Enter Sandbox

Claudio Canella, Mario Werner, Michael Schwarz

claudio.canella@iaik.tugraz.at, we.rner.at,
michael.schwarz@cispa.saarland

Abstract

The complexity of modern applications increases year by year, and with that also the number of vulnerabilities. State-of-the-art defenses and countermeasures can reduce the number of vulnerabilities, but the risk remains that an attacker can still exploit a vulnerability. For such cases, sandboxes can be deployed to limit the post-exploitation impact. The idea behind sandboxing is to reduce the number of resources available to an application to the bare minimum required for it to work. One such resource can be syscalls used by applications to request specific tasks to be performed by the operating system. Linux provides seccomp to restrict syscalls to the bare minimum necessary for the application to work, but this requires complicated and complex manual analysis by a developer to identify the syscalls the application requires. This significantly reduces the applicability of Linux seccomp.

In this talk, we present Chestnut, a two-phase approach that can automatically identify the syscalls an application requires. We first discuss the two components in phase one that perform static analysis either on the source-code level or on an already existing binary. As static analysis has inherent limitations, the second and optional phase of Chestnut performs dynamic analysis to identify syscalls that have either been missed by the static approaches or to refine the list of syscalls further. Finally, we discuss our results in terms of performance, functional correctness, and security. For the performance evaluation, we show that our approach only has an insignificant impact on applications' compilation time and a low extraction overhead in existing binaries. Then, we show that our automatically generated seccomp filters do not inhibit the applications' functional correctness, i.e., we observe no crashes. We extract the syscalls of 18 widely-used applications for our security analysis and determine whether we can block security-critical syscalls such as mprotect and the two exec syscalls. We then cross-reference the found syscalls with a list of known kernel vulnerabilities triggered by syscalls and determine whether our now sandboxed applications could be used to launch such an attack or not.

1 Overview

In this whitepaper, we cover the topics of our talk and also provide technical background. The whitepaper is a pre-print of our paper "Automating Seccomp Filter Generation for Linux Applications" [1]. It presents our talk's content in more detail, such as the challenges that need to be solved by such a system.

It also provides detailed information on how our implementation solves these challenges in our public proof-of-concept [2] and a more detailed evaluation. We provide insight into how an automated system can improve security with next to no overhead through this work. We also discuss how such systems can be further improved, which leads to a further increase in the system's security.

The main takeaways of both the talk and the whitepaper are as follows.

1. Sandboxing is an important mechanism to reduce the attack surface of applications.
2. Effective sandboxing is still a huge manual effort, and thus not applied to many applications.
3. Automated sandboxing is challenging but feasible, and it can help to drastically reduce the attack surface of an application while exhibiting only small overhead.

References

[1] Canella, C., Werner, M., Gruss, D., and Schwarz, M. Automating Seccomp Filter Generation for Linux Applications. *arXiv:2012.02554* (2020).

[2] Canella, C., Werner, M., and Schwarz, M. Chestnut, `https://github.com/chestnut-sandbox/Chestnut` 2020.

Automating Seccomp Filter Generation for Linux Applications

Claudio Canella*, Mario Werner*, Daniel Gruss*, Michael Schwarz‡,

*Graz University of Technology ‡CISPA Helmholtz Center for Information Security

***Abstract*—Software vulnerabilities in applications 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 automated approaches for detecting and installing the necessary filter rules. 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 for Linux userspace applications 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 extract the used syscalls during compilation or in an analysis of the binary. The compiler-based approach of Chestnut is up to factor 73 faster than previous approaches without affecting the accuracy adversely. On the binary analysis level, we demonstrate that the requirement of position-independent binaries of related work is not needed, enlarging the set of applications for which Chestnut is usable. In an optional second phase, Chestnut provides a dynamic refinement tool that allows restricting 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-level analysis on a set of 18 widely used applications. We found that 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 prevents exploitation of more than $62\,\%$ of the 175 CVEs that target the kernel via syscalls. Finally, we perform a 6 month long-term study of a sandboxed Nginx server.

I. Introduction

The complexity of applications is steadily growing [29], [58], and with that, also the number of vulnerabilities found in applications [49]. 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 [73]. While state-of-the-art defensive-programming techniques and countermeasures reduce the number of vulnerabilities, there is still a remaining risk that an attacker can exploit a vulnerability in an application. Especially for privileged applications such as `setuid` binaries, this can, in the worst case, mean that an attacker can take over 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 [43]. 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 [56], [26] and, ideally, untrusted and potentially malicious, or benign but compromised applications cannot escape the sandbox.

On Linux, seccomp [18] and the extended seccomp-bpf can be used by applications to restrict the syscall interface. Seccomp-bpf [18] supports filter rules via developer-defined Berkeley Packet Filters [47]. Each syscall can either be entirely blocked 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. As this is a considerable effort, seccomp is mainly used in applications that implement isolation mechanisms, e.g., sandboxes [32], [22]. Given its complexity, it is rarely used in other applications.

Recent works proposed two methods to automatically generate such seccomp filters [24], [14]. The first approach utilizes the compiler and various external tools to derive the filters during compilation [24]. 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 [14]. 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], [28]. 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) [14]. 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. [24] as it does not require a sophisticated points-to analysis to generate filter rules. Instead, a faster *has address taken* approach can be used that achieves the same accuracy but at a fraction of the performance impact on compilation time. Second, we demonstrate an alternative implementation to the one provided by DeMarinis et al. [14]. In this approach, we demonstrate that the requirement of a PIC binary is not necessary, significantly extending the set of applications to which it can be applied. We implement our method in a proof-of-concept tool, Chestnut.[1] We also advance the state of the art in evaluations of automatic syscall filtering, with a first long-term case study and coverage metrics to confirm our approach's validity.

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

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 [60] to disassemble applications and to locate syscalls. Using symbolic backward execution [46] from the `syscall` instruction, we infer the syscall number used in the identified syscall. Additionally, we use the control-flow graph recovery functionality of angr [79] 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.

