Automating Seccomp Filter Generation for Linux Applications

Claudio Canella
Graz University of Technology
Austria

Mario Werner
Graz University of Technology
Austria

Daniel Gruss
Graz University of Technology
Austria

Michael Schwarz
CISPA Helmholtz Center for Information Security
Germany

ABSTRACT

Software vulnerabilities undermine the security of applications. By blocking unused functionality, the impact of potential exploits can be reduced. While seccomp provides a solution for filtering syscalls, it requires manual implementation of filter rules for each individual application. Recent work has investigated approaches to automate this task. However, as we show, these approaches make assumptions that are not necessary or require overly time-consuming analysis.

In this paper, we propose Chestnut, an automated approach for generating strict syscall filters with lower requirements and limitations. Chestnut comprises two phases, with the first phase consisting of two static components, *i.e.*, a compiler and a binary analyzer, that statically extract the used syscalls. The compiler-based approach of Chestnut is up to factor 73 faster than previous approaches with the same accuracy. On the binary level, our approach extends over previous ones by also applying to non-PIC binaries. An optional second phase of Chestnut is dynamic refinement to restrict the set of allowed syscalls further. We demonstrate that Chestnut on average blocks 302 syscalls (86.5 %) via the compiler and 288 (82.5 %) using the binary analysis on a set of 18 applications. Chestnut blocks the dangerous `exec` syscall in 50 % and 77.7 % of the tested applications using the compiler- and binary-based approach, respectively. For the tested applications, Chestnut blocks exploitation of more than 61 % of the 175 CVEs that target the kernel via syscalls.

CCS CONCEPTS

• **Security and privacy → Operating systems security**.

KEYWORDS

seccomp; Linux; automated syscall filtering

ACM Reference Format:
Claudio Canella, Mario Werner, Daniel Gruss, and Michael Schwarz. 2021. Automating Seccomp Filter Generation for Linux Applications. In *Proceedings of the 2021 Cloud Computing Security Workshop (CCSW '21), November 15, 2021, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3474123.3486762

1 INTRODUCTION

The complexity of applications is steadily growing, and with that, also the number of vulnerabilities found in applications [39]. A consequence is that the attack surface for exploits is also growing. Especially in applications written in memory unsafe languages such as C, bugs often lead to memory safety violations that potentially enable exploits [59]. Even with state-of-the-art defenses, a risk remains that an attacker can exploit a remaining vulnerability in an application. For privileged applications such as `setuid` binaries, this can, in the worst case, fully compromise the entire system.

The remaining exploitation risk can be addressed by reducing the post-exploitation impact (cf. principle of least privilege). With available resources and interfaces limited to those strictly required by the application, a successful exploit cannot use arbitrary other functionality [35]. Especially blocking dangerous syscalls and syscall parameters that are not required by many applications, e.g., the `exec` syscall to execute a new program, reduces an attacker's possibilities in the post-exploitation phase. Application sandboxing limits the resources available to an application [23, 45] and, ideally, untrusted and potentially malicious, or benign but compromised applications cannot escape the sandbox.

On Linux, seccomp [15] and the extended seccomp-bpf can be used by applications to restrict the syscall interface. Seccomp-bpf [15] supports developer-defined filter rules. Each syscall can be blocked entirely or specific arguments for it. However, the correct usage of seccomp-bpf requires the developer to know which syscalls are used by the application and the included libraries. Given the considerable effort, seccomp is mainly used in applications that provide isolation mechanisms, e.g., sandboxes [19, 28].

Recent works proposed two methods to automatically generate such seccomp filters [11, 21]. The first approach utilizes the compiler and various external tools to derive the filters during compilation [21]. To minimize the set of syscalls, the approach relies on sophisticated points-to analysis [2] to generate a call graph of reachable functions and syscalls. The second approach relies on binary analysis to determine the syscalls an existing binary intends to use [11]. While these are first solutions to the problem of automating filter generation, both come with clear limitations. For instance, the first approach does not scale with the program size due to the points-to analysis [2, 25]. In practice, the overheads can be prohibitively large as they would require a massive upscaling of development and build server resources. The second approach comes with a strong requirement that the application is compiled as a position-independent code (PIC) binary (PIE) [11]. While PIE is the default on recent Ubuntu distributions for C and C++ compiled programs, static C and C++ binaries are by default not compiled as

PIE. Other compiled binaries are often not PIE either, e.g., 'golang' binaries such as the popular git server Gogs, which are not supported by these previous works. Both limitations reduce the set of applications that can be protected with these solutions substantially.

In this paper, we present a novel approach that overcomes the limitations of previous ones and automatically generates strict seccomp filters for native Linux userspace applications. We show that our approach is a significant improvement over the compiler-based approach by Ghavamnia et al. [21] without the expensive points-to analysis to generate filter rules. Instead, a faster *has address taken* approach achieves the same accuracy but at a fraction of the performance impact on compilation time. In contrast to DeMarinis et al. [11], we demonstrate that the requirement of a PIC binary is not necessary, significantly extending the set of target applications. We implement our method in a proof-of-concept tool, Chestnut.[1]

Chestnut uses a two-phase process: A static first phase $\mathcal{P}1$ consisting of two static components (**Sourcalyzer** and **Binalyzer**), and an optional dynamic second phase $\mathcal{P}2$ (**Finalyzer**). Using static analysis, Chestnut first identifies unused syscalls without running the application in $\mathcal{P}1$ and dynamically refines this set in $\mathcal{P}2$ to reduce the inherent limitations of the static analysis in $\mathcal{P}1$.

For **Sourcalyzer**, we extend the LLVM framework to detect the syscalls used by the application already during compile- and link-time. The syscall information for each shared library is either extracted using the compiler or using Binalyzer. **Binalyzer** can be used for applications and libraries which are either not compatible with LLVM or where the source code is not available. We rely on capstone [48] to disassemble applications and to locate syscalls. Using symbolic backward execution [37] from the `syscall` instruction, we infer the syscall number used in the identified syscall. Additionally, we use the control-flow graph (CFG) recovery functionality of angr [64] to map exported functions to identified syscalls. Exactly as in previous work, an inherent limitation of static approaches is that they can miss syscalls in rare cases if control-flow cannot be inferred correctly. However, we observe that more frequently, the set of used syscalls is overapproximated. To refine the number of allowed syscalls, we provide a complementary optional dynamic approach in the second phase of Chestnut. In this second phase, **Finalyzer** traces all syscalls of the application and then refines the allowlist to further restrict or relax the seccomp filters.

We demonstrate our approach's feasibility by applying it to various real-world client applications, such as git and busybox, database applications, such as redis and sqlite3, and Nginx as a server application. We show that Chestnut does not impair their functionality but significantly reduces the attack surface. On average, Chestnut blocks 295 syscalls (84.5 %) on Linux kernel 5.0. In the 18 real-world binaries we evaluated, Chestnut blocked the `exec` syscall for 50 % of the applications using Sourcalyzer and in 77.7 % using Binalyzer. We block the `mprotect` syscall in 61.1 % of the tested applications using Sourcalyzer. Furthermore, we evaluate our approach on real-world exploits, showing that Chestnut blocks exploitation of 64 % and 62 % of CVEs using Sourcalyzer and Binalyzer, respectively. We compare our approaches with related previous work [11, 21] and show that we are similarly effective in mitigating CVEs via compiler-based approaches [21]. However, we improve the performance by up to factor 73. In contrast to previous work [11], we show that binary-based approaches can also be applied to non-PIC binaries. We evaluate the functional correctness of Sourcalyzer with test suites as well as a 6-month long-term case study of a Sourcalyzer-protected Nginx production server. Furthermore, we are the first to evaluate how tight automatically generated filter rules are by relying on available test suites. We also advance the state of the art in evaluation of automatic syscall filtering, with a long-term case study and by measuring code coverage to confirm our approach's validity.

Filters generated automatically with a tool might not always be as strict as theoretically possible. However, there is no time investment required by the developer, making it a very inexpensive defense in depth. More importantly, Chestnut can be applied to and improve the security of existing and widely-used technology, *i.e.*, seccomp, making syscall filtering available to commodity applications. The only runtime overhead introduced is the small overhead of using seccomp, similar as containers already do today.

To summarize, we make the following contributions:

(1) We present a new compiler-based approach for automated syscall-filter generation, up to 73x faster than previous work.
(2) We present a dynamic method to refine the filters.
(3) We show that Chestnut blocks the exploitation of more than 61 % of the 175 CVEs in the Linux kernel exploitable via syscalls.
(4) We show that previous requirements can be lifted and show that our approach can be applied to PIC and non-PIC binaries.
(5) In a 6 month long-term study on Nginx, we demonstrate the correctness of our approach and do not observe a single crash.

Outline. Section 2 provides background, Section 3, threat model, and design of Chestnut. Section 4 discusses our static and Section 5 our dynamic approach. We evaluate Chestnut in Section 6 and discuss related work and limitations in Section 7. Section 8 concludes.

2 BACKGROUND

Sandboxing is a security mechanism that constrains software within a tightly controlled environment by restricting the available resources to a required minimum [23, 45]. Hence, the damage in case of exploitation is limited. These restrictions may encompass the ability to access the network, limit the amount of storage, file descriptors, or inhibit the application from issuing specific syscalls. By now, different forms of sandboxing have been adopted by many browser vendors to secure their products [44, 50, 58, 66].

2.1 Linux Seccomp

To facilitate operations that require higher privileges or direct hardware access, the kernel provides syscalls to every userspace application. As with other interfaces, they also contain bugs that can lead to privilege escalation [31–33]. Hence, platform security profits from limiting the amount of syscalls that an application can perform. With Secure Computing (seccomp) [15], Linux provides a filter that allows a userspace program to specify the syscalls it performs over its lifetime. The kernel then blocks the remaining syscalls for the sandboxed application that might originate from an application being hijacked. As seccomp filters do not dereference pointers, so-called time-of-check time-of-use attacks [38] common

[1]The prototype and several demo videos can be found in our anonymous GitHub repository https://github.com/chestnut-sandbox/Chestnut.

