Temporal System Call Specialization for Attack Surface Reduction

Seyedhamed Ghavamnia, Tapti Palit, Shachee Mishra, Michalis Polychronakis
Stony Brook University

Abstract

Attack surface reduction through the removal of unnecessary application features and code is a promising technique for improving security without incurring any additional overhead. Recent software debloating techniques consider an application's entire lifetime when extracting its code requirements, and reduce the attack surface accordingly.

In this paper, we present *temporal specialization*, a novel approach for limiting the set of system calls available to a process depending on its phase of execution. Our approach is tailored to server applications, which exhibit distinct *initialization* and *serving* phases with different system call requirements. We present novel static analysis techniques for improving the precision of extracting the application's call graph for each execution phase, which is then used to pinpoint the system calls used in each phase. We show that requirements change throughout the lifetime of servers, and many dangerous system calls (such as `execve`) can be disabled after the completion of the initialization phase. We have implemented a prototype of temporal specialization on top of the LLVM compiler, and evaluated its effectiveness with six popular server applications. Our results show that it disables 51% more security-critical system calls compared to existing library specialization approaches, while offering the additional benefit of neutralizing 13 more Linux kernel vulnerabilities that could lead to privilege escalation.

1 Introduction

Modern software is complex. Applications typically support a wide range of functionalities for different use cases [28, 49], as evidenced by the existence of multiple features, options, and configuration settings. To support these different features, programs typically require access to a vast range of privileged operations from the OS kernel (e.g., allocating memory, creating new processes, and accessing files or the network), which are made available through the system call interface.

Some of these capabilities, however, are used by the application only once during startup, and are never used again during the lifetime of the program. This is especially true for server applications, which once launched, remain running and serving requests for a long period of time. This means that *all* kernel capabilities (i.e., system calls) remain available to a potentially vulnerable process, and can thus be used as part of exploitation attempts.

Software debloating and specialization has recently gained popularity as a technique for removing or constraining *unused* parts of applications, with the goal of reducing the code and features available to attackers. While some approaches use static analysis to identify unused parts of shared libraries [12, 51], others rely on dynamic analysis and training to identify unneeded parts of the application [13, 21, 48]. Similar techniques have also been applied on containers to constrain the set of system calls available to the hosted programs [22, 38, 59]. A key shared characteristic of the above approaches is that they consider the *entire* lifetime of a program as part of the scope of their analysis.

In this paper, we explore software specialization from a different perspective, and present *temporal system call specialization*, a novel attack surface reduction approach for limiting even further the set of system calls that are available to a process, depending on its *phase of execution*. Instead of treating each application as a single, monolithic entity with an unchanging set of requirements, temporal specialization takes into consideration the changes in an application's requirements throughout its execution lifetime. In particular, we focus on server applications, which typically exhibit two distinct *initialization* and *serving* phases.

Our main motivation is that many dangerous system calls, such as `execve`, which are frequently used as part of exploit code, are often not removed by existing code debloating and specialization techniques, because they are required by the application for legitimate purposes. Crucially, however, operations such as spawning new processes or creating listening sockets are typically only performed during the very first moments of a server's lifetime—the *initialization* phase. Temporal specialization automatically derives the set of system calls required by each execution phase, and restricts the set of

available system calls once the server enters its stable serving phase. This significantly reduces the set of system calls available to an attacker.

A crucial requirement for pinpointing the system calls required in each phase is to construct a sound and precise call graph. As most server applications are developed using C/C++, which support indirect function invocations, we must rely on static code analysis to resolve the possible targets of indirect call sites. Unfortunately, the state-of-the-art implementations of points-to analysis algorithms suffer from severe imprecision and overapproximation, which eventually results in the inclusion of many spurious system calls that are not actually used. To address this challenge, we propose two pruning mechanisms that remove spurious edges from the derived call graph, significantly improving its *precision* while retaining its *soundness*. After identifying the system calls needed in each phase, we use Seccomp BPF to block any system calls that are not needed anymore after the completion of the initialization phase, thereby removing them from the process' attack surface.

We implemented a prototype of temporal specialization for Linux on top of LLVM, and evaluated it with six popular applications (Nginx, Apache Httpd, Lighttpd, Bind, Memcached, and Redis). We show that many dangerous system calls, such as `execve`, can be disabled after the application enters its serving phase, i.e., when the server application starts handling client requests and becomes susceptible to attacks. Our results show that temporal specialization disables 51% more security-critical system calls compared to existing library specialization approaches [12, 51], while in many cases it does not leave room for evasion using alternative system call combinations. As an added benefit, 53 Linux kernel vulnerabilities are neutralized by removing system calls which serve as entry points for triggering them, 13 of which are not preventable by library specialization.

Our work makes the following main contributions:

1. We propose a novel temporal system call specialization approach that considers the different operational characteristics of server applications throughout their different execution phases.

2. We present type-based and address-taken-based pruning mechanisms to improve the precision of static analysis techniques for call graph construction.

3. We evaluate our prototype implementation with six popular applications and a diverse set of 567 shellcode and 17 ROP payload samples, demonstrating its effectiveness in blocking exploit code, as well as in reducing the exposed attack surface of the underlying kernel.

Our prototype implementation is publicly available as an open-source project at `https://github.com/shamedgh/temporal-specialization`.

2 Background and Motivation

User-space applications rely on the system call API to interact with the OS. The Linux kernel v4.15 used in this work provides 333 system calls, while its latest version 5.6 (as of June 2020) provides 349. Applications, however, typically rely only on a subset of these system calls for their operation. Moreover, their requirements change according to the *phase* of execution, e.g., whether the application is being initialized or serving requests. From a security perspective, this overabundance of system calls allows an attacker to i) use the additional system calls to carry out malicious operations as part of exploiting a vulnerability, and ii) exploit underlying kernel vulnerabilities triggered through system calls and achieve privilege escalation [22, 31, 32].

2.1 Static vs. Temporal API Specialization

Previous works in attack surface reduction [21, 26, 34, 48, 50] consider the entire application lifetime, and remove functionality that will never be used at any point. When considering the execution phases of typical server applications, however, we observe that further specialization can be achieved.

In particular, servers typically start handling client requests after performing a series of one-time operations for setting up the process. This *initialization phase* mainly consists of operations such as parsing configuration files, binding network ports, and forking worker processes. After the completion of these tasks, the server enters its main long-term *serving phase* for handling client requests. In this stable state, the server typically performs operations such as reading from and writing to sockets or files, managing memory, and allocating tasks to the worker processes. Nginx [7] is an example of a server which exhibits this behavior. Depending on whether it is started in "single-process" or "multi-process" mode, Nginx either executes the function `ngx_single_process_cycle`, or forks the configured number of worker processes, each of which invokes the function `ngx_worker_process_cycle`. Both functions mark the beginning of the serving phase by entering an infinite loop that processes client requests.

The operations performed in these two phases are distinctively different, and thus the required system calls for carrying them out are also different. For example, if a server only creates a fixed set of long-lived worker processes during the initialization phase, it will not need access to system calls such as `fork` and `execve` during the serving phase.

Figure 1 shows a simplified view of the call graph for Apache `httpd` [15], one of the most popular open source web servers. The different shapes correspond to application functions, library functions, and system calls. The initialization phase begins with `main`, and this phase performs operations such as binding and listening to sockets, and spawning the worker processes through calls to `fork` and `execve`. The forked worker processes begin execution at the func-

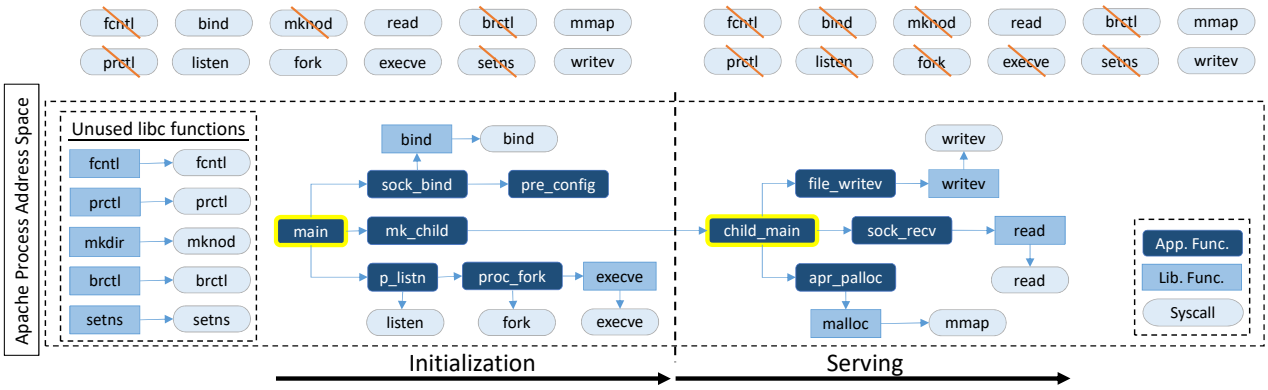

Figure 1: Library debloating [12,51] can only remove system calls that are never used during the entire lifetime of the application (top left). Temporal specialization removes additional system calls that are never used *after* the initialization phase (top right).

tion `child_main`, which denotes the beginning of the serving phase. During this phase, the application performs tasks such as allocating buffers and handling I/O operations.

Library debloating techniques [12, 51] analyze the code of a given application to identify and remove parts of the linked libraries that are not needed by the application, thereby creating specialized versions of each library. However, they consider the entire lifetime of the application, and therefore, in the example of Figure 1, are unable to prevent access to system calls such as `fork` and `execve`—crucial for attackers' exploit code—as they are used during the initialization phase.

