Building Dynamic System Call Sandbox with Partial Order Analysis

QUAN ZHANG, CHIJIN ZHOU, YIWEN XU, ZIJING YIN, MINGZHE WANG, and ZHUO SU, BNRist, School of Software, Tsinghua University, China
CHENGNIAN SUN, Cheriton School of Computer Science, University of Waterloo, Canada
YU JIANG*, BNRist, School of Software, Tsinghua University, China
JIAGUANG SUN, BNRist, School of Software, Tsinghua University, China

Attack surface reduction is a security technique that secures the operating system by removing the unnecessary code or features of a program. By restricting the system calls that programs can use, the system call sandbox is able to reduce the exposed attack surface of the operating system and prevent attackers from damaging it through vulnerable programs. Ideally, programs should only retain access to system calls they require for normal execution. Many researchers focus on adopting static analysis to automatically restrict the system calls for each program. However, these methods do not adjust the restriction policy along with program execution. Thus, they need to permit all system calls required for program functionalities.

We observe that some system calls, especially security-sensitive ones, are used a few times in certain stages of a program's execution and then never used again. This motivates us to minimize the set of required system calls dynamically. In this paper, we propose DynBox, which gradually disables access to unnecessary system calls throughout the program's execution. To accomplish this, we utilize partial order analysis to transform the program into a partially ordered graph, which enables efficient identification of the necessary system calls at any given point during program execution. Once a system call is no longer required by the program, DynBox can restrict it immediately. To evaluate DynBox, we applied it to seven widely-used programs with an average of 615 KLOC, including web servers and databases. With partial order analysis, DynBox restricts an average of 23.50, 16.86, and 15.89 more system calls than the state-of-the-art Chestnut, Temporal Specialization, and the configuration-aware sandbox, C2C, respectively. For mitigating malicious exploitations, on average, DynBox defeats 83.42% of 1726 exploitation payloads with only a 5.07% overhead.

CCS Concepts: • **Security and privacy → Trust frameworks**.

Additional Key Words and Phrases: System Call Sandbox, Program Analysis, Attack Surface Reduction

ACM Reference Format:
Quan Zhang, Chijin Zhou, Yiwen Xu, Zijing Yin, Mingzhe Wang, Zhuo Su, Chengnian Sun, Yu Jiang, and Jiaguang Sun. 2023. Building Dynamic System Call Sandbox with Partial Order Analysis. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 266 (October 2023), 28 pages. https://doi.org/10.1145/3622842

*Yu Jiang is the corresponding author.

Authors' addresses: Quan Zhang, zhangq20@mails.tsinghua.edu.cn; Chijin Zhou, tlock.chijin@gmail.com; Yiwen Xu, xuyiwen14@gmail.com; Zijing Yin, Aurora@europe.com; Mingzhe Wang, wmzhere@gmail.com; Zhuo Su, suzcpp@gmail.com, BNRist, School of Software, Tsinghua University, Beijing, China; Chengnian Sun, cnsun@uwaterloo.ca, Cheriton School of Computer Science, University of Waterloo, Waterloo, ON, Canada; Yu Jiang, jiangyu198964@126.com, BNRist, School of Software, Tsinghua University, Beijing, China; Jiaguang Sun, sthuse20@outlook.com, BNRist, School of Software, Tsinghua University, , China.

2475-1421/2023/10-ART266
https://doi.org/10.1145/3622842

1 INTRODUCTION

As software systems become increasingly complex, it is inevitable that security vulnerabilities will be introduced. Despite the usage of advanced testing and defense techniques [Abadi et al. 2009; Zalewski 2016; Zhang et al. 2023a; Zhou et al. 2022], these programs may still be exposed to potential vulnerabilities, creating a large attack surface for adversaries to exploit and further compromise the host operating system (OS). To mitigate such exploitation attempts, security researchers have begun restricting the capabilities available to attackers. System calls (syscalls) are one of the essential capabilities during the exploitation as attackers need them to achieve their goals, such as socket listening or malicious code execution. In order to reduce the OS's attack surface, it is necessary to restrict the available syscalls of a program by building a syscall sandbox. With a proper policy, such a syscall sandbox can isolate the vulnerable program without interrupting its normal execution. After restricting available syscalls for the program with a strict syscall sandbox policy, even if attackers exploit this vulnerable program, it is much more difficult for them to cause damage to OS. This type of sandbox is widely used to isolate vulnerable sub-components, such as the JavaScript engine of browsers.

Many techniques are proposed to build syscall sandboxes with custom sandbox policies for programs [Canella et al. 2021; Ghavamnia et al. 2020, 2022]. They all aim to accurately resolve the minimal set of syscalls required by a program as the policy. To this end, many efforts have been made to construct a more precise call graph. With the call graph, existing methods can determine the potential reachable functions and collect all syscalls invoked in these functions as the required syscall set. Then, existing methods manually select one or two transition points between the program's different execution phases and enforce syscall restriction policies at these transition points. Most works directly set policy at the beginning of a program, permitting all syscalls that may be used by the program. Apart from this, Temporal Specialization (Temp) [Ghavamnia et al. 2020] attempts to enforce a tighter policy once more at the transition point of initialization completion, so it can further restrict syscalls that are only required during the initialization phase. After enforcing restrictions at these two transition points, the syscall sandbox policies remain fixed and are never changed in the subsequent phases. Existing approaches with such *fixed* policies lack strict constraints, because many complex programs inherently require a large number of critical syscalls, such as `execve`, `fork`, and `setuid`, in different running phases with different functionalities. Thus, existing works have to permit all critical syscalls required for the programs' subsequent execution and will not adjust the policy further, even though these critical syscalls are only required in certain phases and not used thereafter. This allows attackers to utilize these critical syscalls to damage the OS when exploiting a vulnerable program running on it. Therefore, it is crucial to identify more transition points between a program's different execution phases for dynamically forbidding unnecessary syscalls, and existing works leave much to be desired.

The main challenge that prevents existing methods from achieving dynamical syscall restriction is that manually identifying positions where the program's syscall requirement changes is difficult and imprecise. First, manually identifying a transition point of any two phases is labor-consuming and requires expert experience. Moreover, a program usually has a complex lifecycle with many phases, making it impossible to divide all of them manually. Furthermore, through manual identification, these transition points between the program's execution phases cannot accurately capture the reduction positions where the program's required syscall set decreases. Therefore, an urgent demand exists for a dynamic syscall sandbox that can automatically identify these reduction positions and promptly restrict unnecessary syscalls.

Thus, we propose a syscall sandbox called DynBox that dynamically restricts unnecessary syscalls according to the program's requirement. When a program reaches a reduction position

where a syscall becomes no longer required for subsequent execution, DynBox enforces the syscall restriction policy to restrict the syscall. Specifically, the problem of identifying the reduction position is simplified into calculating the possible execution order of any two instructions. Based on the execution order, we can know, after each instruction, which syscall invocations will never be executed, thereby identifying the reduction position where the syscall becomes unnecessary.

