = \{\texttt{main}, \texttt{f1}, \texttt{f3}, \texttt{f9}, \texttt{f10}\}$. Interested readers are referred to the appendix (§ A) for more information about how `sysfilter` handles GNU `IFUNC` and `NSS` symbols, overlapping code, and hand-written assembly.

3.1.3 System Call Set Construction

The x86-64 ABI dictates that system calls are performed using the `syscall` instruction [30].[3] Moreover, during the invocation of `syscall`, the *system call number* is placed in register `%rax`. Armed with the program's VCG, `sysfilter` constructs the system call set in question as follows.

First, it identifies all reachable functions that include `syscall` instructions, by performing a linear sweep in each function $f \in V[VCG]$ to pinpoint `syscall` instances. Once the set of all the reachable `syscall` instructions is established, `sysfilter` continues with performing a simple *value-tracking* analysis to resolve the exact value(s) of `%rax` on every `syscall` site. The process relies on standard live-variable analysis using *use-define* (UD) chains [2, § 9.2.5]. Specifically, `sysfilter` considers that `syscall` instructions "use" `%rax` and leverages the UD links to find all the instructions that "define" it. In most cases, `%rax` is defined via constant-load instructions (e.g., `mov $0x3,%eax`), and by collecting such instructions and extracting the respective constant values, `sysfilter` can assemble system call sets. If `%rax` is defined via instructions that involve memory operands, `sysfilter` aborts (or issues a warning, if invoked accordingly) as the resulting system call set may be incomplete [72]. The output of the syscall-set extraction component is the collected set of system call numbers in JSON format (see Figure 1D).

We opt for applying the analysis above in an intra-procedural manner, as our results indicate that this strategy works well in practice (see § 5); system call invocation is architecture-specific, and typically handled via `libc` using the following pattern (in x86-64): '`mov $SYS_NR, %eax; syscall`', where `$SYS_NR = {$0x0, $0x1, ...}`. One exception is the handling of the `syscall()` function [48], which performs system calls indirectly by receiving the respective system call number as argument. If `syscall()` is not-address taken in VCG, then `sysfilter` first identifies the reachable functions that directly-invoke `syscall()`, and performs intra-procedural, value-tracking on register `%rdi` (first argument, system call number). If the address of `syscall()` is taken in the reachable VCG, then `sysfilter` aborts (or issues a warning, if invoked accordingly) as the resulting system call set may, again, be incomplete.

[3] Performing syscalls via software interrupts (e.g., `int $0x80`), or `sysenter`, is only supported in x86-64 Linux to allow executing 32-bit applications over a 64-bit kernel. `sysfilter` focuses solely on 64-bit applications (i.e., it does not consider syscalls via `int $0x80` or `sysenter`).

```
1  #define ARCH   AUDIT_ARCH_X86_64
2  #define NRMAX  (X32_SYSCALL_BIT - 1)
3  #define ALLOW  SECCOMP_RET_ALLOW
4  #define DENY   SECCOMP_RET_KILL_PROCESS
5
6  struct sock_filter filter[] = {
7  BPF_STMT(BPF_LD | BPF_W | BPF_ABS,
8       (offsetof(struct seccomp_data, arch))),
9  BPF_JUMP(BPF_JMP| BPF_JEQ|BPF_K, ARCH, 0, 7),
10 BPF_STMT(BPF_LD | BPF_W | BPF_ABS,
11      (offsetof(struct seccomp_data, nr))),
12 BPF_JUMP(BPF_JMP|BPF_JGT|BPF_K, NRMAX, 5, 0),
13 BPF_JUMP(BPF_JMP|BPF_JEQ|BPF_K, 0,  3, 0),
14 BPF_JUMP(BPF_JMP|BPF_JEQ|BPF_K, 1,  2, 0),
15 BPF_JUMP(BPF_JMP|BPF_JEQ|BPF_K, 15, 1, 0),
16 BPF_JUMP(BPF_JMP|BPF_JEQ|BPF_K, 60, 0, 1),
17 BPF_STMT(BPF_RET|BPF_K, ALLOW),
18 BPF_STMT(BPF_RET|BPF_K, DENY) };
```

Figure 3: **Classic BPF (cBPF) Program.** Compiled-by sysfilter, enforcing the following syscall set: 0 (read), 1 (write), 15 (exit), and 60 (sigreturn). The filter checks if the value of field nr (syscall number) $\in \{0,1,15,60\}$ via means of linear search.

3.2 System Call Set Enforcement

The input to the syscall-set enforcement component of sysfilter is the set of allowed system calls, as well as the ELF file that corresponds to the main binary of the application (see Figure 1E). Armed with the set of developer-intended syscalls, sysfilter uses seccomp-BPF [36] to enforce it at run-time. The latter receives as input a BPF "program" [54], passed via prctl, or seccomp, which is invoked by the kernel on every system call. Note that BPF programs are executed in kernel mode by an interpreter for BPF bytecode [54], while just-in-time (JIT) compilation to native code is also supported [11]. In addition, the Linux kernel provides support for two different BPF variants: (a) classic (cBPF) and (b) extended (eBPF) [46]; seccomp-BPF makes use of cBPF only.

The input to seccomp-BPF programs (filters) is a fixed-size struct (i.e., seccomp_data; see Figure 8 in Appendix B), passed by the kernel, which contains a snapshot of the system call context: i.e., the syscall number (field nr), architecture (field arch), as well as the values of the instruction pointer and syscall arguments. sysfilter performs filtering based on the value of nr as follows: **if** ($nr \in \{0,1,\ldots\}$) **then** ALLOW **else** DENY, where $\{0,1,\ldots\}$ is the set of allowed system call numbers. Given such a set, sysfilter compiles a cBPF filter that implements the above check via means of *linear* or *skip list*-based search. Figure 3 depicts a filter that uses the linear search approach to enforce the following set of syscalls: read (0), write (1), exit (15), and sigreturn (60). Ln. 7 – 12 implement a standard preamble, which asserts that the architecture is indeed x86-64. This check is crucial as it guarantees that the mapping between the allowed syscall numbers and the syscalls performed is the right one.

For instance, suppose that this check is missing, and getuid (102)—a harmless syscall—exists in the allowed set. If the target process (x86-64) is compromised, and the attacker issues syscall no. 102, via int $0x80 (or sysenter), then the filter will allow the syscall but the kernel will execute socketcall instead: i.e., the syscall with number 102 in x86 (32-bit), effectively giving the attacker network-access capabilities. The check in ln. 9 rejects every architecture different from x86-64, while the check in ln. 12 rejects syscalls that correspond to the x32 ABI [50].[4] The bulk part of the enforcement/search is implemented in ln. 13 – 16 (BPF_JEQ statements). Note that cBPF does not allow loops, and therefore sysfilter implements the linear search using *loop unwinding* (i.e., 'if-else if-...-else' construct). In case of a non-permitted syscall, sysfilter terminates the processes (ln. 4, SECCOMP_RET_KILL_PROCESS). Figure 9, in the appendix (§ B), illustrates a cBPF filter that uses the skip list approach to implement the search.

sysfilter injects the compiled filter as follows. First, it generates a dynamic shared object, namely libsysfilter.so. Next, it links the aforementioned shared object with the main binary, using patchelf [61]; libsysfilter.so includes only a single function, install_filter, registered as a constructor. The net result of the above is that ld.so will automatically load libsysfilter.so, and invoke install_filter, during the initialization of the main binary (see Figure 1F).

install_filter attaches the compiled cBPF filter, at load-time, using the seccomp system call [47]. Importantly, before invoking seccomp (with SECCOMP_SET_MODE_FILTER), the no_new_privs attribute of the calling thread is asserted, via invoking prctl (with PR_SET_NO_NEW_PRIVS), disabling the acquisition of new privileges via further execve-ing programs that make use of set-user-ID, set-group-ID, or other capabilities. Lastly, install_filter passes the argument SECCOMP_FILTER_FLAG_TSYNC [47] to seccomp for making the respective filter visible to *all* executing threads, while it also uses SECCOMP_FILTER_FLAG_SPEC_ALLOW [47] to disable the speculative store bypass (SSB) mitigation. Note that the latter is configurable; however, the SSB mitigation is only relevant when BPF programs of *unknown provenance* are loaded in kernel space to further assist mounting Spectre attacks [37] (variant 4 [25])—sysfilter cBPF programs are not malicious nor attacker-controlled.