2.2 Seccomp BPF

Seccomp BPF [8] is a mechanism provided by the Linux kernel for restricting the set of system calls that are accessible by user-space programs. Specifically, Seccomp BPF uses the Berkeley Packet Filter language [40] for allowing developers to write programs that act as system call filters, i.e., BPF programs that inspect the system call number (as well as argument values, if needed) and allow, log, or deny the execution of the respective system call. Applications can apply Seccomp BFP filters by invoking either the `prctl` or `seccomp` system call from within their own process. After doing so, all system call invocations from within the process itself or any forked child processes will be checked against the installed filters to grant or reject permission. We use this mechanism to reduce the set of system calls available to programs after the completion of their initialization phase.

3 Threat Model

We consider remote adversaries armed with a vulnerability that allows arbitrary code execution. Temporal system call specialization does not rely on any other exploit mitigations, but as an attack surface reduction technique, it is meant to be used along with other code specialization techniques. Our technique limits the set of system calls an attacker can invoke. Therefore, any exploit code (e.g., shellcode or ROP payload) will have limited capabilities, and will not be able to invoke system calls that are not needed by the server after its initialization phase. These typically include security-critical system calls that can be used to spawn additional services, execute shell commands, and so on. Preventing access to these system calls also effectively neutralizes the corresponding kernel code, which may contain vulnerabilities that can lead to privilege escalation [39]—an attacker cannot trigger those vulnerabilities to compromise the kernel, as the respective system calls cannot be invoked in the first place.

Time-of-check to time-of-use (TOCTTOU) [60] and other race condition attacks are out of the scope of this work.

4 Design

Our goal is to reduce the number of system calls available to attackers once a server application has finished its initialization phase, and thus reduce the exposed attack surface. Disabling system calls that remain unused during the serving phase requires the identification of those system calls that the application uses during the initialization phase, and does not need afterwards. To achieve this, our approach performs the following steps, illustrated in Figure 2.

- Build a *sound* call graph of the application, and derive the list of imported functions from external libraries.
- Map the application call graph, as well as the imported external library functions, to system calls.
- Use programmer-supplied information about the functions that mark the beginning of the initialization and serving phases, respectively, to derive the call graph of each of these phases of execution.

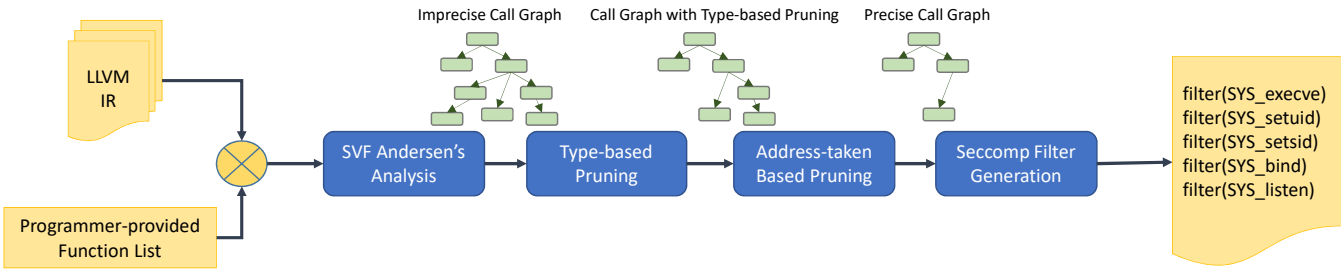

Figure 2: Overview of the process for generating a sound call graph to identify the system calls required by each execution phase.

- Based on these call graphs, identify the list of system calls required by each phase.
- Create Seccomp filters to restrict the use of unneeded system calls, and apply them right after the end of the initialization phase.

4.1 Identifying the Transition Point

We require an expert to identify the *boundary* where the program transitions from the initialization phase to the serving phase, and pass it to our toolchain through a configuration file. This is the point where the server begins its main operation and its system call requirements change. As discussed in Section 2.1, in many applications, such as Apache Httpd [15] and Nginx [7], the transition takes place after the server's main process forks, and child processes are created. In others, such as Memcached [4], which use an event-driven model, this transition takes place at the beginning of the event loop that handles client requests. In case of Apache Httpd, as shown in Figure 1, this transition boundary is defined by the function `child_main`, and once execution reaches this function, many system calls are no longer needed.

Although identifying this transition boundary could perhaps be automated based on heuristics or dynamic analysis, we did not invest the effort to develop such a capability, as this needs to be done only once per application. Manually pinpointing the entry point to the serving phase is relatively easy even if one is not familiar with a given code base. This is the only step where manual intervention is required.

4.2 Call Graph Construction

Applications and libraries written in C/C++ often use indirect function calls via function pointers. For example, the `libapr` and `libapr-util` libraries used by Apache Httpd, use function pointers to register custom memory allocation functions, to register callbacks, and to provide other functionalities that allow the programmer to customize the library. Resolving these indirect function calls in a sound and precise manner is therefore critical for identifying the system calls needed by the application.

Points-to analysis is a static code analysis technique for deriving the possible targets of pointers in a program, and is necessary to soundly identify the target functions of indirect function calls. We use the well-known Andersen's points-to analysis algorithm [14] for this purpose.

Applying Andersen's algorithm to the source code of an application generates a sound call graph, in which all indirect call sites are resolved. However, like all static analysis techniques, points-to analysis suffers from imprecision and overapproximation. For example, Apache's function `ap_run_pre_config` contains an indirect function call. Andersen's points-to analysis reports 136 targets for this function pointer. We manually verified that only seven targets can actually be executed, and the rest 129 are spurious targets that were included due to the imprecision of the analysis.

Previous works [14, 27] have extensively discussed the challenges of scalable and accurate points-to analysis, and an in-depth discussion of these issues is out of the scope of this paper. However, we briefly describe the different sources of overapproximation we faced in our problem space, along with how we mitigated them.

4.2.1 Points-to Analysis Overapproximation

Points-to analysis can be modeled with multiple types of sensitivity, which reflect how objects in memory are modeled. These include *field sensitivity*, *context sensitivity*, and *path sensitivity*. An analysis algorithm employing a higher degree of sensitivity will provide more precise results, and in turn will allow us to gain a more fine-grained view into the system calls required by each execution phase. However, using higher degrees of sensitivity has the fundamental problem of increasing the analysis time, while it requires significant effort to implement such a capability. For example, the popular implementation of Andersen's algorithm, SVF [55], supports field sensitivity (it models every field of a `struct` type uniquely), but not context sensitivity or path sensitivity. This results in imprecision in the results of the points-to analysis.

Context-sensitive analysis considers the calling context when analyzing the target of a function call. When the same function is invoked from different call sites, each function call gets its own "context" and is analyzed independently of the

other function calls. This prevents return values of the called function from propagating into unintended call sites, leading to imprecision. This is critical for functions that allocate or reassign objects referenced by their arguments, or functions that return pointers. Lack of context sensitivity in such cases causes the propagation of analysis results to *all* call sites and *all* return sites of these functions. For example, to allocate memory, Nginx uses a wrapper around memory allocation routines (e.g., `malloc`), called `ngx_alloc`. Because the analysis used by SVF is not context sensitive, its results contain significant overapproximation.

Similarly to context sensitivity, the lack of *path sensitivity* also causes overapproximation in the results of the points-to analysis. Path-sensitive points-to analysis takes into account the predicates of the branch conditions in the control flow graph of the program when solving pointer constraints. Without path sensitivity, the analysis cannot reason about the predicate conditions of a branch.

During our analysis of popular servers, we observed that it was common for libraries (e.g., `libapr`) to provide an option to insert *optional* callback functions at various stages of the life cycle of the library. These callbacks are implemented as indirect function calls, and their call sites are guarded by `NULL` checks on the callback function pointer. We call these *guarded indirect call sites*, and discuss them further in Section 4.2.3.

Due to the lack of context sensitivity, even if no callback function is registered, the points-to analysis can return spurious targets for the guarded indirect call site. Due to the lack of path sensitivity, the analysis cannot detect that the call site is in fact guarded, and will be skipped at runtime. Figure 3 shows an example of a guarded indirect call site. The imprecise call graph contains a spurious edge to `piped_log_maintenance` from a guarded indirect call site accessible in the serving phase. As this function contains a call to the `execve` system call, the overapproximation would prevent it from being removed in the serving phase. Similarly, the lack of context sensitivity and path sensitivity causes overapproximation in the number of possible targets for all indirect call sites, even if they are not optional callback functions guarded by `NULL` checks. A more detailed discussion on overapproximation in points-to analysis is available in Appendix A.

To reduce the overapproximation that the lack of context and path sensitivity introduces in our analysis, and consequently increase the number of system calls that can be removed in the serving phase, we implemented two filtering schemes that prune spurious call edges based on *argument types* and *taken addresses*.

4.2.2 Pruning Based on Argument Types

A naive implementation of Andersen's points-to analysis algorithm does not consider any semantics regarding the type of pointers while solving the constraint graph. For example, SVF's implementation of Andersen's algorithm considers the number of arguments, but not their types, when solving indirect call sites. Due to the lack of context sensitivity and path sensitivity, the results of the points-to analysis often contain imprecision in the form of pointers of one type pointing to memory objects of a different type.

Similarly, when resolving targets for indirect function calls, the results of the points-to analysis often contain functions whose types of arguments do not match those of the call site. For example, in the imprecise call graph of Apache Httpd shown in Figure 3, the guarded indirect call site in function `other_child_cleanup` has two possible targets, `piped_log_maintenance` and `event_run`, despite the fact that only the former matches the types of arguments of the guarded call site.