To this end, the partial order analysis is proposed to resolve the execution order of instructions. The analysis first finds candidate reduction positions to build the candidate graph according to the control flow. Then, the candidate graph is transferred into the partially ordered graph, whose nodes are mapped from instructions of a program, and edges are transferred from the candidate graph's edges. This graph guarantees that for any two nodes n_a and n_b on the graph, if there is an edge from n_a to n_b, then the instruction mapped to n_a can only be executed before the one mapped to n_b. Based on this graph, the analysis determines the execution order of instructions. By obtaining all subsequent syscall invocations for an instruction I_i, we can determine the set of syscalls required by the program for subsequent execution starting from I_i (i.e., the **required syscall set**). Finally, the analysis identifies reduction positions, where the required syscall set reduces, and performs instrumentation at these positions with syscall restriction policies to construct DynBox. During the program's execution, these restriction policies are gradually enforced, and the sandbox is thus tightened. We ensure that DynBox will not interfere with the program's normal execution.

We implement the partial order analysis based on LLVM [Lattner and Adve 2004] and adopt the Linux Seccomp BPF [Kernel 2022] to construct DynBox. To validate the effectiveness and scalability of DynBox, we build DynBox for seven complex applications with an average of 615 KLOC, including web servers (Httpd, Nginx), databases (Redis, Memcached, SQLite), a DNS system (Bind), and a compression tool (Tar). All of them are widely used and prone to many critical vulnerabilities. We collect 43 exploitable vulnerabilities for all programs and locate their possible exploitation positions. On each position, we evaluate how many syscalls that DynBox restricts when the program executes at this position. The results show that DynBox forbids 23.50, 16.86, and 15.89 more syscalls than three state-of-the-art syscall sandboxes, Chestnut [Canella et al. 2021], Temp [Ghavamnia et al. 2020] and C2C [Ghavamnia et al. 2022], respectively. Furthermore, DynBox can precisely restrict critical syscalls, such as `bind`, `listen`, and `socket`, at the proper positions, even if they are required by the core functionalities of the programs. By dynamically adjusting the sandbox policy, DynBox can block 1440 of 1726 (83.42%) malicious payloads on average. Meanwhile, we ensure that DynBox will not interfere with the normal execution of programs, and it only incurs 5.07% runtime costs and 4.23% binary size expansion.

In summary, our work makes the following contributions.

- We propose DynBox, a dynamic syscall sandbox, that gradually restricts unnecessary syscalls to continuously reduce the attack surface during the program's execution.
- We design partial order analysis to build DynBox. The analysis can automatically identify the reduction positions where syscalls become no longer required, and enforce syscall restriction policies at these positions.
- We implement DynBox and evaluate its effectiveness on seven complex programs. On average, On average, DynBox surpasses existing syscall sandboxes by restricting an additional 15.89~23.50 syscalls, effectively mitigating 83.42% of exploitation payloads.

Our paper is organized as follows. In Section 2, we introduce the background and highlight the main focus of this paper. Then, we present a motivation example in Section 3. The design and implementation of DynBox are outlined in Section 4 and Section 5. Next, we evaluate DynBox in Section 6 from three aspects. We discuss the limitation of DynBox and introduce related works in Section 7 and Section 8, respectively. Finally, we conclude the paper in Section 9.

2 BACKGROUND

2.1 Syscall Sandbox

Syscall sandbox is an isolation technique that restricts the usable syscall of a program. Recently, operating systems have introduced new mechanisms to set a syscall sandbox for a program, such as Seccomp BPF for Linux and Pledge for OpenBSD [Kernel 2022; Pal 2018]. By isolating the vulnerable program with a syscall sandbox, operating systems can effectively restrict the damage caused by vulnerability exploitation of this program and protect itself. Here we take the Seccomp BPF as an example to introduce the syscall sandbox.

Seccomp BPF [Kernel 2022] is a mechanism that Linux provides to build the syscall sandbox. By loading a Berkeley Packet Filter (BPF) [McCanne and Jacobson 1993] program into the kernel, we can set a policy for the target program. In the BPF program, we can define the policy for restricting syscall. After loading BPF, each time the target program invokes a syscall, Linux runs the BPF program to check whether the calling complies with the policy. If the syscall is forbidden, Linux will send a signal to the target program and terminate its execution. Seccomp BPF is now widely used to build the sandbox. For example, Docker uses it to restrict the privilege of a container [Merkel 2014]. Browsers use Seccomp BPF to isolate the render process since it contains many vulnerabilities and is frequently exploited [Narayan et al. 2020; Reis et al. 2019].

To set a policy, a program needs to invoke syscalls `prctl` or `seccomp` by itself. By invoking them multiple times, the program can augment existing syscall restriction policies with new ones. Consequently, by appropriately instrumenting `prctl` or `seccomp` syscalls within the program's code, we produce a sandbox-enhanced binary for this program. When executing such binary, the program will execute the instrumented codes at proper execution positions. In this way, it can progressively enforce new policies to restrict the syscalls that just become unnecessary, and finally a dynamic syscall sandbox is built for this program. However, removal of a restriction policy is forbidden, thus making the syscall sandbox only more stringent. Hence, when restricting a syscall, it is crucial to ascertain that the program no longer requires it.

The policy of Seccomp BPF is thread-independent and process-independent. Once a new thread or process is spawned, it must inherit the policy of its parent. This means that a child's policy cannot override the policy of its parent, but can only add new restrictions. Thus, Seccomp BPF allows us to build a dynamic sandbox for each process and thread.

2.2 Vulnerability Exploitation

Vulnerabilities are inevitable during the development of a program. Once the program is deployed to the real world, its vulnerabilities can be utilized as entry points by the attackers to compromise the OS [Kemerlis et al. 2012; Szekeres et al. 2013; Zhang et al. 2021; Zhou et al. 2023]. Usually, attackers exploit a vulnerability in the program and hijack the program's control flow to execute malicious codes, which is also called payloads. The impact of malicious payloads is not limited to the vulnerable program but the host OS. They may steal vital information, execute arbitrary commands or even get the root privilege of the system.

To achieve the attack objectives, attackers must interact with OS and invoke many syscalls. For example, `execve` is widely utilized for executing commands. To escalate privilege, `setuid` is usually called. Additionally, attacks through the Internet need syscalls related to the network (e.g., `listen`, `bind`). For concurrent bugs, `sched_setaffinity` is normally used for improving the probability of a successful attack. On some vulnerabilities, attackers may not get the privilege, but they can still steal information with syscalls like `open` and `read`. If we can build a syscall sandbox to isolate the vulnerable program and restrict program's unnecessary syscalls, we can effectively reduce the attack surface and protect the OS [Li et al. 2017].