Once the filter is installed using the method outlined above, the respective process can execute only developer-indented syscalls. Note that ld.so is included in the analysis scope, and hence the initialization/finalization of additional libraries, at run-time (e.g., via dlopen/dlclose), as well as any other ld.so-related functionality, is supported seamlessly.

[4] If the target binary is going to be executed atop an x86-64 Linux kernel that does not support x86 emulation (CONFIG_IA32_EMULATION=n) nor the x32 ABI (CONFIG_X86_X32=n), then sysfilter can further optimize the generated filters by omitting the arch-related preamble.

Crucially, no_new_privs guarantees that filters are *pinned* to the protected process during its lifetime—i.e., even if the process is completely compromised, attackers cannot remove filters. Recall that the filtering itself takes place in kernel mode using only the syscall number as input; the syscall arguments are not inspected, and user space memory is not accessed, thereby avoiding the pitfalls related to concurrency and (wrapper-based) syscall filtering [17, 92]. In addition, applications that make use of seccomp-BPF are seamlessly supported as well. BPF filters are *stackable*, meaning that more than one filter can be attached to a process; if multiple filters exist, the kernel always enforces the most *restrictive* action.

Handling execve sysfilter prevents enforcement bypasses via the execution of different programs. Specifically, even if a (compromised) process is allowed to invoke execve, it still cannot extend its set of allowed syscalls by invoking a different executable that has a (potentially) larger set of allowed syscalls; the same is also true if the process tries to craft a rogue executable in the filesystem, which allows all syscalls (or some of the blocked ones), and execute it. Filter pinning and stacking are essential for ensuring that processes can only reduce their set of allowed syscalls, in accordance to the principle of least privilege [76], but they do interfere with execve as they are preserved process attributes.

For example, suppose that programs P1 and P2 have the following syscall sets. P1: 0 (read), 1 (write), 15 (exit), and 59 (execve); P2: 0 (read), 1 (write), 2 (open), 3 (close), 8 (lseek), 9 (mmap), 11 (munmap), 56 (clone), 61 (wait4), 79 (getcwd), 96 (gettimeofday), 102 (getuid), 115 (getgroups), 202 (futex), 292 (dup3), and 317 (seccomp). If P2 is normally-invoked, then it will operate successfully. However, if P2 is invoked via P1, then the resulting process will not be able to issue any other syscall than read, write, exit, and execve (the last two are not even required by P2). To deal with this issue, sysfilter supports two different execve modes: (a) *union* and (b) *hierarchical*.

In union mode, given a set programs $\{P_1, P_2, ..., P_N\}$ that can be invoked in any combination, via execve, with $SYS_1, SYS_2, ..., SYS_N$ being their allowed sets of syscalls, sysfilter constructs a filter that enforces the union of $SYS_1, SYS_2, ..., SYS_N$ and attaches it to all of them. This will result in each process functioning correctly, as it has support for the syscalls it requires, but overly-approximates least privilege—every program effectively inherits the privileges (with respect to the syscall API) of all others in the set. In the example above, union would result in executing both P1 and P2 with the following syscall set: $\{0-3, 8, 9, 11, 15, 56, 59, 61, 79, 96, 102, 115, 202, 292, 317\}$.

In hierarchical mode, sysfilter begins with the same approach as above, but further *rectifies* (reduces) syscall sets each time a process invokes execve. In our example, this would result in executing P1 with the set $\{0-3, 8, 9, 11, 15, 56, 59, 61, 79, 96, 102, 115, 202, 292, 317\}$, further reduced to $\{0-3, 8, 9, 11, 56, 61, 79, 96, 102, 115, 202, 292, 317\}$ right before execve-ing P2. Note that the hierarchical mode still results in certain processes being a bit more privileged (with respect to accessing OS services), but not all.

If special treatment regarding execve is required for a particular (set of) program(s), then a *recipe* can be provided to the enforcement tool, along with the respective syscall sets, which describes how sysfilter should operate on execve calls. If union mode is specified, then sysfilter merely splices together a set of different syscall sets (provided via separate JSON files), and compiles a single filter that is attached to all programs in scope. In case of hierarchical mode, the recipe describes the (execve) relationships between callers and callees, allowing sysfilter to construct different filters, one for each program in scope, which adhere to the model above.

More importantly, our results (§ 5) indicate that recipe creation can be automated, to some extent, by employing static value-tracking analysis to resolve the first argument of execve calls. However, note that sysfilter is not geared towards sandboxing applications that invoke *arbitrary* scripts or programs (e.g., command-line interpreters, managed runtime environments); other schemes, like Hails [21], SHILL [56], and the Web API Manager [82], are better suited for this task.

3.3 Prototype Implementation

Our prototype implementation consists of ≈ 2.5 KLOC of C/C++ and ≈ 150 LOC of Python, along with various shell scripts (glue code; ≈ 120 LOC). More specifically, we implemented the extraction tool atop the Egalito framework [95]. Egalito is a binary *recompiler*; it allows rewriting binaries in-place by first lifting binary code into a layout-agnostic, machine-specific intermediate representation (IR), dubbed EIR, and then allowing "tools" to inspect or alter it.

We implemented the extraction tool as an Egalito "pass" (C/C++), which creates the analysis scope, constructs the VCG, and extracts the respective syscall set, using the techniques outlined in § 3.1. Note that we do not utilize the binary rewriting features of Egalito; we only leverage the framework's API to precisely disassemble the corresponding binaries and lift their code in EIR form, which, in turn, we use for implementing the analyses required for constructing the DCG, identifying all AT functions for building the ACG, pruning unreachable parts of the call graph for assembling the VCG, identifying syscall instructions, performing value-tracking, *etc.* We chose Egalito over similar frameworks as it employs the best jump table analysis to date.

The enforcement tool is implemented in Python and is responsible for generating the cBPF filter(s), and libsysfilter.so, and attaching the latter to the main binary using patchelf (see § 3.2). Lastly, during the development of sysfilter we also improved Egalito by adding better support for hand-coded assembly, fixing various symbol resolution issues, and re-architecting parts of the framework to reduce memory pressure. (We upstreamed all our changes.)

