C2C: Fine-grained Configuration-driven System Call Filtering

Seyedhamed Ghavamnia
sghavamnia@cs.stonybrook.edu
Stony Brook University
Stony Brook, USA

Tapti Palit
tpalit@purdue.edu
Purdue University
West Lafayette, USA

Michalis Polychronakis
mikepo@cs.stonybrook.edu
Stony Brook University
Stony Brook, USA

ABSTRACT

Configuration options allow users to customize application features according to the desired requirements. While the code that corresponds to disabled features is never executed, it still resides in process memory and comprises part of the application's attack surface, e.g., it can be reused for the construction of exploit code. Automatically reducing the attack surface of disabled application features according to a given configuration is thus a desirable defense-in-depth capability. The intricacies of modern software design and the complexities of popular programming languages, however, introduce significant challenges in automatically deriving the mapping of configuration options to their corresponding application code.

In this paper, we present *Configuration-to-Code (C2C)*, a generic configuration-driven attack surface reduction technique that automatically maps configuration options to application code using static code analysis and instrumentation. C2C operates at a fine-grained level by pruning configuration-dependent conditional branches in the control flow graph, allowing the precise identification of a given configuration option's code at the basic block level. At runtime, C2C reduces the application's attack surface by filtering any system calls required exclusively by disabled features. Using popular applications, we show how security-critical system calls (such as `execve`) can be automatically disabled when not needed, limiting an attacker's vulnerability exploitation capabilities. System call filtering also reduces the exposed attack surface of the underlying Linux kernel, neutralizing 32 additional CVEs (for a total of 88) compared to previous software specialization techniques.

CCS CONCEPTS

• **Security and privacy → Software and application security**; *Operating systems security*.

KEYWORDS

Attack surface reduction; system call filtering; configuration

ACM Reference Format:
Seyedhamed Ghavamnia, Tapti Palit, and Michalis Polychronakis. 2022. C2C: Fine-grained Configuration-driven System Call Filtering. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS '22), November 7–11, 2022, Los Angeles, CA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3548606.3559366

CCS '22, November 7–11, 2022, Los Angeles, CA, USA

ACM ISBN 978-1-4503-9450-5/22/11…$15.00
https://doi.org/10.1145/3548606.3559366

1 INTRODUCTION

Software applications typically support a continuously increasing set of features and extensions [32, 58] that can be controlled by various runtime configuration options. Through these configuration options, users can alter program execution and tune it towards their needs. Configuration settings vary significantly across different applications in terms of the offered customization granularity. While some options just enable or disable features, others tweak their details, providing users fine-grained control over the program.

Although a given active configuration restricts the features available at runtime, the application code corresponding to these features and the underlying system calls it relies on always remain accessible in the program. This unused code is essentially software bloat, and is often capable of performing privileged operations through the invocation of critical system calls, such as spawning new processes and changing memory permissions.

Software debloating and specialization have recently gained traction as defense-in-depth techniques for reducing the attack surface of applications, by removing or restricting access to unneeded features. These include techniques applied to user-space applications [30, 31, 51, 56, 59], as well as techniques for kernel [71, 72] and container [27, 67] specialization. A common characteristic of these techniques is that they are *configuration agnostic*. They consider all possible configurations of a given application, and therefore overlook debloating and specialization opportunities that depend on different configuration settings. This results in the generation of policies that are more permissive than what is required by typical configurations used in practice.

The problem of statically mapping code snippets to configuration options for software debloating has not received enough attention. Previous works [43, 65] have only attempted to derive such a mapping for test case generation purposes—an admittedly simpler problem, due to the inherent tolerance for false negatives. Moreover, these techniques target simpler programming environments (from a program analysis perspective), such as the C pre-processor and Android applications. The prevalence of popular C/C++ applications forces us to handle the complexities of these languages. To the best of our knowledge, Koo et al. [38] were the first to illustrate the benefits of configuration-driven debloating. While they demonstrated a significant reduction in code size after debloating using various default configurations, their approach relied on differential testing that required significant manual involvement. The programmer is required to provide correct and valid values for *each* configuration option to correctly perform the differential testing. This process is infeasible for large applications such as Nginx and Apache, which have hundreds of configuration options. Moreover, their approach removes code only at the library level, leading to missed opportunities for code removal at a finer granularity.

Our goal in this work is to develop a system for performing *fine-grained* configuration-dependent attack surface reduction by minimizing the involvement of the programmer. To this end, we present *C2C*, a platform for mapping configuration options to application code and disabling unneeded system calls according to a given active configuration, reducing this way the attack surface exposed to adversaries. Although code removal is desirable, simply reducing the number of ROP gadgets has minimal security benefits [10]. In contrast, identifying unnecessary code and using it to build system call profiles has two benefits. First, it hinders user-level exploitation, and second, it neutralizes kernel vulnerabilities. C2C minimizes the burden on programmers by requiring them to provide only two simple inputs, the active configuration with which they wish to launch the application, and the list of *struct types* that store the active configuration values in the application code. Due to the various methods used by developers to store and apply options (configuration files, command-line options), automatically mapping these settings to code is challenging. We have analyzed these methods, and propose an automated framework for mapping a program's configuration options to code.

C2C uses static analysis techniques to derive a fine-grained mapping between runtime options and application code at the basic block level. Our analysis first determines the conditional branches that depend on runtime options, and then instruments the code with monitors that, at runtime, observe the values of the corresponding configuration options. Using these stored values, C2C determines if a configuration-dependent conditional branch is disabled or not, and restricts any system calls used exclusively by unreachable basic blocks. Ideally, static analysis would be able to precisely determine the value of a runtime option and map it accurately to its dependent conditional branch. Unfortunately, we cannot assume this is always the case due to the inherent overapproximation of static analysis. To preserve the soundness of our analysis, we relax our mappings from a *value-based* to a *use-based* approach. In situations where static analysis cannot derive the exact value of a given runtime configuration option, we conservatively consider that runtime option to be *used*, and retain all conditional branches that depend on it.

We implemented a Linux-based prototype of C2C on top of LLVM, and evaluated it with ten popular applications (Nginx, Apache Httpd, Lighttpd, Postgresql, OpenSMTPD, Redis, Memcached, Curl, Wget, Tar). To demonstrate the security benefits of C2C, we first show that many security-critical system calls, such as `execve`, can be disabled for common active configurations used by popular installations of these applications. Then, we evaluate how filtering the unneeded system calls neutralizes critical Linux kernel privilege escalation vulnerabilities by prohibiting access to the vulnerable kernel code. The results of our experimental evaluation show that C2C successfully disables security-critical system calls, including `execve` (Apache, Wget), `setuid` (Nginx), and `bind` (Postgresql), which are retained by previous library debloating [59] and temporal specialization [29] approaches. Similarly, C2C neutralizes 32 additional Linux kernel CVEs compared to previous approaches.

Based on our experience with developing C2C and analyzing popular applications, we observed that certain software design patterns make applications easier to analyze and adapt for configuration-driven debloating. In light of the promising results obtained by C2C for the evaluated applications, we share these insights and provide guidelines that we believe will encourage the integration of configuration-driven debloating in modern applications.

In summary, our work makes the following main contributions:

(1) We present C2C, a novel fine-grained configuration-driven software attack surface reduction technique, which generates restrictive system call filters based on the specific application configuration used at launch time.
(2) We evaluate our prototype implementation with ten popular applications and demonstrate how C2C can filter security-critical system calls, such as `execve`. We further show how C2C reduces the exposed kernel attack surface by neutralizing Linux kernel vulnerabilities.
(3) We provide a set of design guidelines for developers on how to structure and refactor software in a manner that simplifies its analysis and facilitates the mapping of configuration options to code.

Our prototype implementation is publicly available as an open-source prototype at https://github.com/shamedgh/c2c.

2 BACKGROUND AND MOTIVATION

2.1 Security Implications of Configuration Options

Users can modify the execution of applications through configuration options specified either during compilation, or when the application is launched. While many of these options have no side effects (e.g., specifying the path to an output file), others can lead to the execution of sensitive operations, such as spawning new processes or launching new executables. In this section, we motivate the need for configuration-aware attack surface reduction by demonstrating how some configuration options introduce security-critical system calls into the attack surface of an application.

Consider Apache Httpd's essential SSL plugin, which must be enabled to support HTTPS. Among the plethora of options it supports, one allows the invocation of an external binary for the generation of random numbers to seed the key generation process. Line 11 in Listing 1 shows the code that checks if this runtime option has been configured to launch an external program, and line 12 shows the `ssl_util_ppopen` function which internally invokes the `execve` system call to launch the external program. The SSL plugin invokes the security-critical `execve` system call *only* if the `nSrc` field of the `pRandSeed` object is initialized with the `SSL_RSSRC_EXEC` value—`execve` is not required under any other circumstances.

```
1  typedef struct {
2    ssl_rsctx_t  nCtx;
3    ssl_rssrc_t  nSrc;
4    char        *cpPath;
5    int          nBytes;
6  } ssl_randseed_t;
7
8  int ssl_rand_seed(server_rec *s ...){
9    SSLModConfigRec *mc;
10   ssl_randseed_t *pRandSeed = getRandSeed(mc);
11   if (pRandSeed->nSrc == SSL_RSSRC_EXEC)
12     fp = ssl_util_ppopen(s, p, cmd,argv);
13 }
```

Listing 1: Simplified code from Apache Httpd showing a branch that depends on a configuration-related struct type.