To demonstrate our approach's feasibility, we evaluate 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 while it 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 prevent the `mprotect` syscall in 61.1 % of the tested applications using Sourcalyzer. Furthermore, we evaluate our approach with existing real-world exploits, showing that Chestnut prevents exploitation of around 64 % and 62 % of CVEs using Sourcalyzer and Binalyzer, respectively. We also compare our approaches with related previous work [24], [14]. We show that we can achieve similar results in terms of effectiveness, measured in the practically mitigated CVEs, to the compiler-based approach by Ghavamnia et al. [24] but improving the performance by up to factor 73. We are the first to show that binary-based approaches can also be applied to non-PIC binaries by demonstrating that Binalyzer runs successfully on non-PIC binaries. We evaluate the functional correctness of Sourcalyzer in functional tests as well as a 6-month long-term case study: During 6 months of use of a Sourcalyzer-protected Nginx production server, we did not observe a single crash. Furthermore, we are the first to evaluate how tight automatically generated filter rules actually are. We evaluate the functional correctness and the tightness of the filters by executing the available test suites. We substantiate the validity of these experiments by measuring the code coverage of the respective test suites.

Filters generated automatically with a tool might not always be as strict as theoretically possible. However, there is no time investment required from 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 novel compiler-based approach for automatic syscall-filter generation without manual interaction that is up to factor 73 faster than previous work.
2) We present a method to refine the number of allowed syscalls based on dynamic tracing.
3) We demonstrate that Chestnut prevents the exploitation of more than 63 % of the 175 CVEs in the Linux kernel exploitable via syscalls.
4) We show that requirements of previous approaches are not necessary, thus enabling a significantly faster approach that is also applicable to a wider range of applications.
5) We perform a 6 month long-term study using Nginx to demonstrate the functional correctness of our approach where we did not observe a single crash.

Outline. Section II provides background. In Section III, we discuss the threat model, challenges, and design of Chestnut. In Section IV, we detail our compiler-based approach and the extraction from existing binaries. Section V discusses our dynamic refinement approach. We evaluate Chestnut in Section VI. We discuss related work, limitations, and future work in Section VII. We conclude in Section VIII.

II. Background

A. Sandboxing

Sandboxing is a security mechanism that intends to constrain software within a tightly controlled environment by

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

restricting the available resources to a required minimum [56], [26]. 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 [72], [82], [54], [62], [63], [81].

B. 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 [37], [36], [38]. Hence, platform security profits from limiting the amount of syscalls that an application can perform. With Secure Computing (seccomp) [18], 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 exploited. As seccomp filters do not dereference pointers, so-called time-of-check time-of-use attacks [48] common in syscall interposition frameworks are not possible. Examples of applications that rely on seccomp are Chromium [11], Firefox [51], and the zygote process in Android systems [31].

C. 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 referred to as control-flow hijack attacks [73].

Nergal [55] and Shacham [67] describe ROP attacks, which allow an attacker to chain existing code gadgets within an application together to perform malicious tasks. Each such gadget is a sequence of instructions that end 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 [9], [4], [41], [8], [25], [64] or signal handlers [5].

D. Executable and Linkable Format

On Unix-based systems, the Executable and Linkable Format (ELF) [6], [16] is the standard file format for shared libraries and executable files. One advantage is that it is highly flexible and extensible. An ELF file consists of an ELF header that is followed by data. The data itself can consist of a program and a section header table describing segments and sections, respectively. Segments contain information that is necessary for the run-time execution of the ELF binary, while sections contain information relevant for linking and relocating.

Dynamic Linking. The dynamic linker is responsible for loading and linking shared libraries needed by an executable during runtime [16]. For that, the dynamic linker copies 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 during link time and is embedded into the ELF file.

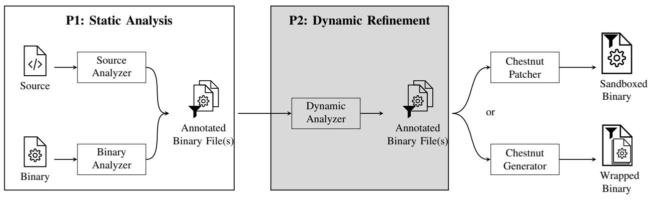


Fig. 1: The components of Chestnut and their interaction. In $\mathcal{P}\mathbf{1}$, source files can be analyzed statically with our LLVM-based analyzer, binaries with our binary analyzer. If necessary, dynamic analysis in the optional $\mathcal{P}\mathbf{2}$ refines the previous filters. After both phases, the binary can either be rewritten to block unused syscalls or a tailored sandbox can be generated that allows only the syscalls used in the binary.

III. Design of Chestnut

In this section, we introduce the threat model used throughout the paper. We outline a set of challenges required to solve automatic filter generation and discuss the high-level idea of Chestnut. Figure 1 illustrates the main components of Chestnut, e.g., the compiler modification *Sourcalyzer*, the binary analyzer *Binalyzer*, and the dynamic refinement tool *Finalyzer*. Furthermore, we discuss how all three components can be combined to further enhance the capabilities of Chestnut.

A. Threat Model and Idea of Chestnut

Chestnut supports Linux applications available as either C source code or as a binary. In the latter case, it is not limited to PIC binaries as previous work has discussed [14]. 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 [52], [33], and pdf readers [19], [20]. Another example where Chestnut can be used to restrict the syscall interface are messenger applications as several attacks have been shown to fully compromise a system [68], [27]. We assume correct usage of Chestnut in one of its variants (cf. Figure 1). Chestnut assumes that the application itself 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 a vulnerable application. We assume that the post-exploitation step targets the trusted system and requires syscalls, e.g., to gain kernel privileges. Chestnut prevents the application from using syscalls it usually would not use, averting harm from the rest of the system. Syscalls provided for file operations can potentially be exploited by an attacker to modify configuration files. Argument-level API specilization [50] can be used to protect against such attacks. In line with related work [24], Chestnut currently cannot protect against such attacks if the application requires these syscalls, and the file permissions are set incorrectly. Augmenting Chestnut with argument-level API specialization is left for future work. 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 [39], [83], [40], [45], [77], [65], [7] are out of scope.

B. Challenges

Automatic filter generation using a static approach requires solving the following four challenges:

$\mathcal{C}1$: Identifying Syscall Numbers for each Syscall. To automatically block unused syscalls, it is necessary to 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) [15]. Hence, the first challenge is identifying the individual syscall number that a specific syscall uses. Syscalls can appear in many different forms within a binary, 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. For example, an application calls *foo()* in *libfoo.so* that calls *open()* in *libc.so*, which finally calls *syscall()* in *libc.so* issuing the syscall. For our approach, we have to detect all syscalls, regardless of how they are called. Extracting the syscall number is the only architecture-dependent part of Chestnut.