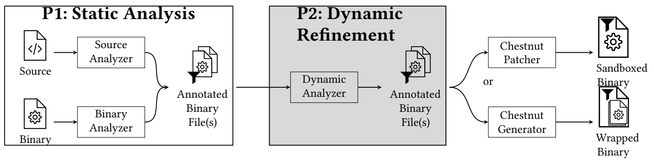


Figure 1: Chestnut consists of $\mathcal{P}1$, a static analysis on source and binary files, and an optional dynamic analysis, $\mathcal{P}2$, refining the filters. Chestnut can either rewrite the binary or generate a tailored sandbox to block unused syscalls.

in syscall interposition frameworks are not possible. Examples of applications that rely on seccomp are Chromium [9], Firefox [41], and the zygote process in Android systems [27].

2.2 Memory Safety

Memory safety is an essential concept in computer security, and its violation can lead to exploitation. One way to exploit a program is to corrupt its memory and to divert control flow to a previously injected code sequence. This code sequence, *i.e.*, the payload, is called *shellcode* and is commonly written in machine code. These types of attacks are commonly known as control-flow hijack attacks [59].

ROP attacks [53] allow chaining existing code gadgets within an application to perform malicious tasks. Each gadget is a sequence of instructions ending with a return instruction. ROP attacks are hard to defend as all the information is already present within the application, *i.e.*, an attacker does not need to inject code. While ROP attacks overwrite saved return addresses, similar attacks exist that overwrite other pointers [4, 6, 7, 22, 51] or signal handlers [5].

2.3 Executable and Linkable Format

On Unix-based systems, the highly flexible and extensible Executable and Linkable Format (ELF) [13] is used for shared libraries and executables. ELF files consist of header and data, including a program and a section header table for segments and sections. Segments contain information for the run-time execution of the binary, while sections contain information for linking and relocating.

Dynamic Linking. The dynamic linker is responsible for loading and linking shared libraries used by an executable during runtime [13]. For that, the dynamic linker maps the shared library's content into memory and ensures its functionality, e.g., filling jump tables and relocating pointers. On Unix-like systems, the dynamic linker is selected at link time and is embedded into the ELF file.

3 DESIGN OF CHESTNUT

In this section, we introduce our threat model, outline challenges of automatic filter generation, and discuss the high-level idea of Chestnut. We introduce the main components of Chestnut (Figure 1), *i.e.*, the compiler modification *Sourcalyzer*, the binary analyzer *Binalyzer*, and the dynamic refinement tool *Finalyzer*.

3.1 Threat Model and Idea of Chestnut

Chestnut supports Linux applications available as either C source code or as a binary, and is not limited to PIC binaries as previous work [11]. These applications can range from server applications to applications executing potentially malicious code that is not controlled by the user, such as browsers, office applications [42], and pdf readers [16, 17]. Chestnut can also restrict the syscall interface of messenger applications which have been used to compromise systems [24, 54]. We assume correct usage of Chestnut in one of its variants (cf. Figure 1). Chestnut assumes that the application is not malicious but potentially vulnerable to exploitation, e.g., due to a memory-safety violation, enabling an attacker to gain arbitrary code execution within the application. We assume that post-exploitation requires syscalls, e.g., to gain kernel privileges. Syscalls provided for file operations can potentially be hijacked by an attacker to modify configuration files. Argument-level API specialization [40] can be used to protect against such attacks. We consider this an orthogonal problem, in line with related work [21], and do not consider such attacks. Chestnut is orthogonal to other defenses such as CFI, ASLR, NX, or canary-based protections and enhances the security in case these other mitigations have been circumvented. Side-channel and fault attacks are out of scope.

3.2 Challenges

Automatic filter generation using a static approach requires solving the following four challenges that we detail in Sections 4 and 5.

$\mathcal{C}1$: Identifying Syscall Numbers for each Syscall. To automatically block unused syscalls, we must identify the syscalls used by the application. The syscall itself is usually a single instruction, e.g., `syscall` (x86_64) or `svc #0` (AArch64). The actual syscall is specified as a number in a CPU register, e.g., `rax` (x86_64) or `x8` (AArch64) [12]. Hence, the first challenge is to identify the syscall number for a specific syscall. Syscalls have many different forms within a program, e.g., inline assembly, assembly file, or issued with the libc *syscall* wrapper function. Moreover, syscalls might not be called directly, but via a call chain through various libraries.

$\mathcal{C}2$: Reconstruct Call Sites of Syscalls. By solving challenge $\mathcal{C}1$, we know which syscalls are potentially called by the target application. Unfortunately, including all detected syscalls of the binary and the used libraries does not suffice. Most binaries link against libc, which provides an implementation of almost all syscalls. Hence, the generated filters would be too permissive as they would basically allow all syscalls. We have to analyze the reachability of the identified syscalls by constructing a call graph for every binary.

$\mathcal{C}3$: Generate Set of Syscalls. To generate the final set of syscalls for our application, the information from $\mathcal{C}1$ and $\mathcal{C}2$ has to be combined for the application and its libraries. By combining the call graph obtained in $\mathcal{C}2$ with the information which functions are used in the application and libraries, we create a set of functions potentially called by the application. In combination with the call graph ($\mathcal{C}2$) and syscall numbers ($\mathcal{C}1$), this set provides the information about all the syscalls that the application can execute.

$\mathcal{C}4$: Install Filters. Our library (libchestnut) relies on seccomp to apply the syscall filters. This library uses the allowlist ($\mathcal{C}3$), generates the seccomp rules, and installs the resulting filters before the actual application starts at the main entry point.

3.3 High-Level Idea

This section discusses how the components of Chestnut solve the challenges, with the implementation details in Sections 4 and 5.

Sourcalyzer. Sourcalyzer is the compiler-based component of Chestnut for static analysis of the application source code. It is a compiler pass in LLVM that identifies all syscalls at compile time.

For statically linked applications, and given that libraries are compiled with Sourcalyzer, the compiler and linker are aware of the entire codebase and can thus identify every syscall instruction of the final binary. As the C standard library implements almost all syscalls, linking against it would allow almost all syscalls, which renders the filters ineffective. Hence, we need to determine further which syscalls are used by the application by analyzing the CFG to solve challenges $\mathcal{C}2$ and $\mathcal{C}3$. While comparable work [21] needs to perform the same task, we demonstrate a solution that is up to factor 73 faster. We discuss this in Section 4.1.

Binalyzer. Sourcalyzer requires the source code of the application and all used libraries. Binalyzer has the same goal but works directly on the binary level. With this, our approach is also applicable to programs where the source code is not available or not compatible with LLVM, retrofitting the approach to binaries. In contrast to previous work [11], Binalyzer is not restricted to PIC binaries.

The idea is to scan binaries and libraries for *syscall* instructions and use symbolic backward execution [37] from these locations to infer the respective syscall number, again solving challenge $\mathcal{C}1$. To reduce overapproximation, Binalyzer leverages CFG analysis of all dependencies to map exported functions to syscall numbers ($\mathcal{C}2$, $\mathcal{C}3$). Binalyzer also works on stripped binaries as all the required information is still included for dynamic linking.

Finalyzer. Working around limitations of static analysis, we propose an optional dynamic phase, Finalyzer, based on syscall tracing, removing or adding filters that cannot be identified statically. Finalyzer is solely intended to refine filters identified by our static approaches in a controlled, benign environment.

The dynamic nature of Finalyzer allows simplifying challenges $\mathcal{C}1$ to $\mathcal{C}4$. Finalyzer extracts the syscall number during runtime ($\mathcal{C}1$) by intercepting all syscalls for the target application. By intercepting the syscall, it is inherent that the syscall is reachable ($\mathcal{C}2$). In this step, missed syscalls are added to refine the installed filter list ($\mathcal{C}4$). We discuss this process in more detail in Section 5.

Combining Components. Chestnut is designed to allow combining all three components (cf. Figure 1). For instance, Finalyzer is intended to be used as an optional step after the static components if they cannot infer the used syscalls due to the static analysis's limitations. An instance where this is necessary is when an application dynamically starts other applications. The child process inherits the parent's filters, which cannot be relaxed anymore. By combining the static approaches with Finalyzer, the syscalls of the child process are identified and added to the application's allowlist. Sourcalyzer can also be used in combination with Binalyzer, e.g., if the source is available for the application but not for a used library.

Applying Syscall Filters. The output of each component is a set of syscalls the application can call. For Sourcalyzer, the syscall filters are directly compiled into the target application. However, if this is either not desirable or possible, e.g., because only the binary is available, we provide two tools to apply the syscall filters (cf. Figure 1). ChestnutGenerator creates a sandbox tailored to the target application. Alternatively, ChestnutPatcher directly patches the target application to include the syscall filters and libchestnut.

4 STATIC FILTER EXTRACTION

We now present the two static approaches of $\mathcal{P}1$ to automatically generate syscall filters. We highlight the necessary steps, cf. Figure 2, for solving the outlined challenges in a fast and efficient way in both a compiler and a binary-based approach in more detail.

4.1 Compiler-Based Approach

Sourcalyzer utilizes the LLVM compiler framework [34] to extract syscalls from source code. It uses module passes (*i.e.*, one analysis and one transformation pass) that operate on the LLVM intermediate representation (IR). Additionally, LLVM's linker lld is extended to combine the extracted information from multiple translation units. We use an unmodified compiler-rt and musl libc. Hence, using Chestnut with the Sourcalyzer approach is as simple as compiling and linking an application with our extended toolchain.