We have mitigated this problem by checking every indirect call site and pruning any call edges to functions with arguments whose types do not match those of the call site. To maintain soundness, when pruning based on argument types, we consider only arguments of `struct` type, as primitive types may have a mismatch due to reasons such as integer promotion. This simple mechanism is extremely effective in reducing the number of edges in our final call graph. Indicatively, for Nginx, it reduces the number of edges by 70%.

4.2.3 Pruning Based on Taken Addresses

Andersen's algorithm considers all functions in the program to be reachable from its entry point. We observed that this leads to an imprecision in the results of the resolution of indirect call sites, with the result set containing functions that are not accessible from `main` at all.

A function can be the target of an indirect call site only if its address is *taken* (and stored in a variable) at some point in the program. Consequently, if the address of a function is taken at some point in the program that is unreachable from `main`, it can never be a target of an indirect call.

Based on this intuition, we prune further the (still) overapproximated graph generated from the previous argument type based pruning step by first identifying all functions whose addresses are taken along any path that is accessible from the `main` function. This gives us all possible functions that can actually be targets of indirect calls. Using this list of potential address-taken functions, we visit each indirect call site in the program and prune all edges towards targets that do not have their address taken along any valid path.

Going back to the example of Figure 3, the address of `piped_log_mnt` is stored in a function pointer within the function `start_module`, but `start_module` is not reachable from the entry point of Apache Httpd. On the other hand, the function `other_child_cleanup` contains a guarded indirect call site, which first checks if that function pointer is not `NULL`, in which case then dereferences it to invoke the target function. At run time, this `NULL` check will always return false, and this indirect call site is never executed.

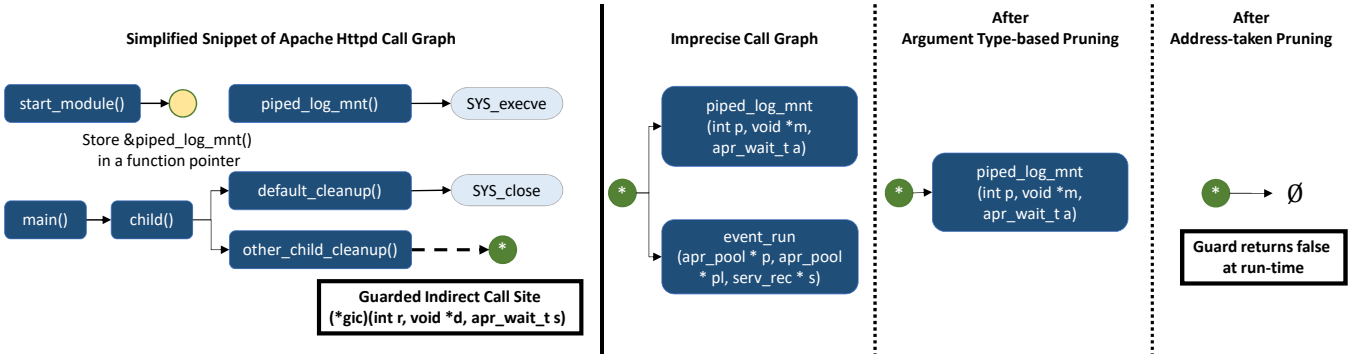

Figure 3: The effect of pruning based on argument types and taken addresses on generating a precise call graph.

Path-insensitive points-to analysis cannot determine whether the guard `NULL` check will fail or not. However, as we prune indirect call sites based on address-taken functions, and given that the address of `piped_log_mnt` is never taken along any reachable path from `main`, we can correctly infer that this guarded indirect call site does not have any valid targets, and will be skipped at run time.

4.3 Mapping System Call Invocations to the Application Call Graph

System calls are typically invoked through the Libc library, which provides corresponding wrapper functions (e.g., the `write` Libc function invokes the `SYS_write` system call). We map each exported Libc function to its relevant system call by first generating the call graph of the entire library, and then augmenting it with information about the system calls of each function as "leaves" on the generated call graph [22].

In addition to using Libc wrappers, applications and libraries can also invoke system calls directly using the `syscall()` glibc function or the `syscall` assembly instruction—we handle both of these cases as well. Finally, we combine the Libc call graph with the call graphs of the main application and all its dependent libraries. Using the resulting unified graph, we extract the set of system calls required by the application for the initialization phase, and then for the serving phase, and identify the system calls that are not needed in the latter. We then use Seccomp to apply the respective filters at the beginning of the serving phase.

5 Implementation

In this section, we describe the implementation details of our framework for temporal system call specialization. Our framework currently supports server applications written in C. Although we currently support only Linux libraries and applications, the concept can easily be applied to other operating systems as well. We use the LLVM [3] compiler toolchain to statically analyze the code of the target application. Because Glibc does not compile with LLVM, we use the GCC toolchain for the compilation and analysis of Glibc.

5.1 Constructing a Sound Call Graph

Our goal is to identify the functions that may be invoked during the initialization and serving phases. To that end, the first step is to construct a sound and precise call graph for the whole application. Accurate points-to analysis for resolving the targets of indirect call sites is the most critical part of this process. We use SVF's [55, 56] implementation of the Andersen's points-to analysis algorithm [14]. SVF operates on the LLVM intermediate representation (IR), so we first lower the C source code into the LLVM IR format using the `clang` compiler and by applying link-time optimization (LTO). We then run SVF on this generated bitcode.

As we discussed in Section 4.2, SVF's implementation of Andersen's algorithm is field sensitive, but not context sensitive or path sensitive, leading to significant imprecision. We also observed that in some cases, the lack of context and array index sensitivity causes objects to lose field sensitivity. We provide more details on this subtle issue in Appendix A.

Solving these imprecision problems would fundamentally require implementing a context-sensitive, path-sensitive, and array-index-sensitive analysis, which increases the complexity of the points-to algorithm, and also requires significant programming effort. Instead, we implemented an alternative lightweight solution that simply prunes call edges in the call graph that are provably added as a result of imprecision.

5.1.1 Pruning Based on Argument Types

SVF begins by iterating over all instructions in the IR bitcode, collecting constraints along the way, and adding them to the *constraint graph*. Then, it iterates over all constraints and solves each of them. At the end of each iteration, it checks if it can successfully find a new target for an indirect call site. For any new target found, it first checks if the number of

arguments in the call site matches the number of arguments in the target function. In case they match (and the target function is not a variadic function), the analysis adds the target function as a possible target of the indirect call site. Then, it begins a new iteration to solve any additional constraints due to the newly discovered target function. As discussed in Section 4.2.2, this results in the inclusion of targets with the same number of arguments, but completely unrelated argument types. We modified SVF to take the argument type into account and only add functions as possible targets when the argument types match.

5.1.2 Pruning Based on Taken Addresses

One of the downsides of using path-insensitive and context-insensitive pointer analysis is that it cannot consider the state of the program when solving the points-to set constraints. In particular, as discussed in Section 4.2.3, if an indirect function call is guarded by a `NULL` check on a function pointer, and the function pointer is not initialized in any function that is reachable from the program's entry point, then the call will be skipped at run time. This is especially useful for modular programs, where initializing a module causes the address of one or multiple functions to be taken, and any housekeeping tasks related to that module are performed after doing the not-`NULL` check on their relevant function pointers. However, due to the imprecision of SVF's static analysis, its results include spurious targets for these guarded indirect call sites.

Using the call graph generated after argument type pruning, we record all functions whose addresses are stored into function pointers. A function's address can be stored into a pointer in three ways: i) by a direct store to a pointer, ii) when passed as an argument to another function, or iii) as part of the initialization of a constant global variable. We implemented an LLVM IR pass to extract functions that have their addresses taken via any of these cases. It traverses the call graph in a depth-first manner, starting at the `main` function, and analyzes every `LoadInst` IR instruction to check if the address of a function is being loaded from memory. To track functions passed as arguments to other functions, it iterates over every IR `Value` passed as an argument at a call site, and checks if it corresponds to a function. Finally, it iterates over all constant `GlobalVariable` objects in the IR to track whether a function is part of their initialized values. Based on the resulting set of address-taken functions, we remove any spurious targets at each indirect call site, while retaining all direct call sites without any modifications.

Algorithm 1 summarizes the steps for both types of pruning based on argument types and taken addresses, which result in a much more precise call graph than the one provided by SVF. Once the final call graph is derived, the next and final step is to identify the system call invocations performed during the initialization phase and the serving phase.

Algorithm 1: Generation of Precise Call Graph

```
Input: LLVM IR bitcode for the target application
Output: precise_cg: precise application call graph
Run SVF's Andersen points-to analysis to get the
  (overapproximated) call graph cg;
/* Perform argument-type pruning */
foreach Indirect-callsite ic in cg do
    foreach Target t of ic in cg do
        if Argument types of t does not match that of ic
          then
            Prune target t for ic;
        end
    end
end
addr_taken_fn_set ← ∅;
reachable_functions ← ∅;
/* Collect address-taken functions */
Traverse cg depth-first, starting from main;
foreach Reachable function func from main do
    reachable_functions ∪ {func};
end
foreach Function f in reachable_functions do
    foreach Address-taken function f_addr_tk in f do
        addr_taken_fn_set ∪ {f_addr_tk};
    end
end
/* Perform address-taken pruning */
foreach Indirect-callsite ic in cg do
    foreach Target t of ic in cg do
        if t ∉ addr_taken_fn_set then
            Prune target t for ic;
        end
    end
end
precise_cg ← cg;
```

5.2 Pinpointing System Call Invocations

System calls are typically invoked through library function calls implemented in the standard C library—the most common implementation of which is glibc. Since glibc cannot be compiled with LLVM, we do not use points-to analysis to generate the call graph and rely on a more overapproximated mechanism, which considers any function having its address taken as a potential target of any indirect call site in its own module. This is only performed once to generate the glibc call graph, and is then used for all applications.