2.3 Focus of This Paper

DynBox aims to defend the operating system against attacks using vulnerable programs as entry points by dynamically minimizing the exposed attack surface. We assume that programs are not malicious but contain exploitable vulnerabilities. Deployed with these vulnerable programs, the OS exposes a wide attack surface to attackers. They can take over the program and use it as a stepping stone to compromise the entire system.

If we can apply a suitable syscall sandbox for a program, even when attackers successfully hijack its control flow, the accessible syscalls are still restricted by the sandbox. Thus, we can block such attacks from taking over vulnerable programs and reduce the damage that a compromised program can cause. Moreover, DynBox is not specific for certain kinds of vulnerabilities (e.g., memory safety bug or logic bug.) [Nagarakatte et al. 2009], or exploitation methods (e.g., code injection, code reuse) [Schuster et al. 2015; Shacham 2007; Younan et al. 2012], thus bringing universal protection for operating systems.

In this paper, DynBox primarily focuses on programs whose code remains unchanged. It is because DynBox requires analyzing their potential control flow. In addition, we need to preprocess the dynamic libraries of a program. Therefore, currently, programs with dynamically loaded libraries [Project 2023] or self-modifying code are not supported. In addition, DynBox relies on two static analysis tools that are essential components of our trusted computation base (TCB), enabling the resolution of indirect calls and analysis of dynamic library binaries.

It is important to note that the syscall sandbox should not interrupt the normal execution of the program. Therefore, a syscall can only be restricted when it is deemed unnecessary. If an attacker exploits a vulnerability by utilizing limited syscalls at the exploitation position where these syscalls are still required for the program's subsequent execution, DynBox may not be able to provide protection in such cases.

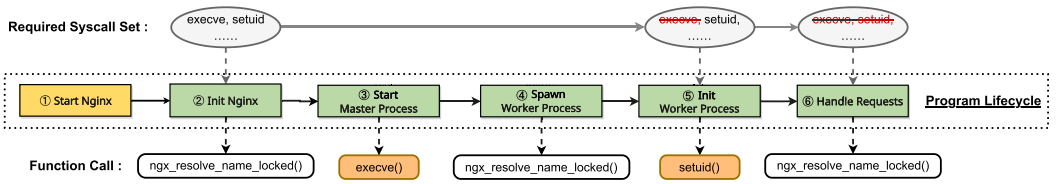


Fig. 1. The variety of required syscall set during the execution of Nginx. The syscalls within the ellipses are required for the program's execution at the corresponding positions. The "Function Call" are functions invoked in different phases. The vulnerability CVE-2016-0746 is located at `ngx_resolve_name_locked`.

3 MOTIVATING EXAMPLE

In this section, we illustrate the motivation of our work by analyzing a vulnerability in Nginx. CVE-2016-0746 [MITRE 2016] is a use-after-free (UAF) vulnerability in Nginx, which is one of the most critical types of vulnerability and is widely used for remote code execution. As the white boxes show in Figure 1, the vulnerability is located at the function `ngx_resolve_name_locked`, which is called several times during execution. The program has distinct required syscall sets at different positions where the function is called. Specifically, during initialization in ②, Nginx starts and calls `ngx_resolve_name_locked` for the first time. At this position, Nginx has not spawned the worker process, and thus the syscalls `execve` and `setuid` are required. After creating the worker process in ⑤, `ngx_resolve_name_locked` is called again. Now, `execve` is not needed for the subsequent execution and is excluded from the required syscall set. However, the initialization of the child

process does not finish at this position, and `setuid` is still necessary for the subsequent execution. Finally, the worker process starts to handle requests from users after finishing all the initialization in ⑥. In this phase, Nginx repeatedly calls the `ngx_resolve_name_locked` to process the requests, and neither `execve` nor `setuid` is required. Notably, Figure 1 only describes the lifecycle of Nginx, and different programs have different execution phases that need to be analyzed individually.

Attackers can exploit the UAF vulnerability in the function `ngx_resolve_name_locked` to compromise the system. To defend against it, many researchers have designed customized syscall sandboxes for Nginx. To achieve effective protection, where to enforce new sandbox policies is extremely important. Most existing works [Canella et al. 2021; DeMarinis et al. 2020; Quach et al. 2018] set the policy after Nginx initially starts in ①, so they have to permit the syscalls `execve` and `setuid` to Nginx. However, since these syscalls are widely used by attackers to execute malicious code, existing sandboxes cannot effectively isolate vulnerable applications like Nginx.

Nginx handles requests in the ⑥, so attackers are most likely to exploit the vulnerability at that time. If we can dynamically adjust the sandbox along the execution of Nginx, we can forbid the usage of `execve` and `setuid` at the sixth phase. In that case, attackers will be severely hindered from causing damage to the system. Thus, building a dynamic sandbox to restrict syscalls at proper positions can significantly reduce the attack surface for operating systems.

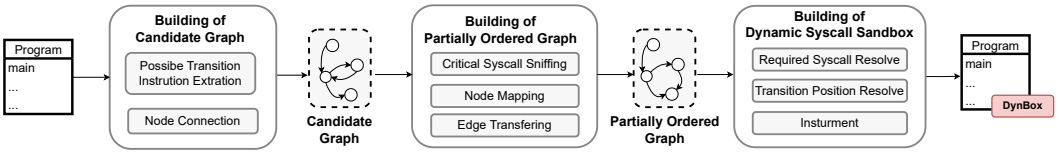


Fig. 2. Overview of the partial order analysis. First, the analysis extracts candidate set-reduce instructions and connects them to the candidate graph. Then, the candidate graph is transferred to the partially ordered graph. Finally, based on the partially ordered graph, the analysis resolves required syscalls, finds reduction positions, and dynamically enforces the sandbox policy to construct DynBox.

4 DESIGN

In this section, we illustrate the design of DynBox. To construct DynBox, we should find the reduction positions where the required syscall set reduces and instrument the program to build a sandbox-enhanced binary. For the sake of simplification, we first transfer the problem of finding reduction positions into resolving the execution order of instruction, so that we can know the subsequent syscall invocations after each instruction. To meet this end, we propose partial order analysis, which involves three steps depicted in Figure 2.