$\mathcal{C}1$: Identify System Call Numbers. To invoke a syscall, x86_64 provides the *syscall* and AArch64 the *svc #0* instruction. The extraction of the syscall number is the only architecture-dependent part of Chestnut. Given the syscall number, the rest of the approach is the same for all architectures supported by LLVM. Syscalls are typically abstracted by the standard C library via the *syscall()* function. musl additionally provides the function *__syscall_cp()*. To detect all invocations of syscalls, we need to detect all three cases, *i.e.*, inline assembly, the *syscall* function, and the *__syscall_cp()* function.

The LLVM analysis pass iterates over all functions within a translation unit. For each function, we iterate over every LLVM IR instruction to check whether it is a call site. If it is, we check whether it is an inline syscall assembly statement or a call to either one of the *syscall* or *__syscall_cp* functions. In all three cases, we extract the first argument as it is the number of the requested syscall. Due to the way we traverse the IR, we also know precisely what function performs the respective syscall.

Our proof-of-concept implementation currently does not parse assembly files as they are treated differently by LLVM than normal source files. Hence, if a syscall is implemented in one, e.g., *clone*, we cannot detect it, but a full implementation can handle this case.

$\mathcal{C}2$: Reconstruct Syscall Call Sites. Sourcalyzer uses the syscall numbers extracted in $\mathcal{C}1$ as a starting point for further analyzing which syscalls are used based on the call graph. The main challenge here is extracting a reasonably precise call graph without inducing huge performance overheads or by requiring changes to the common compilation model (e.g., by demanding link-time optimization (LTO)). In particular, restricting indirect function call sites to a set of possible call targets is a necessary, but typically quite expensive, task that commonly relies on inter-procedural pointer analysis (e.g., Andersen [2], Steensgaard [57]). This type of analysis requires access to the whole program and often does not scale efficiently to larger program sizes [2, 25]. An automated syscall-detection system based on this form of analysis and its impact on the compile-time performance has been demonstrated by Ghavamnia et al. [21]. This approach also requires changes to the common compilation model, which is not supported by every application.

Hence, as we want to avoid changes to the compilation model and given that our application can tolerate some imprecision, we do not use sophisticated pointer analysis in our prototype implementation and opt for a function-signature based heuristic to determine

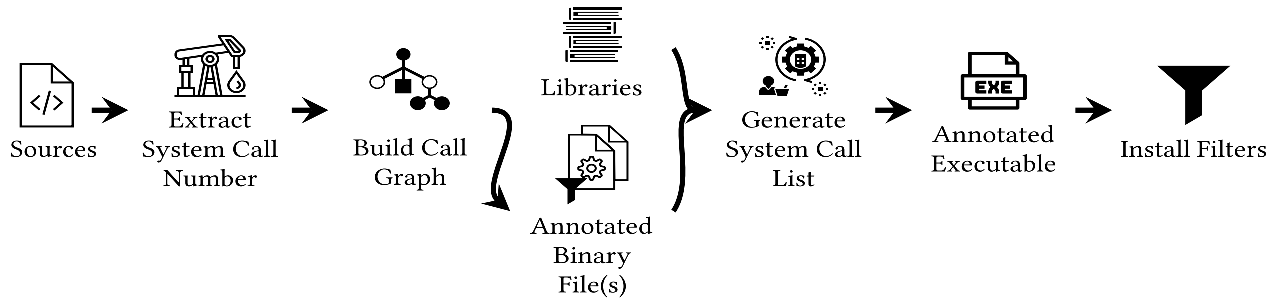


Figure 2: The different steps of Sourcalyzer, which starts with the source and ends with a fully sandboxed application.

possible call targets. Every function in the program where the function type of the call site matches the function type of the definition is considered a possible call target. For correctly typed programs, this heuristic is an overapproximation of the actual possible call targets, which corresponds to permitting more syscalls than are actually needed (cf. Section 6.4.2).

Both the LLVM IR passes and the linker are involved in mapping syscall numbers to functions. Our analysis pass traverses over all defined functions within the module. It extracts function signature, direct and indirect function calls, and functions referenced in the code (for which function pointers exist, *i.e.*, functions that have their address taken). The latter is similar to what LLVM uses for software-based CFI. This gives rise to the assumption that the resulting call graph is precise enough as applications that use software-based CFI would otherwise not work correctly.

We perform our analysis in the same traversal in which we locate the used syscall numbers (see $\mathcal{C}1$). We also support function aliases by treating them as copies of the original function. Finally, references to functions in global initializers are extracted, as they are used, e.g., for global file structures.

Our IR transformation pass stores the information collected from the analysis pass in the ELF object for the linker to use.

$\mathcal{C}3$: Generate Syscall Set. By solving challenges $\mathcal{C}1$ and $\mathcal{C}2$, we generate object files containing the serialized syscall and call graph information. The linker extracts this information from all the provided input files to perform the actual call graph construction and syscall number propagation. Finally, the linker can either generate the set of relevant syscalls for the application or a flattened call graph for further processing.

In more detail, after loading the call graph metadata, all reachable functions are resolved according to their symbol's linkage specification (e.g., local or global, strong or weak), and a list of indirect callable functions is generated. In the next step, a call graph is constructed in which each node represents a function, and each directed edge represents a possible control-flow transfer from the caller to the callee. The linker transforms this call graph into a directed acyclic graph (DAG) using Tarjan's algorithm [60], enabling efficient propagation of the information. Namely, each graph node has to be updated only once by visiting the DAG in post-order. Using the discovered strongly coupled components, circular call dependencies are directly resolved by merging the information from all functions that are part of the respective cycle in the original call graph. As a result, the linker has access to a flattened call graph in which, for every function, all reachable syscall numbers are known.

With the flattened call graph, we determine which syscalls our final application needs. For a static binary, we extract all syscalls that can be reached from the *main* and the *exit* function and embed them as a simple list of numbers into the final ELF binary. For dynamic binaries or shared libraries, we instead embed the flattened call graph into the linked binary for further processing.

$\mathcal{C}4$: Install Seccomp Filters. After linking with Sourcalyzer, the binary contains annotations for the application's used syscall numbers directly or its flattened call graph that still needs to be combined with the additional dynamic libraries. For static linkage, we delegate the processing to the application itself by additionally linking against our libchestnut library. This library contains a constructor that extracts the syscall numbers and installs the seccomp filters using libseccomp [14] before the application starts executing.

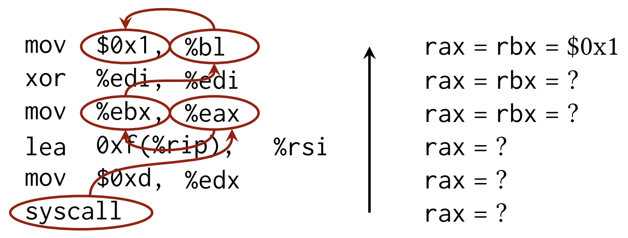
In the second case, dynamic linkage, we provide two options. ChestnutPatcher extracts the embedded call graph from all library dependencies and determines all syscalls from functions that are reachable from the *main* and *exit* function. Finally, the tool adds a new note section with information on syscall numbers. As the compiler has generated the dynamic binary, we can already link libchestnut against it automatically. ChestnutGenerator performs the same steps except that it does not modify the binary but creates a launcher that sets up the filters before executing the actual binary.

4.2 Binary Syscall Extraction

The second static approach of Chestnut, Binalyzer, works on the binary level. With less semantic information available than on the compiler level, Binalyzer works without access to the source code and even for stripped binaries. In contrast to previous work [11], we demonstrate our approach on PIC and non-PIC binaries.

$\mathcal{C}1$: Identify Syscall Numbers. The syscall number is not encoded in the instruction but is provided in a register, *i.e.*, `rax` on x86_64 or x8 on AArch64. Hence, Binalyzer has to reconstruct the syscall number by inferring the content of this register.

Binalyzer uses the capstone framework [48] to disassemble a binary as this framework supports various ISAs, e.g., x86_64 and AArch64. Starting from a syscall instruction, Binalyzer leverages symbolic backward execution [37]. Tracking back from the syscall instruction, Binalyzer tracks the register's symbolic value containing the syscall number. In many cases, the immediate for the syscall



```
mov  $0x1, %bl              rax = rbx = $0x1
xor  %edi, %edi             rax = rbx = ?
mov  %ebx, %eax             rax = rbx = ?
lea  0xf(%rip),   %rsi      rax = ?
mov  $0xd, %edx             rax = ?
syscall                     rax = ?
```

Figure 3: Symbolic backward execution starts from the syscall instructions and finds the syscall number by symbolically tracking the corresponding CPU register.

number is directly moved to the register before the syscall instruction as it is a constant value. However, in some cases, there is at least some form of register-to-register transfer involved. These transfers also include register copies where only a lower part of the register is involved. Thus, as illustrated in Figure 3, Binalyzer keeps the content of the register symbolic and steps back through the binary, symbolically evaluating operations. This symbolic backward execution is repeated until either a concrete immediate for the syscall number is identified, or after a user-definable number of instructions have been analyzed without successfully identifying the immediate.[2] One failure reason can be that the syscall instruction is a call or jump target, *i.e.*, there are potentially multiple call sites reaching the instruction with different syscall numbers. Luckily, the syscall instruction is usually inlined, and thus, we do not consider such situations for our proof of concept.

$\mathcal{C}2$: Reconstruct Syscall Call Sites. To reduce the overapproximation of syscalls, Binalyzer analyzes the CFG to map syscalls to (exported) functions. We rely on angr [64] to statically create a CFG of the binary. Based on the basic blocks of all functions in the CFG, we assign every syscall identified in $\mathcal{C}1$ to a function. Binalyzer traverses the CFG from each exported function as the root node to identify reachable functions with a syscall instruction. Assuming a correctly reconstructed CFG from angr and correctly identified syscall numbers ($\mathcal{C}1$), this yields a set of possible syscalls per exported function.