Across the many configurations used by different production Docker images that we analyzed, including Bitnami Wordpress [16], Mediawiki [18], and Drupal [17], we could not find a single deployment that configured the SSL plugin to launch an external binary. Yet, under previous coarse-grained library debloating [59], and system call specialization [29] techniques, the `execve` system call would be retained in Httpd's attack surface as long as the SSL plugin is enabled, irrespectively of whether the external key generation option is used or not.

Similar situations arise with client applications too. The utility program Wget, used to download remote web pages and files, provides the runtime option `–use-askpass`, which allows the user to launch an external program to handle user credentials. Similar to Httpd's SSL plugin, if this option is specified, the `execve` system call is invoked to launch the external program. Although `execve` is *solely* required by this runtime option, existing approaches cannot remove it from the attack surface, because it is a valid part of the program's control flow graph.

In C2C, our goal is to map these runtime options to code. Performing this fine-grained mapping will in turn allow us to specialize the application at runtime by *filtering* system calls depending on the particular configuration used to launch the application.

2.2 Seccomp BPF

Our fine-grained application specialization technique filters unneeded system calls by applying Seccomp BPF programs to the target application. Seccomp BPF is a mechanism implemented in the Linux kernel which allows a user-space program to restrict its own access to the system calls provided by the kernel. Specifically, Seccomp BPF uses the Berkeley Packet Filter language [45] to allow 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 the `prctl` or `seccomp` system calls from within their own process. After doing so, all system call invocations from within the process and any of its forked child processes will be checked against the installed filters and will either be permitted or blocked, depending on the specified policy.

3 THREAT MODEL

Similarly to previous attack surface reduction techniques [27, 29], our threat model assumes remote adversaries armed with a vulnerability that allows arbitrary code execution. C2C does not rely on any other exploit mitigations to be deployed. 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 the system calls that are not required by the current active configuration. These may include security-critical system calls that can be used to spawn additional processes, execute shell commands, and so on. Preventing access to a system call also effectively neutralizes the corresponding kernel code that carries out its functionality. This can prevent the exploitation of vulnerabilities that can lead to privilege escalation [42]—an attacker cannot trigger those vulnerabilities to compromise the kernel, as the respective system calls cannot be invoked in the first place.

4 DESIGN

The goal of C2C is to accurately map configuration options to application code, and then to block system calls used exclusively by disabled configuration options at launch time. In the rest of this paper, we use the term *runtime options* to refer to configuration options that can be specified while launching a binary (e.g., through command line arguments), and *active configuration* to refer to the particular runtime settings used by a process.

By analyzing the source code of different applications, we observed that in most cases configuration-dependent code follows one of two main design patterns:

- Basic blocks following conditional branches or switch statements that depend on whether a runtime option is set in the active configuration.
- Functions reachable through function pointers that are initialized depending on whether a runtime option is set in the active configuration.

Figure 1 presents an overview of how our system handles each of these cases. In addition to the source code of the application, C2C requires the programmer to provide the list of struct types or scalar variables that are used to store active configuration settings. Applications typically store the active configuration in dedicated struct types, such as `ngx_core_conf_t` in case of the popular Nginx server. These configuration-related types (*not* their instances) must be provided as input to our C2C toolchain. We also require the programmer to annotate the point in the application's code where the parsing of runtime configuration options is complete, with the `start_processing` annotation. Typically, the code that parses and loads the configuration values is straightforward, and thus identifying both the types that store the active configuration and the completion of the configuration parsing phase is a simple task. Once provided with the source code and the names of these struct types, C2C performs basic block specialization through the following analysis phases:

- *Augmented Control Flow Graph (ACFG) Creation:* The ACFG contains detailed information about configuration-dependent branches (or switch statements), including the (field of the) configuration object that the branch depends on, and the targets of the *True* and *False* conditions.
- *Conditional Branch Pruning:* Instruments every store instruction that writes values to fields of the objects of the annotated configuration types, to record the written values during program execution. Then, depending on a given active configuration, branches of configuration-dependent conditional branches are enabled or disabled.
- *Runtime Address-Taken Pruning:* Filters the targets of indirect function calls depending on whether the address of the target function is taken during program initialization, and whether it can be taken post-initialization, based on the enabled conditional branches.
- *Filter Generation:* Using the results of the previous phases, identifies the code that can be removed, and generates corresponding Seccomp filters.

At each of these stages, our design ensures the soundness of the remaining code by performing conservative analysis—some code associated with disabled configuration options may be retained,

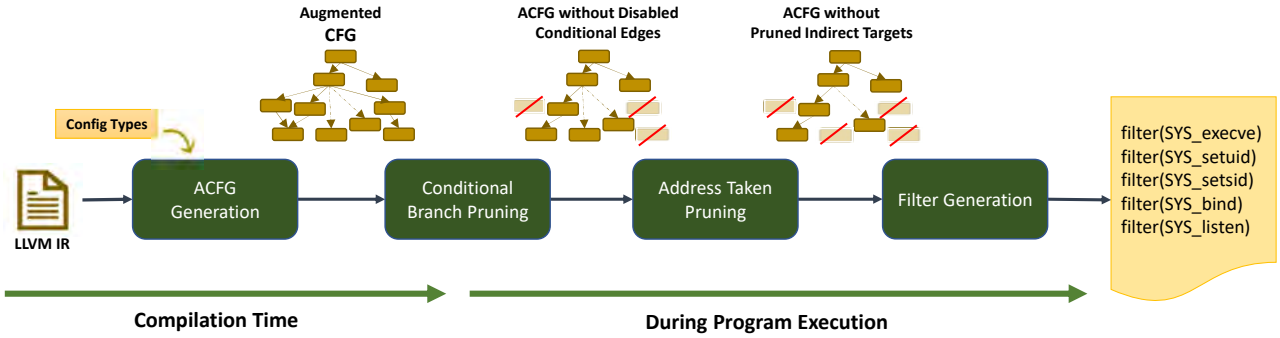

Figure 1: Overview of how C2C maps configuration options to code and applies restrictive system call filters to reduce the application's attack surface.

but no basic block is pruned from the graph unless its associated configuration option is also disabled. The following section will discuss the first step of performing basic block specialization through building the ACFG.

4.1 Augmented Control Flow Graph Construction

We first build a sound interprocedural control flow graph (CFG) that includes the conditional branches and switch statements. We call this the *Augmented Control Flow Graph (ACFG)*. Traditional control flow graphs contain only the edges of the control flow, but not the *condition* under which an edge is taken. The ACFG augments the graph by labeling the edges of a conditional branch as *True* and *False*, according to the result of the condition. Also, each case of a switch statement is augmented with a *Switch* label specifying only one case can be true at runtime.

In order to build the ACFG, we start with the CFG of every function and then augment the edges corresponding to conditional branches with both their condition, and whether the edge is taken if the condition is *True* or *False*. As shown in Figure 2, the two control flow edges connecting basic block bb0 to bb1 and bb2 are labeled with the result of the condition that causes the branch to be executed. To simplify the handling of complex branch conditions that involve logical operators (such as conjunction, disjunction, and negation), we expand complex conditions into nested if-statements so that each if-statement has a single condition involving a simple comparison operator. For example, in Figure 3, the complex condition involving the conjunction operator is expanded into two `if` statements, each having a single comparison.

Switch statements require special handling because each case only has a *True* edge that can be enabled or disabled. We do this by augmenting the branch of each case with a *Switch* label and check whether the value matches each case at runtime, only keeping the taken case branch.[1]

4.2 Conditional Branch Pruning

Application code that depends on runtime options is predicated by conditional branches and switch statements. These branches

[1]From this point forward, any reference to conditional branches covers switch statements as well.

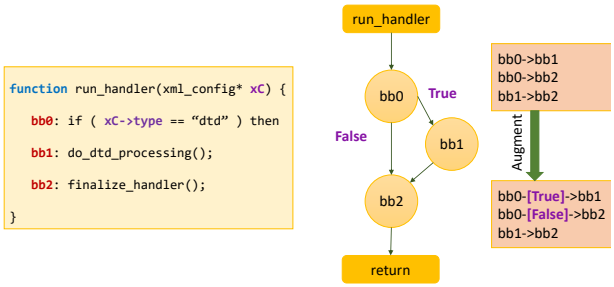

Figure 2: The augmented control flow graph for a sample Nginx code snippet. Every control flow edge is augmented with the value of the condition.

Figure 3: Converting a complex branch condition with conjunctions into nested simple branches.

check whether a runtime option is enabled or initialized to a particular value, and depending on the result of the check, execute the configuration-specific code. Using the annotated struct types, C2C first derives which conditional branches depend on runtime option values. Then, the values written to these configuration objects are recorded at runtime. Finally, based on the configuration values written the inaccessible system calls are filtered.