$\mathcal{C}2$: Reconstruct Call Sites of Syscalls. By solving challenge $\mathcal{C}1$, we know which syscalls can be 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. This call graph contains the information obtained from $\mathcal{C}1$ for each function and the information about edges between them.

$\mathcal{C}3$: 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. We rely on seccomp to apply the syscall filters as seccomp is natively supported on Linux. To install the filters, we provide a library (libchestnut) that has to be added to the application. 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.

Once all these challenges have been solved, we automatically obtain an application that can only execute the required syscalls. We solve the challenges in detail in Sections IV and V.

C. High-Level Idea

This section briefly discusses the three components Chestnut provides in the two phases ($\mathcal{P}1$ and $\mathcal{P}2$) and how they solve the challenges. Sections IV and V provide more detail on the implementation of each component.

Sourcalyzer. Chestnut contains Sourcalyzer, a compiler-based component for static analysis of the application source code. Based on LLVM, Sourcalyzer is a compiler pass that extracts all syscalls identified during compilation.

This proves to be a practical approach for statically linked binaries. For such 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. Unfortunately, just extracting the numbers and installing a seccomp filter for all found syscalls is not enough, as this would lead to almost all syscalls being allowed. The reason is that the C standard library implements almost all syscalls. By linking against it, our generated filters would allow almost all syscalls, which renders the filters ineffective. Hence, we need to determine further which syscalls are actually used by the application by analyzing the control-flow graph to solve challenges $\mathcal{C}2$ and $\mathcal{C}3$. While comparable work [24] needs to perform the same task, we demonstrate a solution that is up to factor 73 faster. We discuss this in Section IV-A.

Binalyzer. The compiler-based approach's limitation is that it requires the source code of the application and all used libraries. Binalyzer has the same goal as Sourcalyzer but works directly on the binary level. With this, our approach is also applicable to programs where the source code is not available or where the source code is not compatible with LLVM, retrofitting the approach to binaries. In contrast to previous work [14], Binalyzer is also not restricted to PIC binaries.

The idea of this binary-level analyzer is to scan binaries and libraries for *syscall* instructions and then use symbolic backward execution [46] from these locations to infer the respective syscall number, again solving challenge $\mathcal{C}1$. Similar to the compiler-based approach, the basic binary-level approach also suffers from overapproximation. To reduce overapproximation, Binalyzer leverages control-flow-graph analysis of all dependencies to map exported functions to the identified syscall numbers ($\mathcal{C}2$). Finally, based on the required symbols of the binary and the libraries and the syscall-to-function mapping, Binalyzer infers a set of syscalls reachable by the application ($\mathcal{C}3$). Note that Binalyzer can also be applied to stripped binaries as all the required information is still included for dynamic linking, *i.e.*, the list of exported functions to which we add syscalls. We detail this in Section IV-B.

Finalyzer. To work around the limitations of static analysis, we propose Finalyzer, an approach based on dynamic syscall tracing. Finalyzer is solely intended to refine filters identified by our static approaches in a developer-controlled, benign environment. In this optional phase, Finalyzer removes or adds additional filters that cannot be identified statically.

The dynamic nature of Finalyzer allows us to simplify challenges $\mathcal{C}1$ to $\mathcal{C}4$ by inspecting syscalls just-in-time. 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 ($\mathcal{C}2$). These dynamically collected syscalls can then be checked against the statically identified syscalls. In this step, missed syscalls can be added to refine the installed filter list ($\mathcal{C}4$). We discuss this process in more detail in Section V.

Combining Components. Chestnut is designed in a way that allows combining all three components, as shown in 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 parents' syscall filters, which cannot be relaxed anymore. By combining the static approaches with Finalyzer, the syscalls of the child process can be 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 file containing the syscalls the application can call. For Sourcalyzer, the syscall filters can be 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.

IV. Static Filter Extraction

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

A. Compiler-Based Approach

Sourcalyzer utilizes the LLVM compiler framework [42] 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

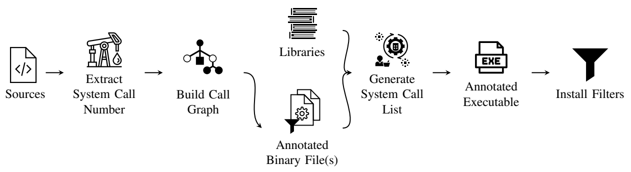


Fig. 2: The different steps of Sourcalyzer, which starts with the source and ends with a fully sandboxed application.

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, *i.e.*, *read* invokes the *syscall()* function with the *SYS_read* syscall number. musl additionally provides the function *__syscall_cp()* that adds a cancellation point to a syscall. 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. Note that, 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 in this regard is extracting a reasonably precise call graph without strongly degrading usability due to 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 [71]). This type of analysis requires access to the whole program and often does not scale efficiently to larger program sizes [2], [28]. 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. [24]. 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 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. Note that, 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 VI-D2).

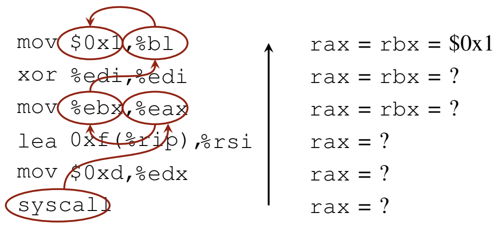
Both the LLVM IR passes and the linker are involved in mapping syscall numbers to functions. Our analysis pass traverses over all defined functions within the module. It extracts their function signature, functions that are directly called, the function signatures for indirect call sites, and the functions that are referenced by the code (for which function pointers exist, *i.e.*, functions that have their address taken). The latter is similar to what LLVM uses in its implementation of 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.

Note that we perform our analysis in the same traversal where we also locate the used syscall numbers (see $\mathcal{C}1$), meaning that a single pass over the IR is sufficient. As function aliases are widely used in musl, we also support them by treating them like copies of the original function. Finally, references to functions in global initializers are extracted. In musl, e.g., global file structures are used on which functions for reading, writing, and seeking are registered.

Our IR transformation pass serializes the information collected from the analysis pass into a note section of the emitted ELF object for the linker to use this information. To simplify inspection of the extracted data, we use human-readable JSON as encoding format. For a production-ready compiler, binary encoding is preferable to reduce performance overheads.

$\mathcal{C}3$: 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 [74], 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 can be 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 in the program, all reachable syscall numbers are known.



```
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 = ?
```

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

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

$\mathcal{C}4$: Install Seccomp Filters. After linking with the Sourcalyzer toolchain, the binary contains annotations containing 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 [17] 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.