$\mathcal{C}3$: Generate Syscall Set. To solve challenge $\mathcal{C}3$, we have to combine the information created from solving $\mathcal{C}2$ for all binaries, *i.e.*, the application binary and all of its libraries. We cannot create a complete CFG over the application binary and its libraries as this would take multiple hours to days, depending on the size of the application and the number of dynamic libraries. Instead, we chose to overapproximate the number of possible syscalls by relying on individual CFGs that we merge. We only consider functions that are defined in the dynamic symbol table of the ELF file. These functions are found in the dynamic libraries loaded by the application. Hence, we search for these functions in the shared object dependencies and look up the used syscalls for the function in the respective library. As shared libraries can also have a dynamic symbol table if they call functions from other libraries, this process is repeated for all dynamic symbols of all shared object dependencies.

Solving challenge $\mathcal{C}3$ yields a set of syscalls that the application can potentially call. This assumes that no dynamic libraries are loaded at runtime, e.g., via `dlopen`, and that the application does not execute a different binary at runtime, e.g., via `exec`. In such cases, we need to resort to $\mathcal{P}2$ as the complete set of syscalls cannot be determined statically.

[2]For the evaluation, we set this number to 30, which was sufficiently large.

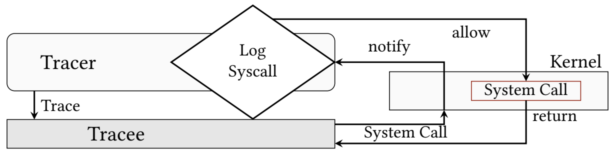


Figure 4: The tracer gets notified by the kernel when the tracee executes a syscall. The tracer logs the syscall and informs the kernel to execute the syscall.

$\mathcal{C}4$: Install Seccomp Filters. From the full set of syscalls, Binalyzer has to create filter rules and apply them to the binary. We cannot simply compile the filters with the application (cf. Sourcalyzer). Instead, Binalyzer supports binary rewriting or alternatively building a sandbox wrapper (cf. Figure 1). ChestnutGenerator is a simple tool that sets up the filter rules and starts the target program.

With binary rewriting, Binalyzer stores the syscall numbers in the ELF binary and injects a new shared object dependency, libchestnut. The library provides a constructor function, which is called before the actual application starts and which parses the filters stored in the binary to apply the seccomp filter rules. The advantage of a rewritten binary is that it does not need any launcher.

5 DYNAMIC REFINEMENT

In this section, we discuss the optional $\mathcal{P}2$ component Finalyzer, a method to dynamically refine the previously detected syscall filters. It simplifies the challenges $\mathcal{C}1$ to $\mathcal{C}4$ by inspecting syscalls just-in-time in a secure and controlled environment during development.

5.1 Limitations of Static Approaches

While our approach for statically detecting an application's syscalls works well for most binaries (cf. Section 6), there are inherent limitations to a static approach. Dynamically loaded libraries, e.g., codecs, plugins, self-modifying, or JIT-compiled code, often cannot be analyzed statically. Moreover, seccomp is not flexible enough to handle scenarios involving child processes with a different set of syscalls, as a child inherits its parent's filters and can only further restrict but not relax them. Hence, the parent also needs to install filters for the child's syscalls.

5.2 Implementation Details

In our prototype, Finalyzer is an strace-like syscall-tracing component linked against the target application or used as a standalone wrapper for a binary (cf. Figure 1). This allows Finalyzer to work with Binalyzer and Sourcalyzer. If desired, it can also be used without the static components to identify required syscalls.

In either case, Finalyzer, *i.e.*, the *tracer*, first creates a child process, the *tracee*. Finalyzer then installs seccomp filters for all syscalls in a way that informs the tracer about a seccomp violation. To enable this behavior, the tracer needs to attach itself to the tracee. The tracee then stops execution until it receives the continue signal from the tracer to ensure that it successfully attached itself. If the child process creates a new child process, the tracer is automatically attached to the newly created child process. The tracer is then also informed of the unsuccessful execution of the child's syscalls.

When receiving the notification of a violating syscall, Finalyzer extracts the syscall number ($\mathcal{C}1$), logs it ($\mathcal{C}3$), and allows it for all future occurrences (cf. Section 3.3), as illustrated in Figure 4. As the syscall is indeed executed, it is inherent that it is reachable ($\mathcal{C}2$).

Once Finalyzer has finished tracing the application, it cross-references the set of obtained syscalls with the ones obtained in $\mathcal{P}1$. If a syscall is missing, it adds the newly detected syscall to the allowlist. Optionally, it can also be used to remove syscalls that $\mathcal{P}1$ identified but which were never executed during $\mathcal{P}2$.

6 EVALUATION

In this section, we evaluate the performance, functional correctness, and security of Chestnut. Our evaluation is in line with related work [11, 21] while improving on it in several points, *i.e.*, we evaluate several parts that were not yet evaluated in these works. In the performance evaluation, we evaluate the one-time overheads of Chestnut, such as compile time and binary-analysis time. We also discuss the runtime overhead seccomp introduces. For the functional correctness, we evaluate whether Chestnut causes any issues in terms of functionality of existing real-world software, e.g., crashes. We also perform a 6-months long evaluation of an Nginx server with its syscall interface restricted by Sourcalyzer. In the security evaluation, we evaluate the ability of Chestnut to block the dangerous exec syscall and the overapproximation of syscalls in general. With the latter, we are the first to demonstrate how tight the automatically generated filters are. Furthermore, we evaluate how well Chestnut can mitigate real-world exploits. Finally, we detail differences between our work and related works in this field.

6.1 Setup

For the evaluation of Chestnut, we focus on x86_64. Note that the only architecture-dependent part of Chestnut is the extraction of the syscall number. Hence, we do not expect significant differences for other architectures. We also verified that the general approach works across architectures by successfully extracting the syscall numbers from musl libc for both x86_64 and AArch64.

We evaluate Chestnut on various real-world applications (cf. Table 1), including client, server, and database software. While busybox may be seen as a non-obvious choice, it is in line with previous work that used coreutils for the evaluation [47]. We instead chose busybox as the number of provided utilities is 3 times higher, making it a better choice for our evaluations. For evaluating Sourcalyzer, we compile the binaries statically with and without Chestnut enabled using our modified compiler. For Binalyzer, we compile the applications dynamically using GCC 7.5.0-3 on Ubuntu 18.04.4. For the sake of brevity, we do not evaluate every combination of components and sandboxes but focus on libchestnut for Sourcalyzer and ChestnutGenerator for Binalyzer.

6.2 Performance Evaluation

In this section, we evaluate the performance of Chestnut. This includes the one-time overheads for compiling (Section 6.2.1) or binary analysis (Section 6.2.2), the increase in binary size (Section 6.2.3), and runtime overheads (Section 6.2.4).

6.2.1 Compile-Time Overhead. We analyze the impact Sourcalyzer has on the compile time of an application. To make comparison possible, we compile the application 10 times with and without our modification enabled, always using our modified compiler, and use the average compile time over these runs.

As the results show, we observe the worst-case overhead for the git application with an increase from an average of 65.5 s ($\sigma_{\bar{x}} = 0.094$, $N = 10$) to 84 s ($\sigma_{\bar{x}} = 0.054$, $N = 10$), an increase of 28 %. For the busybox utilities combined, the average increases from 10.94 s ($\sigma_{\bar{x}} = 2.88$, $N = 10$) to 10.99 s ($\sigma_{\bar{x}} = 2.79$, $N = 10$). When compared to related work [21], we observe a speedup of factor 73 for Nginx when using Sourcalyzer. This low overhead makes it a feasible approach to be used in everyday development cycles.

6.2.2 Binary Extraction Runtime. For Binalyzer, we evaluate the time it takes to extract the syscalls from the dynamic binary. We assume that default dependencies like *libc.so* have already been processed, *i.e.*, their extracted call graph is available. For completeness, we timed the extraction of syscalls from *libc.so*, which takes on average 44.66 s ($\sigma_{\bar{x}} = 0.18$, $N = 10$). For the applications themselves, we can see in Table 1 that the extraction process is in the range of 2 to 10 s for the individual busybox utilities, with an average time of 3.4 s ($\sigma_{\bar{x}} = 0.73$, $N = 10$). For large binaries like FFmpeg (> 100 MB) and its dynamic dependencies, the extraction takes ≈11 min.

6.2.3 Binary Size Analysis. The code size does not increase with added filters, but the binary size increases by the meta-information.

Compiler. We analyze the size of the binary produced by Sourcalyzer compared to a vanilla application. Chestnut needs to treat static and dynamic ELF files differently as syscall numbers of externally linked libraries are not known. In a static binary, we only add the set of syscall numbers to the binary and link against libchestnut and libseccomp. As both libraries are of fixed size, the maximum overhead in a static binary is limited by the number of syscalls Linux provides, *i.e.*, 349 on Linux 5.0. Table 1 shows the overhead for statically linked binaries. As expected, the overhead is quite small in large binaries, e.g., FFmpeg. In the small busybox utilities, the overhead appears to be huge (> 177 %), but as these binaries sizes are in the lower kilobyte range (40-100 kB), linking against two additional libraries drastically increases the size. Nevertheless, the binaries remain in the kilobyte range.

For dynamic binaries and shared libraries, we have to embed the entire call graph as we need the information later on to determine the required syscalls. Table 2 shows the size increase for three shared libraries. In *libcrypto.so*, we observe a worst-case increase from 4.1 MB to 23 MB (460 %). The overhead also increases with the size of the binary as the call graph is larger for the larger codebase.

Binary. Table 1 shows the increase for Binalyzer. We opted to generate a binary that needs to be launched by ChestnutGenerator instead of rewriting the binary. Still, for simplicity, we embed the detected syscalls in the binary from where our wrapper extracts the information. As we embed only the numbers, the overhead in all 18 applications is less than 2 %. Binary rewriting incurs the overhead of the dependency on libchestnut and libseccomp, but this increase is again negligible beneath the other overheads.