Application		Version	Syscalls	Tests	Pass?
FFmpeg	(124)	4.2	167	3756	✓
GNU Core.	(100)	8.31	148	672	✓
GNU M4	(1)	1.14	70	236	✓
MariaDB	(156)	10.3	153	2059	✓
Nginx	(1)	1.16	128	356	✓
Redis	(6)	5.0	104	81	✓
SPEC CINT.	(12)	1.2	82	12	✓
SQLite	(7)	3.31	139	31190	✓
Vim	(3)	8.2	163	255	✓
GNU Wget	(1)	1.20	107	130	✓

Table 1: **Correctness Test.** The numbers in parentheses count the different binaries included in the application/package. "Syscalls" indicates the number of system calls in the allowed set; in case of applications with multiple binaries that number corresponds to the unique syscalls across the syscall sets of all binaries in the package. "Tests" denotes the number of tests run from the validation suite of the application.

4 Evaluation

We evaluate `sysfilter` in terms of (1) correctness, (2) run-time performance overhead, and (3) effectiveness.

Testbed We used two hosts for our experiments: (a) run-time performance measurements were performed on an 8-core Intel Xeon W-2145 3.7GHz CPU, armed with 64GB of (DDR4) RAM, running Devuan Linux (v2.1, kernel v4.16); (b) analysis tasks were performed on an 8-core AMD Ryzen 2700X 3.7GHz CPU, armed with 64GB of (DDR4) RAM, running Arch Linux (kernel v5.2). All applications (except SPEC CINT2006) were obtained from Debian `sid` (development distribution) [86], as it provides the latest versions of upstream packages along with debug/symbol information [93].

Correctness We used 411 binaries from various packages/-projects with `sysfilter`, including GNU Coreutils, Nginx, Redis, SPEC CINT2006, SQLite, FFmpeg, MariaDB, Vim, GNU M4, and GNU Wget, to extract and enforce their corresponding syscall sets. The results are shown in Table 1. Once sandboxed, we stress-tested them with ≈38.5K tests from the projects' validation suites. (Note that we did not include tests that required the application to execute arbitrary external programs, such as tests with arbitrary commands used in Vim scripts, Perl scripts in Nginx, and arbitrary shell scripts to load data in SQLite and M4.) In all cases, `sysfilter` managed to extract a complete and tight over-approximation of the respective syscall sets, demonstrating that our prototype can successfully handle complex, real-world software.

Performance To assess the run-time performance impact of `sysfilter`, we used SPEC CINT2006 (std. benchmarking suite), Nginx (web server), and Redis (data store)—i.e., 19 program binaries in total; the selected binaries represent the most performance-sensitive applications in our set and are well-suited for demonstrating the relative overhead(s). We also explored different settings and configurations, including interpreted vs. JIT-compiled `BPF` filters, and filter code that implements sandboxing using a linear search vs. filter code that utilizes a skip list-based approach (§ 3.2). In the case of SPEC, we observed a run-time slowdown ≤1% under all conditions and search methods.

Figure 4 and Figure 5 illustrate the impact of `sysfilter` on Nginx (128) and Redis (103)—the numbers in parentheses indicate the corresponding syscall set sizes, while "Binary" corresponds to skip list-based filters. We configured Nginx to use 4 worker processes and measured its throughput using the `wrk` tool [23], generating requests via the loopback interface from 4 threads, over 256 simultaneous connections, for 1 minute. Overall, `sysfilter` diminishes reduction in throughput by using skip list-based filters (compared to linear search-based ones) when JIT is disabled, with maximum reductions in throughput of 18% and 7%, respectively. The differences in compilation strategy appear to be normalized by jitting, which showed a maximum drop in throughput of 6% in all conditions. We evaluated Redis similarly, using the `memtier` tool [73], performing a mix of `SET` and `GET` requests with a 1:10 req. ratio for 32 byte data elements. The requests were issued from 4 worker threads with 128 simultaneous connections, per thread, for 1 minute. `sysfilter` incurs maximum throughput reductions of 11% and 3%, with and without JIT support, respectively. Lastly, toggling `SECCOMP_FILTER_FLAG_SPEC_ALLOW` [47] (for enabling the SSB mitigation) incurs an additional ≈10% overhead in all cases.

Effectiveness To assess the security impact of syscall filtering, we investigated how `sysfilter` reduces the attack surface of the OS kernel, by inquiring what percentage of all C/C++ applications in Debian `sid` (≈30K main binaries) can exploit 23 (publicly-known) Linux kernel vulnerabilities—ranging from memory disclosure and corruption to direct privilege escalation—even after hardened with `sysfilter`. A list of the number of binaries in our dataset affected by each CVE is shown in Table 2. Depending on the exact vulnerability, the percentage of binaries that can still attack the kernel ranges from 0.09% – 64.34%. Although `sysfilter` does not defend against particular types of attacks (e.g., control- or data-flow hijacking [87]), our results demonstrate that it can mitigate real-life threats by means of least privilege enforcement and (OS) attack surface reduction.

5 Large-scale System Call Analysis

We conclude our work with a large scale study regarding the syscall profile of all C/C++ applications in Debian `sid`, reporting insights regarding syscall set sizes (e.g., the number of syscalls per binary), most- and least-frequently used syscalls, syscall site distribution (libraries vs. main binary), and more. The results of this analysis not only justify our design, but can also aid that of future syscall policing mechanisms.