- In the first step, the analysis extracts all *candidate set-reduce instructions* ψ. After these instructions' execution, the required syscall set may reduce. Next, these instructions are connected as nodes to build candidate graph $CG(\psi, s)$ based on the program's control flow.
- In the second step, nodes ψ of CG, i.e., *candidate set-reduce instructions*, are mapped to nodes of the partially ordered graph $PG(N, E)$, and CG's edges S are also transferred to edges E of PG. Meanwhile, critical syscall sniffing is applied during PG construction to improve the analysis precision. In this way, PG is ensured to be partially ordered and can be used to resolve the instructions' execution order.
- In the third step, the analysis employs PG to determine the set of syscalls needed for the program's execution after each instruction is executed (namely, the required syscall set). By identifying where the required syscall set reduces, the analysis finally finds appropriate reduction positions and performs instrumentation at these positions to construct DynBox.

After instrumentation, we obtain an enhanced binary that possesses the capability to dynamically adjust the syscall policy throughout its execution, thereby restricting unnecessary syscalls. We prove that the analysis produces no false positives, meaning that DynBox does not restrict any required syscall and thus will not interfere with the normal execution of programs.

4.1 Problem Formulation

We define a program P's instruction at address i as I_i. For a specific input, a running program can be formulated as a trace $\tau = (I_i, I_j, ..., I_n)$. $\tau[k]$ is the k-th executed instruction of τ. Considering all possible inputs, the set of all possible traces constitutes a trace set T. For a multithreading or multiprocessing program, one τ represents the execution flow of one of the threads or processes.

To determine whether we can reduce a syscall *sc* after a certain instruction I_i's execution, for any trace τ that contains I_i, we should ensure that *sc* will not be invoked after I_i. In other words, we should find all possible instructions executed after I_i on any trace $\tau \in T$, and make sure they are not invocation instructions of *sc*. Thus, we need the execution order of I_i and other instructions. In conclusion, our goal can be converted to a problem of resolving the execution order of any two instructions. To describe the execution order, the successor relation $\succcurlyeq$ is first introduced as follows.

Definition 4.1. *(**Relation $\succcurlyeq$ of Instructions**) The instruction $I_j \succcurlyeq I_i$ holds if and only if there exists at least one τ among all traces T that satisfies $\tau[a] = I_i, \tau[b] = I_j, b \geq a$.*

Instruction I_j is denoted as the successor of I_i. Notably, $I_i \succcurlyeq I_i$. With the execution order, the required syscall set of instruction I_i can be further calculated by finding I_i's all successor instructions that invoke syscalls. We determine the syscall invoked by instruction I_i using mapping $sys(I_i)$. It returns the syscall ID only if the instruction I_i invokes a syscall. Otherwise, it returns nothing. Next, Definition 4.2 formally defines the required syscall set ϕ.

Definition 4.2. *(**Required Syscall Set** $\phi(P, I_i)$) Given an instruction $I_i \in P$, we can determine the required syscall set $\phi(P, I_i)$ for the program P such that for any $\tau \in T$, if and only if there exists an instruction I_j satisfying $I_j \succcurlyeq I_i$ and $sys(I_j) = sc$, then $sc \in \phi(P, I_i)$.*

The required syscall set $\phi(P, I_i)$ is a minimal set including all syscalls that the program may invoke when it executes instruction I_i and I_i's subsequent instructions among all traces T. Syscalls in $\phi(P, I_i)$ should be permitted to guarantee that the program can normally execute after I_i.

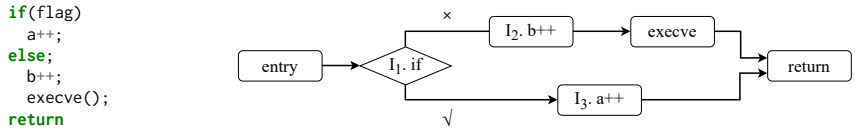


Fig. 3. An example of a reduction position. $r(I_1, I_3)$ is a reduction position of restricting execve, but $r(I_1, I_2)$ is not a reduction position in this example.

It is known that the required syscall set monotonically reduces during programs' execution. The position where the required syscall set reduces is defined as reduction position $r(I_i, I_j)$, with its set denoted as $\mathbb{R}$.

Definition 4.3. *(**Reduction Position Set** $\mathbb{R}$) If $r(I_i, I_j) \in \mathbb{R}$ is a reduction position, there should exist $\tau \in T$ that satisfies $I_i = \tau[a], I_j = \tau[a+1]$, and the required syscall set $\phi(P, I_j) \subsetneq \phi(P, I_i)$.*

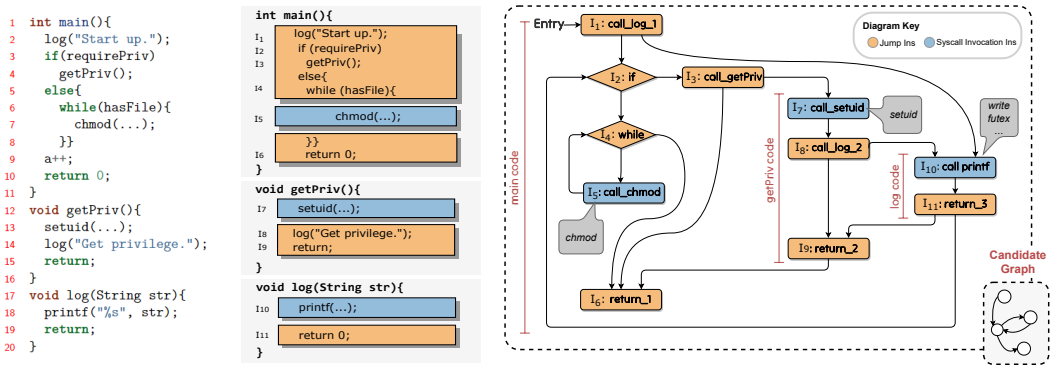
The reduction position $r(I_i, I_j)$ is a pair of two adjacently executed instructions, indicating that the required syscall set of program reduces when the program runs from I_i to I_j. Take the program

in Figure 3 as an example. When it executes from "`if`" to "`a++`", `execve` should be removed from the required syscall set of the program. However, from "`if`" to "`b++`", `execve` is still necessary for the program, and the required syscall set does not change. Thus, only pair $r(I_1, I_3)$, i.e., the edge from node I_1 to node I_3, can be identified as a reduction position.

In Definition 4.1, we utilize the relation $\geqslant$ to represent the execution order between instructions, based on which we can resolve the required syscall set and reduction positions according to Definition 4.2 and Definition 4.3 respectively. Therefore, the most crucial part of building DynBox is to determine the successor relation $\geqslant$ of instructions. To resolve the $\geqslant$ relation, we aim to transfer the program to the partially ordered graph, where the $\geqslant$ relation of instructions can be easily determined. The detailed analysis steps are introduced in the following sections.