We implemented an analysis pass written in GCC's RTL (Register Transfer Language) intermediate representation to extract the call graph and system call information from glibc. Our analysis pass first builds the call graph using the Egypt tool [24], which operates on GCC's RTL IR. Then, the anal-

ysis pass iterates over every call instruction in the IR and records any inline assembly code containing the native x86-64 `syscall` instruction. These are then added as the "leaves" of the functions in the call graph.

In addition to making direct system calls via inline-assembly, glibc also makes system calls via *wrapper macros* such as `T_PSEUDO`, `T_PSEUDO_ERRNO`, and `T_PSEUDO_ERRVAL`. We identify these wrappers and add the system calls invoked through them to the call graph.

Glibc also uses *weak symbols* and *versioned symbols* to support symbol versioning. Both `weak_alias` and `versioned_symbol` provide aliases for functions. We statically analyze the source code to collect all such aliases, and add them to the call graph. In this way we can map Glibc function calls to system calls.

System calls can also be invoked directly by the application through the `syscall()` glibc function, inline-assembly, or the use of assembly files. We analyze the IR bitcode of the application for invocations of the `syscall()` function, and add the corresponding syscall number information to the call graph. To track the directly invoked system calls in inline assembly, we analyze the LLVM IR for `InlineAsm` blocks. If an `InlineAsm` block contains the `syscall` instruction, we extract the system call number and add it to the functions that call the inline assembly block.

To scan assembly files for `syscall` instructions, we developed a tool that extracts the corresponding system call number. In 64-bit systems, the `syscall` instruction reads the system call number from the RAX register. Starting from every `syscall` instruction, we perform backwards slicing to identify the initialization point of RAX with the system call number. The process continues tracing backwards in the assembly code to find the value (or set of values) that RAX can take at runtime. While glibc does use inline assembly, we did not encounter any custom assembly-level system call invocations in the set of applications we evaluated.

Once we have mapped the Glibc interface to system calls, and have extracted the direct system calls, we combine this information with the previously generated precise call graph, to obtain the list of system calls required by the initialization phase and the serving phase.

5.3 Installing Seccomp Filters

Finally, we create and apply Seccomp filters that disable the unneeded system calls at the transition boundary from the initialization to the serving phase. We use the `prctl` system call to *install* the Seccomp filters. We currently require manual intervention to install the Seccomp filters, but this can be easily automated as part of the compilation process. Seccomp filters are expressed as BPF programs, and once installed, they cannot be modified. However, if the `prctl` system call is not blocked, then it is possible to install new Seccomp filters. When two installed BPF programs contradict each other, the least permissive of the two takes precedence. Therefore, once a system call is prohibited, the attacker cannot remove it from the deny list. For example, if invoking `execve` is prohibited, and an attacker is able to install another BPF program that allows it, the deny list will have priority and `execve` will remain blocked. Furthermore, an installed Seccomp filter cannot be uninstalled without killing the process it has been applied to.

As an additional safeguard, the invocation of the `prctl` and `seccomp` system calls is prohibited as part of our Seccomp filtering at the beginning of the serving phase, if the application no longer needs them. This means that an attacker cannot install any new filters at all once the serving phase begins.

6 Experimental Evaluation

The main focus of our experimental evaluation lies on assessing the *additional* attack surface reduction achieved by temporal specialization compared to library specialization techniques, and evaluating its security benefits. For all experiments, we used a set of six very popular server applications: Nginx, Apache Httpd, Lighttpd, Bind, Memcached, and Redis.

Existing library specialization techniques [12, 51] only remove unused code, and do not actually perform any kernel-backed system call filtering (e.g., using Seccomp). That is, although the Libc functions corresponding to some system calls may be removed, the attacker is still able to directly invoke those system calls, e.g., as part of injected shellcode or a code reuse payload. Still, such a capability is relatively easy to implement once the unused Libc functions have been identified. In fact, for our evaluation purposes, we developed our own library specialization tool, similar to piecewise compilation [51], and on top of it implemented the capability of applying Seccomp filters to actually block the execution of system calls that correspond to removed Libc functions (unless they are also invoked directly by other parts of the application, in which case they cannot be disabled). Piecewise compilation leverages the SVF [55] tool to perform points-to analysis and generate the call graph for each library. Our custom library specialization tool also uses SVF to create call graphs for each library and further extends them to extract the list of system calls required for each application.

For our security evaluation, we explore two aspects of the protection offered by temporal specialization. First, we evaluate its effectiveness in blocking exploit code using a large set of shellcode and ROP payload samples. To account for potential evasion attempts using alternative system call combinations, we also exhaustively generate all possible variants of each sample. Second, given that system calls are the gateway to exploiting kernel vulnerabilities, we also look into the number of Linux kernel CVEs that are neutralized once the relevant system calls have been blocked.

We also validated the correctness of our implementation by applying temporal specialization and running each application with various workloads. For each application, we performed

Table 1: "Argument type" and "address taken" pruning reduce the number of spurious edges on the call graph significantly.

Application	SVF	+ Arg. Type	+ Address Taken
Nginx	38.2K	11.6K	11.5K
Apache Httpd	23.8K	12.4K	11.1K
Lighttpd	3.0K	2.7K	2.7K
Bind	67.9K	33.7K	33.3K
Memcached	7.6K	6.2K	5.8K
Redis	33.8K	18.6K	18.6K

100 client requests and validated the responses, without encountering any issues. We also compared the server logs in both cases to further ensure the absence of any internal errors that are not visible at the client side.

6.1 Call Graph Analysis

Identifying the transition boundary between the initialization and serving phase for the applications we consider is straightforward. We begin by providing some further details for each application. For Nginx [7], we use the default configuration with all the default modules enabled. Nginx has three functions that can act as transition points to the serving phase: `ngx_worker_process_cycle` and `ngx_single_process_cycle` are used for handling client requests, while `ngx_cache_manager_process_cycle` is responsible for cache management. Each of them runs in its own separate thread.

We use the vanilla configuration of Apache Httpd [15], statically compiled with `libapr` and `libapr-util` to make our analysis simpler. Our configuration enables all default modules. The transition boundary of the serving phase is the `child_main` function.

Lighttpd has an event-driven architecture, not relying on a primary–secondary process model. It can be launched with a configurable number of processes, and each process executes the `server_main_loop` function to handle client requests.

Bind is one of the most widely used DNS servers, acting as both an authoritative name server and as a recursive resolver. Bind uses multi-threading to handle client requests and enters the serving phase after creating the secondary threads, by invoking the `isc_app_ctxrun` function.

Memcached and Redis are both in-memory key-value databases. Similarly to Lighttpd, Memcached also has an event-driven architecture and executes the `worker_libevent` function to serve client requests. In Redis, the `aeMain` function serves as the event processing loop.

As shown in Table 1, these applications vary in complexity, with the number of edges in the initial call graph (generated by SVF) ranging from 3K for Lighttpd to 67.9K for Bind. By applying our pruning techniques based on argument types and taken addresses, the precision of the points-to analysis improves significantly, reducing the number of spurious edges to half or even less, especially for the most complex applications. This improvement allows us to disable more system calls during the serving phase of each application. For example, in case of Apache Httpd, using the more imprecise results of SVF alone does not allow the removal of security-sensitive system calls such as `execve`.

Table 2: Breakdown of the time (in minutes) required for each step of our analysis.

Application	Bitcode Size(MB)	Default (min)	SVF w. Arg. Type	+ Addr. Taken	Temp. Total
Nginx	1.9	1	+80	+2	**83**
Apache	2.1	3	+13	+1	**17**
Lighttpd	1.0	1	+1	+1	**3**
Bind	11.0	3	+554	+5	**562**
Memcached	1.6	1	+1	+1	**3**
Redis	9.2	1	+21	+1	**23**

Table 3: Number of system calls retained (out of 333 available) after applying library debloating and temporal specialization.

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

The complexity of each application affects the analysis time required to generate the call graph. Table 2 shows the breakdown of the amount of time required for generating the call graph in each step, with the total time for the whole toolchain in the last column. We compiled each application 10 times and report the average time. The compilation time is only a few seconds different in each case for all applications. The analysis time ranges from three minutes for Lighttpd to more than nine hours for Bind. The most time-consuming aspect of our approach is running Andersen's points-to analysis algorithm, which is expected. While one could use other algorithms, such as Steensgard's [54], which are both more efficient and more scalable, they come at the price of precision. We discuss other algorithms and tools which can be used to generate call graphs in Section 7.

6.2 Filtered System Calls

We compare our approach with library specialization [12, 51] to show the benefit of applying temporal specialization. As shown in Table 3, once entering the long-term serving phase, temporal specialization retains fewer system calls than static

Table 4: Critical system calls removed by library specialization ("Lib.") and temporal specialization ("Temp.").