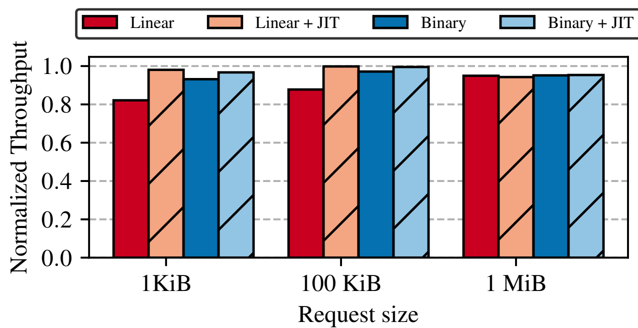

Figure 4: **Impact of `sysfilter` on Nginx.**

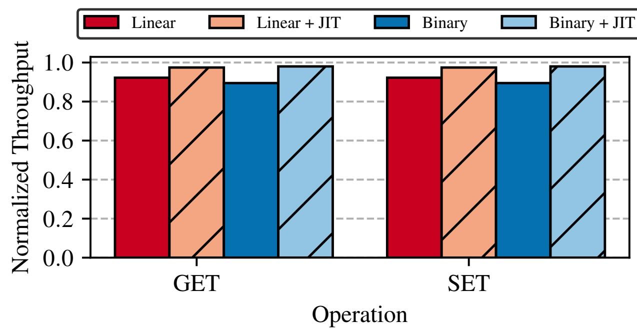

Figure 5: **Impact of `sysfilter` on Redis.**

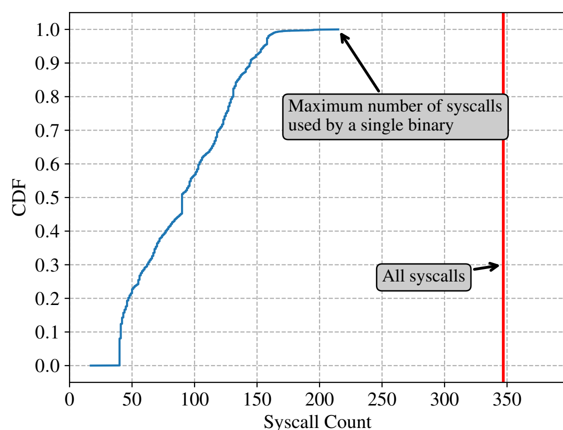

Figure 6: **Number of Syscalls per Binary.**

We consider packages from the three major Debian repositories, namely `main`, `contrib`, and `nonfree`. At the time of writing, the Debian `sid` distribution contains over 50K packages listed for the x86-64. We exclude, however, packages that do not contain executable code, such as documentation packages (`*-doc`), development headers (`*-dev`), and debug symbols (`*-dbg`, `*-dbgsym`); meta-packages and virtual packages; architecture-agnostic packages that do not include x86-64 binaries; and packages containing only shared libraries. Note that shared libraries and other excluded packages can be installed during processing, as dependencies of main application packages. We processed a total of 33829 binaries across 8922 packages, 30398 (91.3%) of which could be analyzed successfully. The median runtime for the extraction tool is about 30s per binary, with 90% of binaries completing within 200s. For a single FCG pass, the median runtime is reduced to about 10s per binary.

Number of Syscalls per Binary Many binaries only use a small portion of the syscall API. Figure 6 shows the distribution of the number of syscalls used by each binary processed. Out of the total set of 347 syscalls, the maximum number of syscalls used by one binary is only 215 syscalls, which represents 62% of the total syscall API. We observe that this distribution has a long tail: i.e., the median syscall count per binary is 90 syscalls, with the 90th percentile at 145 syscalls.

With one exception, we find that *all* binaries processed use at least 40 syscalls. The sole exception is the statically-linked binary `mtcp_restart` that uses only 17 syscalls—this binary performs syscalls directly, without using any library wrappers. In the general case, even the simplest of programs, such as `/bin/false`, utilize 40 syscalls due to the paths included by initialization functions that load shared libraries: e.g., `mmap` and `mprotect` are ubiquitous as they are always reachable from `_start`, even before `main` is invoked.

Syscalls from Libraries When extracting syscalls from each binary, we record which of its shared libraries contained a `syscall` instruction for each invocation. After `libc`, the second most common library is `libpthread` (used by 68.7% of binaries), in which we observe 40 syscalls—upon further investigation we found that this is due to `glibc`'s use of macros to directly invoke syscalls. The next most common libraries that invoke syscalls directly are `libstdc++` (37% of binaries), which invokes `futex` (202) directly; `libnss_resolve` (32% of binaries, 3 syscalls), and `libglib-2.0` (17%, 5 syscalls).

While we leave a comprehensive analysis of library syscalls to a future study, we note a few common themes. Libraries tend to directly invoke syscalls with no `libc` wrapper function available, such as `futex`, and `gettid`. We also observe many libraries directly invoking syscalls specific to their core functionality, perhaps to handle circumstances where existing wrappers are unavailable or insufficient. For instance, we note cryptographic libraries, such as `libgcrypt` and `libcrypto`, directly invoking cryptographic-related syscalls, like `getrandom` and `keyctl`.

Effectiveness of FCG Approximation Figure 7 shows the number of syscalls we extract from each binary using the three FCG approximation methods (§ 3.1.2): DCG, $DCG \cup ACG$, and VCG, sorted by the count for the VCG. Each binary represents three points on the figure (i.e., one for each method).

CVE	Syscall(s) Involved	Vulnerability Type	Binaries (%)
CVE-2019-11815	clone, unshare	Memory corruption	19558 (64.34)
CVE-2013-1959	write	Direct privilege escalation	19558 (64.34)
CVE-2015-8543	socket	Type confusion	19128 (62.93)
CVE-2017-17712	sendto, sendmsg	Memory corruption	19057 (62.69)
CVE-2013-1979	recvfrom, recvmsg	Direct privilege escalation	18968 (62.40)
CVE-2016-4998	setsockopt	Memory disclosure	17360 (57.11)
CVE-2016-4997	setsockopt	Memory corruption	17360 (57.11)
CVE-2016-3134	setsockopt	Memory corruption	17360 (57.11)
CVE-2017-18509	setsockopt, getsockopt	Memory corruption	17416 (57.29)
CVE-2018-14634	execve, execveat	Memory corruption on suid program	16775 (55.18)
CVE-2017-14954	waitid	Memory disclosure	14064 (46.27)
CVE-2014-5207	mount	Direct privilege escalation	11412 (37.54)
CVE-2018-12233	setxattr	Memory corruption	3356 (11.04)
CVE-2016-0728	keyctl	Memory corruption	2827 (9.30)
CVE-2014-9529	keyctl	Memory corruption	2827 (9.30)
CVE-2019-13272	ptrace	Direct privilege escalation	127 (0.42)
CVE-2018-1000199	ptrace	Memory corruption	127 (0.42)
CVE-2014-4699	fork, clone, ptrace	Register value corruption	121 (0.40)
CVE-2014-7970	pivot_root	DoS	79 (0.26)
CVE-2019-10125	io_submit	Memory corruption	58 (0.19)
CVE-2017-6001	perf_event_open	Direct privilege escalation	51 (0.17)
CVE-2016-2383	bpf	Memory corruption	35 (0.12)
CVE-2018-11508	adjtimex	Memory disclosure	26 (0.09)

Table 2: **Effectiveness Analysis**. The column "Binaries" indicates the number (and percentage) of binaries observed in the large scale analysis on Debian sid applications that use the system calls related to the respective vulnerability. (Underlined entries correspond to vulnerabilities that involve namespaces.)

For all binaries, the count for the VCG is always in between that of DCG and *DCG* ∪ *ACG*. Thus, for our dataset, VCG represents a safe, *tight* over-approximation of the FCG.

dl{open, sym} and execve By employing our value-tracking approach (see § 3.1.3), sysfilter can resolve ≈89% of all dlsym arguments, ≈37% of all dlopen arguments, and ≈30% of all execve arguments. We observed a few cases in common libraries where value-tracking fails, which may benefit from special handling (§ A): e.g., ≈50% of dlsym failures relate to NSS functionality, while ≈5% of dlopen failures involve Kerberos plugins. Lastly, we found two isolated cases where sysfilter was unable to construct syscall sets: Qemu and stress-ng contain arbitrary syscall dispatchers (like glibc's syscall()), which is expected given their functionality. Otherwise, we find that syscall sites follow strictly the pattern 'mov $SYS_NR, %eax; syscall'.

6 Related Work

Syscall-usage Analysis Tsai et al. [88] performed a study similar to ours (on binaries in Ubuntu v15.04) to characterize the usage of the syscall API, as well as that of ioctl, fcntl, prctl, and pseudo-filesystem APIs. Their study focuses on quantifying API complexity and security-related usage trends, such as unused syscalls and adoption of secure APIs over the legacy ones. Our study focuses specifically on the syscall API as a means of evaluating our extraction tool. We consider this work complementary, and focus on making the analysis more scalable, precise, and complete. Specifically, and in antithesis to sysfilter, the call graph construction approach of Tsai et al. does not consider initialization/finalization code nor does it identify AT functions that are part of global struct/union/C++ object initializers.