Table 1: Results for the compiler- (</>) and binary-based (⚙) approach of Chestnut, respectively. We show the number of detected syscalls in $\mathcal{P}1$, used syscalls, and added syscalls in $\mathcal{P}2$, the size overhead of the annotations, compile-time overhead (for Sourcalyzer), and binary analysis time (for Binalyzer). The `exec` and `mprotect` columns indicate whether Chestnut blocks (✓) the respective syscalls or not (✗). We also show the percentage of fully mitigated CVEs and the individual subvariants. Syscalls added in $\mathcal{P}2$ are only necessary for edge cases in our proof-of-concept implementation of Chestnut.

	Software	#Syscalls Found / Used / $\mathcal{P}2$ Added		Size Overhead		Analysis Time		exec		mprotect		Fully Mitigated		Subvariant Mitigated	
		</>	⚙	Compiler (</>)	Binary (⚙)	Compiler (</>)	Binary (⚙)	</>	⚙	</>	⚙	</>	⚙	</>	⚙
Client	ls	24 / 14 / 0	39 / 18 / 1	+173 kB (253 %)	+288 B (1.08 %)	+0.38 s (1.72 %)	3.041 s	✓	✓	✓	✗	81.1 %	81.1 %	87.5 %	87.5 %
	chown	22 / 11 / 0	36 / 14 / 0	+174 kB (369 %)	+280 B (1.52 %)	+0.29 s (1.35 %)	2.777 s	✓	✓	✓	✗	81.7 %	81.7 %	87.8 %	87.8 %
	cat	18 / 6 / 0	34 / 13 / 0	+174 kB (397 %)	+272 B (1.9 %)	+0.08 s (2.29 %)	2.576 s	✓	✓	✓	✗	82.3 %	81.7 %	88.1 %	87.8 %
	pwd	16 / 4 / 0	34 / 14 / 0	+175 kB (430 %)	+272 B (1.92 %)	+0.21 s (0.98 %)	2.507 s	✓	✓	✓	✗	85.1 %	81.7 %	90.3 %	87.8 %
	diff	25 / 9 / 0	36 / 16 / 0	+173 kB (304 %)	+280 B (1.25 %)	+0.06 s (1.44 %)	2.946 s	✓	✓	✓	✗	81.1 %	81.7 %	87.5 %	87.8 %
	dmesg	15 / 5 / 0	34 / 14 / 0	+176 kB (439 %)	+272 B (1.92 %)	+0.08 s (2.17 %)	2.452 s	✓	✓	✓	✗	85.1 %	81.7 %	90.3 %	87.8 %
	env	15 / 3 / 0	33 / 13 / 0	+175 kB (416 %)	+272 B (1.92 %)	+0.07 s (1.88 %)	2.416 s	✗	✓	✓	✗	81.7 %	81.7 %	88.4 %	87.8 %
	grep	20 / 11 / 0	34 / 16 / 0	+174 kB (177 %)	+272 B (1.49 %)	+0.41 s (1.88 %)	2.748 s	✓	✓	✓	✗	81.7 %	81.7 %	87.8 %	87.8 %
	true	3 / 1 / 0	32 / 12 / 0	+200 kB (3277 %)	+264 B (4.4 %)	+0.09 s (2.55 %)	9.908 s	✓	✓	✓	✗	98.3 %	83.4 %	98.4 %	88.7 %
	head	17 / 7 / 0	33 / 13 / 0	+174 kB (434 %)	+272 B (1.92 %)	+0.06 s (1.64 %)	2.436 s	✓	✓	✓	✗	82.3 %	81.7 %	88.1 %	87.8 %
	git	82 / 42 / 1	85 / 42 / 2	+219 kB (4.5 %)	+448 B (0.003 %)	+18.5 s (28.18 %)	247 s	✗	✗	✗	✗	34.3 %	58.3 %	55.9 %	73.4 %
	FFmpeg	63 / 27 / 1	91 / 27 / 2	+190 kB (0.21 %)	+472 B (0 %)	+268 s (27.14 %)	643 s	✗	✓	✗	✗	33.1 %	34.9 %	57.8 %	44.1 %
	mutool	61 / 16 / 1	69 / 15 / 0	+189 kB (0.48 %)	+376 B (0.001 %)	+3.17 s (0.69 %)	164 s	✗	✓	✗	✗	52.0 %	38.3 %	69.4 %	60.3 %
	memcached	88 / 54 / 1	102 / 59 / 4	+216 kB (28.9 %)	+456 B (0.13 %)	+0.35 s (5.5 %)	8 s	✗	✓	✗	✗	30.3 %	33.7 %	50.0 %	41.9 %
DB	redis-server	85 / 35 / 1	93 / 42 / 3	+216 kB (11.2 %)	+472 B (0 %)	+2.4 s (2.7 %)	41 s	✗	✗	✗	✗	30.3 %	32.0 %	54.1 %	54.4 %
	sqlite3	92 / 72 / 1	102 / 72 / 13	+215 kB (5.9 %)	+456 B (0.02 %)	+0.8 s (7.2 %)	45 s	✗	✗	✗	✗	62.9 %	32.6 %	75.3 %	57.5 %
Server	Nginx	105 / 48 / 0	106 / 51 / 4	+217 kB (1.5 %)	+528 B (0.003 %)	+7.9 s (10.53 %)	277 s	✗	✓	✓	✗	32.0 %	30.9 %	38.8 %	40.0 %
	httpd	98 / 50 / 1	106 / 46 / 0	+218 kB (8.3 %)	+504 B (0.04 %)	+4.1 s (5 %)	16.8 s	✗	✗	✗	✗	29.7 %	30.3 %	50.9 %	44.4 %

Table 2: We evaluate the size overhead of Sourcalyzer on shared libraries compared to a vanilla version.

Shared library	Vanilla	Annotated	Overhead
musl libc.so	815 kB	1007 kB	23.63 %
libssl.so	657 kB	1.7 MB	161 %
libcrypto.so	4.1 MB	23 MB	460 %

6.2.4 Runtime Overhead and Seccomp. For the static approaches, the only overhead compared to manually crafted seccomp filters is the parsing of the syscall numbers. As this is done during application startup, it is a one-time overhead that depends on the number of rules that need to be set up. We investigate the overhead for setting up the application with the smallest (*true*) and largest (*Nginx*) number of syscalls based on Sourcalyzer. For Nginx, the setup time takes on average 9.92 ms ($\sigma_{\bar{x}} = 0.007$, $N = 10\,000$) while it only takes 0.58 ms ($\sigma_{\bar{x}} = 0.004$, $N = 10\,000$) for *true*. The remaining slowdown is then introduced by seccomp itself, which is unavoidable if a developer decides to use it for syscall filtering.

6.2.5 Dynamic Refinement Overhead. As a microbenchmark, we analyze the impact of Finalyzer on the syscall latency. We first benchmark the latency of the *getppid* syscall without Finalyzer in place 1 million times. The latency of *getppid* on our test system (Ubuntu 18.04.4, kernel 5.0.21-050021-generic) is 1358 ($\sigma_{\bar{x}} = 0.91$, $N = 1\,000\,000$) cycles. With Finalyzer, we observe an average latency of 17 103 ($\sigma_{\bar{x}} = 5.52$, $N = 1\,000\,000$) cycles, an increase of approximately 1160 %. While this increase seems large, it is intended as an optional step during development. Hence, we consider this less of a problem without impact on the released application.

6.3 Functional-Correctness Evaluation

Binaries sandboxed with Chestnut must be guaranteed to still work as intended. Related work [21] tested each application 100 times using various workloads. For a fair comparison, we perform the same tests. For applications where a test suite is available, we execute them to reach higher coverage, ensuring that we do not miss edge cases. Beyond previous work [11, 21], we evaluate code coverage to show that large parts of the application are executed. Finally, we perform a 6-month long test of Nginx sandboxed by Sourcalyzer.

In more detail, we first apply Chestnut to the binaries, cf. Table 1. Obtaining a sound ground truth of whether all syscalls are detected is infeasible and would require time-consuming formal proofs that are out-of-scope for this paper. Hence, we rely on executing the available test suites that should cover many of the different code paths available in the tested application. This is, for instance, possible for FFmpeg, memcached, redis, Nginx, and sqlite3. In other cases, we execute the binaries with different configurations to reach as many different code paths as possible [21]. We observed no crashes in applications sandboxed with Chestnut. Even if a syscall is missed in $\mathcal{P}1$, $\mathcal{P}2$ can be used to add it, ensuring correct functionality.

While this is not an exhaustive test, it can be assumed that test suites for large applications are designed for complete functionality coverage and thorough testing of critical components in particular. Based on the latter, it is a reasonable assumption that our test tests whether all syscalls in the core functionality of the application are found. To further substantiate this, we perform line and function coverage tests for a selection of applications, cf. Table 1. We perform these tests for FFmpeg (Lines: 59.3 %, Functions: 61.7 %), memcached (77 %, 91.9 %), and redis (77 %, 61.5 %). Additionally, the sqlite developers always maintain 100 % branch and 100 % MC/DC coverage [56]. While not perfect, the results indicate that large parts of the respective applications are executed and, to a certain degree, demonstrate that Chestnut does not impede them. In future work, we want to employ coverage-guided fuzzing to better estimate whether all required syscalls are found.

Programs using `fork+exec`, e.g., *git-diff*, exhibit the inherent problem of seccomp, namely that a child program inherits its parent's filters. If the child uses a syscall blocked by the parent, the child crashes. For such applications, $\mathcal{P}2$ is necessary to ensure functionality. Out of the 18 tested applications, $\mathcal{P}2$ was only necessary for two of them, namely *git-diff* and *git-log* as they performed syscalls blocked by their parent. After refining the filters using Finalyzer, both successfully completed their task.