B. *Binary Syscall Extraction*

The second static approach of Chestnut, Binalyzer, works on the binary level. While there is 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 [14], we demonstrate that the requirement of a PIC binary is not necessary.

$\mathcal{C}1$: Identify Syscall Numbers. The syscall number specifying the type of syscall is not encoded in the syscall instruction itself. Instead, the syscall number is provided by a general-purpose 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. This reconstruction results in a list of all syscalls and their virtual addresses.

Binalyzer uses the capstone framework [60] to disassemble a binary as this framework supports various ISAs, e.g.,

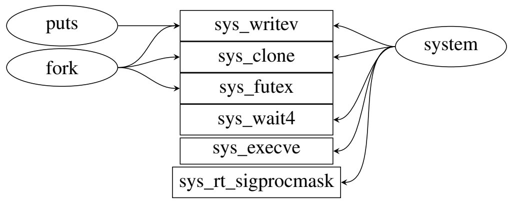


Fig. 4: Binalyzer creates an $n : m$ mapping between exported functions (ellipse) and syscalls (rectangle).

x86_64 and AArch64. The disassembly is used as the base for identifying syscall numbers. Starting from a syscall instruction, Binalyzer leverages symbolic backward execution [46]. 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 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. Such a situation would require a more complex symbolic execution. 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 used syscalls, Binalyzer analyzes the binary's control-flow graph (CFG) to map identified syscalls to (exported) functions. After this analysis, Binalyzer has a set of all possible syscalls per exported function (cf. Figure 4).

We rely on angr [79] 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. Such a function might not be an exported function or even a named function, but any block identified as 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 (cf. Figure 4).

$\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 dynamically-linked 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. As a tradeoff, 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. Note that shared libraries can also have a dynamic symbol table if they call functions from other libraries. Thus, 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 would have to resort to $\mathcal{P}2$, as the complete set of syscalls cannot be determined statically.

$\mathcal{C}$4: Install Seccomp Filters. From the complete 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 two different variants: binary rewriting, and building a sandbox wrapper (cf. Figure 1). ChestnutGenerator is a simple application that sets up the filter rules and starts the target application.

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. In the constructor function, the library 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 application.

V. Dynamic Refinement

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

A. Limitations of Static Approaches

While our approach for statically detecting an application's syscalls works well for most binaries (cf. Section VI), there are inherent limitations to a static approach. Dynamically loaded libraries, e.g., codecs, plugins, self-modifying, or just-in-time-compiled code, often cannot be analyzed statically.

Moreover, the current implementation of 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. For a child to work as intended, the parent also needs to install a seccomp filter for the syscalls the child uses as otherwise the operating system kills the child.

B. Implementation Details

In our prototype, Finalyzer is a strace-like syscall-tracing component linked against the target application or used as a

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

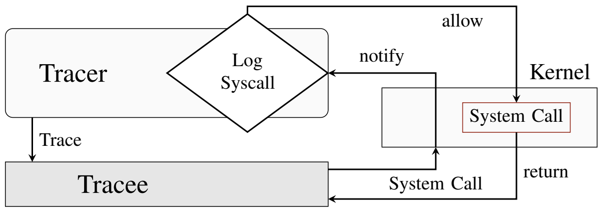


Fig. 5: The tracer gets notified by the kernel when the tracee executes a syscall. The tracer logs the syscall and informs the kernel to execute the syscall.

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 any of the two static components to identify the 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.

Upon receiving the notification of a violating syscall, Finalyzer extracts the syscall number ($\mathcal{C}\mathbf{1}$), logs it ($\mathcal{C}\mathbf{3}$), and allows it for all future occurrences (cf. Section III-C). As the syscall is indeed executed, it is inherent that it is reachable ($\mathcal{C}\mathbf{2}$). We illustrate this whole process in Figure 5.

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

VI. Evaluation

In this section, we evaluate the performance, functional correctness, and security of Chestnut. Our evaluation is in line with related work [24], [14] while improving on it in several points, *i.e.*, we evaluate several parts that were omitted by 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 prevent 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 discuss related works in this field to show the main differences to our work.

A. 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 shown in Table I, 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 [59]. 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.

B. Performance Evaluation

In this section, we evaluate the performance of Chestnut. This includes the one-time overheads for compiling (Section VI-B1) or binary analysis (Section VI-B2), the increase in binary size (Section VI-B3), and runtime overheads (Section VI-B4).

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 $65.5\,\mathrm{s}$ ($\sigma_{\bar{x}} = 0.094$, $N = 10$) to $84\,\mathrm{s}$ ($\sigma_{\bar{x}} = 0.054$, $N = 10$), an increase of $28\,\%$. For the busybox utilities combined, the average increases from $10.94\,\mathrm{s}$ ($\sigma_{\bar{x}} = 2.88$, $N = 10$) to $10.99\,\mathrm{s}$ ($\sigma_{\bar{x}} = 2.79$, $N = 10$). When compared to related work [24], 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.

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

	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 %
Client	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 %
Client	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 %
Client	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 %
Client	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 %
Client	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 %
Client	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 %
Client	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 %
Client	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 %
Client	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 %
Client	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 %
Client	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 %
Client	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 %
DB	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 %
Server	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 I: Results for the compiler- (</>) and binary-based (⚙) approach of Chestnut, respectively. For each software, 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 `execve` and `execveat` or `mprotect` syscalls respectively. We also show the percentage of fully mitigated CVEs and individual subvariants of these CVEs targeting the kernel. Note that the syscalls added in $\mathcal{P}2$ are only necessary in our proof-of-concept implementation of Chestnut as they appear in edge cases, which can be handled in a production-ready implementation.

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 %

TABLE II: We evaluate the size overhead of the compiler-based approach of Chestnut on shared libraries. The overhead is based on a vanilla version of the respective shared library.

100 MB) and its dynamic dependencies, the extraction takes around 11 min.

3) Binary Size Analysis: The code size does not increase with adding filters, only the binary size increases by the meta-information. Moreover, libchestnut and libseccomp are potentially linked to the application.

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 I 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 II 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 I 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 additional overhead of adding the dependency on libchestnut and libseccomp, but this increase is again insignificant and similar to the observed result of Sourcalyzer.

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. Hence, 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 sandbox an application with it. Previous work has shown that its performance depends on the number of filters that are installed and the rule's position within the filter list [30], [76]. The Linux developers are currently working on improving the performance of seccomp [12].