4.2.1 Configuration-dependent Conditional Branch Identification. We perform data flow analysis starting from the configuration objects (objects of programmer annotated struct type), to identify the conditional branches that depend on runtime options. Only the top-level configuration-related struct type needs to be provided by the developer. Our analysis automatically infers the fields of these objects as well as other related struct types. C2C performs forward analysis, iteratively joining *use-def* chains to identify all values derived from configuration-related objects. When the forward analysis reaches a conditional branch, C2C concludes that the branch depends on the configuration value and records the dependency of

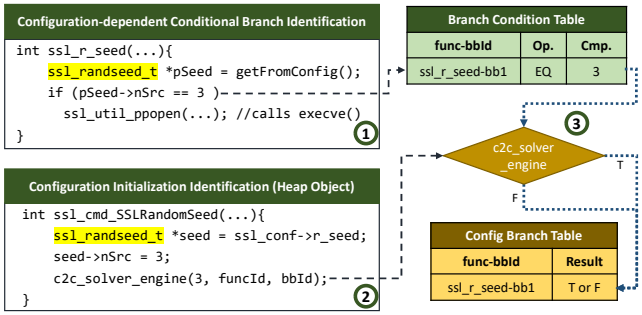

Figure 4: The different steps of conditional branch pruning. 1) Identify the configuration-dependent conditional branches. 2) Instrument the configuration initialization instructions. 3) Evaluate the results of configuration-dependent conditional branches (at runtime). (Dotted lines represent operations performed at runtime.)

the conditional branch and the fields of the configuration objects in a table that we call *ConfigBranchTbl*. At runtime, the result of the branch condition is updated to this table and this mapping is used to determine which conditional branches can be safely removed.

Consider the simplified code snippet from Apache Httpd presented in Figure 4. Assuming that the type `ssl_randseed_t` is derived from the programmer-specified top-level configuration type, our forward analysis can successfully identify that the system call `execve` is guarded by a configuration-dependent conditional branch, and that this branch depends on the configuration value stored in the `nSrc` field of an object of type `ssl_randseed_t`.

Note that while our analysis can identify most of the runtime options in the applications we tested, missing an option does not affect the soundness of our approach—it only results in creating more permissive filters.

4.2.2 Configuration Initialization Identification. Finally, C2C extracts the values stored in the configuration objects before the start of the processing phase. Using these values, at runtime, the instrumentation added by C2C computes whether the branch conditions that depend on these objects evaluate to *True* or *False*, and updates the ConfigBranchTbl accordingly. This step differs depending on whether the configuration object is a heap object or a global variable.

Global Configuration Objects. Some applications store their runtime options in global objects. For example, `wget` uses an object of the struct type `options`. C2C instruments the start of the *processing phase* to iteratively read the values of the global configuration objects. C2C uses the type information available at compilation time to insert instructions that traverse the various fields of each of the configuration objects, and invoke an evaluation routine that computes whether the values result in the *True* or the *False* branch to be taken. This information is recorded in the ConfigBranchTbl entry for each configuration-dependent conditional branch. During program execution, when this code is executed, the configuration values are read and evaluated, and the configuration-dependent conditional branches will be enabled or disabled depending on the active configuration of the process.

Heap Configuration Objects. Reading configuration objects on the heap presents certain challenges. Unlike global objects which are always valid during the program's lifetime, reading the values of heap objects requires a valid pointer to refer to them. This is not always straightforward because the pointer itself may be stored as a field of another object. To avoid these complexities, C2C simply identifies the instructions that parse runtime options and stores them into heap configuration objects, and instruments them to record the values being stored.

Similar to identifying configuration-related conditional branches, C2C performs forward analysis, traversing the *def-use* chains originating from the configuration objects to *store* instructions. C2C instruments these memory store instructions to call an evaluation routine that inspects the value written and evaluates the branch condition. Depending on the result of this evaluation, we update whether the *True* or *False* branch will be taken. (See Section 4.5 for exceptions.) Thus, by analyzing the state of ConfigBranchTbl, C2C can map the active configuration to code and determine which branch targets can be safely disabled for the current invocation of the application.

In Figure 4, our analysis identifies that `ssl_cmd_SSLRandomSeed` parses and stores the configuration value to the `nSrc` field of the `ssl_randseed_t` object. Using the previously derived information that the conditional branch in function `ssl_r_seed` depends on this object field, C2C instruments the store instruction to invoke the C2C Solver Engine, which at runtime, evaluates the result of that conditional branch and updates ConfigBranchTbl.

Once every branch in ConfigBranchTbl is analyzed, C2C identifies the inaccessible code. Starting from the disabled branch targets of the configuration-dependent branches, C2C follows the function call edges in the call graph to identify the functions that no longer have any caller and therefore can be safely removed. To do this, C2C starts with a precomputed call graph that we build offline, as discussed in Section 4.2.3, and iteratively removes functions that are no longer reachable. In line with previous works [29], we also perform additional filtering on the targets of indirect function calls based on functions which have their addresses taken. We discuss this process in detail in Section 4.3. Finally, based on the remaining functions, C2C enforces restrictions which we discuss further in Section 4.4.

4.2.3 Callgraph Construction. We construct the application's callgraph offline to reduce start time overhead. The presence of indirect function calls via function pointers requires the use of static pointer analysis. We use a field-sensitive version of the well-known Andersen's [8] pointer analysis algorithm to resolve the targets of function pointers. Because our analysis is field-sensitive, it treats each field of a complex object (e.g., `struct`), distinctly. This provides a higher degree of precision for our analysis.

4.3 Runtime Address Taken Pruning

As discussed in Section 4.2.3, we perform offline pointer analysis to resolve indirect function calls. Performing the callgraph construction offline inherently forces us to consider *all* code as *in-scope*, and therefore our callgraph contains some overapproximation—the targets of indirect call sites include functions that will never be invoked via a function pointer. For example, a runtime option might

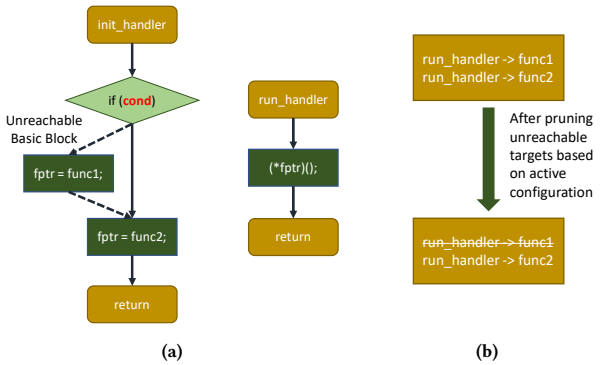

Figure 5: Pruning targets of indirect call sites based on function pointers initialized at runtime.

initialize a function pointer with multiple targets, none of which are actually feasible if the option is disabled at runtime.

To address this issue, C2C performs an additional level of pruning of the targets of indirect call sites at runtime. Our approach is inspired by the address-taken pruning approach presented in Temporal Specialization [29]. We observe that a function can be a valid target of an indirect function call, only if its address is taken (and stored to another variable) in code that is accessible at runtime. If a function's address is not taken before the processing phase starts *and* thereafter is *solely* taken inside an unreachable basic block, it cannot be a valid target for an indirect function call. Therefore, C2C removes these functions from the target sets of all indirect call sites.

Figure 5 shows an example. Assume that for a given active configuration the conditional branch in function `init_handler` depends on the variable `cond` that always evaluates to *False*, and thus the function pointer `fptr` can never point to `func1` (Figure 5a). Before the start of the processing phase, our analysis determines this, and thus prunes `func1` from the list of possible callees of `run_handler`, where `fptr` is dereferenced in an indirect call (Figure 5b).

4.4 Filter Generation

Using the mapping between runtime options to code, and given the active configuration of a process, we use system call filtering to reduce its attack surface. Similarly to previous works [13, 27, 29], we mainly focus on reducing the attack surface in terms of minimizing the set of system calls available to the process based on its active configuration. We consider two phases of execution for all applications: the *initialization* phase and the *processing* phase. We use the open-source framework by Ghavamnia et al. [29] to extract the list of required system calls for each phase.

As discussed in Section 8, in our current implementation C2C solely filters system calls and does not remove any code, due to the minimal benefits of the latter at the basic block level. Still, implementing support for removing any code deemed inaccessible is straightforward (e.g., by zeroing out).

After performing the steps outlined in Section 4.2, we know which configuration-dependent conditional branches will always evaluate to *False* during the current execution. With this information, we proceed to find the functions that become inaccessible and derive the system calls accessed *exclusively* by these functions. The result of this step is an interprocedural *pruned* control flow graph.

We use our generated pruned callgraph to extract the set of system calls required by the program. Based on the open-source frameworks provided by previous works [29, 59], we map the retained functions to their system calls. Finally, at runtime, we install a Seccomp BPF filter [1] at the start of the processing phase to filter the system calls identified as unnecessary by our analysis.

4.5 Ensuring Analysis Soundness

It is not always possible to correctly identify the value stored at the configuration object, or the exact configuration-dependent value that is the predicate of a conditional branch. The presence of pointers in C/C++ code also presents an additional level of challenge in ensuring soundness, as the address of the configuration object may be passed as an argument to a function invoked via an indirect call using a function pointer. The results of pointer analysis techniques usually include imprecision, making them unsuitable for our use—if the analysis informs us that a branch condition *might* refer to a configuration value disabled in the active configuration, we still cannot remove it safely, as it might, in reality, refer to a different object altogether.