Adding Missed Syscalls using $\mathcal{P}2$. We evaluate how many syscalls the static approaches miss. For Sourcalyzer, Finalyzer adds 4 syscalls to musl libc, which then propagate to applications if the corresponding function is used, e.g., *clone*. Table 1 shows how many syscalls are added in $\mathcal{P}2$. For Binalyzer and busybox, $\mathcal{P}2$ only adds a syscall in *ls*. Sqlite3 misses the most as Finalyzer needs to add 13 syscalls. These missed syscalls are only a limitation of our proof-of-concept implementation, occurring in edge cases that can be handled in a full implementation. Hence, Chestnut also works without Finalyzer.

Long-Term Study using Nginx. To demonstrate the functional correctness of Chestnut, we performed a long-term study of 6 months using Nginx. In this test, we compiled a static version of Nginx using Sourcalyzer (105 allowed syscalls), which we then deployed to a real-world server to host a website. Within 6 months, the server handled ≈ 100 000 requests without ever triggering a seccomp violation. This shows that Sourcalyzer can infer all syscalls necessary for a successful operation on a real-world system.

6.4 Security Evaluation

To evaluate how Chestnut increases the security of sandboxed applications, we analyze how often dangerous syscalls, e.g., `exec`, are blocked (Section 6.4.1), the number of syscalls not blocked even though they are not used by the application (Section 6.4.2), the number of mitigated real-world exploits (Section 6.4.3), and how malicious SGX enclaves can be blocked (Section 6.4.4).

6.4.1 Blocking Dangerous Syscalls. Three of the more dangerous syscalls that Linux provides are the two syscalls in the `exec` group, *i.e.*, `execve` and `execveat`, and the `mprotect` syscall. With the `exec` syscalls available, an attacker can execute an arbitrary binary in the presence of an exploitable memory safety violation [6]. In fact, most libc versions even contain a ROP gadget that leverages the `exec` syscall to open a shell [30]. Hence, an attacker can execute a new program in the context of the current one. With `mprotect`, an attacker can modify the permissions of existing memory, *i.e.*, make it executable. While `mmap` can be used to map memory as executable, we did not consider it in our evaluation. We consider attacks not relying on syscalls [8] as out of scope.

Even with Chestnut, certain attacks are still possible, e.g., adding an ssh key if a privileged application is hijacked and the *open*/*write* syscalls are allowed. These attacks are also possible with Chestnut, but other attacks are blocked, improving the overall system security. Hence, Chestnut still improves the status quo.

Compiler. We evaluate how often Sourcalyzer can block `exec` and `mprotect` (Table 1). In busybox, we block the `exec` syscalls in 9 out of 10 cases and `mprotect` in all 10. Additionally, we also evaluated all the remaining busybox utilities and blocked `exec` in 313 out of 396 (79.0 %) of them and `mprotect` in all 396 (100 %). In Nginx, we cannot block `exec`, but we block `mprotect`. In the other applications, we can block neither of them as our compiler detects a potential call to a function that contains the respective syscalls.

Binary. Binalyzer blocks the `exec` syscalls in all busybox utilities, cf. Table 1, where Sourcalyzer could not block the `exec` syscall in the *env* utility. We manually verified that the syscalls are indeed not required. The `mprotect` analysis showed the opposite behavior as it is not blocked in any of the applications. For Nginx, memcached, mutool, and FFmpeg, we also block the `exec` syscalls without crashing the application, but not `mprotect`. We could not block either one of them in git, httpd, redis, and sqlite3. For git and the `exec` syscalls, the reason is that some of git commands rely on other applications, *i.e.*, the configured pager for commands like *diff* or *log*. The explanation of why we cannot block `mprotect` using Binalyzer is the point of time at which we start blocking syscalls. In Sourcalyzer, we block syscalls that are reachable only from the *main* and *exit* functions, while we block them from the start of the application in Binalyzer. Hence, we need to allow `mprotect` as it is required for setting up the application. In a full implementation, functions necessary for program startup can be removed from the analysis, e.g., the *mprotect* syscall.

6.4.2 Overapproximation of Syscalls. Chestnut can drastically reduce the number of syscalls available to an application (cf. Table 1). For our 18 tested applications, Nginx and httpd block the least number of syscalls with 106 being allowed. However, without Chestnut, 349 syscalls in Linux 5.0 would be available [36]. While Chestnut drastically reduces the attack surface, both Sourcalyzer and Binalyzer often allow more syscalls than necessary. In this section, we estimate how tight automatically generated syscall filters are that an automated approach can generate. We are the first to demonstrate this for an automated seccomp-filter generation tool. We apply this analysis also to related work [11] to compare the different approaches (cf. Section 6.5).

Setup. To evaluate our static components' overapproximation, we leverage the functionality of Finalyzer in libchestnut. This has the advantage over *strace* that we do not include syscalls that are needed for setting up the application, *i.e.*, we only log syscalls after the main entry point. We then either execute the applications test suite or execute the program with different arguments to trigger different code paths, *i.e.*, try to trigger as many of the existing syscalls as possible. The accuracy of our results depends on the code coverage of the respective test suites. As was the case in Section 6.3, we argue that despite this not being an exhaustive test, test suites typically cover at least the core functionality of the application and its critical components. We substantiate this claim with the code coverage metrics discussed in Section 6.3 The results show that large parts of the respective applications are executed, demonstrating that this is an adequate but not perfect approach to detect overapproximation. This depicts a first step to the measurement of the tightness of automatically generated filters.

Using the aforementioned approach, we obtain a list of syscalls that the evaluated program issued. We calculate that list's intersection with the allowed syscalls as detected by Sourcalyzer or Binalyzer. This gives us a list of syscalls that our approach allows but that are never executed by the application in our tests. If a syscall was triggered that our static approaches block, Finalyzer automatically refines the application's filter list.

Compiler. As Table 1 shows, overapproximation varies between different applications when using Sourcalyzer. In the busybox utilities, we observed the largest overapproximation for *env*, where only 20 % of the detected syscalls are actually used. For the larger applications, we observe the largest overapproximation in mutool, with only 26.23 % being used. Note that the results cannot be compared to Binalyzer due to different libc versions being used, *i.e.*, musl libc for applications compiled with Sourcalyzer and glibc in Binalyzer.

Binary. For the evaluation of Binalyzer, we slightly deviate from the outlined setup just to ease the evaluation. Instead of using libchestnut, we use the standalone implementation of Finalyzer. Hence, we observe a larger amount of syscalls as we also record syscalls executed during program startup, similar to *strace*.

In busybox, we overapproximate the most in the *true* utility, where only 37.5 % are being used. In the larger applications, we observe the lowest percentage of actually used syscalls in mutool, with only 21.74 % being used. Future work could extend the functional-correctness evaluation, estimating overapproximation using coverage-guided fuzzing.

6.4.3 Mitigating Real-World Exploits. For evaluating the effectiveness of Chestnut in mitigating real-world exploits, we assume an attacker that can either inject shellcode or mount a ROP attack [59] in one of our target applications. We define an exploit as successful if the attacker can exploit a kernel bug from the application context. These bugs either trigger a privilege escalation or result in a denial of service. As seccomp filters restrict the available syscalls, they reduce the attack surface of the kernel.

For the evaluation, we extract a list of 175 CVEs from the CVE database [61] that exploit syscalls on the x86_64 Linux kernel. From this list, we extract the necessary syscalls, resulting in a list of 231 malicious samples. The reason is that a CVE can be triggered by different syscalls that are independent of each other. As some syscalls have equivalent versions, we extend our list of samples to 320 by substituting the syscall numbers where applicable. We provide a list of these equivalent syscalls in Appendix A. As we want to show that Chestnut-sandboxed applications impede the exploitation of unpatched kernel vulnerabilities, we assume a kernel that is vulnerable to all these CVEs.

To determine the effectiveness of Chestnut, we cross-reference the syscall numbers from each sample with the ones we block in Table 1. If one of the syscalls required for the exploit is blocked, we determine that this application cannot trigger the exploit in the kernel, indicating that Chestnut increased the security of the system. We consider both the number of CVEs that we fully mitigate and the number of subvariants mitigated by Chestnut.

Compiler. With Sourcalyzer, we can fully mitigate 84.04 % of the CVEs and 89.42 % of the subvariants in the case of busybox. The reason for that is that the busybox utilities are rather small, allowing only a few syscalls. Even with larger applications, our compiler still increases the system's security, fully mitigating 38.08 % of CVEs and 56.53 % of the subvariants.

Binary. In busybox, Binalyzer mitigates 81.8 % of the CVEs fully and 87.9 % of the subvariants. In the larger binaries, Binalyzer can fully mitigate 36.38 % of the CVEs and 52 % of the subvariants.

6.4.4 Blocking Malicious SGX Enclaves. Intel SGX enclaves cannot directly execute any syscalls, but only use functionality provided by the host application. The host application can use syscalls to provide this functionality to the enclave. Schwarz et al. [52] presented a technique to execute arbitrary syscalls from an SGX enclave via a ROP attack on the host application. This allows malicious or hijacked enclaves to mount attacks on the kernel.

Weiser et al. [65] presented SGXJail as a generic defense for malicious enclaves, blocking them from executing arbitrary syscalls. Binalyzer achieves a similar goal without affecting the performance of required syscalls. For the evaluation, we used the public proof-of-concept exploit provided by Schwarz et al. [52]. The Intel SGX SDK currently does not support LLVM; hence, we can only evaluate Binalyzer. As enclaves cannot contain syscalls, Binalyzer only has to scan the host application and allow only syscalls legitimately used by the host application. Out of the 349 syscalls provided by Linux 5.0, 279 (79.9 %) are blocked, including exec. We verified that the benign functionality of the host and enclave is not impacted. As a result, the malicious (or hijacked) enclave cannot run arbitrary programs anymore, and the attack surface is drastically reduced.

6.5 Comparison to Other Approaches

Two recent approaches on automating seccomp filter generation [11, 21] were published after the start of our 6-month long-term case study of Chestnut on Nginx. In this section, we compare our work to these two approaches and discuss the differences.