	Syscall	Nginx		Apache Httpd		Lighttpd		Bind		Memcached		Redis	
		Lib.	Temp.	Lib.	Temp.	Lib.	Temp.	Lib.	Temp.	Lib.	Temp.	Lib.	Temp.
Cmd Execution	`clone`	✗	✓	✗	✗*	✗	✗*	✗	✗*	✗	✗*	✗	✗*
	`execveat`	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	`execve`	✗	✓	✗	✓	✗	❖	✓	✓	✓	✓	✗	❖
	`fork`	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	`ptrace`	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Permission	`chmod`	✗	✗	✓	✓	✗	✓	✗	✓	✓	✓	✗	✓
	`mprotect`	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
	`setgid`	✗	✗	✗	✓	✗	✓	✗	✓	✗	✓	✓	✓
	`setreuid`	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	`setuid`	✗	✗	✗	✓	✗	✓	✗	✓	✗	✓	✓	✓
Networking	`accept4`	✗	✗	✗	✗	✗	✗	✓	✓	✗	✗	✓	✓
	`accept`	✗	✗	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗
	`bind`	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗
	`connect`	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
	`listen`	✗	✗	✗	✓	✗	✓	✗	✗	✗	✗	✗	✓
	`recvfrom`	✗	✗	✓	✓	✗	✗	✗	✗	✗	✗	✓	✓
	`socket`	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗

✓: System call is removed. ✗: System call is not removed.

❖: Can be mitigated by applying configuration-driven debloating [34] (details in Section 6.2).

* : Can be mitigated by applying API specialization (details in Section 6.3).

library debloating. Although in most cases the reduction is not significant (in the best case for Bind, the number of system calls drops from 127 to 85, while in the worst case for Nginx, only 7 system calls are removed), a more crucial question is whether the removed system calls are "critical" or not, i.e., whether they will hinder the execution of exploit code that relies on them.

As a first step towards answering this question, Table 4 shows which critical system calls are filtered in each application after applying library debloating and temporal specialization. We chose a set of 17 security-critical system calls which are used as part of shellcode and ROP payloads (more details on this data set are provided in Section 6.3). As shown in Table 4, temporal specialization removes a total of 53 critical system calls across all applications, compared to just 35 for library debloating alone—an overall increase by 51%.

We group these system calls according to their functionality into three categories to analyze further the impact of temporal specialization. *Command Execution* includes system calls used to execute arbitrary code. *Permission* includes system calls which can be used to modify user, file, or memory permissions. *Networking* contains system calls mostly used in establishing network connections.

Command Execution The system calls `execveat`, `fork` and `ptrace` can be filtered across all applications by both techniques. No application uses `execveat` or `ptrace`. In place of the former most use `execve`, while the use of the latter is rare. The reason no application uses the `fork` system call is that Libc's `fork` function actually uses the `clone` system call. The widely used by exploit code `execve` system call is also used in many applications to spawn child processes, so it can not be removed by library debloating.

After entering the serving phase, however, most servers do not need to invoke `execve` anymore, and thus temporal specialization can remove it. This has significant security benefits, as also discussed in the next section. For Lighttpd and Redis, we manually verified that `execve` was invoked only if the application was launched with a specific run-time configuration option that is *disabled by default*. Therefore, the prior application of some form of configuration-driven debloating [34] would allow temporal specialization to successfully remove `execve` from all six applications.

Permissions Four of the permission system calls (`chmod`, `setgid`, `setuid`, and `setreuid`) can be filtered in all applications, except Nginx. As allocating memory and setting its permissions is a crucial operation for most applications, the `mprotect` system call cannot be filtered under any circumstances. As we discuss in Section 6.3, we could still enforce a more restrictive invocation policy for this system call by limiting the allowable permissions to be applied on memory pages, as after the initialization phase it is unlikely that *executable* memory will need to be allocated.

Table 5: Equivalent system calls.

System call	Equivalent System call(s)
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

Networking Neither approach can filter system calls used for creating network connections (socket, connect). This is because server applications may establish connections with other backend services, such as databases.

Although we expected listen and bind to be removed by temporal specialization, as these operations are typically part of the initialization phase, they are only removed in Apache Httpd, Lighttpd and Redis (only listen). We suspect that the reason they remain in the rest is related to the remaining overapproximation in the call graph, and we plan to further analyze these cases as part of our future work.

6.3 Exploit Code Mitigation

To evaluate the security benefits of temporal specialization, we collected a large and diverse set of exploit payloads. This set consists of 53 shellcodes from Metasploit [5], 514 shellcodes from Shell-storm [9], and 17 ROP payloads (from PoCs and in-the-wild exploits). Shellcodes are generic and can work against every application. Although the ROP payload of a given proof-of-concept exploit is meant to be used against a specific application, since all these payloads use one or more system calls to interact with the operating system, their final intent can be generalized irrespective of the target application. Thus, for ROP payloads, we make the conservative assumption that each can be used against any of our test applications.

6.3.1 Shellcode Analysis

For Metasploit, we use the msfvenom utility to generate a binary for each of the 53 available Linux payloads. We then disassemble each generated file to extract the system calls used. Similarly, we extract the system calls used by the 514 payloads collected from Shell-storm. Finally, we compare the set of system calls used in each payload with the set of system calls available in each application after applying library specialization and temporal specialization, to get the number of shellcodes "broken" in each case. We consider a payload broken if at least one of the system calls it relies on is removed. For instance, the bind_tcp shellcode uses six system calls: setsockopt, socket, bind, mprotect, accept and listen. Temporal specialization blocks bind in Lighttpd and Apache Httpd, and the attacker can no longer successfully run this shellcode.

To account for potential evasion attempts by swapping blocked system calls with equivalent ones, we also exhaustively generate all possible variants of each shellcode using other system call combinations that provide the same functionality. For instance, replacing accept with accept4 maintains the same functionality, but would allow an attacker to bypass a filter that restricts only one of them. Starting from our initial set of 567 shellcodes, we generate 1726 variants according to the equivalent system calls listed in Table 5.

We have summarized the results regarding the number of blocked shellcodes for each application by each specialization technique in Table 6. As shown in the row titled "All Shellcodes," for each of the six tested applications, temporal specialization successfully breaks a higher number of shellcode variants compared to library debloating. The improvement is significant in Lighttpd (1248 with temporal vs. 919 with library specialization), Apache Httpd (1466 vs. 1097), Nginx (1249 vs. 923), and Redis (1307 vs. 1165), while it is marginal for Memcached (1319 vs. 1258) and Bind (1341 vs. 1258).

Payloads can be categorized according to the task they perform. The broad categories include i) payloads that open a port and wait for the attacker to connect and launch a shell, ii) payloads that connect back and launch a reverse shell, iii) payloads that execute arbitrary commands, and iv) payloads that perform system operations, e.g., access a file or add a user. The first four rows in Table 6 provide the number of broken payloads in each of these categories. We see that 90% of the payloads that open a port are broken with temporal specialization. For Apache Httpd, although 88% of the "connect" and 91% of the "execute" shellcodes are broken with our approach, none of the two specialization schemes perform well for payloads that perform file operations. This is because file system operations are required by applications during both the initialization and the serving phases.

Achieving arbitrary remote code execution provides an attacker the ultimate control over a target system. Removing the ability to execute commands thus has a more significant impact on restricting an attacker's actions compared to blocking payloads of other categories, e.g. payloads that open a port. The execve system call is the most crucial for executing arbitrary commands, and as shown in Table 4, it can be removed in Apache Httpd, Nginx, Memcached and Bind by applying temporal specialization. This can also be seen in the row titled "Execute Command" in Table 6, where more than 80% of the shellcodes that aim to achieve arbitrary command execution are broken in Nginx, Apache Httpd, Bind, and Memcached. In these cases, the attacker is heavily restricted, and even if payloads in other categories (e.g., network connection establishment) are successful, the capability of executing arbitrary commands is still restricted.

Table 6: Number (and percentage) of payloads broken by library ("Lib.") and temporal ("Temp.") specialization for each category.

Payload Category	Count	Nginx		Apache Httpd		Lighttpd		Bind		Memcached		Redis	
		Lib.	Temp.	Lib.	Temp.	Lib.	Temp.	Lib.	Temp.	Lib.	Temp.	Lib.	Temp.
Open Port	560	334 (60%)	471 (84%)	199 (71%)	546 (98%)	330 (59%)	525 (94%)	500 (89%)	505 (90%)	471 (84%)	479 (85%)	439 (78%)	527 (94%)
Create Connection	366	245 (67%)	313 (86%)	268 (73%)	321 (87%)	263 (71%)	271 (74%)	313 (85%)	314 (85%)	289 (79%)	314 (85%)	280 (76%)	293 (80%)
Execute Command	408	223 (54%)	340 (83%)	247 (60%)	370 (91%)	223 (54%)	273 (67%)	338 (83%)	358 (88%)	352 (86%)	362 (89%)	259 (63%)	274 (67%)
System Operations	392	121 (30%)	125 (32%)	183 (46%)	229 (58%)	103 (26%)	179 (46%)	107 (27%)	164 (42%)	146 (37%)	164 (42%)	187 (47%)	213 (54%)
All Shellcodes	1726	923 (53%)	**1249 (72%)**	1097(63%)	**1466 (85%)**	919 (53%)	**1248 (72%)**	1258 (72%)	**1341 (78%)**	1258 (72%)	**1319 (77%)**	1165(68%)	**1307 (76%)**
Change Permission	3	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)
Execute Command	14	7 (50%)	14 (100%)	7 (50%)	14 (100%)	7(50%)	7 (50%)	14 (100%)	14 (100%)	14 (100%)	14 (100%)	7 (50%)	7 (50%)
All ROP Payloads	17	7 (41%)	**14 (82%)**	7 (41%)	**14 (82%)**	7 (41%)	7 (41%)	14 (82%)	14 (82%)	14 (82%)	14 (82%)	7 (41%)	7 (41%)

6.3.2 ROP Payload Analysis

We collected a set of 17 publicly available ROP payload samples for Linux (details for each one are provided in Table 8 in the appendix). We follow the same strategy as with the shellcodes and make the generic assumption that each of the payloads can be used against any of our tested applications.