Another source of uncertainty about the result of a conditional branch is when C2C cannot statically determine either predicate of the comparison. This can either be due to the configuration object's value undergoing uncertain computation before being compared in the conditional branch, or because the configuration value is compared with a value that cannot be determined statically. Consider Listing 2 from Nginx's codebase. The value in the configuration object `ngx_http_gzip_conf_t` can be derived statically, but since it is being compared with `content_length_n` which changes dynamically, C2C cannot reason about the resolution of this branch.

```
int ngx_gzip_head_filter(ngx_req_t *r){
    ngx_http_gzip_conf_t  *c = ...;
    if (r->head_out.cont_len < c->min_len)
        process_filter();
}
```

Listing 2: A configuration-dependent conditional branch in which the left-hand operand of the comparison operation cannot be statically determined.

In situations where C2C cannot identify the exact value that is being stored at a configuration object, or the exact value that is the predicate of the conditional branch, it falls back to a *use-based* approach—our analysis simply decides if the configuration value is "used." In particular, our analysis defaults to use-based analysis under the following conditions.

(1) The configuration object or field is passed as an argument at an indirect function call site.
(2) A value that cannot be determined statically is written to the configuration object or field.
(3) Our analysis concludes that a value derived from the configuration object is used as a predicate in a conditional branch, but the value cannot be statically determined.
(4) The value derived from the configuration can be statically determined, but it is compared with a variable which cannot be statically determined.

Table 1: Decision Table for Value-based and Use-based conditional branches. "T or F" indicates that the *True* or *False* branch will be retained depending on the result of the comparison performed by our solving engine. "T & F" indicates that both the *True* and the *False* branch will be retained, to preserve soundness. For switch statements, "T & F" means that all cases will be enabled, and "T or F" means that only one case will be enabled, depending on the value of the configuration variable.

Approach	Constraint	Result
	Global Configuration Variables	
Value-based	-	T or F
Use-based	Non-deterministic comparison	T & F
	Heap Configuration Variables	
Value-based	Executed at runtime	T or F
Value-based	Not executed at runtime	None
Use-based	Argument to indirect call site	T & F
Use-based	Non-deterministic comparison	T & F
Use-based	Not executed at runtime	None

If C2C concludes that the configuration value is indeed used in the active configuration, it preserves both the *True* and the *False* branches for conditional branches and keeps *all* case branches for switch statements. Similarly, if C2C concludes that a configuration value is not used in the active configuration, it removes both branches in the case of conditional branches and removes *all* case edges for switch statements, except the *default* edge (if one exists). The complete decision table for global and heap configuration objects is provided in Table 1.

5 IMPLEMENTATION

We implemented C2C using the LLVM 12 toolchain. C2C requires interprocedural, whole program analysis to perform basic block specialization. To this end, we use LLVM's Link Time Optimization (LTO) [35] to obtain the linked Intermediate Representation (IR) of the entire application and its dependent libraries. We use this linked IR bitcode to construct the Augmented Control Flow Graph (ACFG). Conditional branches are represented by `BranchInst` instructions and switch statements are represented by `SwitchInst` in the IR bitcode. LLVM's Clang frontend automatically decomposes the complex branch conditionals discussed in Section 4.1, into simple nested branch instructions that depend on a single value. Iterating over each `BranchInst` in a function's Control Flow Graph (CFG) our analysis annotates each conditional branch edge as *True* or *False*, depending on the resolution value that determines which control flow edge is traversed. For switch statements, C2C annotates each case edge with a *Switch* label, enabling only one at runtime depending on the observed value. This information is stored in a global variable named `ConfigBranchTbl`.

We perform forward analysis starting from the configuration-related objects to identify both the configuration-dependent conditional branches (and switch statements), as well as (in case of configuration objects on the heap) the `StoreInst` instructions that write the active configuration values into them. LLVM automatically provides the *def-use* chains that link the definition of a variable to its use, and we recursively join these def-use chains to implement the forward analysis algorithms. Once our forward analysis identifies the `StoreInst` instructions that store the active configuration, we instrument them with calls to the *C2C Solver Engine* and pass the stored configuration value to it. The solver engine uses the ACFG stored in the ConfigBranchTbl to determine if the value stored disables any configuration-dependent branch edge. For global configuration objects, we instrument the start of the processing phase to retrieve the values stored in them and invoke the solver engine, which in turn updates the ConfigBranchTbl.

To resolve the targets of indirect call sites, we generate the interprocedural callgraph of the application at compile time using the SVF [64] framework. The runtime address taken pruning discussed in Section 4.3 is implemented by instrumenting each function's entry to record its invocation in a runtime table named `RuntimeExecuteTbl`. After the application completes parsing its configuration, this table is used to determine which functions could not possibly have their addresses taken and therefore can be safely pruned from the target-sets of all indirect call sites. As the final step, we use the framework provided by previous works [27, 29] to derive the system call profile for the application and apply it to the application, as a Seccomp profile.

6 EXPERIMENTAL EVALUATION

In this section, we evaluate the attack surface reduction achieved by C2C. We use C2C to derive system call profiles for ten popular applications based on commonly used active configurations, and compare the results with the profiles generated by existing library specialization [59] and temporal specialization [29] techniques.

6.1 Applications and Active Configurations

We have chosen seven server applications (Apache Httpd, Nginx, Lighttpd, Postgresql, OpenSMTPD, Redis, and Memcached) and three client applications (Wget, Curl, and Tar) for our evaluation. The manual effort required for identifying the required annotations was less than one hour per application.

Our approach applies system call specialization depending on a given active configuration. Because the results depend on the active configuration used, we attempt (whenever possible) to use configurations that are representative of real-life workloads, such as those used in official Docker images. For example, to evaluate the benefits of applying C2C to Apache Httpd, we used the active configurations from the Bitnami Wordpress [16], Drupal [17], and Mediawiki [18] Docker images that use Httpd as their web server. To build our baseline binaries, we use the compilation options used to build the application binaries in the Docker images (if available), or the ones used to build the binary installed by the Ubuntu package manager. For client applications, in which the active configuration typically depends on the user-supplied command line options, we use typical configurations considered in previous works [31].

To illustrate the benefits of C2C over previous approaches, we specialize the same set of ten applications with library specialization [59] and temporal specialization [29], using the toolchains provided by the temporal specialization open-source framework [28]. In the following, we provide more details for the applications we evaluated using our C2C toolchain.

Table 2: Conditional branch pruning statistics. The first two columns show the number of annotations required and the location of the configuration object. The next columns show the number of conditional edges, configuration-dependent edges, configuration-dependent with initialization identified, followed by the number of edges with imprecision (non-deterministic comparison). The last column shows the number of disabled edges at runtime based on the configuration used.

Application (Configuration Used)	Config. Annotations	Global (G) or Heap (H)	Conditional Edges	Config. Dependent	Heap Init. Identified	Edges with Non-Det. Cmp.	Disabled
Apache (Bitnami Wordpress)	1	G/H	54K	4.9K	3.5K	2.5K	**665 (%13.3)**
Nginx (Zend Server)	2	H	39K	4.0K	3.8K	0.6K	**1674 (%41.0)**
Lighttpd	1	H	11K	0.1K	0.1K	16	**70 (%49.2)**
Postgresql	7	G/H	206K	2.5K	0.4K	0.4K	**1230 (%48.5)**
OpenSMTPD	1	H	23K	0.1K	0.1K	98	**33 (%19.8)**
Redis (Redis)	1	G	34K	3.2K	-	0.6K	**641 (%19.8)**
Memcached	1	G	15K	0.5K	-	98	**228 (%41.0)**
Curl (no options)	4	H	24K	1.0K	0.9K	80	**881 (%90.6)**
Wget (no options)	1	G	15K	1.5K	-	64	**732 (%46.8)**
Tar	1	G	25K	85	-	2	**25 (%29.4)**

6.1.1 Apache Httpd. Apache is a popular web server which supports a long list of configuration options. It uses multiple levels of struct types and objects to store these options, rooted in the top-level struct type `server_rec`. We use the active configurations included in the Docker images for Bitnami Wordpress [16], Drupal [17], and Mediawiki [18] for our evaluation. Of the top-50 most popular Docker images, these images have the most variety in their active configurations. While the Drupal and Mediawiki images have similar active configurations, the Bitnami Wordpress image enables different options (e.g., SSL-related settings) not used by the others.

6.1.2 Nginx. Nginx is a popular web server that is highly configurable. The `ngx_core_conf_t` and `ngx_conf_s` top-level struct types are used to store runtime options. Nginx creates objects of these and other related struct types on the heap. From the top-50 popular images on Docker Hub, the Zend Server [20] image uses Nginx as its underlying web server. We used its active configuration for our evaluation as it is representative of real-world use cases.

6.1.3 Lighttpd. Lighttpd is a lightweight web server which also supports a broad set of runtime settings. The `server_config` top-level struct type is used to store high-level runtime options, with more detailed options stored in key-value maps. For example, whether the server should run as a daemon is specified by the `dont_daemonize` field of the `server_config` struct type. We note that the configuration parsing code was refactored in the latest version (v1.4.63), making it amenable to C2C's analysis.

6.1.4 Postgresql. Postgresql is one of the most popular database servers, supporting many runtime options. With 804 KLOC, it is the largest application codebase in our test suite. The active configuration for Postgresql is stored in two primary struct types, `HbaLine` and `Port`, and also multiple scalar global variables (e.g. `bool EnableSSL`). Pointers to these global variables are stored in struct objects of type `config_bool`, `config_int`, `config_real`, `config_string`, and `config_enum`. We thus require the programmer to annotate these five struct types, after which our analysis can derive the branches associated with the rest of the configuration-related variables. We use the default runtime settings provided with the source code for our evaluation.