5) Dynamic Refinement Overhead: As a microbenchmark, we analyze the impact of Finalyzer on the syscall latency.

Software	Coverage Lines	Coverage Functions
FFmpeg	59.3 %	61.7 %
memcached	77 %	91.9 %
redis	77 %	61.5 %

TABLE III: Coverage results for selected applications.

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 supposed to be used as an optional step during development. Hence, we consider this to be less of a problem as it does not impact the released application.

C. Functional-Correctness Evaluation

A critical aspect of Chestnut is that the sandboxed binaries still work as intended without observing crashes. Related work [24] tested each application 100 times using various workloads. For a fair comparison, we perform the same tests. Moreover, for those applications where a test suite is available, we execute these test suites to reach higher coverage, ensuring that we do not miss edge cases. Beyond previous work [24], [14], we also extend our evaluation with code coverage results to show that large parts of the application are actually executed. We also perform a 6 month long test of Nginx sandboxed by Sourcalyzer.

In more detail, we first apply Chestnut to the binaries shown in Table I. Note that 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 ensure that we reach as many different code paths as possible, similar to what related work has done [24]. We observed no crashes in applications sandboxed with Chestnut. Even if a syscall is missed in $\mathcal{P}\mathbf{1}$, $\mathcal{P}\mathbf{2}$ 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 functional-correctness test tests whether all syscalls in the core functionality of the tested application are found. To further substantiate this, we perform a coverage test for a selection of applications shown in Table I. We show the result of these coverage tests in Table III. Additionally to our results, the sqlite developers always maintain 100 % branch and 100 % MC/DC coverage [70]. While not perfect, the results indicate that large parts of the respective applications are executed and, to a certain degree, demonstrate the functional correctness of the applications after Chestnut has been applied. In future work, we would like 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, $\mathcal{P}\mathbf{2}$ is necessary to ensure functionality. Out of the 18 tested applications, $\mathcal{P}\mathbf{2}$ 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 $\mathcal{P}\mathbf{2}$. We evaluated how many syscalls the static approaches missed. For Sourcalyzer, Finalyzer added 4 syscalls to musl libc, which are then propagated to the individual applications if the corresponding function is used, e.g., *clone*. Table I shows for each application how many syscalls were added in $\mathcal{P}\mathbf{2}$.

For Binalyzer and busybox, $\mathcal{P}\mathbf{2}$ only had to add a syscall in *ls*. The largest amount of added syscalls appears in sqlite3, where Finalyzer adds 13 syscalls.

These missed syscalls are currently only a limitation of our proof-of-concept implementation. All these syscalls occur in edge cases that our implementations do not, but a full implementation can cover. Hence, they do not necessitate Finalyzer to be a mandatory phase of Chestnut.

Long-Term Study using Nginx. To further 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, which we then deployed to a real-world server to host a website. The number of blocked syscalls for that Nginx binary is shown in Table I, *i.e.*, 105. Over this period of 6 months, the server handled around 100 000 requests without ever triggering a seccomp violation. This further demonstrates that Sourcalyzer can infer all syscalls necessary for a successful operation of Nginx on a real-world system.

D. Security Evaluation

To evaluate how Chestnut increases the security of sandboxed applications, we analyze how often dangerous syscalls, e.g., `exec`, are blocked (Section VI-D1), the number of syscalls not blocked even though they are not used by the application (Section VI-D2), the number of mitigated real-world exploits (Section VI-D3), and how malicious SGX enclaves can be prevented (Section VI-D4).

1) Preventing 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 [8]. In fact, most libc versions even contain a ROP gadget that leverages the `exec` syscall to open a shell [35]. 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 [10] as out of scope.

Even with Chestnut, certain attacks are still possible, e.g., adding an ssh key if a privileged application is exploited and the *open*/*write* syscalls are allowed. Note that these attacks are also possible with Chestnut, but other attacks are prevented, improving the overall system security. Hence, Chestnut improves the status quo, which is the goal of this work.

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

Binary. As Table I shows, Binalyzer prevents the `exec` syscalls in all of the shown busybox utilities. The analysis of the remaining busybox utilities showed that we can also prevent it in all of them. This result differs from Sourcalyzer which could not block the `exec` syscall in the *env* utility. We manually verified that the syscalls are indeed not required in the application. The `mprotect` analysis showed the opposite behavior as it is not blocked in any of the applications. For Nginx, memcached, mutool, and FFmpeg, we were also able to 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 the git commands rely on other applications to run, *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, the functions necessary for program startup can be removed from the analysis, potentially removing the *mprotect* syscall.

2) Overapproximation of Syscalls: As Table I shows, Chestnut can drastically reduce the number of syscalls available to a userspace application. For our 18 tested applications, Nginx and httpd block the least number of syscalls with 106 being allowed. However, without Chestnut, all 349 syscalls that Linux 5.0 provides would be available [44]. While Chestnut drastically reduces the attack surface, both Sourcalyzer and Binalyzer often allow more syscalls than necessary. We estimate our approaches' overapproximation to determine how tight the syscall filters are that an automated approach can generate. As such, we are the first to demonstrate this for an automated seccomp filter generation tool.

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. Using this setting, 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. Note that the accuracy of our results depends on the code coverage of the respective test suites. As was the case in Section VI-C, we again argue that despite this not being an exhaustive test, test suites typically cover at least the core functionality of the tested applications and its critical components. We again substantiate this claim with the code coverage metrics for selected applications shown in Table III. As the coverage metrics indicate, large parts of the respective applications are executed. This demonstrates that this is an adequate but not perfect approach to detect the overapproximation of Chestnut. Furthermore, we are the first to provide an insight into 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. For git, FFmpeg, memcached, redis, sqlite3, httpd, and Nginx, Binalyzer allows on average 2 syscalls more than Sourcalyzer. Sourcalyzer outperforms it due to two points: the heuristics based on function signatures that we apply in the compiler to infer potential indirect call targets, and as it only includes syscalls that are reachable from *main* and *exit*. While Binalyzer has to rely on heuristics as well, the amount of available information is smaller as it needs to disassemble generated code and infer the call graph from it using angr instead of doing it on the source code level where more meta-information is available. If angr incorrectly infers a call target, it potentially merges syscalls that are not necessary into a function that is later on used by the target application, which results in overapproximation.

As Table I shows, overapproximation varies between different applications. Out of the shown 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.