From a system call perspective, ROP payloads are much simpler and usually aim towards either allocating executable memory for enabling the execution of second-stage shellcode, or invoking `execve` or similar system calls for direct command execution. ROP payloads can thus be broadly categorized into these two categories. The last three rows in Table 6 provide the number of ROP payloads in the two categories and their combined results. Ten ROP payloads attempt to execute commands and temporal specialization blocks all of them in four applications (Apache Httpd, Nginx, Memcached and Bind). In case of Lighttpd and Redis, because `execve` is used even in the serving phase (when used with a specific non-default configuration), temporal specialization cannot filter it. Neither library nor temporal specialization can block any payloads that try to change in-process memory permissions. This is because `mprotect` is used by all applications for memory allocation and permission assignment.

6.3.3 What Else can Attackers Do?

Assuming command execution (e.g., through `execve` and the like) has been blocked, attackers may resort to other system calls to achieve their goals. Meterpreter [6] is an advanced payload that uses DLL injection to inject malicious code into a process. Using such a payload would remove the requirement of using `execve` directly to launch external binaries, and instead allows the attacker to inject the necessary code to perform any operation as part of the vulnerable process itself. While Meterpreter (in its original form) is only available for Windows, there are equivalents for Linux which use the `ptrace` system call. However, none of the applications in our dataset require this system call, so it can be filtered in all cases. Furthermore, by default, this capability is limited to processes that have a predefined relationship with the target process since Linux kernel v3.2 due to the associated security risks. The traced process should either be a child process of the tracer, or should have tracing enabled using `prctl`.

Even if `ptrace` is not available, there are other system call combinations that could be leveraged to perform DLL injection. For example, Linux applications have the option of dynamically loading shared objects after program launch, using the `dlopen` and `dlsym` functions. Even if these two functions are not available, the attacker can simply emulate their functionality using the `open`, `mmap`, and `close` system calls to inject a malicious library. Given that these are very basic operations, it is unlikely that library or temporal specialization will be able to remove these system calls. However, a crucial requirement of DLL injection is to place the injected DLL in *executable* memory. When an application enters its serving phase it will definitely need `mmap` to allocate memory, but this memory is typically used for data which is not executable. Applying argument-level API specialization [41] in this case would prevent the attacker from mapping executable memory once the application enters the serving phase, thereby preventing these attacks.

The set of system calls used for file operations can also be leveraged by an attacker to gain command execution. Consider the case of an attacker writing to a file in the `crontab` folder by invoking `open`, `write`, and `close`. In this case, the `crond` service will run an attacker-controlled script which gives them the capability of executing arbitrary commands. While applying argument-level API specialization [41] can potentially protect against such a scenario (assuming the file paths can be predetermined), our approach cannot prevent such cases in general if file permissions are not set properly. For instance, regular programs should not have write access to sensitive folders like `crontab`.

6.4 Kernel Security Evaluation

System calls are the main entry point into the kernel. Although system calls (especially security-critical ones) are mainly used by attackers to perform unauthorized operations as part of exploiting a vulnerable process, they can also be used to exploit vulnerabilities in the underlying kernel. Previous works [31–33, 46] have shown that malicious users can target the kernel to perform privilege escalation or leak sensi-

Table 7: Kernel CVEs mitigated by filtering unneeded system calls.

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

tive information. In most cases, these attacks are performed by exploiting a kernel vulnerability that is triggered through a system call, when invoked with specially crafted arguments. By disabling system calls associated with kernel vulnerabilities we can thus reduce the attack surface of the kernel that is exposed to attackers. While filtering *security-critical* system calls is of importance in case of user-space vulnerability exploitation, it is important to note that *any* system call associated with a kernel vulnerability can be exploited to mount privilege escalation attacks.

To gain a better understanding of how filtering individual system calls impacts mitigating potential kernel vulnerabilities, we constructed the Linux kernel's call graph using KIRIN [64]. This allows us to identify all functions that are invoked as a result of specific system call invocations, and thus reason about which part of the kernel's code—and therefore which vulnerabilities—become inaccessible when blocking a given set of system calls.

To perform our analysis, we crawled the CVE website [1] for Linux kernel vulnerabilities using a custom automated tool. Our tool extracts each CVE's relevant commit, and after parsing it in the Linux kernel's Git repository, finds the corresponding patch, and retrieves the relevant file and function that was modified by the patch. We discovered that while there were only a few CVEs directly associated with filtered system call code, many CVEs were associated with files and functions that were invoked exclusively by filtered system call code. By matching the CVEs to the call graph created by KIRIN, we were able to pinpoint all the vulnerabilities that are related to the set of system calls filtered by a given application under each specialization mechanism. This provides us with a metric to assess the attack surface reduction achieved by temporal specialization at the kernel level. This reduction is reflected in the number of CVEs neutralized for a given application after applying our Seccomp filters at the beginning of the serving phase.

Based on our analysis, a total of 53 CVEs are effectively removed in at least one of the six applications (i.e., the respective vulnerabilities cannot be triggered by the attacker) by temporal specialization. Out of the 53 vulnerabilities that can be mitigated by temporal specialization, 40 can be mitigated by system call filtering based on library debloating as well. Table 6 shows the 13 CVEs that are neutralized by temporal specialization, and which cannot be neutralized by library specialization in some or all applications. The last two columns correspond to the number of applications for which the CVE is neutralized for library debloating and temporal specialization, respectively.

7 Discussion and Limitations

Our approach does not remove any *code* from the protected program, and consequently cannot mitigate any vulnerabilities in the application itself, or reduce the code that could be reused by an attacker.

Similarly to other attack surface reduction techniques, the effectiveness of temporal specialization varies according to the specific requirements of a given application, and as our results show, it may not prevent all possible ways an attacker can perform harmful interactions with the OS. Our equivalent system call analysis attempts to quantify the evasion potential by replacing system calls with others, but depending on the attacker's specific goals, there may be more creative ways to accomplish them using the remaining system calls. For example, without our technique, an attacker could read the contents of a file simply by executing the `cat` program. Once the `execve`-family of system calls are blocked, the attacker would have to implement a more complex shellcode to open and read the file and write it to an already open socket. As part of our future work, we plan to extend our analysis by extracting the arguments passed to system calls and constraining them as well [41, 42]. This would further limit the attacker's capabilities when using the remaining system calls.

Although we have considered only server applications in this work, there could be benefit in applying temporal specialization to some client applications. In general, any application that follows the *initialization/serving* phase execution model can benefit from our approach. Examples of desktop

applications which follow this model are ssh-agent [61] and syslog-ng [10]. Further analysis of how well these applications follow the two-stage execution model has been left for future work.

Due to multiple inheritance with support for polymorphism in C++, our type-based matching currently supports only C code. We plan to extend our approach to support applications developed in C++ as part of our future work.

Additionally, we plan to investigate the use of alternative points-to analysis algorithms. In particular, the authors of TeaDSA [36], which is the type-aware implementation of SeaDSA [23], report better accuracy than SVF in some cases (typically for C++ applications) and worse in others (C applications). The authors acknowledge that TeaDSA is more precise for C++ applications than SVF. However, for C applications (e.g., OpenSSL), their results show that it is less precise than SVF. Moreover, the comparison in the paper is with the type-unaware SVF. Because most server applications are written in C, we anticipate the accuracy of our type-based pruning to be better than type-aware SeaDSA. Unfortunately we could not get TeaDSA to work with our applications due to crashes. We will explore TeaDSA and other points-to analysis algorithms as part of our future work.

Applications can dynamically load libraries through the `dlopen` and `dlsym` functions. Due to the dynamic nature of this feature, our current prototype does not support it.

8 Related Work

System call filtering based on policies derived through static or dynamic analysis has been widely used in host-based intrusion detection [18–20, 29, 35, 44, 52, 58]. Since in this paper we focus on attack surface reduction through software specialization, we mainly discuss related works in this context.

Application Debloating Many previous works have focused on reducing the attack surface by removing unused code from the application's process address space. Mulliner and Neugschwandtner [43] proposed one of the first approaches for preforming library debloating by removing non-imported functions from shared libraries at load time. Quach et al. [51] improve library debloating by extending the compiler and the loader to remove all unused functions from shared libraries at load time. Agadakos et al. [12] propose a similar library debloating approach at the binary level, through function boundary detection and dependency identification.

Porter et al. [47] also perform library debloating, but load library functions only when requested by the application. While this is similar to our approach in taking the program execution phase into account, library functions are loaded and unloaded based on the need of the application, whereas we install restrictive filters (which cannot be removed) after the execution enters the serving phase.

Davidsson et al. [16] analyze the complete software stack for web applications to create specialized libraries based on the requirements of both the server application binaries and PHP code. Song et al. [53] apply data dependency analysis to perform fine-grained library customization of statically linked libraries. Shredder [41] instruments binaries to restrict arguments passed to critical system API functions to a predetermined legitimate of possible values. Saffire [42] performs call-site-specific argument-level specialization for functions at build time.

Another line of research on debloating focuses on using training to identify unused sections of applications. Qian et al. [48] use training and heuristics to identify unnecessary basic blocks and remove them from the binary without relying on the source code. Ghaffarinia and Hamlen [21] use a similar approach based on training to limit control flow transfers to unauthorized sections of the code.

Other works explore the potential of debloating software based on predefined feature sets. CHISEL [26] uses reinforcement learning to debloat software based on test cases generated by the user. TRIMMER [25] finds unnecessary basic blocks using an inter-procedural analysis based on user-defined configurations. DamGate [63] rewrites binaries with *gates* to prevent execution of unused features.

While the above works focus on C/C++ applications, other works specifically focus on the requirements of other programming languages [30, 57, 62]. Jred [62] uses static analysis on Java code to identify and remove unused methods and classes. Jiang et al. [30] used data flow analysis to implement a feature-based debloating mechanism for Java. Azad et al. [13] propose a framework for removing unnecessary features from PHP applications through dynamic analysis.