6.1.5 OpenSMTPD. OpenSMTPD is a SMTP-based email server. It uses a single struct type (`smtpd`) to store its active configuration. The active configuration includes several options through which the user can specify various limits, such as message-queue size and number of recipients per transaction. We use the sample runtime settings available with the source code for our evaluation.

6.1.6 Redis and Memcached. Redis and Memcached are in-memory data stores. They both store all their runtime options in a global struct object of type `redisServer` and `settings`, respectively. We use the compilation options used to build the Redis and Memcached binaries for their official Docker images [19, 21], and the active configurations used in them.

6.1.7 Wget, Curl, and Tar. Wget, Curl, and Tar are widely-used command-line utilities. Wget and Curl are used for retrieving the static content of websites, and Tar is used for creating and extracting file archives. Wget uses the single-level global struct object `options` to store its configuration, whereas Curl uses the structs `GlobalConfig`, `OperationConfig`, `ssl_primary_config`, and `ssl_general_config`. Tar, on the other hand, uses a global enum variable named `subcommand_option` to store the main runtime options provided by the user. For Wget, we used three different active configurations: 1) no options; 2) with an output file; and 3) with a third-party authentication application enabled. In case of Curl, along with the simplest case of requesting a URL without any options, we considered two other active configurations: 1) using the output file option, and 2) specifying the request type. For Tar, we considered three active configurations: 1) creating an archive, 2) extracting files from a pre-generated archive, and 3) testing the correctness of an archive. Although not server applications, these utilities have a history of vulnerabilities [15, 50, 52] that can be exploited by a malicious website or archive to attack the client. As we will discuss in Section 6.4.1, this is particularly important

because all these applications can potentially invoke the execve system call, which could allow an attacker to run unauthorized programs on the client machine.

6.2 Conditional Branch Pruning

The ACFG created by C2C contains configuration-dependent conditional branch edges. Table 2 provides detailed statistics about the number of configuration-dependent branches, how C2C identifies them, and what type of matching it performs for each edge. The first column shows the number of annotations required to specify the configuration-related data types. The next column specifies whether the application stores its configuration objects on the heap or as a global variable.

The subsequent columns show the number of conditional branch edges, the subset of configuration-dependent edges, and for how many edges C2C could identify an initialization instruction (if stored on the heap). As mentioned in Section 4.5, C2C falls back to use-based matching if the application performs a non-deterministic comparison (column "Edges with Non-Det. Cmp."), or if it passes the configuration object as an argument to an indirect function call (not observed across any of the applications). The final column shows the number of edges that are disabled depending on a sample active configuration.

For example, Apache has 54K conditional edges, of which 4.9K are configuration-dependent. It uses both the heap and global variables for storing its configuration objects. Out of the 4.9K configuration-dependent edges, C2C can identity the initialization instruction of 3.5K edges. For 2.5K edges, C2C is forced to resort to use-based matching due to a non-deterministic comparison at the configuration read site. Our runtime analysis concludes that in the case of Bitnami Wordpress [16], more than 13% of its configuration-dependent edges can be disabled. Across all applications, C2C disables 19% (for OpenSMTPD) to 90% (for Curl) of all configuration-dependent edges.

To further evaluate the security benefit of C2C, we measure the number of system calls that remain accessible, and compare it to previous approaches [29, 59]. We use the framework provided by temporal specialization [28] to perform this comparison. Table 3 shows the number of system calls remaining in the attack surface for C2C, library specialization [59], and temporal specialization [29]. The results in the C2C column correspond to the active configurations specified in the "Active Configuration" column. Although the additional reduction in the number of system calls offered by C2C is small (e.g., for Bitnami, just six additional system calls are removed in the processing phase), as we will discuss in Section 6.4.1, the security benefit of this reduction is significant, as these system calls include security-critical ones (e.g., execve).

6.3 Overhead

Compilation Time. C2C performs a one-time analysis to build the application's ACFG, which increases the compilation time. Pointer analysis is the most time-consuming part of this analysis and is needed to resolve the targets of indirect function calls. The complexity of each application affects the time required to perform this analysis. Specifically, building the ACFG takes less than one hour for most applications. The only exception is Postgresql, for which, due to its larger codebase, the analysis takes approximately 22 hours. Although the overhead seems significant, we do not believe it is a major barrier for applying C2C in practice. First, this is a one-time analysis which does not depend on the active configuration used by the end-user at runtime. Second, other pointer analysis algorithms can be used which have lower algorithmic complexity but have lower precision (e.g., Steensgaard's algorithm [63]).

Code Size Increase. C2C instruments each application to keep track of the configuration-related object values and perform callgraph pruning at runtime. Since this instrumentation adds new instructions to the program, it increases the code size. We used the LLVM IR instruction count to measure the code size increase. Based on our assessment, C2C increases the IR instruction count by less than 3% for most applications. Nginx is the only application for which the LLVM IR instruction count increases by 10%. This is mainly because each configuration initialization instruction in Nginx affects multiple conditional branches, and our current implementation uses basic instrumentation that does not take advantage of any available optimizations. Applying these optimizations can reduce the code size overhead.

Runtime Overhead. C2C performs conditional branch pruning once the program is launched, and can therefore incur some runtime overhead. However, since the pruning is performed *only* during the initialization phase, it *solely* affects the application's startup time. Hence, it does not affect the runtime performance of the processing phase of server applications (e.g., handling client requests). Nonetheless, for client applications (that are usually launched to handle a single request), the entire program execution is affected. The startup time for the applications in our dataset ranges from one second (for smaller applications, e.g., Wget) to 100 seconds (for larger applications, e.g., Postgresql) after applying C2C. While this overhead is still significant for client applications, it can be reduced by improving the efficiency of our pruning algorithm, or by caching the pruned callgraph for frequently used active configurations. We leave the implementation of these techniques as part of our future work.

6.4 Security Evaluation

C2C offers two main security benefits. First, by filtering security-critical system calls, C2C breaks payloads used to perform malicious operations. Second, filtering system calls reduces the exposed attack surface of the underlying kernel, rendering certain kernel vulnerabilities unreachable to the adversary. We evaluate both of these aspects in the following sections.

6.4.1 Security-critical System Call Filtering. Attackers use system calls to perform their intended operation after successfully exploiting vulnerabilities in user-space applications. Security-critical system calls such as execve are widely used in exploit shellcode and ROP payloads. Previous studies [29] have shown that 46% (800 out of 1726) of the exploit payloads available in Metasploit [46] and Shell-storm [61] perform remote code execution and other system operations that can be broken by disabling security-critical system calls. In this section, we study the *additional* security-critical system calls that C2C removes, compared to previous approaches.

Table 3: Number of system calls retained (out of 333 available) after applying library debloating [59], temporal specialization [29], and C2C. "Init." indicates the application's initialization phase and "Proc." the processing phase.

Application	Active Config.	Lib. Spec.	Temp. Spec.		C2C		Example Syscalls Filtered Exclusively by C2C
			Init.	Proc.	Init.	Proc.	
Apache	Bitnami Wordpress	111	110	103	108	97	`execve`,* `setsid`, `mremap`
Apache	Drupal/Mediawiki	110	109	102	107	95	`execve`,* `dup`, `dup3`
Nginx	Zend Server	116	116	111	113	108	`setuid`, `setpriority`, `setrlimit`
Lighttpd	Ubuntu Default	100	100	78	97	74	`recvmsg`, `readv`, `getsockname`, `uname`
Postgresql	Source Code Default	120	120	109	118	105	`bind`, `setsockopt`, `getsockopt`
OpenSMTPD	Source Code Default	110	109	109	109	109	-
Redis	Redis	92	92	84	88	82	`recvfrom`,`sendto`
Memcached	Ubuntu Default	102	102	84	97	83	`nanosleep`
Curl	`-X GET, -i -o`	100	100	100	98	98	`utimes`, `fsetxattr`
Wget	`[no opt], -O`	92	92	92	78	78	`execve`, `setresuid`, `setresgid`, `pipe`
Wget	`–use-askpass`	92	92	92	92	92	-
Tar	`-czvf`	97	97	97	84	84	`mkdir`, `setxattr`, `fchmod`, `fchown`
Tar	`-xzvf`	97	97	97	94	94	`flistxattr`, `fgetxattr`, `llistxattr`
Tar	`–test-label`	97	97	97	73	73	`socket`, `sendmsg`, `connect`, `mkdir`

*Previous works [29] evaluated temporal specialization with an active configuration that did not require `execve` after initialization.

Apache Httpd. As shown in Table 3, C2C restricts access to security-critical system calls such as `execve`. The active configurations we considered for Apache load the SSL module, which requires the `execve` system call *only* if the `SSLRandomSeed` subconfiguration is initialized with the value `exec`. However, none of the considered configurations set this value to `exec`. Therefore, by applying C2C, the `execve` system call can be safely filtered. Note that non-configuration-aware approaches [29, 38, 59] cannot remove `execve` in these scenarios, as it remains reachable in the CFG. Given that `execveat` is already filtered by applying library specialization, an attacker now cannot launch executables through a single system call invocation.