Static System Call Filtering Syscall filtering has been extensively studied in the past, in various contexts. Indeed, sysfilter shares many of the problems, and proposed solutions, with the seminal work by Wagner and Dean [89], which uses static analysis techniques to model sequences of valid syscalls as a non-deterministic finite automaton (NDFA). This work, as well as others from its era [22], aim at building models of program execution for intrusion detection purposes. In contrast, sysfilter focuses on building optimized (OS-enforceable) seccomp-BPF filters by determining the total set of syscalls, independent of ordering, which provides a more compact representation and eliminates the challenges related to control flow modeling. Moreover, sysfilter employs binary analysis, whereas Wagner and Dean's work requires recompiling target binaries and shared libraries, which severely limits the deployability of their scheme.

Shredder [55] performs static analysis on Windows binaries to identify API calls, and arguments, used by applications.

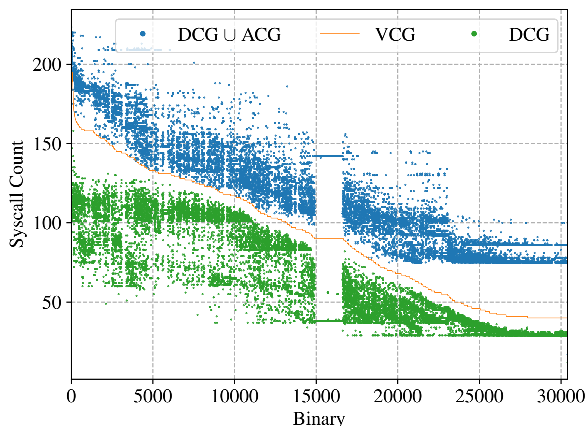

Figure 7: **Syscall Count for Different FCG Construction Methods.** The number of syscalls reported for each binary is shown, sorted by the count for the VCG.

Specifically, it restricts calls to syscall wrapper functions, via trampolines, but requires CFI for effective protection. Independently and concurrently to our work, Ghavamnia et al. proposed Confine [20]: a (mostly) static analysis-based system for automatically extracting and enforcing syscall policies on "containerized" (i.e., Docker) applications. Confine requires access to C library code (e.g., `glibc` or `musl`), while its call graph construction approach considers every function in non-`libc` code within scope. In addition, it relies on `objdump`, and, hence, requires symbols for precise disassembly. `sysfilter` can operate on stripped binaries, while our FCG construction approach produces much tighter syscall sets.

Similar to `sysfilter`, Zeng et al. [98] identify valid sets of syscalls using binary analysis, but their approach lacks soundness: its call graph approximation method relies, in part, on points-to analysis to resolve the targets of function pointers. In antithesis, `sysfilter` identifies all address-taken functions in order to avoid the impression issues associated with this method. Further, Zeng et al. perform the enforcement using a customized Linux kernel to provide per-process system call tables, whereas our `seccomp-BPF` based approach is available in stock Linux kernel v3.5 or later.

Dynamic System Call Filtering Systrace [69] uses dynamic tracing to generate system call policies and implements a userspace daemon for enforcement. Mutz et al. [58] and Maggi et al. [51] develop statistical models for host-based intrusion detection, which as a design choice inherently gives false negatives, potentially impeding valid program execution. Ostia [18] provides a system call sandboxing mechanism that delegates policy decisions to per-process agents, while a plethora of earlier work on container debloating [90], and sandboxing [42, 90], also relies on dynamic tracing. In contrast, `sysfilter` does not rely on dynamic syscall tracing or statistical models, which can generate incomplete policies—instead, `sysfilter` safely over-approximates a program's true syscall set and thus will not break program execution.

`seccomp-BPF` in Existing Software Firefox [57], Chrome [8], and OpenSSH [63] use `seccomp-BPF` to sandbox themselves using manually-crafted policies, while container runtimes, such as Docker and Podman, allow the use `seccomp-BPF` policies to filter container syscalls. By default, Docker applies a filter that disables 44 syscalls [15], and Podman has support for tracing syscalls dynamically with `ptrace` to build a profile for containers. Both also fully support user-specified filters [75]. `sysfilter` can be seamlessly integrated with such software, providing the apparatus for generating the respective syscall sets automatically/precisely.

Binary Debloating `sysfilter` shares goals and analysis approaches with recent software debloating techniques. Quach et al. [71] propose a compiler-based approach that embeds dependency information into programs, and uses a custom loader to selectively load only required portions of shared libraries in memory. TRIMMER [80] specializes LLVM bytecode based on a user-defined configuration, while the work of Koo et al. [39] utilizes coverage information to remove code based on feature directives. C-Reduce [74], Perses [85], and CHISEL [27] use delta-debugging techniques to compile minimized programs using a series of provided test cases. Unlike previous approaches, Razor [70] does not require source code, and implements a dynamic tracer to reconstruct the program's FCG from a set of test cases. The analysis used by Nibbler [1] is the most similar to `sysfilter`. However, Nibbler requires symbols, whereas `sysfilter` operates on stripped binaries.

7 Conclusion

We presented `sysfilter`: a static (binary) analysis-based framework that automatically limits what OS services attackers can (ab)use, by enforcing the principle of least privilege, and reduces the attack surface of the OS kernel, by restricting the syscall set available to userland processes. We introduced a set of program analyses for constructing syscall sets in a scalable, precise, and complete manner, and evaluated our prototype in terms of correctness using 411 binaries, from various real-world C/C++ projects, and ≈38.5K tests to stress-test their functionality when armored with `sysfilter`. Moreover, we assessed the impact of our syscall enforcement mechanism(s) using SPEC CINT2006, Nginx, and Redis, demonstrating minimal run-time slowdown. Lastly, we concluded with a large scale study about the syscall profile of ≈30K C/C++ applications (Debian `sid`). We believe that `sysfilter` is a practical and robust tool that provides a solution to the problem of unlimited access to the syscall API.

Availability

The prototype implementation of `sysfilter` is available at: `https://gitlab.com/brown-ssl/sysfilter`

Acknowledgments

We thank our shepherd, Aravind Prakash, and the anonymous reviewers for their valuable feedback. This work was supported by the Office of Naval Research (ONR) and the Defense Advanced Research Projects Agency (DARPA) through awards N00014-17-1-2788 and HR001118C0017. 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, or DARPA.

References

[1] Ioannis Agadakos, Di Jin, David Williams-King, Vasileios P. Kemerlis, and Georgios Portokalidis. Nibbler: Debloating Binary Shared Libraries. In *Annual Computer Security Applications Conference (ACSAC)*, pages 70–83, 2019.

[2] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Pearson Education, 2nd edition, 2006.

[3] Jim Alves-Foss and Jia Song. Function Boundary Detection in Stripped Binaries. In *Annual Computer Security Applications Conference (ACSAC)*, pages 84–96, 2019.

[4] Dennis Andriesse, Xi Chen, Victor Van Der Veen, Asia Slowinska, and Herbert Bos. An In-Depth Analysis of Disassembly on Full-Scale x86/x64 Binaries. In *USENIX Security Symposium (SEC)*, pages 583–600, 2016.

[5] Dennis Andriesse, Asia Slowinska, and Herbert Bos. Compiler-agnostic Function Detection in Binaries. In *IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 177–189, 2017.

[6] Tiffany Bao, Jonathan Burket, Maverick Woo, Rafael Turner, and David Brumley. BYTEWEIGHT: Learning to Recognize Functions in Binary Code. In *USENIX Security Symposium (SEC)*, pages 845–860, 2014.