Temporal Syscall Specialization. Ghavamnia et al. [21] propose an automated approach to detect the used syscalls during compilation. Their approach is limited to applications that can be split into an initialization and serving phase, *i.e.*, server applications. The idea is to detect syscalls used after the server's initialization phase, *i.e.*, the point in time where it starts handling requests. Thus, this approach is not directly applicable to applications that cannot be easily split into these two phases, potentially enabling attacks through browsers, malicious PDFs [16, 17], messengers [24, 54], and office applications [42]. We explicitly consider such applications in our approach as our threat model is broader and includes local attackers additionally to remote ones. Similar to Chestnut, they also extract a sufficiently precise call graph to be able to extract which syscalls are reachable by the application. Their approach relies on Andersen's points-to analysis, which is known to not scale with program size [2, 25]. We evaluated an orthogonal *has address taken* approach as is used by LLVM's CFI implementation. As this is already used for the CFI implementation of LLVM, we know that the resulting CFG is reasonably precise as otherwise applications that rely on software-based CFI would not work. Our approach achieves similar results in terms of detected syscalls as the more complex and slower approach used by Ghavamnia et al. [21]. In contrast to our approach, they require a multitude of tools for the compilation and link-time optimization that is not supported by every application. As neither Andersen's points-to nor our address taken approach can guarantee a complete CFG, we rely on the more practical address-taken algorithm. This choice significantly reduces the compile time. For instance, syscall extraction for Nginx using Andersen's algorithm shows an increase in compilation time from 1 min to 83 min (+8300 %) [21] compared to an increase of 7.9 s (+10.53 %) with Chestnut.

In summary, we improved the approach's performance significantly while maintaining accuracy and security. Additionally, our approach is applicable to a broader range of applications, including local applications that are commonly hijacked. We also provide an evaluation of the tightness of the resulting filters.

Sysfilter. A second approach, *sysfilter* [11] focuses on extracting syscalls from existing binaries. While sysfilter and Binalyzer share the same goal, the approaches differ in the used tools, *i.e.*, Binalyzer relies on the angr framework that already supports parts of what sysfilter manually implemented. Both approaches show similar success rates in mitigating exploits in their respective test sets.

Sysfilter provides no analysis of the approach's overapproximation, making it hard to estimate how tight the resulting syscall filters are. Hence, we perform such an analysis to show differences between the approaches. As we discussed in Sections 6.3 and 6.4.2, obtaining a ground truth is infeasible and would require computational intensive formal proofs. Hence, we need another source for a reliable baseline to which we can compare the results of the evaluation for Binalyzer and sysfilter.

To provide this baseline, we rely on the results of Sourcalyzer when generating a static binary, for two reasons: First, the compiler has the most information about the application as it needs to generate a functioning binary, *i.e.*, it needs to know which functions are actually required and called. The second reason is based on what a compiler like clang does when it generates the static binary that we use. When generating this binary, the compiler already removes all unnecessary functions, *i.e.*, functions that are never called and never have their address taken, from the binary. So the resulting binary only contains functions and their respective syscalls if the compiler determined a potential path to the respective function. Therefore, any syscall found by the two binary tools within the static binary can be reached and is necessary for the application to work correctly. This number may differ from the one detected by Sourcalyzer due to the inherent overapproximation of the function signature heuristic, *i.e.*, read and write have the same function signature, so if one is used, the other one is automatically included in the set. In this case, the syscall of a function is included even though the compiler removed the function's actual code. Nevertheless, we expect the numbers to be in a similar range.

In this evaluation, both sysfilter and Binalyzer work on the exact same static binaries. We ensured that the binary still contains the stack unwinding information (*.eh_frame*) and other necessary sections (*.init, .fini*) on which sysfilter relies for its precise disassembly. While sysfilter notes that one requirement is a PIC binary, we note that the additional tasks that sysfilter performs for PIC binaries, *i.e.*, relocations or checking the dynamic symbol table, are by the design of static binaries simply not necessary. Building the call graph does not depend on these steps either. In fact, for binary analysis tools like sysfilter and Binalyzer, a static binary can be

Table 3: The number of extracted syscalls by Sourcalyzer, Binalyzer, and the two modes of sysfilter.

Binary	Sourcalyzer	Binalyzer	sysfilter (vacuumed-fcg)	sysfilter (universal-fcg)
FFmpeg	63	53	18	53
busybox	163	144	15	152
Redis-server	85	74	12	74

considered the most straightforward use case as all information is already contained within the static binary.

We consider two different modes of sysfilter, *i.e.*, the default behavior that prunes the call graph based on a reachability analysis and the universal approach that assumes that every function is reachable by every other function. As the binary is compiled statically, we expect that both modes produce the same result as only functions that are reachable from the main entry point are included. We show the result of this analysis in Table 3.

As our analysis shows, the assumption that both modes of sysfilter produce the same result does not hold as the pruning-based mode significantly underapproximates in all three evaluated binaries. The low number of detected syscalls hints at some mistake in the pruning algorithm as the number is too low for such complex applications. In two out of three binaries, Binalyzer and sysfilter using the universal approach produce the exact same result while the third binary only shows a small difference of 8 syscalls. In this case, the difference to Sourcalyzer is within an expected range due to the overapproximation of Sourcalyzer. This is not true for the pruning-based approach of sysfilter as the difference is too large, and the number of detected syscalls is lower than the number of syscalls that are actually used (cf. Table 1). Interestingly, the universal-fcg implementation of sysfilter also supports our observation that a PIC binary is not a requirement for these types of binary analysis tools as it produces similar results to Binalyzer, contradicting the statement by its developers. Nevertheless, there is still a difference in the operation between the universal-fcg approach of sysfilter and Binalyzer as the latter achieves this result by not assuming that every function is reachable by every other function. Instead, it still builds a correct call graph and derives the information from it, which fails for the vacuum-fcg approach of sysfilter.

We investigated the low number of syscalls found in the pruning-based approach of sysfilter. This analysis showed that during the pruning, the main function is removed from the set of reachable functions, which results in the whole application being removed from the analysis. We leave the analysis of whether this is purely an implementation bug or a hint that this is a general problem in the approach for future work as this is out of scope for this paper.

7 DISCUSSION

Limitations and Future Work. Fast and reliable points-to analysis with limited overapproximation is still an unsolved problem [25]. In some cases, we also exhibit the opposite effect in *angr* that it is not able to detect the call target of an indirect call, hence missing a potentially reachable syscall. In contrast to previous work, we evaluate this problem by measuring the code coverage on real-world code examples. Future work may extend our analysis with coverage-guided fuzzing to obtain more precise estimates for the overapproximation of automated seccomp filter generation tools.

Future work may investigate the possibility of extending the syscall filtering with argument tracking. While detecting constant syscall arguments is possible, the precise propagation of this information throughout the call graph is not trivial. Solving this problem would allow restricting syscalls further, e.g., only allow certain hardcoded paths for `exec` or limit possible permissions passed to `mprotect`, e.g., no executable permissions. Finally, one limitation is the performance of seccomp [26, 62] imposed by the underlying system. Since this is not a weakness of Chestnut itself, we consider improving the performance of seccomp out of scope for this paper.

Related Work. Several related works also discuss the problem of automating sandboxing mechanisms, e.g., reducing the attack surface of applications by removing unused code. One of the first approaches for library debloating is based on removing non-imported functions from a shared library during load time [43]. This approach has been further improved by removing all unused functions from shared libraries during load time by extending the compiler and the loader [47]. Agadakos et al. [1] proposed a binary-level approach for library debloating, based on function boundary detection and dependency identification to identify and erase unused functions. Davidson et al. [10] analyzed the entire software stack for web applications to create specialized libraries based on the requirements for PHP code and the server binaries. *Shredder* [40] instruments binaries to restrict arguments to system APIs to a predefined allowlist. Another approach is to apply data dependency analysis for fine-grained customization of static libraries [55].

More closely related to our work is the approach by Ghavamnia et al. [21]. However, their approach suffers from a significantly higher execution time for the analysis during compilation while achieving a comparable accuracy in detecting syscalls. Wagner and Dean [63] propose a static approach to build an IDS that uses a similar approach to Sourcalyzer for pointer analysis to extract a model of expected application behavior. In general, several papers have proposed static analysis of syscalls for anomaly detection and IDS [18]. Rajagopalan et al. [49] propose to replace syscalls with authenticated syscalls that specify a policy and provide a cryptographic MAC that guarantees the integrity of the syscall. sysfilter [11] uses the Egalito framework to statically extract the syscalls from the binary similar to Binalyzer, but has a strong requirement on PIC binaries which Binalyzer does not.

Other approaches reduce the attack surface using training to identify the unused code sections, e.g., Ghaffarinia and Hamlen [20]. Without access to the source code, training and heuristics can be used to identify and remove unnecessary basic blocks [46].

Previous work focused mostly on C/C++ software with few solutions for software in other languages. For Java, one approach uses static code analysis to remove unused classes and methods [29]. For PHP, Azad et al. [3] proposed a framework using dynamic analysis to remove superfluous features.

8 CONCLUSION

Chestnut is an automated approach to block unused syscalls in applications, identified using static analysis and an optional dynamic refinement. The compiler-based approach is up to factor 73 faster than previous work without any loss in accuracy. On the binary level, our approach extends over previous ones by also applying to non-PIC binaries and thus a broader set of applications. Chestnut increases platform security without manual effort as shown in our evaluation of correctness and overapproximation, using test suites, code coverage, and a 6-month long-term evaluation. Chestnut blocks more than 82.5 % of all syscalls and 61 % corresponding kernel CVEs from these applications.

ACKNOWLEDGMENTS

This work has been supported by the Austrian Research Promotion Agency (FFG) via the project ESPRESSO, which is funded by the province of Styria and the Business Promotion Agencies of Styria and Carinthia. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No 681402). Additional funding was provided by generous gifts from ARM, Intel, and Red Hat. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding parties.