Nginx. C2C restricts access to the `setuid` system call for Nginx, which is used across different user-level exploits [29] to change the effective user ID of the process. Since we can also filter `setreuid` by applying library specialization [4, 59], a local attacker cannot easily modify the effective user ID and elevate privileges without performing a privilege escalation attack against the kernel.

Lighttpd and Postgresql. As shown in Table 3, C2C can filter network-related security-critical system calls for Lighttpd and Postgresql. For example, it removes `bind` for Postgresql and `recvfrom` for Lighttpd. However, C2C cannot filter the `execve` system call for these two applications. In the case of Lighttpd, this is due to the imprecision of our analysis. Lighttpd supports executing a third-party program as its logger, configurable at runtime. Nevertheless, it uses pointer arithmetic (which C2C does not support) to perform string matching in the respective conditional branch. We leave implementing support for these more complex situations as part of our future work. In Postgresql, the log archiver invokes the `execve` system call to move the archive logs to backup folders. While `execve` cannot be filtered entirely in this scenario, the arguments passed to it (specifically, the pathname) could potentially be restricted by applying API specialization techniques [47], preventing the attacker from executing arbitrary programs.

Wget. As mentioned in Section 2.1, Wget provides a command-line option for specifying a third-party application to handle credentials when accessing web pages that require authentication. C2C maps this `–use-askpass` option to its respective field in the `options` global object, and successfully restricts the `execve` system call when the active configuration does not use this option. Although Wget is a client application, it could still be a target for a remote code execution attack against the user, e.g., as was the case with CVE-2019-5953 [14].

Tar. C2C can filter three and 13 system calls for Tar when decompressing and creating an archive, respectively. These include system calls related to file access and permissions. However, the `execve` system call cannot be filtered due to the overapproximation of our approach. The `–file` option can be provided with a remote host or a local file. If a remote host is specified, Tar uses `execve` to spawn `rsh` to connect to the remote location. It uses an array-index-based comparison to compare the configuration value stored in the `–file` option. Our current prototype does not support comparisons based on array indices, therefore we are unable to reason about the value stored in this configuration option. We plan to handle these situations in future iterations of our prototype.

Curl. In case of Curl, the imprecision of our analysis prevents us from removing the security-critical `execve` system call, which is used only when NTLM authentication is requested. In particular, because the respective configuration value is copied to multiple variables before being used in a conditional branch, we would need complex data flow analysis to map the conditional branch instruction to the runtime configuration. We leave the implementation of such analysis as part of our future work.

OpenSMTPD, Redis, and Memcached. OpenSMTPD has many runtime settings and C2C can capture the values stored in them, but they do not affect the underlying system calls used. Redis and Memcached do not provide many runtime options, and our manual verification shows that the invocation of security-critical system calls does not depend on the runtime configuration. Therefore, there is limited benefit in applying C2C to these applications regarding critical system call filtration. However, as we discuss next, even filtering system calls that are not security-critical reduces the underlying kernel attack surface.

6.4.2 Kernel Security Evaluation. While system calls can be abused by attackers to perform unwanted operations in user space, they also provide an entry into the kernel. Kernel vulnerabilities accessible through system calls allow attackers to mount privilege escalation attacks. These attacks can potentially disable security mechanisms imposed by the kernel and result in root access. Because these kernel vulnerabilities are related to arbitrary system calls, and not only the security-critical ones, filtering *any* system call can potentially reduce the kernel's attack surface.

We evaluated the security benefit of applying C2C in terms of kernel attack surface reduction by analyzing the number of Linux kernel CVEs mitigated through the system call filters generated by C2C. Our approach for mapping CVEs to kernel code differs from previous work [29] because of inherent overapproximations in their callgraph construction techniques, resulting in an overly conservative analysis. Instead, we rely on a public manually-curated list of Linux kernel CVEs [44] and apply semi-automated text analysis to derive a more accurate mapping. The description for each CVE on this list contains information about the system calls that can trigger it. We automatically parse these descriptions and use them to map system calls to CVEs.

Our results show that C2C mitigates a total of 88 CVEs, each in one or more of the ten applications. More importantly, 32 of the vulnerabilities could not be mitigated by previous approaches in at least one or more applications. Table 4 shows a summary of these 32 CVEs where C2C offers a benefit over library and temporal specialization. The last three columns of Table 4 show the number of applications for which the CVE is effectively mitigated by library specialization, temporal specialization, and C2C, respectively.

7 LESSONS LEARNED

During the course of our work, we studied extensively how different applications parse and store runtime options. We discovered that the effectiveness of C2C depends largely on the ease with which static program analysis techniques can be applied to the application, and that certain software design decisions complicate the mapping of configuration options to application code. These design decisions impact not only C2C, but also other systems that perform configuration-dependent program analysis, such as test-case generation [43, 65]. While most applications are designed in a manner that is amenable to static analysis, some are not. In light of the benefits of automatically mapping runtime options to application code, we believe adopting design practices that make the application easier to analyze statically is a worthwhile endeavor. To that end, in this section we describe our insights and present guidelines on how to design and refactor applications in a way that simplifies configuration-dependent program analysis.

Separate Configuration Parsing From Request Processing. An essential requirement for configuration-dependent debloating is the separation of configuration parsing from the actual processing of user inputs. If the application parses runtime options *on demand*, interspersed with the processing of user inputs, then we cannot determine which configuration-dependent features should be removed *before* the untrusted inputs enter the system. This prevents the specialization of the application binary based on the active configuration. The FTP server Vsftpd is one such application that parses the active configuration on-the-fly upon receiving a client request. This prevented us from applying C2C to reduce its attack surface. Despite this complexity, after manually analyzing its code, we found that it could be easily refactored to parse the runtime options before serving any FTP requests.

Avoid Compulsory Initialization of Linked Modules. To ensure maximum dynamic configurability, it is preferable to perform all software specialization at runtime at the basic block level. In this way, developers could ship a binary containing all features, which can be self-specialized at runtime depending on the active configuration. However, certain design patterns adopted by popular web applications prevent runtime-only specialization. These include storing both enabled and disabled plugins in the same array, and unconditionally invoking each plugin's *initialization* routine irrespective of whether the active configuration requires it.

Mindful of these findings, we recommend transitioning to a modular architecture, where *any* plugin-specific code is invoked only if the active configuration *actually* requires the plugin. Finally, enabled and disabled plugins should not be stored in the same array, to facilitate simple static analysis that does not need to distinguish between individual array elements.

8 DISCUSSION AND LIMITATIONS

Manual Effort. Our approach requires a limited amount of manual effort by the programmer, namely providing the transition point and the information about the struct types used to store the active configuration. The transition point requirement is not unique to C2C, and previous multi-phase system call filtering approaches [29] need it too, while there have been efforts to automate its identification [6]. Identifying the struct types used to store the active configuration does not require significant effort either, because C2C only needs the top-level types to be specified, and can automatically derive any dependent configuration-related sub-types. Moreover, missing a configuration-related struct type does not result in false positives, i.e., required system calls are not mistakenly filtered—it only results in the generation of more permissive policies.

Code Removal. C2C creates system call filters based on the generated fine-grained mapping between runtime settings and code. Although we identify the unneeded basic blocks, they are not removed because the amount of code that can be disabled does not justify the effort needed at runtime. On the other hand, some applications support optional modules that can be enabled or disabled during compilation. By mapping each runtime configuration option to its respective module, we can disable unnecessary modules *entirely*

Table 4: Number of applications (last three columns) in which kernel CVEs are mitigated by filtering unneeded system calls by applying library debloating, temporal specialization, and C2C.

System Call(s)	# CVEs	CVE Examples	Lib.	Temp.	C2C
`execveat, execve`	8	CVE-2015-3339, CVE-2016-1576, CVE-2015-8543, CVE-2016-2853	1	1	3
`getxattr`	1	CVE-2013-4591	9	9	10
`uname`	1	CVE-2012-0957	0	0	1
`bind`	6	CVE-2015-8956, CVE-2013-7421, CVE-2014-2678, CVE-2014-9644	1	2	3
`recvmsg`	1	CVE-2015-8019	0	0	1
`setsockopt`	12	CVE-2017-6346, CVE-2013-0310, CVE-2017-16939, CVE-2013-4470	1	1	2
`recvfrom, sendto`	1	CVE-2015-2686	1	1	2
`getsockopt`	1	CVE-2013-1828	1	1	2
`sendto`	1	CVE-2016-8645	1	1	2

based on runtime settings at compile time. This coarse-grained code reduction can remove large chunks of code at the module and library level without affecting runtime performance [38]. While mapping runtime configuration settings to compile-time options that affect the inclusion of certain modules is straightforward for some applications (e.g., Apache and Nginx, for which we actually implemented this capability), it is challenging for others. We leave the design and implementation of a generic solution for this purpose as part of our future work.

Sources of Imprecision. The callgraph generation of C2C is susceptible to overapproximation due to the need for resolving the targets of indirect function calls. This is not a limitation of our approach, as any technique that relies on sound static analysis would face the same issue. Furthermore, C2C has limited support for conditional branches which perform array-index-based comparisons and falls back to use-based matching under these circumstances. We leave improving accuracy for these cases as part of our future work.