4.2 Candidate Graph Building

To identify reduction positions, we first build the candidate graph $CG(\psi, S)$. The nodes ψ of CG are instructions whose execution may lead to the reduction of the required syscall set, and they are indicated as *candidate set-reduce instructions*. The nodes should be either *syscall invocation instructions* or *jump instructions*. Then, edges S are added according to the control flow of the program. Each edge $s(I_i, I_j) \in S$ represents a candidate reduction position of program P. Such CG construction is the first step of the partial order analysis to find real reduction positions. It is proved that our analysis based on CG considers all possible execution traces of a program (Section 4.2.3). This is because CG completely represents the control flow of the program, and each program's trace $\tau \in T$ can be represented as a traverse path on CG.



(a) Source Code (b) Candidate Set-Reduce Instruction (c) Candidate Graph

Fig. 4. An example for the candidate graph. From the source code, the partial order analysis first extracts candidate set-reduce instructions, which consist of *jump instructions* in orange boxes and *syscall invocation instructions* in blue boxes. Then, the candidate graph is built according to the control flow of the program. The syscalls in callouts are utilized by the corresponding interfaces of dynamic libraries. For better visualization, the figure is drawn based on statements rather than instructions.

4.2.1 Nodes of Candidate Graph. The nodes of CG are all *candidate set-reduce instructions* ψ, after whose execution, a syscall may become unnecessary. Formally, $\forall r(I_i, I_j) \in \mathbb{R}, I_i \in \psi$. In this section, we find all ψ from the program as the nodes of CG. Specifically, only in two situations, a syscall can become unnecessary. First, the program runs into a branch in which the program can never reach the invocation instructions of this syscall. Second, this syscall is just invoked for the last time.

Jump Instruction. The first situation only happens when the program changes its control flow with *jump instructions*. As shown in Figure 4, all statements in orange boxes will be compiled as *jump instructions*. One part of them are conditional and unconditional jumps, like the "`if`" and "`while`" statements in Figure 4. Another part of them are jumps caused by call and return of internal functions, such as call and return of function `getPriv` and `log` in Figure 4. Considering the first situation, the analysis collects all these *jump instructions*, including `br`, `switch`, `ret`, and `call` targeting internal functions, and adds them to nodes ψ.

Syscall Invocation Instruction. The second situation may occur when a *syscall invocation instruction* is executed. Thus, *candidate set-reduce instructions* ψ also contain *syscall invocation instructions*. In most cases, they are `call` instructions targeting interfaces of dynamic libraries, like statements of invoking glibc's interfaces `chmod`, `setuid`, and `printf` in blue boxes of Figure 4. This is because programs do not directly invoke syscalls, but utilize high-level interfaces defined in dynamic libraries (glibc, libpthread, etc.). For example, developers usually prefer using `printf` in glibc instead of the `write` syscall when outputting to `stdout`. However, even if a program may have used `write` for the last time during the calling of `printf`, it is not possible to instrument glibc to restrict `write`. The reason is that glibc is utilized by many other programs as well, and the instrumentation in glibc may interrupt their execution. Therefore, we identify the `call` instructions that invoke interfaces of dynamic libraries as *syscall invocation instructions* and add them into ψ.

Moreover, a dynamic library interface may involve multiple syscalls, like `printf` that utilizes several syscalls including `futex` and `write`. Hence, a `call` to a dynamic library interface should be regarded as a node that invokes multiple syscalls, as the callout of `printf` in Figure 4 shows. To obtain the set of syscalls used by each dynamic library interface, we need to preprocess each dynamic library. To handle it efficiently, referencing Claudio et al.'s work [Canella et al. 2021], we utilize angr [Shoshitaishvili et al. 2016] to analyze the binaries of dynamic libraries. Specifically, with angr, a call graph is constructed for a dynamic library based on its "`.so`" file. Then, we set each interface as an entry point for traversing the call graph. During this traversal, the analysis collects all the syscalls that are possibly invoked by each interface. Meanwhile, if one dynamic library relies on another one, we will recursively analyze the required library preferentially. Finally, a mapping from dynamic libraries' interfaces to their required syscalls is established. After determining the syscalls used by *syscall invocation instructions* that invoke dynamic libraries interfaces, we collect them to be *CG*'s nodes. In a few cases, developers may use `syscall` function from "unistd.h" or write assemble code to invoke syscalls, both of which are properly handled in our analysis.

In the end, by identifying all *jump instructions* and *syscall invocation instructions*, our analysis extracts all *candidate set-reduce instructions* ψ as the nodes of *CG*. From ψ, we could further find the real reduction positions.

4.2.2 Edges of Candidate Graph. After collecting all *candidate set-reduce instructions* ψ, the partial order analysis adds the edges between them according to the control flow of the program to build $CG(\psi, S)$. Thus, *CG* is a subgraph of a control flow graph only with ψ as nodes. Moreover, based on Definition 4.3, each edge $s(I_i, I_j)$ of *CG* represents a candidate reduction position that can become a real reduction position if the program's required syscall set changes during the execution from I_i to I_j. Next, we separately introduce how to connect edges for two different types of instructions in ψ.

Jump Instruction. Jump instructions alter the program's control flow according to different conditions, so there should exist edges from them to their jump targets. First, for *jump instructions* except `call` and `ret`, the analysis adds edges from them to all their possible targets. Second, if *jump instruction* I_i is a `call` instruction targeting internal functions, an edge will be connected from it to the target callee function f_i's entry. Meanwhile, each `ret` instruction in f_i should have

an edge pointing to $I_k \in \psi$. Here I_k is the first *candidate set-reduce instruction* that the program executes after returning from f_i. For example, in Figure 4, the function `main` invokes the function `getPriv`, so there is an edge from "I_3: `call_getPriv`" in the function `main` to the callee function `getPriv`'s first instruction "I_7: `call_setuid`". Meanwhile, an edge is also connected from "I_9: `return_2`" in `getPriv` to "I_6: `return_1`" in `main`, which is executed right after returning from `getPriv`. Sometimes, I_i may be an indirect call, for which we utilize the existing well-performed method, SVF [Sui and Xue 2016], to resolve I_i's all possible callees. In addition, for some commonly used dynamic library interfaces with callbacks, like `phread_create` in pthread and `malloc_hook` in glibc, we properly handle them by resolving function pointers to identify the possible callback functions. And edges indicating such control flow are added to *CG*.

Syscall Invocation Instruction. Different from the `call` instructions targeting the program's internal functions, `call` instructions targeting dynamic library interfaces are regarded as *syscall invocation instructions*. As illustrated in Section 4.2.1, for a *syscall invocation instruction* I_i that invokes an external interface f_e, instead of diving into the implementation of f_e, the analysis views the entire execution of f_e as a single node containing the syscalls required by f_e. For example, "I_{10}. `call_printf`" along with the callout describes the calling of `printf` as one node invoking syscalls like `write` and `futex`. Accordingly, the "I_{10}. `call_printf`" is connected to "I_{11}. `return_3`". Likewise, for a *syscall invocation instructions* I_i, the analysis adds an edge from I_i to its next *candidate set-reduce instructions* according to the control flow.