[7] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazières, and Dan Boneh. Hacking Blind. In *IEEE Symposium on Security and Privacy (S&P)*, pages 227–242, 2014.

[8] Chromium Blog. A safer playground for your linux and chrome os renderers. `https://blog.chromium.org/2012/11/a-safer-playground-for-your-linux-and.html`.

[9] Nathan Burow, Scott A. Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. Control-Flow Integrity: Precision, Security, and Performance. *ACM Computing Surveys (CSUR)*, 50(1):1–33, 2017.

[10] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. Return-Oriented Programming without Returns. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 559–572, 2010.

[11] Jonathan Corbet. BPF: the universal in-kernel virtual machine. `https://lwn.net/Articles/599755/`.

[12] Gabriel Corona. The ELF file format. `https://www.gabriel.urdhr.fr/2015/09/28/elf-file-format/`.

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

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

[15] Docker Documentation. Seccomp Security Profiles for Docker. `https://docs.docker.com/engine/security/seccomp/`.

[16] Common Weakness Enumeration. CWE-123: Write-what-where Condition. `https://cwe.mitre.org/data/definitions/123.html`.

[17] Tal Garfinkel. Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools. In *Network and Distributed System Security Symposium (NDSS)*, pages 163–176, 2003.

[18] Tal Garfinkel, Ben Pfaff, and Mendel Rosenblum. Ostia: A Delegating Architecture for Secure System Call Interposition. In *Network and Distributed System Security Symposium (NDSS)*, 2004.

[19] Jason Geffner. VENOM: Virtualized Environment Neglected Operations Manipulation. `http://venom.crowdstrike.com`.

[20] Seyedhamed Ghavamnia, Tapti Palit, Azzedine Benameur, and Michalis Polychronakis. Confine: Automated System Call Policy Generation for Container Attack Surface Reduction. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2020.

[21] Daniel B Giffin, Amit Levy, Deian Stefan, David Terei, David Mazières, John C Mitchell, and Alejandro Russo. Hails: Protecting Data Privacy in Untrusted Web Applications. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 47–60, 2012.

[22] Jonathon T. Giffin, Somesh Jha, and Barton P. Miller. Detecting Manipulated Remote Call Streams. In *USENIX Security Symposium (SEC)*, pages 61–79, 2002.

[23] Will Glozer. wrk – a HTTP benchmarking tool. `https://github.com/wg/wrk`.

[24] Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. Out Of Control: Overcoming Control-Flow Integrity. In *IEEE Symposium on Security and Privacy (S&P)*, pages 575–589, 2014.

[25] Google Project Zero. speculative execution, variant 4: speculative store bypass. `https://bugs.chromium.org/p/project-zero/issues/detail?id=1528`.

[26] LLVM Developer Group. The LLVM Compiler Infrastructure. `https://llvm.org`.

[27] Kihong Heo, Woosuk Lee, Pardis Pashakhanloo, and Mayur Naik. Effective Program Debloating via Reinforcement Learning. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 380–394, 2018.

[28] Gerard J. Holzmann. Code Inflation. *IEEE Software*, (2):10–13, 2015.

[29] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. Data-Oriented Programming: On the Expressiveness of Non-Control Data Attacks. In *IEEE Symposium on Security and Privacy (S&P)*, pages 969–986, 2016.

[30] Intel. System V Application Binary Interface. `https://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf`.

[31] GNU Compiler Collection (GCC) Internals. Common Function Attributes. `https://gcc.gnu.org/onlinedocs/gcc/Common-Function-Attributes.html#Common-Function-Attributes`.

[32] Kyriakos K. Ispoglou, Bader AlBassam, Trent Jaeger, and Mathias Payer. Block Oriented Programming: Automating Data-Only Attacks. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1868–1882, 2018.

[33] Vasileios P. Kemerlis. *Protecting Commodity Operating Systems through Strong Kernel Isolation*. PhD thesis, Columbia University, 2015.

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

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

[36] The Linux Kernel. Seccomp BPF (SECure COMPuting with filters). https://www.kernel.org/doc/html/latest/userspace-api/seccomp_filter.html.

[37] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. In *IEEE Symposium on Security and Privacy (S&P)*, pages 1–19, 2019.

[38] Hyungjoon Koo, Yaohui Chen, Long Lu, Vasileios P. Kemerlis, and Michalis Polychronakis. Compiler-assisted Code Randomization. In *IEEE Symposium on Security and Privacy (S&P)*, pages 461–477, 2018.

[39] Hyungjoon Koo, Seyedhamed Ghavamnia, and Michalis Polychronakis. Configuration-Driven Software Debloating. In *European Workshop on Systems Security (EuroSec)*, pages 1–6, 2019.

[40] Volodymyr Kuznetzov, László Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. Code-Pointer Integrity. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 147–163. 2014.

[41] Per Larsen, Andrei Homescu, Stefan Brunthaler, and Michael Franz. SoK: Automated Software Diversity. In *IEEE Symposium on Security and Privacy (S&P)*, pages 276–291, 2014.

[42] Lingguang Lei, Jianhua Sun, Kun Sun, Chris Shenefiel, Rui Ma, Yuewu Wang, and Qi Li. Speaker: Split-Phase Execution of Application Containers. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, pages 230–251, 2017.

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

[44] Percy Liang and Mayur Naik. Scaling Abstraction Refinement via Pruning. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 590–601, 2011.

[45] The GNU C Library. System Databases and Name Service Switch. https://www.gnu.org/software/libc/manual/html_node/Name-Service-Switch.html.

[46] Linux Programmer's Manual. bpf – perform a command on an extended BPF map or program. http://man7.org/linux/man-pages/man2/bpf.2.html.

[47] Linux Programmer's Manual. seccomp – operate on Secure Computing state of the process. http://man7.org/linux/man-pages/man2/seccomp.2.html.

[48] Linux Programmer's Manual. syscall – indirect system call. http://man7.org/linux/man-pages/man2/syscall.2.html.

[49] Generic Part Linux Standard Base Core Specification. Exception Frames. https://refspecs.linuxbase.org/LSB_5.0.0/LSB-Core-generic/LSB-Core-generic/ehframechpt.html.

[50] H. J. Lu and Mike Frysinger. x32 System V Application Binary Interface. https://sites.google.com/site/x32abi/.

[51] Federico Maggi, Matteo Matteucci, and Stefano Zanero. Detecting Intrusions through System Call Sequence and Argument Analysis. *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 7(4):381–395, 2008.

[52] Linux Programmer's Manual. syscalls – Linux system calls. http://man7.org/linux/man-pages/man2/syscalls.2.html.

[53] Ali Jose Mashtizadeh, Andrea Bittau, Dan Boneh, and David Mazières. CCFI: Cryptographically Enforced Control Flow Integrity. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 941–951, 2015.

[54] Steven McCanne and Van Jacobson. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *USENIX Winter Conference*, pages 259–270, 1993.

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

[56] Scott Moore, Christos Dimoulas, Dan King, and Stephen Chong. SHILL: A Secure Shell Scripting Language. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 183–199, 2014.

[57] MozillaWiki. Security/Sandbox/Seccomp. https://wiki.mozilla.org/Security/Sandbox/Seccomp.