Remaining Attack Surface. Like other attack surface reduction techniques, we do not claim to prevent the attacker from crafting their exploits by reusing the code and system calls that remain in the attack surface after specialization. Depending on the goal, these may still be enough for successful exploitation. However, by filtering many security-critical system calls such as `execve`, C2C significantly limits the attacker's flexibility. Similarly, filtering system calls reduces the number of entry points into the underlying Linux kernel available to the attacker.

Configuration File Integrity. When the active configuration is read from a file, as is the case with web servers such as Nginx and Apache Httpd, C2C relies on the integrity of this configuration file. If the configuration file is compromised, the attacker could mount a two-step attack, first enabling runtime options that enable access to the desired security-critical system call, and then forcing the application to relaunch with the altered active configuration. The simple solution to prevent this is to revoke the *write* permission from the configuration file before launching the application. Furthermore, we assume the active configuration is not modified by the user after the launch of the program. Although some programs (e.g., Apache) provide features for the user to modify the active configuration of a running process, we leave this out of scope for our current work.

Dynamically Loaded Libraries. Similarly to previous works [13, 29] that generate system call policies based on static analysis, we assume all library dependencies can be identified statically at compile time. If the application dynamically loads a library (e.g., by invoking `dlopen` and `dlsym`), the programmer must manually provide these libraries to the C2C toolchain.

9 RELATED WORK

Previous works on host-based intrusion detection systems [23–25, 33, 39, 54, 60, 66], have used system call policies to detect anomalous behavior. However, because the focus of our work is attack surface reduction, we primarily discuss the related work in the context of debloating and specialization.

9.1 Application Debloating

Various approaches have employed static analysis to identify and disable inaccessible application code. However, these techniques do not consider the active configuration used to launch the application. Mulliner and Neugschwandtner [51] proposed one of the first techniques for application debloating, by removing the non-imported functions from dependent shared libraries. Quach et al. [59] and Agadakos et al. [4] perform library debloating at source-code level and binary-level respectively, by computing callgraphs for both the application and its libraries. Porter et al. [55] also perform library debloating, but use an *oracle* to predict the library functions required during an application's lifetime.

Davidsson et al. [12] analyze the full-stack requirements of web applications and specialize dependent libraries based on the requirements of the web application. LightBlue [68] provides a framework to perform automatic profile-aware debloating of Bluetooth stacks, spanning from Bluetooth host code to the firmware. Song et al. [62] perform fine-grained library customization of statically linked libraries using data-dependency analysis.

Temporal specialization [29] technique splits the server application lifetime into two phases (*Initialization* and *Serving*), and tailors the system call filters to the profiles of each phase. Sysfilter [13] and Chestnut [11] present binary analysis frameworks that restrict

the system calls available to user-space processes. Abhaya [53] also creates allow lists for system calls and their arguments by applying static analysis to the target program. Shredder [47] and Saffire [48] restrict the arguments passed through library API calls, for Windows and Linux applications respectively. SGXPecial [49] specializes SGX enclave interfaces.

Other works use dynamic analysis and training to identify unused application code. Qian et al. [56] remove unused code from binaries using heuristics and *training* the application with likely user inputs. Ghaffarinia and Hamlen [26] use a similar training-based approach to prevent control flow transfers to unauthorized code. CHISEL [31] uses reinforcement learning to debloat software based on user-provided test cases. DamGate [70] prevents the execution of unused features by rewriting binaries with *gates* that restrict access to disabled code. Slimium [57] debloats the Chromium web browser based on pre-generated profiles for multiple popular websites. Ancile [9] leverages fuzzing to identify the code necessary for performing the operations required by the user. However, since these techniques rely on dynamic analysis, they cannot provide any soundness guarantees.

The previous work closest to our approach is the technique presented by Koo et al. [38]. They propose a hybrid technique, combining static and dynamic analysis, that uses differential testing to identify the library dependencies of each runtime option supported by the application. While effective, this approach requires extensive manual effort to generate specially crafted active configurations that exercise each runtime option. Moreover, they support removing code only at the coarser library level. More recently, TRIMMER [5] also presented a configuration-based software debloating technique that relies on dynamic taint tracking to identify and remove code paths which are related to unused configuration options. Similarly, LMCAS [6] performs partial-evaluation to propagate constants captured in the initialization phase and identify any unneeded code.

While the above techniques focus on C/C++ applications, application debloating techniques have been proposed for other languages too. Jred [69] statically analyzes Java applications to identify and remove unused methods and classes. Jiang et al. [34] present a feature-based debloating technique for Java using data flow analysis. Azad et al. [7] remove excessive features from PHP applications by profiling the web application using dynamic analysis. MiniNode [37] presents a framework for reducing the attack surface of NodeJS applications by removing unnecessary packages.

9.2 Kernel and Container Debloating

There has been significant interest in debloating and specializing the Linux kernel with respect to specific application or workload profiles. KASR [71] and FACE-CHANGE [72] use dynamic analysis to create application-specific kernel profiles. Kurmus et al. [40] present a configuration-based debloating technique for the Linux kernel and automatically generate compilation configuration files that can be used to build custom Linux kernels tailored for specific workloads. Similarly, Acher et al. [3] create customized Linux kernels using a statistical supervised learning method. SHARD [2] uses context-aware hardening to specialize the Linux kernel.

Wan et al. [67] use training to profile the required system calls of a container and generate the corresponding Seccomp filters. Due to the incompleteness of dynamic analysis, Confine [27] uses static analysis to derive these Seccomp system-call profiles. MiniCon [36] uses dynamic analysis to identify the minimal set of capabilities required by a container. DockerSlim [22], an open-source tool, removes unnecessary files from Docker images by applying dynamic analysis. SPEAKER [41] separates the required system calls of containers into two main phases, *booting* and *runtime* and uses dynamic training to generate system call profiles for each phase.

10 CONCLUSION

We presented Configuration-to-Code (C2C), a novel approach for specializing applications and their system calls based on the active configuration. Compared to previous configuration-based debloating approaches, C2C requires significantly less manual effort and performs finer-grained specialization, leading to the identification of unnecessary code at the basic block level and the generation of more restrictive filters. We demonstrated the effectiveness of C2C by evaluating it with ten popular applications. By performing basic block specialization, we show how C2C can filter more system calls compared to previous approaches, and also how it can mitigate 32 additional Linux kernel CVEs.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive comments and feedback. This work was supported by the Office of Naval Research (ONR) through award N00014-17-1-2891, the Defense Advanced Research Projects Agency (DARPA) through award D18AP00045, and the National Science Foundation (NSF) through awards CNS-1749895, CNS-2104148, as well as award CNS-2127309 to the Computing Research Association for the CIFellows Project.

REFERENCES

[1] 2022. Seccomp BPF (SECure COMPuting with filters). https://www.kernel.org/doc/html/v4.16/userspace-api/seccomp_filter.html.
[2] Muhammad Abubakar, Pedro Ahmad, Adil Fonseca, and Dongyan Xu. 2021. SHARD: Fine-Grained Kernel Specialization with Context-Aware Hardening. In *Proceedings of the 30th USENIX Security Symposium*. 2435–2452.
[3] Mathieu Acher, Hugo Martin, Juliana Pereira, Arnaud Blouin, Jean-Marc Jézéquel, Djamel Khelladi, Luc Lesoil, and Olivier Barais. 2019. *Learning Very Large Configuration Spaces: What Matters for Linux Kernel Sizes*. Technical Report HAL-02314830. Inria Rennes - Bretagne Atlantique.
[4] Ioannis Agadakos, Di Jin, David Williams-King, Vasileios P Kemerlis, and Georgios Portokalidis. 2019. Nibbler: Debloating binary shared libraries. In *Proceedings of the 35th Annual Computer Security Applications Conference (ACSAC)*. 70–83.
[5] Aatira Anum Ahmad, Abdul Rafae Noor, Hashim Sharif, Usama Hameed, Shoaib Asif, Mubashir Anwar, Ashish Gehani, Junaid Haroon Siddiqui, and Fareed M Zaffar. 2021. TRIMMER: An Automated System for Configuration-based Software Debloating. *IEEE Transactions on Software Engineering* (2021).
[6] Mohannad Alhanahnah, Rithik Jain, Vaibhav Rastogi, Somesh Jha, and Thomas Reps. 2022. Lightweight, Multi-Stage, Compiler-Assisted Application Specialization. In *Proceedings of the 7th European Symposium on Security and Privacy (EuroS&P)*. 251–269.
[7] Babak Amin Azad, Pierre Laperdrix, and Nick Nikiforakis. 2019. Less is More: Quantifying the Security Benefits of Debloating Web Applications. In *Proceedings of the 28th USENIX Security Symposium*. 1697–1714.
[8] Lars Ole Andersen. 1994. *Program analysis and specialization for the C programming language*. Ph. D. Dissertation. University of Cophenhagen.
[9] Priyam Biswas, Nathan Burow, and Mathias Payer. 2021. Code Specialization through Dynamic Feature Observation. In *Proceedings of the 11th ACM Conference on Data and Application Security and Privacy (CODASPY)*. 257–268.
[10] Michael D Brown and Santosh Pande. 2019. Is less really more? towards better metrics for measuring security improvements realized through software debloating. In *Proceedings of the 12th USENIX Workshop on Cyber Security Experimentation and Test (CSET)*.