4.2.3 Soundness of CG. In this section, we prove that our analysis based on *CG* is sound and considers a program's all possible execution traces, because *CG* is built based on the control flow of the program, and we can map each possible trace to a traverse path on *CG*, as shown in Theorem 4.4. For better illustration, we utilize the mapping $\Psi(I_i)$ to associate the instruction I_i with its closest candidate set-reduce instruction. $\Psi(I_i) = I_i$ when $I_i \in \psi$.

Theorem 4.4. (**Soundness of Candidate Graph**) *For a trace $\tau = (I_i, I_j, ..., I_n)$ of the program P, we could find a corresponding traverse path $(\Psi(I_i), \Psi(I_j), ..., \Psi(I_n))$ on CG built from P.*

According to Section 4.2.1, ψ contains all jump instructions of a program, so there is no jump instruction between $\Psi(I_i)$ and I_i, which means $\Psi(I_i)$ should map I_i to exactly one candidate set-reduce instruction. Then, for any two adjacent instructions $I_i = \tau[a]$ and $I_j = \tau[a+1]$, they are either mapped to the same instruction in ψ, or they are mapped to two instructions connected by edge $s(\Psi(I_i), \Psi(I_j))$ of *CG*. Thus, there exists a traverse path $(\Psi(I_i), \Psi(I_j), ..., \Psi(I_n))$ on *CG* for the trace $\tau = (I_i, I_j, ..., I_n)$. In conclusion, based on *CG*, the following steps of our analysis can cover all possible execution traces of a program and will not omit any reduction position.

4.3 Partially Ordered Graph Construction

In this section, we transfer *CG* to the partially ordered graph $PG(N, E)$ to resolve the execution order of instructions. As some illusory traces cause precision degradation, we first adopt critical syscall sniffing to improve the analysis precision during the *PG*'s construction. Then, the analysis maps *candidate set-reduce instructions* ψ of *CG* to the nodes *N* of *PG*, and transfers edges *S* of *CG* to edges *E* of *PG*. We further define the $\succcurlyeq$ relation based on the reachability of two nodes on *PG*. If there is a path on *PG* from n_a to n_b, then $n_b \succcurlyeq n_a$. By design, the nodes in *PG* are partially ordered under the $\succcurlyeq$ relation, assuring that any two nodes either have one determined the $\succcurlyeq$ relation, or they have no relation. Moreover, it is demonstrated in Theorem 4.7 that the $\succcurlyeq$ relation reflects the execution order of instructions. Thus, $n_b \succcurlyeq n_a$ indicates that each instruction mapped to node n_a can only be executed before each instruction mapped to n_b once both instructions are executed during a single run of the program. Finally, the execution order between instructions can be resolved

based on the mapping between *candidate set-reduce instructions* and nodes of PG. As shown in Theorem 4.6, the construction of PG ensures that the $\succcurlyeq$ relation between every two instructions is not omitted. Therefore, the analysis will not omit any required syscalls after any instructions, based on which the syscall restriction will not interrupt the program's normal execution.

The $\succcurlyeq$ relation between two instructions I_i and I_j has three cases. First, I_i and I_j are the successors of each other, where the required syscall set of two instructions are the same. This is typically the case for instructions in a loop or recursion. When a syscall is invoked within a loop or recursion, it will be continuously required until the program exits the loop. As a result, there are no reduction positions among these instructions. Second, if either $I_i \succcurlyeq I_j$ or $I_j \succcurlyeq I_i$, there may exist a reduction position between these instructions. Third, I_i and I_j have no relation, which usually indicates that two instructions are in two exclusive branches, like branches after "`if`" in line 3 of source code in Figure 4. Hence, no reduction position can be found between I_i and I_j. In conclusion, to locate reduction positions, the partially ordered graph is built to identify which pairs of instructions have the relation in the second case, i.e., exactly one $\succcurlyeq$ relation holds between two instructions.

Table 1. ID and name of critical syscalls. They are displayed in three categories according to their functionality.

Network Operation				Permission Control				Command Execution			
ID	Name	ID	Name	ID	Name	ID	Name	ID	Name	ID	Name
42	connect	49	bind	10	setuid	106	mprotect	57	fork	322	execveat_getscheduler
43	accept	50	listen	90	setgid	113	chmod	59	execve	435	clone
44	sendto	53	socket	91	setreuid	114	fchmod	101	ptrace		
45	recvfrom	288	accept4	105	setregid						

4.3.1 Critical Syscall Sniffing. We introduce critical syscall sniffing to mitigate the precision degradation caused by illusory traces. Figure 4 displays an example of illusory trace, which is $(I_1, I_2, I_3, I_7, \underline{I_8}, I_{10}, I_{11}, I_2, I_3, \underline{I_7}, I_8, I_9, I_6)$. As shown by the underlined instructions, it is possible for "I_7" to be executed after "I_8", indicating that instruction "I_7: `call_setuid`" is the successor of "I_8: `call_log_2`". Therefore, the pair $r(I_7, I_8)$ is not a reduction position where we can restrict `setuid`. According to the trace, `setuid` should be restricted at $r(I_8, I_9)$. However, according to the source code, `setuid` is only invoked once and should be restricted at reduction position $r(I_7, I_8)$. Thus, these illusory traces compromise the precision of analysis, causing delays in restricting critical syscalls. This is because the partial order analysis is context-insensitive and cannot distinguish the caller when analyzing in the function `log`.

The basic idea of critical syscall sniffing is to improve the analysis precision of critical syscalls by sacrificing the analysis precision on other syscalls. This strategy is designed based on two observations. First, after analyzing extensive malicious payloads, we find some critical syscalls are extremely important to both attackers and operating systems. As shown in Table 1, critical syscalls are usually related to the network IO, permission control, and command execution, on which a tight restriction can significantly limit the ability of attackers. As for other syscalls, such as `read` and `write`, they are responsible for basic functionalities that typically have limited impact if exploited. Meantime, these syscalls tend to be required throughout the program's entire lifecycle and rarely become unnecessary. Second, many functions defined for general and basic operations (such as writing logs and pushing tasks into a queue) are frequently used, resulting in extensive illusory traces. Meanwhile, these functions only need insensitive syscalls such as `read, write` and `mmap`. It is reasonable to concentrate more on the critical syscalls and cluster functions only using insensitive syscalls to improve the analysis precision.