Kernel and Container Debloating KASR [65] and FACE-CHANGE [66] use dynamic analysis to create kernel profiles for each application by using training to identify used parts of the kernel. Kurmus et al. [37] propose a method to automatically generate kernel configuration files to tailor the Linux kernel for specific workloads. Similarly, Acher et al. [11] use a statistical supervised learning method to create different sets of kernel configuration files. Sysfilter [17] is a static binary analysis framework that reduces the kernel's attack surface by restricting the system calls available to user-space processes.

Wan et al. [59] use dynamic analysis to profile the required system calls of a container and generate relevant Seccomp filters. Due to the incompleteness of dynamic analysis, Confine [22] uses static analysis to create similar Seccomp profiles to filter unnecessary system calls from containers. DockerSlim [2] is an open source tool which also relies on dynamic analysis to remove unnecessary files from Docker images. Similar to temporal debloating, SPEAKER [38] separates the required system calls of containers in two main phases, booting and runtime. The approach only targets containers and relies on training to identify the system calls for each phase.

9 Conclusion

We presented temporal system call specialization, a novel approach for limiting the system calls that are available to server applications after they enter their serving or stable state. Compared to previous software specialization approaches, which consider the whole lifetime of a program, temporal specialization removes many additional system calls, including dangerous ones such as `execve`, which are typically required by server applications only during their initialization phase.

For a given server application, we perform static analysis of the main program and all imported libraries to extract the set of system calls which are no longer used after the transition into the serving phase. As precise call graph generation is a known problem in static analysis, we perform multiple optimizations on top of existing points-to analysis algorithms to reduce the imprecision of the call graph, which helps in identifying a near-accurate set of used system calls.

We demonstrate the effectiveness of temporal specialization by evaluating it with six well known server applications against a set of shellcodes and ROP payloads. We show that temporal specialization disables 51% more security-critical system calls compared to existing library specialization approaches, breaking 77% of the shellcodes and 68% of the ROP payloads tested. In addition, 53 Linux kernel CVEs are mitigated once temporal specialization comes into effect, 13 of which are not preventable by library specialization.

As a best-effort attack surface reduction solution, temporal specialization is practical, easy to deploy, and significantly restricts an attacker's capabilities.

Acknowledgments

We thank our shepherd, Claudio Canella, the anonymous reviewers, and the members of the artifact evaluation committee for their helpful feedback. This work was supported by the Office of Naval Research (ONR) through award N00014-17-1-2891, the National Science Foundation (NSF) through award CNS-1749895, and the Defense Advanced Research Projects Agency (DARPA) through award D18AP00045. Any opinions, findings, and conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the ONR, NSF, or DARPA.

References

[1] Common vulnerabilities and exposures database. `https://www.cvedetails.com`.

[2] DockerSlim. `https://dockersl.im`.

[3] The LLVM compiler infrastructure. `http://llvm.org`.

[4] Memcached. `https://memcached.org/`.

[5] Metasploit framework. `http://www.metasploit.com`.

[6] Meterpreter. `https://github.com/rapid7/metasploit-framework/wiki/Meterpreter/`.

[7] Nginx. `https://www.nginx.com/`.

[8] Seccomp BPF (SECure COMPuting with filters). `https://www.kernel.org/doc/html/v4.16/userspace-api/seccomp_filter.html`.

[9] Shell-storm. `http://www.shell-storm.org`.

[10] Syslog NG. `https://www.syslog-ng.com/`.

[11] Mathieu Acher, Hugo Martin, Juliana Pereira, Arnaud Blouin, Jean-Marc Jézéquel, Djamel Khelladi, Luc Lesoil, and Olivier Barais. Learning very large configuration spaces: What matters for Linux kernel sizes. Technical Report HAL-02314830, Inria Rennes - Bretagne Atlantique, 2019.

[12] Ioannis Agadakos, Di Jin, David Williams-King, Vasileios P Kemerlis, and Georgios Portokalidis. Nibbler: Debloating binary shared libraries. In *Proceedings of the 35th Annual Computer Security Applications Conference (ACSAC)*, pages 70–83, 2019.

[13] Babak Amin Azad, Pierre Laperdrix, and Nick Nikiforakis. Less is more: Quantifying the security benefits of debloating web applications. In *Proceedings of the 28th USENIX Security Symposium*, 2019.

[14] Lars Ole Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, University of Cophenhagen, 1994.

[15] Apache. Apache Httpd, 2019. `https://httpd.apache.org/`.

[16] Nicolai Davidsson, Andre Pawlowski, and Thorsten Holz. Towards automated application-specific software stacks. In *Proceedings of the 24th European Symposium on Research in Computer Security (ESORICS)*, 2019.

[17] Nicholas DeMarinis, Kent Williams-King, Di Jin, Rodrigo Fonseca, and Vasileios P. Kemerlis. Sysfilter: Automated system call filtering for commodity software. In *Proceedings of the International Conference on Research in Attacks, Intrusions, and Defenses (RAID)*, 2020.

[18] Henry Hanping Feng, Jonathon T Giffin, Yong Huang, Somesh Jha, Wenke Lee, and Barton P Miller. Formalizing sensitivity in static analysis for intrusion detection. In *Proceedings of the IEEE Symposium on Security & Privacy (S&P)*, pages 194–208, 2004.

[19] Stephanie Forrest, Steven A Hofmeyr, Anil Somayaji, and Thomas A Longstaff. A sense of self for Unix processes. In *Proceedings of the IEEE Symposium on Security & Privacy (S&P)*, pages 120–128, 1996.

[20] Tal Garfinkel, Ben Pfaff, and Mendel Rosenblum. Ostia: A delegating architecture for secure system call interposition. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2004.

[21] Masoud Ghaffarinia and Kevin W. Hamlen. Binary control-flow trimming. In *Proceedings of the 26th ACM Conference on Computer and Communications Security (CCS)*, 2019.

[22] Seyedhamed Ghavamnia, Tapti Palit, Azzedine Benameur, and Michalis Polychronakis. Confine: Automated system call policy generation for container attack surface reduction. In *Proceedings of the International Conference on Research in Attacks, Intrusions, and Defenses (RAID)*, 2020.

[23] Arie Gurfinkel and Jorge A Navas. A context-sensitive memory model for verification of C/C++ programs. In *Proceedings of the International Static Analysis Symposium*, pages 148–168. Springer, 2017.

[24] Andreas Gustafsson. Egypt. `https://www.gson.org/egypt/egypt.html`.

[25] Ashish Gehani Hashim Sharif, Muhammad Abubakar and Fareed Zaffar. Trimmer: Application specialization for code debloating. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*, 2018.

[26] Kihong Heo, Woosuk Lee, Pardis Pashakhanloo, and Mayur Naik. Effective program debloating via reinforcement learning. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, 2018.

[27] Michael Hind. Pointer analysis: Haven't we solved this problem yet? In *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, pages 54–61, 2001.

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

[29] Kapil Jain and R Sekar. User-level infrastructure for system call interposition: A platform for intrusion detection and confinement. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2000.

[30] Yufei Jiang, Can Zhang, Dinghao Wu, and Peng Liu. Feature-based software customization: Preliminary analysis, formalization, and methods. In *Proceedings of the 17th IEEE International Symposium on High Assurance Systems Engineering (HASE)*, 2016.

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

[32] Vasileios P. Kemerlis, Michalis Polychronakis, and Angelos D. Keromytis. ret2dir: Rethinking kernel isolation. In *Proceedings of the 23rd USENIX Security Symposium*, pages 957–972, 2014.

[33] Vasileios P. Kemerlis, Georgios Portokalidis, and Angelos D. Keromytis. kguard: Lightweight kernel protection against return-to-user attacks. In *Proceedings of the 21st USENIX Security Symposium*, 2012.

[34] Hyungjoon Koo, Seyedhamed Ghavamnia, and Michalis Polychronakis. Configuration-driven software debloating. In *Proceedings of the 12th European Workshop on Systems Security*, 2019.

[35] Christopher Kruegel, Engin Kirda, Darren Mutz, William Robertson, and Giovanni Vigna. Automating mimicry attacks using static binary analysis. In *Proceedings of the USENIX Security Symposium*, 2005.

[36] Jakub Kuderski, Jorge A Navas, and Arie Gurfinkel. Unification-based pointer analysis without oversharing. In *Proceedings of the Formal Methods in Computer Aided Design (FMCAD)*, pages 37–45. IEEE, 2019.

[37] Anil Kurmus, Reinhard Tartler, Daniela Dorneanu, Bernhard Heinloth, Valentin Rothberg, Andreas Ruprecht, Wolfgang Schroder-Preikschat, Daniel Lohmann, and Rudiger Kapitza. Attack surface metrics and automated compile-time OS kernel tailoring. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2013.

[38] Lingguang Lei, Jianhua Sun, Kun Sun, Chris Shenefiel, Rui Ma, Yuewu Wang, and Qi Li. SPEAKER: Split-phase execution of application containers. In *Proceedings of the 12th Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, pages 230–251, 2017.

[39] 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 *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2017.

[40] Steven McCanne and Van Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *Proceedings of the USENIX Winter Conference*, 1993.

[41] Shachee Mishra and Michalis Polychronakis. Shredder: Breaking Exploits through API Specialization. In *Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC)*, 2018.

[42] Shachee Mishra and Michalis Polychronakis. Saffire: Context-sensitive function specialization against code reuse attacks. In *Proceedings of the 5th IEEE European Symposium on Security and Privacy (EuroS&P)*, 2020.