[11] Claudio Canella, Mario Werner, Daniel Gruss, and Michael Schwarz. 2021. Automating Seccomp Filter Generation for Linux Applications. In *Proceedings of the Cloud Computing Security Workshop (CCSW)*. 139–151.

[12] Nicolai Davidsson, Andre Pawlowski, and Thorsten Holz. 2019. Towards Automated Application-Specific Software Stacks. In *Proceedings of the 24th European Symposium on Research in Computer Security (ESORICS)*. 88–109.

[13] Nicholas DeMarinis, Kent Williams-King, Di Jin, Rodrigo Fonseca, and Vasileios P Kemerlis. 2020. Sysfilter: Automated System Call Filtering for Commodity Software. In *Proceedings of the 23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*. 459–474.

[14] CVE Details. 2019. CVE Details (CVE-2019-5953). https://www.cvedetails.com/cve/CVE-2019-5953/.

[15] CVE Details. 2021. CVE Details - Wget. https://www.cvedetails.com/product/332/GNU-Wget.html?vendor_id=72.

[16] DockerHub. 2022. Bitnami Wordpress - Docker Hub. https://hub.docker.com/r/bitnami/wordpress/.

[17] DockerHub. 2022. Drupal - Docker Hub. https://hub.docker.com/_/drupal.

[18] DockerHub. 2022. Mediawiki - Docker Hub. https://hub.docker.com/_/mediawiki.

[19] DockerHub. 2022. Memcached - Docker Hub. https://hub.docker.com/_/memcached.

[20] DockerHub. 2022. php-zendserver - Docker Hub. https://hub.docker.com/_/php-zendserver.

[21] DockerHub. 2022. Redis - Docker Hub. https://hub.docker.com/_/redis.

[22] DockerSlim. 2022. DockerSlim. https://dockersl.im.

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

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

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

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

[27] Seyedhamed Ghavamnia, Tapti Palit, Azzedine Benameur, and Michalis Polychronakis. 2020. 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)*.

[28] Seyedhamed Ghavamnia, Tapti Palit, Shachee Mishra, and Michalis Polychronakis. 2020. Temporal Specialization. https://github.com/shamedgh/temporal-specialization.

[29] Seyedhamed Ghavamnia, Tapti Palit, Shachee Mishra, and Michalis Polychronakis. 2020. Temporal System Call Specialization for Attack Surface Reduction. In *Proceedings of the 29th USENIX Security Symposium*.

[30] Md Mehedi Hasan, Seyedhamed Ghavamnia, and Michalis Polychronakis. 2022. Decap: Deprivileging Programs by Reducing Their Capabilities. In *Proceedings of the 25th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*.

[31] Kihong Heo, Woosuk Lee, Pardis Pashakhanloo, and Mayur Naik. 2018. Effective Program Debloating via Reinforcement Learning. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*.

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

[33] Kapil Jain and R Sekar. 2000. 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)*.

[34] Yufei Jiang, Can Zhang, Dinghao Wu, and Peng Liu. 2016. Feature-Based Software Customization: Preliminary Analysis, Formalization, and Methods. In *Proceedings of the 17th IEEE International Symposium on High Assurance Systems Engineering (HASE)*.

[35] Teresa Johnson. 2016. ThinLTO: Scalable and Incremental LTO. http://blog.llvm.org/2016/06/thinlto-scalable-and-incremental-lto.html.

[36] Haney Kang, Jinwoo Kim, and Seungwon Shin. 2021. MiniCon: Automatic Enforcement of a Minimal Capability Set for Security-Enhanced Containers. In *Proceedings of the 2021 IEEE International IOT, Electronics and Mechatronics Conference (IEMTRONICS)*.

[37] Igibek Koishybayev and Alexandros Kapravelos. 2020. Mininode: Reducing the Attack Surface of Node.js Applications. In *Proceedings of the 23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*. 121–134.

[38] Hyungjoon Koo, Seyedhamed Ghavamnia, and Michalis Polychronakis. 2019. Configuration-Driven Software Debloating. In *Proceedings of the 12th European Workshop on Systems Security (EuroSec)*.

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

[40] Anil Kurmus, Reinhard Tartler, Daniela Dorneanu, Bernhard Heinloth, Valentin Rothberg, Andreas Ruprecht, Wolfgang Schroder-Preikschat, Daniel Lohmann, and Rudiger Kapitza. 2013. Attack Surface Metrics and Automated Compile-Time OS Kernel Tailoring. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*.

[41] Lingguang Lei, Jianhua Sun, Kun Sun, Chris Shenefiel, Rui Ma, Yuewu Wang, and Qi Li. 2017. SPEAKER: Split-Phase Execution of Application Containers. In *Proceedings of the 12th Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*. 230–251.

[42] Yiwen Li, Brendan Dolan-Gavitt, Sam Weber, and Justin Cappos. 2017. Lock-in-Pop: Securing privileged operating system kernels by keeping on the beaten path. In *Proceedings of the USENIX Annual Technical Conference (ATC)*.

[43] Max Lillack, Christian Kästner, and Eric Bodden. 2018. Tracking Load-Time Configuration Options. *IEEE Transactions on Software Engineering* 44, 12 (2018), 1269–1291.

[44] Nicholas Luedtke. 2022. Linux Kernel CVEs. https://https://www.linuxkernelcves.com/.

[45] Steven McCanne and Van Jacobson. 1993. The BSD Packet Filter: A New Architecture for User-Level Packet Capture. In *Proceedings of the USENIX Winter Conference*.

[46] Metasploit. 2022. Metasploit Framework. http://www.metasploit.com.

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

[48] Shachee Mishra and Michalis Polychronakis. 2020. Saffire: Context-sensitive Function Specialization against Code Reuse Attacks. In *Proceedings of the 5th IEEE European Symposium on Security and Privacy (EuroS&P)*. 17–33.

[49] Shachee Mishra and Michalis Polychronakis. 2021. SGXPecial: Specializing SGX Interfaces against Code Reuse Attacks. In *Proceedings of the 14th European Workshop on Systems Security (EuroSec)*. 48–54.

[50] Mitre. 2016. CVE-2016-6321 - GNU Tar. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-6321.

[51] Collin Mulliner and Matthias Neugschwandtner. 2015. Breaking Payloads with Runtime Code Stripping and Image Freezing. Black Hat USA.

[52] NIST. 2021. CVE-2021-22890 - Curl. https://nvd.nist.gov/vuln/detail/CVE-2021-22890.

[53] Shankara Pailoor, Xinyu Wang, Hovav Shacham, and Isil Dillig. 2020. Automated Policy Synthesis for System Call Sandboxing. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 135 (nov 2020).

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

[55] Chris Porter, Girish Mururu, Prithayan Barua, and Santosh Pande. 2020. BlankIt library debloating: Getting what you want instead of cutting what you don't. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 164–180.

[56] Chenxiong Qian, Hong Hu, Mansour Alharthi, Pak Ho Chung, Taesoo Kim, and Wenke Lee. 2019. RAZOR: A Framework for Post-deployment Software Debloating. In *Proceedings of the 28th USENIX Security Symposium*.

[57] Chenxiong Qian, Hyungjoon Koo, ChangSeok Oh, Taesoo Kim, and Wenke Lee. 2020. Slimium: Debloating the Chromium Browser with Feature Subsetting. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 461–476.

[58] Anh Quach, Rukayat Erinfolami, David Demicco, and Aravind Prakash. 2017. 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)*. 65–70.

[59] Anh Quach, Aravind Prakash, and Lok Yan. 2018. Debloating Software through Piece-Wise Compilation and Loading. In *Proceedings of the 27th USENIX Security Symposium*. 869–886.

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

[61] Shellstorm. 2022. Shell-Storm. http://www.shell-storm.org.

[62] Linhai Song and Xinyu Xing. 2018. Fine-Grained Library Customization. In *Proceedings of the 1st ECOOP International Workshop on Software Debloating and Delayering (SALAD)*.

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

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

[65] Reinhard Tartler, Daniel Lohmann, Christian Dietrich, Christoph Egger, and Julio Sincero. 2011. Configuration Coverage in the Analysis of Large-Scale System Software. In *Proceedings of the 6th Workshop on Programming Languages and Operating Systems (PLOS)*.

[66] David Wagner and Drew Dean. 2001. Intrusion detection via static analysis. In *Proceedings of the IEEE Symposium on Security & Privacy (S&P)*. 156–168.

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

[68] Jianliang Wu, Ruoyu Wu, Daniele Antonioli, Mathias Payer, Nils Ole Tippenhauer, Dongyan Xu, Dave (Jing) Tian, and Antonio Bianchi. 2021. LIGHTBLUE: Automatic Profile-Aware Debloating of Bluetooth Stacks. In *Proceedings of the 30th USENIX Security Symposium*. 339–356.

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

[70] Tian Lan Yurong Chen and Guru Venkataramani. 2017. DamGate: Dynamic Adaptive Multi-feature Gating in Program Binaries. In *Proceedings of the Workshop on Forming an Ecosystem Around Software Transformation (FEAST)*.

[71] Zhi Zhang, Yueqiang Cheng, Surya Nepal, Dongxi Liu, Qingni Shen, and Fethi Rabhi. 2018. 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)*. 691–710.

[72] Xiangyu Zhang Zhongshu Gu, Brendan Saltaformaggio and Dongyan Xu. 2014. 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)*.