Binary. For the evaluation of Binalyzer, we slightly deviate from the outlined setup just to ease the evaluation. The only difference is that we do not rely on the linked libchestnut library, but instead use the standalone implementation of Finalyzer. Hence, we observe a larger amount of syscalls as we

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

TABLE IV: Syscalls and their equivalents.

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.

As was the case with the functional-correctness evaluation, we would like to investigate the possibility of better estimating the overapproximation using a coverage-guided fuzzer. Unfortunately, this is out-of-scope for this paper.

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 code-reuse attack [73] 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 in the kernel 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 CVEs from the mitre database [75] that exploit syscalls on the x86_64 Linux kernel. This results in a total of 175 CVEs since 2003. From this list, we extract the necessary syscalls and map them to the corresponding syscall numbers, resulting in a list of 231 malicious samples. The reason for the higher number is that a CVE can be triggered by different syscalls that are independent of each other. As some syscalls have equivalent versions that perform the same action, we extend our list of samples to 320 by substituting the syscall numbers where applicable. We provide a list of these equivalent syscalls in Table IV. 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 prohibited by our various test applications in Table I. If one of the syscalls required for the exploit is not allowed by the application, 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 mitigate around 84.04 % of the CVEs completely and around 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. With the larger applications, our compiler performs worse with mitigating around 38.08 % of CVEs completely and around 56.53 % of the subvariants. Even though Sourcalyzer does not perform as well for the larger binaries, it still increases the system's security.

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

4) Preventing 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. [66] presented a technique to execute arbitrary syscalls from an SGX enclave by mounting a ROP attack on the host application. This allows malicious or exploited enclaves to mount attacks on the kernel.

Weiser et al. [80] presented SGXJail as a generic countermeasure for malicious enclaves, preventing 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. [66]. 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 exploited) enclave cannot run arbitrary programs anymore, and the attack surface is drastically reduced.

E. Comparison to Other Approaches

Recently, the field of automating seccomp filter generation has gained traction with the publication of two works [24], [14]. Note that these works were only published after the start of our 6-month long-term case study of Chestnut on Nginx. In this section, we want to take a closer look at these two approaches and discuss the differences between them and our work.

Temporal Syscall Specialization. Ghavamnia et al. [24] propose an automated approach to detect the used syscalls during compilation. Contrary to our approach, they require a multitude of tools for the compilation and link-time optimization that is not supported by every application. Their approach is limited to applications that can be split into an initialization and serving phase, such as server applications. The basic 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 an initialization and serving phase, potentially enabling attacks through browsers, malicious PDFs [19], [20], messengers [68], [27], and office applications [52], [33]. We explicitly consider such applications in our approach (cf. Section III-A). 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], [28]. 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 evaluation showed that this approach achieves similar results in terms of detected syscalls as the more complex and slower approach used by Ghavamnia et al. [24]. 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 $1\,\mathrm{min}$ to $83\,\mathrm{min}$ (+8300 %) [24] compared to an increase of $7.9\,\mathrm{s}$ (+10.53 %) with Chestnut. Another difference is that our approach allows for a larger threat model as we also include a potential local attacker instead of just a remote one.

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

Sysfilter. A second approach, *sysfilter* [14] 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 VI-C and VI-D2, 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. The reason why we do this is twofold. 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.

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

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

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 position-independent binary, we note that there is no reason for such a requirement as 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 considered the most straightforward use case as all information is already contained within the single binary.

In the evaluation, 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 V.

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 I). 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 further investigated why the number of found syscalls is so low for 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 fault or a hint that this is a general fault in the approach for future work as this is out of scope for this paper.

VII. Discussion

Limitations and Future Work. One of the limitations is the performance of seccomp as it is quite slow [30], [76]. Unfortunately, this is a limitation imposed by the underlying system and not a weakness of Chestnut itself. Improving the performance of seccomp is considered out of scope for this paper, and there is no suitable alternative.

Another limitation is the overapproximation of both the compiler- and binary-based approaches. Fast and reliable points-to analysis with limited overapproximation is still an unsolved problem as previous work has shown [28]. 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 future work, we will investigate the possibility of extending the syscall filtering with argument tracking. While detecting the constant arguments from a syscall is possible, the problem is the propagation of the information throughout the call graph so that in the end, only the argument remains that is needed. If we can solve this problem, we can restrict syscalls even further. For instance, for the `exec` syscalls, we can detect hardcoded paths and install filters that only allow it with this path. For `mprotect`, we can then further restrict the possible set of permissions so that only those are allowed that are used, potentially preventing `mprotect` with executable permissions. Furthermore, we will investigate performance improvements for seccomp as well as alternatives. We also want to investigate the possibility of using coverage-guided fuzzers to estimate the overapproximation of automated seccomp filter generation tools.

Related Work. For related work, we mostly focus on work that tried to automate a sandboxing process as this is also what was done in this work. Previous work has already focused on reducing the attack surface of an application by removing unused code from an application. One of the first approaches for library debloating was proposed by Mulliner and Neugschwandtner [53] based on removing non-imported functions from a shared library during load time. This approach has been further improved by Quach et al. [59] by removing all unused functions from shared libraries during load time by extending the compiler and the loader. Agadakos et al. [1] proposed a binary-level approach for library debloating. It is based on function boundary detection and dependency identification to identify and erase unused functions. Davidson et al. [13] performed an analysis of the complete software stack for web applications to create specialized libraries based on the requirements for PHP code and the server binaries. Mishra and Polychronakis [50] proposed *Shredder*, which instruments binaries to restrict arguments to critical system APIs to a predetermined allowlist. Another approach is to apply data dependency analysis for fine-grained library customization of static libraries [69]. Ghavamnia et al. [24] propose a similar approach to Sourcalyzer, but at the cost of a much higher analysis time during compilation but with similar accuracy in detecting syscalls. Wagner and Dean [78] 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 [21]. Rajagopalan et al. [61] propose to replace syscalls with authenticated syscalls that specify a policy and provide a cryptographic message authentication code that guarantees the integrity of the syscall.

Other approaches propose to reduce the attack surface by relying on training to identify the unused code sections. Ghaffarinia and Hamlen [23] rely on training to limit control flow transfers to not authorized sections of code. An approach without access to the source code uses training and heuristics to identify and remove unnecessary basic blocks [57].

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 [34]. For PHP, Azad et al. [3] proposed a framework using dynamic analysis to remove superfluous features.