[43] Collin Mulliner and Matthias Neugschwandtner. Breaking payloads with runtime code stripping and image freezing, 2015. Black Hat USA.

[44] Chetan Parampalli, R Sekar, and Rob Johnson. A practical mimicry attack against powerful system-call monitors. In *Proceedings of the ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, pages 156–167, 2008.

[45] Fernando Magno Quintao Pereira and Daniel Berlin. Wave propagation and deep propagation for pointer analysis. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 126–135, 2009.

[46] Marios Pomonis, Theofilos Petsios, Angelos D. Keromytis, Michalis Polychronakis, and Vasileios P. Kemerlis. kRˆX: Comprehensive kernel protection against just-in-time code reuse. In *Proceedings of the 12th European Conference on Computer Systems (EuroSys)*, pages 420–436, 2017.

[47] Chris Porter, Girish Mururu, Prithayan Barua, and Santosh Pande. 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 (PLDI)*, pages 164–180, 2020.

[48] Chenxiong Qian, Hong Hu, Mansour Alharthi, Pak Ho Chung, Taesoo Kim, and Wenke Lee. RAZOR: A framework for post-deployment software debloating. In *Proceedings of the 28th USENIX Security Symposium*, 2019.

[49] Anh Quach, Rukayat Erinfolami, David Demicco, and Aravind Prakash. A multi-OS cross-layer study of bloating in user programs, kernel and managed execution environments. In *Proceedings of the Workshop on Forming an Ecosystem Around Software Transformation (FEAST)*, pages 65–70, 2017.

[50] Anh Quach and Aravind Prakash. Bloat factors and binary specialization. In *Proceedings of the 3rd ACM Workshop on Forming an Ecosystem Around Software Transformation (FEAST)*, pages 31–38, 2019.

[51] Anh Quach, Aravind Prakash, and Lok Yan. Debloating software through piece-wise compilation and loading. In *Proceedings of the 27th USENIX Security Symposium*, pages 869–886, 2018.

[52] Mohan Rajagopalan, Matti Hiltunen, Trevor Jim, and Richard Schlichting. Authenticated system calls. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, pages 358–367, 2005.

[53] Linhai Song and Xinyu Xing. Fine-grained library customization. In *Proceedings of the 1st ECOOP International Workshop on Software Debloating and Delayering (SALAD)*, 2018.

[54] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 32–41, 1996.

[55] Yulei Sui and Jingling Xue. SVF: interprocedural static value-flow analysis in LLVM. In *Proceedings of the 25th International Conference on Compiler Construction*, 2016.

[56] Yulei Sui, Ding Ye, and Jingling Xue. Detecting memory leaks statically with full-sparse value-flow analysis. *IEEE Transactions on Software Engineering*, 40(2):107–122, 2014.

[57] Kanchi Gopinath Suparna Bhattacharya and Mangala Gowri Nanda. Combining concern input with program analysis for bloat detection. In *Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*, 2013.

[58] David Wagner and Drew Dean. Intrusion detection via static analysis. In *Proceedings of the IEEE Symposium on Security & Privacy*, pages 156–168, 2001.

[59] Zhiyuan Wan, David Lo, Xin Xia, Liang Cai, and Shanping Li. Mining sandboxes for Linux containers. In *Proceedings of the 10th IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 92–102, 2017.

[60] Jinpeng Wei and Calton Pu. TOCTTOU vulnerabilities in UNIX-style file systems: An anatomical study. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2005.

[61] Tatu Ylönen. SSH Agent. `https://www.ssh.com/ssh/agent`.

[62] Dinghao Wu Yufei Jiang and Peng Liu. Jred: Program customization and bloatware mitigation based on static analysis. In *Proceedings of the 40th Annual Computer Software and Applications Conference (ACSAC)*, 2016.

[63] Tian Lan Yurong Chen and Guru Venkataramani. Damgate: Dynamic adaptive multi-feature gating in program binaries. In *Proceedings of the Workshop on Forming an Ecosystem Around Software Transformation (FEAST)*, 2017.

[64] Tong Zhang, Wenbo Shen, Dongyoon Lee, Changhee Jung, Ahmed M. Azab, and Ruowen Wang. PeX: A

permission check analysis framework for linux kernel. In *Proceedings of the 28th USENIX Security Symposium*, pages 1205–1220, 2019.

[65] Zhi Zhang, Yueqiang Cheng, Surya Nepal, Dongxi Liu, Qingni Shen, and Fethi Rabhi. KASR: A reliable and practical approach to attack surface reduction of commodity OS kernels. In *Proceedings of the International Conference on Research in Attacks, Intrusions, and Defenses (RAID)*, pages 691–710, 2018.

[66] Xiangyu Zhang Zhongshu Gu, Brendan Saltaformaggio and Dongyan Xu. Face-change: Application-driven dynamic kernel view switching in a virtual machine. In *Proceedings of the 44th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2014.

A Appendix

Imprecision of Points-to Analysis

In Sections 4.2 and 5.1, we discussed how context sensitivity and path sensitivity contribute to the overapproximation problem and make the results of Andersen's analysis imprecise. While our experiences show that the lack of context sensitivity and path sensitivity are the primary contributors to this imprecision, other factors too contribute to overapproximation in the results of the points-to analysis.

Field Sensitivity The points-to analysis provided by the SVF library is field-sensitive. Field sensitivity allows every field of a `struct` to be uniquely modeled, which is critical for the precision of the analysis. For example, in case of Apache Httpd, the `cleanup_t` type contains function pointers for cleaning memory allocated on various heaps. To distinguish between the different function pointers in this structure, we must model the individual fields of the `struct cleanup_t` as field-sensitive. However, there are certain circumstances under which SVF forsakes field sensitivity in lieu of simplicity of implementation and reduction in analysis time.

Array Index Sensitivity SVF's implementation of Andersen's algorithm is not array-index-sensitive. Individual elements of an array are not modeled uniquely. Therefore, if multiple `struct` objects are stored in a array, the individual `struct` objects become field-insensitive, because the array elements themselves are not modeled uniquely.

For example, objects of type `ap_listen_rec` are stored in the array of pointers `listen_buckets`. The type `ap_listen_rec` has a field `accept_func` which stores a pointer to the function that is invoked on the *accept* event. As these objects are stored in an index-insensitive array, they lose their field sensitivity, and SVF cannot distinguish between the targets of the `accept_func` field and the targets of the other fields in `ap_listen_rec` that hold function pointers. Moreover, due to array index insensitivity, it is impossible to distinguish the `accept_func` field of one `ap_listen_rec` object, from the `accept_func` field of another object, stored in the same array.

Table 8: Linux ROP payloads used in our evaluation.

1)	Return Oriented Programming and ROPgadget tool http://shell-storm.org/blog/Return-Oriented-Programming-and-ROPgadget-tool/
2)	ARM Exploitation - Defeating DEP - executing mprotect() https://blog.3or.de/arm-exploitation-defeating-dep-executing-mprotect.html
3)	64-bit ROP | You rule 'em all! https://0x00sec.org/t/64-bit-rop-you-rule-em-all/1937
4)	64-bit Linux Return-Oriented Programming https://crypto.stanford.edu/~blynn/rop/
5)	Return-Oriented-Programming(ROP FTW) http://www.exploit-db.com/docs/english/28479-return-oriented-programming-(rop-ftw).pdf
6)	PMS 0.42 - Local Stack-Based Overflow (ROP) https://www.exploit-db.com/exploits/44426/
7)	Crashmail 1.6 - Stack-Based Buffer Overflow (ROP) https://www.exploit-db.com/exploits/44331/
8)	PHP 5.3.6 - Local Buffer Overflow (ROP) https://www.exploit-db.com/exploits/17486/
9)	HT Editor 2.0.20 - Local Buffer Overflow (ROP) https://www.exploit-db.com/exploits/22683/
10)	Bypassing non-executable memory, ASLR and stack canaries on x86-64 Linux https://www.antoniobarresi.com/security/exploitdev/2014/05/03/64bitexploitation/
11)	Bypassing non-executable-stack during Exploitation (return-to-libc) https://www.exploit-db.com/papers/13204/
12)	Exploitation - Returning into libc https://www.exploit-db.com/papers/13197/
13)	Bypass DEP/NX and ASLR with Return Oriented Programming technique https://medium.com/4ndr3w/linux-x86-bypass-dep-nx-and-aslr-with-return-oriented-programming-ef4768363c9a/
14)	ROP-CTF101 https://ctf101.org/binary-exploitation/return-oriented-programming/
15)	Introduction to return oriented programming (ROP) https://codearcana.com/posts/2013/05/28/introduction-to-return-oriented-programming-rop.html/
16)	Simple ROP Exploit Example https://gist.github.com/mayanez/c6bb9f2a26fa75261a9a26a0a637531b/
17)	Analysis of Defenses against Return Oriented Programming https://www.eit.lth.se/sprapport.php?uid=829/

Positive Weight Cycles Due to context insensitivity, especially for memory allocation wrappers, it is possible for the constraint graph to contain cycles. Cycle elimination [27] is a popular optimization in points-to analysis—the key idea being that constraint nodes that are part of a cycle in the constraint graph share the same solution, and therefore can be collapsed into a single node. However, cycle elimination is not trivial in field-sensitive analysis, because the edges between the constraint nodes are weighted (where the weight of the edge is the index of the field being accessed).

Moreover, SVF implements an optimization of Andersen's algorithm, called Wave Propagation [45]. This optimization requires the constraint graph to be topologically sorted, and that there are no edges. Due to this requirement, at the end of each iteration, SVF converts every field-sensitive `struct` object that is involved in a cycle into field-insensitive.