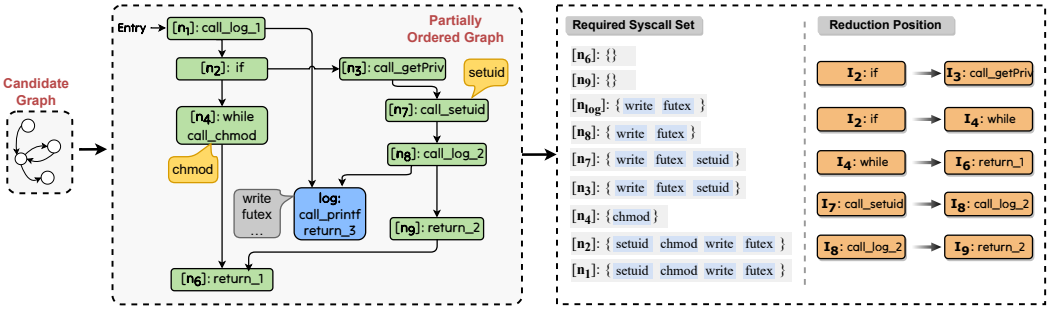


Fig. 5. Examples of partially ordered graph construction and DynBox Enhancement. First, the candidate graph in Figure 4 is transferred to the partially ordered graph. Then, based on the graph, the required syscall set and reduction positions are resolved. The syscalls in yellow callouts are critical ones. The dark-blue box is the *cluster node* of the function `log`. The required syscall sets for nodes are listed in reverse topological order. Reduction positions are pairs of instructions from Figure 4.

We utilize the mapping $isInCluster(I_i)$ to determine whether I_i on CG should be clustered into a *cluster node* on PG. If I_i belongs to a function that invokes no critical syscalls, then I_i and all instructions from the same function of I_i are clustered into a *cluster node*. Meanwhile, since we do not aim to analyze the required syscalls of instructions clustered to a *cluster node*, edges from *cluster node* to others are ignored. As a result, illusory traces are removed during critical syscall sniffing. Specifically, during the execution of a function, if it does not invoke any critical syscall, it will be clustered. Moreover, if there is a recursion during which no critical syscall is invoked either, all functions in the recursion will be clustered to a node. For example, the function `log` in Figure 4 is clustered to a dark-blue *cluster node* n_{log} in the partially ordered graph of Figure 5, and node n_{log} in Figure 5 has no outgoing edges. By doing so, the analysis will not timely restrict syscalls in the function `log`, so the restriction of syscalls `write` and `futex` are delayed until the program returns from the function `log`. However, as the illusory traces related to instructions of the function `log` are removed, critical syscalls like `setuid` can be restricted timely.

4.3.2 Node Mapping and Edge Transferring. To construct $PG(N, E)$, *candidate set-reduce instructions* ψ in $CG(\psi, S)$ are mapped to nodes N of PG according to the reachability of instructions on CG. If two instructions are reachable to each other on CG, they will be mapped to the same node on PG. In terms of instructions clustered by critical syscall sniffing, they will be regarded as a *cluster node* on PG. The edges E on PG ensure that nodes N of PG are partially ordered under the $\succcurlyeq$ relation defined in Definition 4.5.

First, based on the reachability of instructions on CG, we map instructions from CG to nodes of PG through mapping $\mu(CG, I_i)$, which is presented in Equation 1.

$$\mu(CG, I_i) = \begin{cases} n_f, & isInCluster(I_i). \\ n_i, & \nexists j, I_j \xrightarrow{CG} I_i \wedge I_i \xrightarrow{CG} I_j. \\ \mu(CG, I_j), & \exists j, I_j \xrightarrow{CG} I_i \wedge I_i \xrightarrow{CG} I_j. \end{cases} \tag{1}$$

For each instruction $I_i \in \psi$, we first determine whether it should be mapped into a *cluster node* on PG based on mapping $isInCluster(I_i)$. According to the critical syscall sniffing strategy in Section 4.3.1, if I_i belongs to a function f invoking no critical syscalls, then $isInCluster(I_i) = true$ and I_i will be mapped to a *cluster node* n_f along with all instructions in f. Subsequently, we verify if there exists an instruction I_j such that both I_i and I_j are reachable from each other on CG. If I_j

exists, I_j and I_i will be mapped to the same node in PG, otherwise, a new node n_i is generated for I_i. Here, $I_i \xrightarrow{CG} I_j$ indicates that there is a path on CG from I_i to I_j.

After creating nodes of DynBox, we transfer all edges S of $CG(\psi, S)$ to edges E of $PG(N, E)$. In detail, for an edge $s(I_i, I_j) \in S$, if $isInCluster(I_i) = false$, and I_i and I_j are separately mapped to distinct nodes n_a and n_b on PG, we transfer $s(I_i, I_j)$ from CG to edge $e(n_a, n_b)$ on PG. When $isInCluster(I_i) = true$, then $s(I_i, I_j)$ will be ignored during edge transferring. It implies that partial order analysis disregards the edges from *cluster nodes* to other nodes, because the analysis is not intended to precisely identify reduction positions in these clustered functions. Note that the edge targeting a cluster node is preserved, as the edge $e(n_8, n_{log})$ in Figure 5 shows.

Here we illustrate the complexity of PG construction with node mapping and edge transferring. The most consuming part of the construction is to determine reachability between ψ on CG. Through caching intermediate results appropriately, the complexity of resolving all inter-connected nodes is $O(m + n)$, where n is the size of ψ and m is the number of CG's edges. Taking into account that establishing edges of PG requires traversing all edges on CG, the overall complexity for constructing the program graph is $O(m + n)$.

Based on the aforementioned node mapping and edge transferring, we can construct PG, on which the execution order of the program's instructions can be easily resolved. To this end, we first define the $\succcurlyeq$ relation between PG's nodes, as Definition 4.5 depicts.

Definition 4.5. *(**Relation $\succcurlyeq$ on PG's Nodes**) For n_a and n_b on PG, the relation $n_b \succcurlyeq n_a$ holds if and only if there is a traverse path from n_a to n_b on PG.*

Intuitively, the $\succcurlyeq$ relation is established based on the reachability between PG's nodes. A traverse path on PG from n_a to n_b implies $n_b \succcurlyeq n_a$. For example, in Figure 5, PG has a path that from node n_2 to node n_7, indicating $n_7 \succcurlyeq n_2$.

Under the $\succcurlyeq$ relation, PG has two important properties that facilitate the resolution of instructions' execution order. First, as demonstrated in Section 4.3.3, the $\succcurlyeq$ relation of PG's nodes can directly represent the execution order of two instructions. To be specific, $n_b \succcurlyeq n_a$ indicates that the instructions mapped to n_a can be executed before instructions mapped to n_b. Then, PG promises that instructions mapped to n_a will never be executed after instructions mapped to n_b. It is because PG's nodes are partially ordered under the $\succcurlyeq$ relation, which means that for two different nodes, if $n_b \succcurlyeq n_a$ holds, then $n_a \not\succcurlyeq n_b$. For instance, we have $n_7 \succcurlyeq n_2$ in Figure 5. Thus, the "`if`" statement in node n_2 can be executed before the `setuid`'s invocation in node n_7, and "`if`" statement will never be executed after "`call_setuid`". The above two properties enable quick calculation of instruction execution order, allowing for the resolution of the required syscall set to be streamlined.