VIII. Conclusion

Chestnut is an automated approach for sandboxing native Linux userspace applications on the syscall level. It mainly relies on static analysis to identify syscalls required by an application and blocks the unused syscalls. Chestnut also supports an optional dynamic refinement phase that can be used to restrict the syscall filters further. In contrast to existing solutions, Chestnut has fewer requirements and limitations. For instance, the compiler-based approach is up to factor 73 faster than previous work without any loss in accuracy. For the binary-level analysis, we lifted the requirement of a position-independent binary, making our approach applicable to a broader set of applications. We demonstrated that in a selection of 18 real-world applications, Chestnut can, on average, block 302 syscalls (86.5 %) when used on the source level, and 288 (82.5 %) when used on the binary level. Our analysis showed that the compiler- and binary-based approach prevent exploitation of more than 63 % and 61 %, respectively, of Linux kernel CVEs, which can be triggered via syscalls, in case the selected applications have been exploited. Moreover, Chestnut can reduce the attack surface by blocking the dangerous `exec` syscall in the majority of tested applications. We have provided insights into the functional correctness and the overapproximation of Chestnut, which we substantiated by code coverage metrics of the respective tests. Additionally, we showed the results of a 6month long-term evaluation of Nginx on a real server. Finally, our work shows that automated sandboxing is feasible and increases platform security without manual effort.

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] I. Agadakos, D. Jin, D. Williams-King, V. P. Kemerlis, and G. Portokalidis, "Nibbler: debloating binary shared libraries," in *ACSAC*, 2019.

[2] L. O. Andersen, "Program Analysis and Specialization for the C Programming Language," Ph.D. dissertation, May 1994.

[3] B. A. Azad, P. Laperdrix, and N. Nikiforakis, "Less is more: Quantifying the security benefits of debloating web applications," in *USENIX Security Symposium*, August 2019.

[4] T. K. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, "Jump-oriented programming: a new class of code-reuse attack," in *AsiaCCS*, 2011.

[5] E. Bosman and H. Bos, "Framing signals - A return to portable shellcode," in *S&P*, 2014.

[6] D. P. Bovet, "Special sections in Linux binaries," January 2013. [Online]. Available: https://lwn.net/Articles/531148/

[7] C. Canella, J. Van Bulck, M. Schwarz, M. Lipp, B. von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss, "A Systematic Evaluation of Transient Execution Attacks and Defenses," in *USENIX Security Symposium*, 2019, extended classification tree and PoCs at https://transient.fail/.

[8] N. Carlini and D. A. Wagner, "ROP is still dangerous: Breaking modern defenses," in *USENIX Security Symposium*, 2014.

[9] S. Checkoway, L. Davi, A. Dmitrienko, A. Sadeghi, H. Shacham, and M. Winandy, "Return-oriented programming without returns," in *CCS*, 2010.

[10] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer, "Non-control-data attacks are realistic threats." in *USENIX Security Symposium*, 2005.

[11] Chromium, "Linux Sandboxing." [Online]. Available: https://chromium.googlesource.com/chromium/src/+/0e94f26e8/docs/linux_sandboxing.md

[12] J. Corbet, "Constant-action bitmaps for seccomp()," 2020.

[13] N. Davidsson, A. Pawlowski, and T. Holz, "Towards automated application-specific software stacks," in *ESORICS*, 2019.

[14] N. DeMarinis, K. Williams-King, D. Jin, R. Fonseca, and V. P. Kemerlis, "sysfilter: Automated System Call Filtering for Commodity Software," in *RAID*, October 2020.

[15] D. Drysdale, "Anatomy of a system call, part 2," 2014. [Online]. Available: https://lwn.net/Articles/604515/

[16] ——, "How programs get run: ELF binaries," February 2015. [Online]. Available: https://lwn.net/Articles/631631/

[17] J. Edge, "A library for seccomp filters," April 2012. [Online]. Available: https://lwn.net/Articles/494252/

[18] ——, "A seccomp overview," September 2015. [Online]. Available: https://lwn.net/Articles/656307/

[19] J. M. Esparza, "Static analysis of a CVE-2011-2462 PDF exploit," 2012.

[20] ——, "Quick analysis of the CVE-2013-2729 obfuscated exploits," 2014.

[21] H. H. Feng, J. T. Giffin, Y. Huang, S. Jha, W. Lee, and B. P. Miller, "Formalizing sensitivity in static analysis for intrusion detection," in *S&P*, 2004.

[22] Firejail, "Firejail Security Sandbox," 2016. [Online]. Available: https://firejail.wordpress.com/

[23] M. Ghaffarinia and K. W. Hamlen, "Binary control-flow trimming," in *CCS*, 2019.

[24] S. Ghavamnia, T. Palit, S. Mishra, and M. Polychronakis, "Temporal System Call Specialization for Attack Surface Reduction," in *USENIX Security Symposium*, August 2020.

[25] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis, "Out of control: Overcoming control-flow integrity," in *S&P*, 2014.

[26] I. Goldberg, D. Wagner, R. Thomas, E. A. Brewer *et al.*, "A secure environment for untrusted helper applications: Confining the wily hacker," in *USENIX Security Symposium*, 1996.

[27] S. Groß, "Remote iPhone Exploitation Part 1: Poking Memory via iMessage and CVE-2019-8641," 2020.

[28] M. Hind, "Pointer analysis: Haven't we solved this problem yet?" in *PASTE*, 2001.

[29] G. J. Holzmann, "Code inflation," *IEEE Software*, 2015.

[30] T. Hromatka, "seccomp and libseccomp performance improvements," 2018.

[31] G. Inc., "Seccomp filter in Android O," July 2017. [Online]. Available: https://android-developers.googleblog.com/2017/07/seccomp-filter-in-android-o.html

[32] ——, "Sandbox2," 2019. [Online]. Available: https://developers.google.com/sandboxed-api/docs/sandbox2/overview

[33] A. Inführ, "Libreoffice (CVE-2018-16858) - Remote Code Execution via Macro/Event execution," February 2019.

[34] Y. Jiang, D. Wu, and P. Liu, "Jred: Program customization and bloatware mitigation based on static analysis," in *COMPSAC*, 2016.

[35] M. Jurczyk and G. Coldwind, "Permissions overview," *Insomni'hack*, 2015.

[36] V. Kemerlis, "Protecting commodity operating systems through strong kernel isolation," Ph.D. dissertation, Columbia University, 2015.