REFERENCES

[1] Ioannis Agadakos, Di Jin, David Williams-King, Vasileios P Kemerlis, and Georgios Portokalidis. 2019. Nibbler: debloating binary shared libraries. In *ACSAC*.
[2] Lars Ole Andersen. 1994. *Program Analysis and Specialization for the C Programming Language.* Ph.D. Dissertation.
[3] Babak Amin Azad, Pierre Laperdrix, and Nick Nikiforakis. 2019. Less is More: Quantifying the Security Benefits of Debloating Web Applications. In *USENIX Security Symposium.*
[4] Tyler K. Bletsch, Xuxian Jiang, Vincent W. Freeh, and Zhenkai Liang. 2011. Jump-oriented programming: a new class of code-reuse attack. In *AsiaCCS.*
[5] Erik Bosman and Herbert Bos. 2014. Framing Signals - A Return to Portable Shellcode. In *S&P.*
[6] Nicholas Carlini and David A. Wagner. 2014. ROP is Still Dangerous: Breaking Modern Defenses. In *USENIX Security Symposium.*
[7] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. 2010. Return-oriented programming without returns. In *CCS.*
[8] Shuo Chen, Jun Xu, Emre Can Sezer, Prachi Gauriar, and Ravishankar K Iyer. 2005. Non-Control-Data Attacks Are Realistic Threats.. In *USENIX Security Symposium.*
[9] Chromium. [n.d.]. Linux Sandboxing. https://chromium.googlesource.com/chromium/src/+/0e94f26e8/docs/linux_sandboxing.md
[10] Nicolai Davidsson, Andre Pawlowski, and Thorsten Holz. 2019. Towards automated application-specific software stacks. In *ESORICS.*
[11] Nicholas DeMarinis, Kent Williams-King, Di Jin, Rodrigo Fonseca, and Vasileios P. Kemerlis. 2020. sysfilter: Automated System Call Filtering for Commodity Software. In *RAID.*
[12] David Drysdale. 2014. Anatomy of a system call, part 2. https://lwn.net/Articles/604515/
[13] David Drysdale. 2015. How programs get run: ELF binaries. https://lwn.net/Articles/631631/
[14] Jake Edge. 2012. A library for seccomp filters. https://lwn.net/Articles/494252/
[15] Jake Edge. 2015. A seccomp overview. https://lwn.net/Articles/656307/
[16] Jose Miguel Esparza. 2012. Static analysis of a CVE-2011-2462 PDF exploit.
[17] Jose Miguel Esparza. 2014. Quick analysis of the CVE-2013-2729 obfuscated exploits.
[18] 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 *S&P.*
[19] Firejail 2018. Firejail Security Sandbox. https://firejail.wordpress.com/
[20] Masoud Ghaffarinia and Kevin W Hamlen. 2019. Binary Control-Flow Trimming. In *CCS.*
[21] Seyedhamed Ghavamnia, Tapti Palit, Shachee Mishra, and Michalis Polychronakis. 2020. Temporal System Call Specialization for Attack Surface Reduction. In *USENIX Security Symposium.*
[22] Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. 2014. Out of Control: Overcoming Control-Flow Integrity. In *S&P.*
[23] Ian Goldberg, David Wagner, Randi Thomas, Eric A Brewer, et al. 1996. A secure environment for untrusted helper applications: Confining the wily hacker. In

USENIX Security Symposium.
[24] Samuel Groß. 2020. Remote iPhone Exploitation Part 1: Poking Memory via iMessage and CVE-2019-8641.
[25] Michael Hind. 2001. Pointer analysis: Haven't we solved this problem yet?. In *PASTE.*
[26] Tom Hromatka. 2018. seccomp and libseccomp performance improvements.
[27] Google Inc. 2017. Seccomp filter in Android O. https://android-developers.googleblog.com/2017/07/seccomp-filter-in-android-o.html
[28] Google Inc. 2019. Sandbox2. https://developers.google.com/sandboxed-api/docs/sandbox2/overview
[29] Yufei Jiang, Dinghao Wu, and Peng Liu. 2016. JRed: Program Customization and Bloatware Mitigation Based on Static Analysis. In *COMPSAC.*
[30] Mateusz Jurczyk and Gynvael Coldwind. 2015. Permissions overview. *Insomni'hack* (2015).
[31] Vasileios Kemerlis. 2015. *Protecting Commodity Operating Systems through Strong Kernel Isolation.* Ph.D. Dissertation. Columbia University.
[32] Vasileios P Kemerlis, Michalis Polychronakis, and Angelos D Keromytis. 2014. ret2dir: Rethinking kernel isolation. In *USENIX Security Symposium.*
[33] Vasileios P. Kemerlis, Georgios Portokalidis, and Angelos D. Keromytis. 2012. kGuard: Lightweight Kernel Protection against Return-to-User Attacks. In *USENIX Security Symposium.*
[34] Chris Lattner and Vikram S. Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *IEEE / ACM International Symposium on Code Generation and Optimization – CGO.*
[35] Xin Lin, Lingguang Lei, Yuewu Wang, Jiwu Jing, Kun Sun, and Quan Zhou. 2018. A measurement study on Linux container security: Attacks and countermeasures. In *ACSAC.*
[36] Linux. 2019. 64-bit system call numbers and entry vectors. https://github.com/torvalds/linux/blob/master/arch/x86/entry/syscalls/syscall_64.tbl
[37] Kin-Keung Ma, Khoo Yit Phang, Jeffrey S Foster, and Michael Hicks. 2011. Directed symbolic execution. In *International Static Analysis Symposium.*
[38] W. S. McPhee. 1974. Operating system integrity in OS/VS2. *IBM Systems Journal* (1974).
[39] Matt Miller. 2019. Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape. *Bluehat IL* (2019).
[40] Shachee Mishra and Michalis Polychronakis. 2018. Shredder: Breaking exploits through API specialization. In *ACSAC.*
[41] Mozilla. 2016. Seccomp filter in Android O. https://wiki.mozilla.org/Security/Sandbox/Seccomp
[42] Jens Müller, Fabian Ising, Christian Mainka, Vladislav Mladenov, Sebastian Schinzel, and Jörg Schwenk. 2020. Office Document Security and Privacy. In *WOOT.*
[43] Collin Mulliner and Matthias Neugschwandtner. 2015. Breaking Payloads with Runtime Code Stripping and Image Freezing. *BlackHat USA* (2015).
[44] Shravan Narayan, Craig Disselkoen, Tal Garfinkel, Nathan Froyd, Eric Rahm, Sorin Lerner, Hovav Shacham, and Deian Stefan. 2020. Retrofitting Fine Grain Isolation in the Firefox Renderer. In *USENIX Security Symposium.*
[45] Vassilis Prevelakis and Diomidis Spinellis. 2001. Sandboxing Applications. In *USENIX ATC.*
[46] Chenxiong Qian, Hong Hu, Mansour Alharthi, Pak Ho Chung, Taesoo Kim, and Wenke Lee. 2019. RAZOR: A Framework for Post-deployment Software Debloating. In *USENIX Security Symposium.*
[47] Anh Quach, Aravind Prakash, and Lok Yan. 2018. Debloating Software through Piece-Wise Compilation and Loading. In *USENIX Security Symposium.*
[48] Nguyen Anh Quynh. 2014. Capstone: Next-gen disassembly framework. *Black Hat USA* (2014).
[49] Mohan Rajagopalan, Matti Hiltunen, Trevor Jim, and Richard Schlichting. 2005. Authenticated system calls. In *DSN.*
[50] Charles Reis, Alexander Moshchuk, and Nasko Oskov. 2019. Site Isolation: Process Separation for Web Sites within the Browser. In *USENIX Security Symposium.*
[51] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. 2015. Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications. In *S&P.*
[52] Michael Schwarz, Samuel Weiser, and Daniel Gruss. 2019. Practical Enclave Malware with Intel SGX. In *DIMVA.*
[53] Hovav Shacham. 2007. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *CCS.*
[54] Natalie Silvanovich. 2020. Exploiting Android Messengers with WebRTC: Part 1.
[55] Linhai Song and Xinyu Xing. 2018. Fine-grained library customization. In *SALAD.*
[56] SQLite. 2020. How SQLite Is Tested.
[57] Bjarne Steensgaard. 1996. Points-to Analysis in Almost Linear Time. In *POPL.*
[58] Nicolas Sylvain. 2008. A new approach to browser security: the Google Chrome Sandbox.
[59] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. SoK: Eternal War in Memory. In *S&P.*
[60] Robert Tarjan. 1972. Depth-first search and linear graph algorithms. *SIAM journal on computing* (1972).
[61] The MITRE Corporation. [n.d.]. Common Vulnerabilities and Exposures. http://cve.mitre.org/
[62] Tizen. 2018. Security:Seccomp. https://wiki.tizen.org/Security:Seccomp
[63] David Wagner and R Dean. 2000. Intrusion detection via static analysis. In *S&P.*
[64] Fish Wang and Yan Shoshitaishvili. 2017. Angr - The Next Generation of Binary Analysis. In *2017 IEEE Cybersecurity Development (SecDev).*
[65] Samuel Weiser, Luca Mayr, Michael Schwarz, and Daniel Gruss. 2019. SGXJail: Defeating Enclave Malware via Confinement. In *RAID.*
[66] Mozilla Wiki. 2019. Security/Sandbox. https://wiki.mozilla.org/Security/Sandbox

A LIST OF EQUIVALENT SYSCALLS

In this appendix, we provide a list of equivalent syscalls (cf. Table 4).

Table 4: Syscalls and their equivalents.

Syscall	Equivalents
munlockall	munlock
listxattr	llistxattr, flistxattr
epoll_create	epoll_create1
mlockall	mlock, mlock2
execve	execveat
recvfrom	recvmsg, recvmmsg
writev	pwritev
mknod	mknodat
open	openat
accept	accept4
getdents	getdents64
sendto	sendmmsg, sendmsg
getxattr	fgetxattr, lgetxattr
rename	renameat, rename2
epoll_ctl	epoll_ctl_old