[58] Darren Mutz, Fredrik Valeur, Giovanni Vigna, and Christopher Kruegel. Anomalous System Call Detection. *ACM Transactions on Information and System Security (TISSEC)*, pages 61–93, 2006.

[59] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 245–258, 2009.

[60] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. CETS: Compiler Enforced Temporal Safety for C. In *ACM SIGPLAN International Symposium on Memory Management (ISMM)*, pages 31–40, 2010.

[61] NixOS. patchelf – A small utility to modify the dynamic linker and RPATH of ELF executables. https://github.com/NixOS/patchelf.

[62] A. Jefferson Offutt and J. Huffman Hayes. A Semantic Model of Program Faults. *ACM SIGSOFT Software Engineering Notes (SEN)*, 21(3):195–200, 1996.

[63] OpenSSH. Release Notes. https://www.openssh.com/txt/release-6.0.

[64] Oracle Solaris, Linker and Libraries Guide. Position-Independent Code. https://docs.oracle.com/cd/E26505_01/html/E26506/glmqp.html.

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

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

[67] GNU Project. The GNU Compiler Collection. https://gcc.gnu.org.

[68] Sergej Proskurin, Marius Momeu, Seyedhamed Ghavamnia, Vasileios P. Kemerlis, and Michalis Polychronakis. xMP: Selective Memory Protection for Kernel and User Space. In *IEEE Symposium on Security and Privacy (S&P)*, pages 584–598, 2020.

[69] Niels Provos. Improving Host Security with System Call Policies. In *USENIX Security Symposium (SEC)*, pages 257–272, 2003.

[70] Chenxiong Qian, Hong Hu, Mansour Alharthi, Pak Ho Chung, Taesoo Kim, and Wenke Lee. RAZOR: A Framework for Post-deployment Software Debloating. In *USENIX Security Symposium (SEC)*, pages 1733–1750, 2019.

[71] Anh Quach, Aravind Prakash, and Lok Yan. Debloating Software through Piece-Wise Compilation and Loading. In *USENIX Security Symposium (SEC)*, pages 869–886, 2018.

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

[73] Redis Labs. NoSQL Redis and Memcache traffic generation and benchmarking tool. https://github.com/RedisLabs/memtier_benchmark.

[74] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. Test-Case Deduction for C Compiler Bugs. In *ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, pages 335–346, 2012.

[75] Valentin Rothberg. Generate SECCOMP Profiles for Containers Using Podman and eBPF. `https://podman.io/blogs/2019/10/15/generate-seccomp-profiles.html`.

[76] Jerome H. Saltzer and Michael D. Schroeder. The Protection of Information in Computer Systems. *IEEE*, 63(9):1278–1308, 1975.

[77] Pawel Sarbinowski, Vasileios P. Kemerlis, Cristiano Giuffrida, and Elias Athanasopoulos. VTPin: Practical VTable Hijacking Protection for Binaries. In *Annual Computer Security Applications Conference (ACSAC)*, pages 448–459, 2016.

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

[79] Hovav Shacham. 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)*, pages 552–561, 2007.

[80] Hashim Sharif, Muhammad Abubakar, Ashish Gehani, and Fareed Zaffar. TRIMMER: Application Specialization for Code Debloating. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 329–339, 2018.

[81] Kevin Z. Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization. In *IEEE Symposium on Security and Privacy (S&P)*, pages 574–588, 2013.

[82] Peter Snyder, Cynthia Taylor, and Chris Kanich. Most Websites Don't Need to Vibrate: A Cost-Benefit Approach to Improving Browser Security. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 179–194, 2017.

[83] Brad Spengler. PaX: The Guaranteed End of Arbitrary Code Execution. In *G-Con2*, 2003.

[84] Guy Lewis Steele Jr. Debunking the "Expensive Procedure Call" Myth or, Procedure Call Implementations Considered Harmful or, LAMBDA: The Ultimate GOTO. In *ACM National Conference*, pages 153–162, 1977.

[85] Chengnian Sun, Yuanbo Li, Qirun Zhang, Tianxiao Gu, and Zhendong Su. Perses: Syntax-Guided Program Reduction. In *International Conference on Software Engineering (ICSE)*, pages 361–371, 2018.

[86] Debian The Universal Operating System. The unstable distribution ("sid"). `https://www.debian.org/releases/sid/`.

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

[88] Chia-Che Tsai, Bhushan Jain, Nafees Ahmed Abdul, and Donald E. Porter. A Study of Modern Linux API Usage and Compatibility: What to Support When You're Supporting. In *European Conference on Computer Systems (EuroSys)*, pages 1–16, 2016.

[89] David Wagner and Drew Dean. Intrusion Detection via Static Analysis. In *IEEE Symposium on Security and Privacy (S&P)*, pages 156–168, 2000.

[90] Zhiyuan Wan, David Lo, Xin Xia, Liang Cai, and Shanping Li. Mining Sandboxes for Linux Containers. In *International Conference on Software Testing, Verification and Validation (ICST)*, pages 92–102, 2017.

[91] Richard Wartell, Yan Zhou, Kevin W. Hamlen, Murat Kantarcioglu, and Bhavani Thuraisingham. Differentiating Code from Data in x86 Binaries. In *European Conference on Machine Learning and Knowledge Discovery in Databases (ECML-PKDD)*, pages 522–536, 2011.

[92] Robert N. M. Watson. Exploiting Concurrency Vulnerabilities in System Call Wrappers. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2007.

[93] Debian Wiki. Using Symbols Files. `https://wiki.debian.org/UsingSymbolsFiles`.

[94] David Williams-King, Graham Gobieski, Kent Williams-King, James P. Blake, Xinhao Yuan, Patrick Colp, Michelle Zheng, Vasileios P. Kemerlis, Junfeng Yang, and William Aiello. Shuffler: Fast and Deployable Continuous Code Re-Randomization. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 367–382, 2016.

[95] David Williams-King, Hidenori Kobayashi, Kent Williams-King, Graham Patterson, Frank Spano, Yu Jian Wu, Junfeng Yang, and Vasileios P. Kemerlis. Egalito: Layout-Agnostic Binary Recompilation. In *ACM SIGPLAN International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 133–147, 2020.

[96] Xiaoyang Xu, Masoud Ghaffarinia, Wenhao Wang, Kevin W. Hamlen, and Zhiqiang Lin. CONFIRM: Evaluating Compatibility and Relevance of Control-flow Integrity Protections for Modern Software. In *USENIX Security Symposium (SEC)*, pages 1805–1821, 2019.

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

[98] Qiang Zeng, Zhi Xin, Dinghao Wu, Peng Liu, and Bing Mao. Tailored Application-specific System Call Tables. Technical report, Pennsylvania State University, 2014.

[99] Hanqing Zhao, Yanyu Zhang, Kun Yang, and Taesoo Kim. Breaking Turtles All the Way Down: An Exploitation Chain to Break out of VMware ESXi. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2019.

A VCG Special Cases

GNU IFUNC GCC, along with the GNU Binutils and `glibc`, provide support for (a GNU-specific feature, named) *indirect functions*. Such symbols are of type `STT_GNU_IFUNC` and have an associated *resolver* function that will "return" the actual/target function to be used in lieu of the indirect (`IFUNC`) symbol [31]. The resolution takes place via PLT, at run-time; `IFUNC`s are typically used for customizing the symbol resolution of `ld.so`, and selecting among different function implementations that use processor-specific features. `sysfilter` links every call via an `IFUNC` PLT entry with: (1) the respective resolver function, and (2) *all* its potential targets in VCG—the latter are easily identifiable as they are functions whose address is taken in the resolver.