[37] V. P. Kemerlis, M. Polychronakis, and A. D. Keromytis, "ret2dir: Rethinking kernel isolation," in *USENIX Security Symposium*, 2014.

[38] V. P. Kemerlis, G. Portokalidis, and A. D. Keromytis, "kguard: Lightweight kernel protection against return-to-user attacks," in *USENIX Security Symposium*, 2012.

[39] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors," in *ISCA*, 2014.

[40] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre Attacks: Exploiting Speculative Execution," in *S&P*, 2019.

[41] B. Lan, Y. Li, H. Sun, C. Su, Y. Liu, and Q. Zeng, "Loop-oriented programming: a new code reuse attack to bypass modern defenses," in *IEEE Trustcom/BigDataSE/ISPA*, 2015.

[42] C. Lattner and V. S. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *IEEE / ACM International Symposium on Code Generation and Optimization – CGO*, 2004.

[43] X. Lin, L. Lei, Y. Wang, J. Jing, K. Sun, and Q. Zhou, "A measurement study on Linux container security: Attacks and countermeasures," in *ACSAC*, 2018.

[44] Linux, "64-bit system call numbers and entry vectors," 2019. [Online]. Available: https://github.com/torvalds/linux/blob/master/arch/x86/entry/syscalls/syscall_64.tbl

[45] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading Kernel Memory from User Space," in *USENIX Security Symposium*, 2018.

[46] K.-K. Ma, K. Y. Phang, J. S. Foster, and M. Hicks, "Directed symbolic execution," in *International Static Analysis Symposium*, 2011.

[47] S. McCanne and V. Jacobson, "The BSD Packet Filter: A New Architecture for User-level Packet Capture," in *USENIX Winter*, January 1993.

[48] W. S. McPhee, "Operating system integrity in os/vs2," *IBM Systems Journal*, 1974.

[49] M. Miller, "Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape," *Bluehat IL*, February 2019.

[50] S. Mishra and M. Polychronakis, "Shredder: Breaking exploits through api specialization," in *ACSAC*, 2018.

[51] Mozilla., "Seccomp filter in Android O," July 2016. [Online]. Available: https://wiki.mozilla.org/Security/Sandbox/Seccomp

[52] J. Müller, F. Ising, C. Mainka, V. Mladenov, S. Schinzel, and J. Schwenk, "Office document security and privacy," in *WOOT*, 2020.

[53] C. Mulliner and M. Neugschwandtner, "Breaking payloads with runtime code stripping and image freezing," *BlackHat USA*, August 2015.

[54] S. Narayan, C. Disselkoen, T. Garfinkel, N. Froyd, E. Rahm, S. Lerner, H. Shacham, and D. Stefan, "Retrofitting Fine Grain Isolation in the Firefox Renderer," in *USENIX Security Symposium*, 2020.

[55] Nergal, "The advanced return-into-lib(c) explits: PaX case study," 2001.

[56] V. Prevelakis and D. Spinellis, "Sandboxing applications," in *USENIX ATC*, 2001.

[57] C. Qian, H. Hu, M. Alharthi, P. Ho Chung, T. Kim, and W. Lee, "RAZOR: A framework for post-deployment software debloating," in *USENIX Security Symposium*, 2019.

[58] A. Quach, R. Erinfolami, D. Demicco, and A. Prakash, "A multi-OS cross-layer study of bloating in user programs, kernel and managed execution environments," in *FEAST*, 2017.

[59] A. Quach, A. Prakash, and L. Yan, "Debloating software through piece-wise compilation and loading," in *USENIX Security Symposium*, 2018.

[60] N. A. Quynh, "Capstone: Next-gen disassembly framework," *Black Hat USA*, 2014.

[61] M. Rajagopalan, M. Hiltunen, T. Jim, and R. Schlichting, "Authenticated system calls," in *DSN*, 2005.

[62] C. Reis and S. D. Gribble, "Isolating web programs in modern browser architectures," in *EuroSys*, 2009.

[63] C. Reis, A. Moshchuk, and N. Oskov, "Site Isolation: Process Separation for Web Sites within the Browser," in *USENIX Security Symposium*, 2019.

[64] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A. Sadeghi, and T. Holz, "Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications," in *S&P*, 2015.

[80] S. Weiser, L. Mayr, M. Schwarz, and D. Gruss, "SGXJail: Defeating enclave malware via confinement," in *RAID*, 2019.

[65] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss, "ZombieLoad: Cross-Privilege-Boundary Data Sampling," in *CCS*, 2019.

[66] M. Schwarz, S. Weiser, and D. Gruss, "Practical Enclave Malware with Intel SGX," in *DIMVA*, 2019.

[67] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *CCS*, 2007.

[68] N. Silvanovich, "Exploiting Android Messengers with WebRTC: Part 1," 2020.

[69] L. Song and X. Xing, "Fine-grained library customization," in *SALAD*, 2018.

[70] SQLite, "How SQLite Is Tested," 2020.

[71] B. Steensgaard, "Points-to analysis in almost linear time," in *POPL*, 1996.

[72] N. Sylvain, "A new approach to browser security: the Google Chrome Sandbox," October 2008. [Online]. Available: https://blog.chromium.org/2008/10/new-approach-to-browser-security-google.html

[73] L. Szekeres, M. Payer, T. Wei, and D. Song, "SoK: Eternal War in Memory," in *S&P*, 2013.

[74] R. Tarjan, "Depth-first search and linear graph algorithms," *SIAM journal on computing*, June 1972.

[75] The MITRE Corporation, "Common vulnerabilities and exposures." [Online]. Available: http://cve.mitre.org/

[76] Tizen, "Security:Seccomp," 2018. [Online]. Available: https://wiki.tizen.org/Security:Seccomp

[77] S. van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida, "RIDL: Rogue In-flight Data Load," in *S&P*, 2019.

[78] D. Wagner and R. Dean, "Intrusion detection via static analysis," in *S&P*, 2000.

[79] F. Wang and Y. Shoshitaishvili, "Angr - The Next Generation of Binary Analysis," in *2017 IEEE Cybersecurity Development (SecDev)*, 2017.

[81] M. Wiki, "Project Fission," 2019. [Online]. Available: https://wiki.mozilla.org/Project_Fission

[82] ——, "Security/Sandbox," December 2019. [Online]. Available: https://wiki.mozilla.org/Security/Sandbox

[83] Y. Yarom and K. Falkner, "Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack," in *USENIX Security Symposium*, 2014.