GNU NSS The Name Service Switch (NSS) [45] is used by `glibc` to select among different *name resolution* mechanisms (e.g., flat-file databases, DNS, LDAP). Specifically, `glibc` consults `nsswitch.conf` to determine the mapping between various databases (i.e., `passwd`, `shadow`, `group`, `hosts`, *etc.*) and resolution mechanisms (e.g., `files`, `dns`, `ldap`). Each such mechanism corresponds to a different dynamic shared

object (e.g., libnss_files.so, libnss_dns.so, libnss_ldap.so), which provides a specific implementation of the NSS API. Depending on the contents of nsswitch.conf, glibc loads the analogous .so ELF file, using dlopen, and invokes the relevant functions (NSS), after obtaining their addresses via dlsym. sysfilter parses nsswitch.conf, and makes use of the implicit library/function dependency mechanism to add the matching ELF object(s) and function(s) in the analysis scope (§ 3.1.1) and VCG (§ 3.1.2), respectively.

Overlapping Functions Certain versions of glibc include functions whose body overlaps with that of other functions. In particular, in v2.24 of glibc, ≈ 30 functions are completely embedded inside others (e.g., connect wraps __connect_nocancel). In cases where, say, f1() overlaps with f2(), and &f1() < &f2(), both functions can be (in)directly-invoked by others, but if f1() gets executed, f2() will be invoked as well, as the execution will fall through to the latter. sysfilter supports such cases by carefully inspecting function boundaries (.eh_frame section; § 3.1.2), and connecting the respective functions in DCG, accordingly.

Hand-written Assembly ASM code is not problematic for sysfilter as long as it adheres to our hardening assumptions (§ 2). sysfilter does not support non-PIC objects (§ 3.1.1); hence, if ASM code that is embedded in binaries is analyzed, it will be PIC (by construction). If .eh_frame records (for hand-written ASM) are missing, or code and data are mixed, then the precision of our analyses will be affected. Thankfully, however, the (hand-written) ASM code that is linked-with popular binaries/libraries oftentimes contains annotations to support stack unwinding and (C++) exception handling [49]. Lastly, if partial information regarding function boundaries is available, sysfilter will resort to using a combination of linear and recursive disassembly techniques, and state-of-the-art heuristics [94], for approximating function boundaries.

B Enforcement Details

```
1 struct seccomp_data {
2   int nr;             /* syscall number        */
3   __u32 arch;         /* architecture (x86-64) */
4   __u64 instruction_pointer; /* IP value      */
5   __u64 args[6];      /* syscall arguments     */
6 };
```

Figure 8: **struct seccomp_data.** Passed by the Linux kernel to seccomp-BPF filters on every syscall. The field nr is filled with the system call number, while arch and instruction_pointer are filled with the respective architecture, and value of the instruction pointer, during the time of executing syscall (i.e., AUDIT_ARCH_X86_64 and %rip, in x86-64). Likewise, args[6] is a six-element array, filled with the syscall arguments (i.e., the values of registers %rdi, %rsi, %rdx, %r10, %r8, and %r9, in x86-64).

```
1  #define ARCH  AUDIT_ARCH_X86_64
2  #define NRMAX (X32_SYSCALL_BIT - 1)
3  #define ALLOW SECCOMP_RET_ALLOW
4  #define DENY  SECCOMP_RET_KILL_PROCESS
5
6  struct sock_filter filter[] = {
7  BPF_STMT(BPF_LD | BPF_W | BPF_ABS,
8      (offsetof(struct seccomp_data, arch))),
9  BPF_JUMP(BPF_JMP| BPF_JEQ|BPF_K, ARCH, 0, 7),
10 BPF_STMT(BPF_LD | BPF_W | BPF_ABS,
11     (offsetof(struct seccomp_data, nr))),
12 BPF_JUMP(BPF_JMP|BPF_JGE|BPF_K, 61, 11,  0),
13 BPF_JUMP(BPF_JMP|BPF_JGE|BPF_K, 8,   5,  0),
14 BPF_JUMP(BPF_JMP|BPF_JGE|BPF_K, 2,   2,  0),
15 BPF_JUMP(BPF_JMP|BPF_JEQ|BPF_K, 1,   19, 0),
16 BPF_JUMP(BPF_JMP|BPF_JEQ|BPF_K, 0,   18, 19),
17 BPF_JUMP(BPF_JMP|BPF_JEQ|BPF_K, 3,   17, 0),
18 BPF_JUMP(BPF_JMP|BPF_JEQ|BPF_K, 2,   16, 17),
19 BPF_JUMP(BPF_JMP|BPF_JGE|BPF_K, 11,  2,  0),
20 BPF_JUMP(BPF_JMP|BPF_JEQ|BPF_K, 9,   14, 0),
21 BPF_JUMP(BPF_JMP|BPF_JEQ|BPF_K, 8,   13, 14),
22 BPF_JUMP(BPF_JMP|BPF_JEQ|BPF_K, 56,  12, 0),
23 BPF_JUMP(BPF_JMP|BPF_JEQ|BPF_K, 11,  11, 12),
24 BPF_JUMP(BPF_JMP|BPF_JGE|BPF_K, 115, 5,  0),
25 BPF_JUMP(BPF_JMP|BPF_JGE|BPF_K, 96,  2,  0),
26 BPF_JUMP(BPF_JMP|BPF_JEQ|BPF_K, 79,  8,  0),
27 BPF_JUMP(BPF_JMP|BPF_JEQ|BPF_K, 61,  7,  8),
28 BPF_JUMP(BPF_JMP|BPF_JEQ|BPF_K, 102, 6,  0),
29 BPF_JUMP(BPF_JMP|BPF_JEQ|BPF_K, 96,  5,  6),
30 BPF_JUMP(BPF_JMP|BPF_JGE|BPF_K, 292, 2,  0),
31 BPF_JUMP(BPF_JMP|BPF_JEQ|BPF_K, 202, 3,  0),
32 BPF_JUMP(BPF_JMP|BPF_JEQ|BPF_K, 115, 2,  3),
33 BPF_JUMP(BPF_JMP|BPF_JEQ|BPF_K, 317, 1,  0),
34 BPF_JUMP(BPF_JMP|BPF_JEQ|BPF_K, 292, 0,  1),
35 BPF_STMT(BPF_RET|BPF_K, ALLOW),
36 BPF_STMT(BPF_RET|BPF_K, DENY) };
```

Figure 9: **Classic BPF (cBPF) Program.** Compiled-by sysfilter, enforcing the following syscall set: 0 (read), 1 (write), 2 (open), 3 (close), 8 (lseek), 9 (mmap), 11 (munmap), 56 (clone), 61 (wait4), 79 (getcwd), 96 (gettimeofday), 102 (getuid), 115 (getgroups), 202 (futex), 292 (dup3), and 317 (seccomp). The filter checks if the value of field nr $\in \{0,1,2,3,8,9,11,56,61,79,96,102,115,202,292,317\}$ via means of (deterministic) ***skip list***-based search. The BPF_JEQ statements assert if the value of nr is one of the 16 allowed syscalls, whereas BPF_JGE statements implement the "shortcuts" in the search process.