4.3.3 Soundness and Partial Order Property of PG. Here we prove two properties of PG. First, as illustrated in Theorem 4.6, the $\succcurlyeq$ relation can be used to *over-approximate* the execution order of instructions, which means the execution order between any two instructions can be resolved on PG. Thus, the required syscall set resolved based on PG will not omit any syscalls. Second, Theorem 4.7 shows that PG is *partially ordered* under relation $\succcurlyeq$, which facilitates subsequent analysis.

Theorem 4.6. *(**Soundness of PG**) For any two instructions $I_i, I_j \in \psi, I_j \succcurlyeq I_i$, and $isInCluster(I_i) = false$, if $n_a = \mu(CG, I_i)$, $n_b = \mu(CG, I_j)$ and $n_a \neq n_b$, then we have $n_b \succcurlyeq n_a$. Otherwise, I_i and I_j are mapped to the same node.*

Theorem 4.6 demonstrates that PG preserves the execution order of any two instructions except ones that belong to a clustered function, which is proved in the following. According to the Definition 4.1 and Theorem 4.4, for instructions $I_i, I_j \in \psi, I_j \succcurlyeq I_i$, there exists traverse paths on CG from I_i to I_j. Assuming that no instruction between I_i and I_j is clustered on these traverse paths, for

any two adjacent instructions between I_i and I_j, they should either be mapped to the same node on PG, or an edge $s \in S$ between them will be transferred to an edge $e \in E$ on PG based on Equation 1 and edge transferring algorithm. Thus, there exists a corresponding traverse path from n_a to n_b on PG. In addition, on traverse paths of CG from I_i to I_j, some instructions between I_i and I_j may be clustered. According to the construction of PG, a corresponding traverse path can be found on PG by ignoring clustered instructions. In conclusion, for any two instructions $I_j \succcurlyeq I_i$, we can find a traverse path from $n_a = \mu(CG, I_i)$ to $n_b = \mu(CG, I_j)$ on PG. If $n_a \neq n_b$, then $n_b \succcurlyeq n_a$. Thus, PG preserves all $\succcurlyeq$ relations between the program's instructions, ensuring that the subsequent analysis will not omit any required syscalls.

Theorem 4.7. *(**Partially Ordered Property of** PG**)** PG*'s nodes* N *form a partially ordered set with respect to the relation* $\succcurlyeq$.

In order for Theorem 4.7 to hold, $(N, \succcurlyeq)$ must satisfy reflexivity, transitivity, and antisymmetry. First, according to Definition 4.5, for $n_a \in N$, we have $n_a \succcurlyeq n_a$, so reflexivity holds. Second, we prove that PG holds transitivity. Assuming that $n_b \succcurlyeq n_a$ and $n_c \succcurlyeq n_b$, we can obtain two traverse paths on CG from I_i to I_j and I_j to I_k, where $n_a = \mu(CG, I_i), n_b = \mu(CG, I_j)$, and $n_c = \mu(CG, I_k)$. Thus, there is a path from I_i to I_k on CG, which implies that $n_c \succcurlyeq n_a$ by Theorem 4.6. Third, we provide proof of antisymmetry. When $n_a \succcurlyeq n_b$ and $n_b \succcurlyeq n_a$, according to Section 4.3.2, I_i and I_j should be reachable to each other on CG where $n_a = \mu(CG, I_i)$, $n_b = \mu(CG, I_j)$, so $n_a = n_b$. In conclusion, with the antisymmetry, reflexivity, and transitivity, $(N, \succcurlyeq)$ is a non-strict partially ordered set. Thus, if PG's two distinct nodes $n_b \succcurlyeq n_a$, we can infer that $n_a \not\succcurlyeq n_b$, which means for any two instructions $I_i, I_j \in \psi$, $n_a = \mu(CG, I_i)$ and $n_b = \mu(CG, I_j)$, we have $I_j \succcurlyeq I_i$ and $I_i \not\succcurlyeq I_j$.

4.4 Dynamic Syscall Sandbox Enhancement

Based on the partially ordered graph PG, we can get the required syscall set after each instruction and find reduction positions. (1). The partial order analysis resolves the required syscall set *after instructions* I_i, which is termed as $\phi(P, I_i)$ in Definition 4.2. First, we define the required syscall set *for* PG*'s node* n_a, denoted as $\phi(PG, n_a)$ (Definition 4.8). Since the $\succcurlyeq$ relation on PG's nodes preserves the execution order of instructions, in Section 4.4.4, we prove that $\phi(PG, n_a)$ can be utilized to over-approximate $\phi(P, I_i)$ when I_i is mapped to n_a. To calculate the $\phi(PG, n_a)$, the analysis establishes a topological order for PG's nodes with the partial order property of PG. By processing these nodes in reverse topological order, we can efficiently calculate $\phi(PG, n_a)$ by reusing the pre-computed results of n_a's successor nodes. (2). The analysis checks the required syscall set of every two adjacent instructions, and marks the positions where the syscall set reduces as reduction positions. (3). We enforce dynamically-adjusted syscall sandbox policy with instrumentations at the reduction positions to build DynBox for the program.

4.4.1 Required Syscall Set Resolution. First, we introduce the required syscall set of a PG's node n_a, which is denoted as $\phi(PG, n_a)$ in Definition 4.8.

Definition 4.8. *(**Required Syscall Set of** PG**'s Node** $\phi(PG, n_a)$**)** If there exists an instruction I_i that invokes syscall sc, and I_i is mapped to the node n_a or n_a's successor node n_b, then we can conclude that $sc \in \phi(PG, n_a)$.*

Intuitively, $\phi(PG, n_a)$ contains syscalls that are invoked by each instruction I_i mapped to n_a and all required syscalls set for n_a's subsequent nodes, as shown in Equation 2.

$$\phi(PG, n_a) = \left(\bigcup_{n_b \succcurlyeq n_a} \phi(PG, n_b) \right) \cup \left(\bigcup_{\mu(CG, I_i) = n_a} sys(I_i) \right) \tag{2}$$

We utilize $\phi(PG, n_a)$ to over-approximate $\phi(P, I_i)$ when I_i is mapped to n_a. Formally, for an instruction $I_i \in \psi$ mapped to n_a, we have $\phi(P, I_i) \subseteq \phi(PG, n_a)$. As demonstrated in Theorem 4.9, this approximation is sound, and no required syscall is omitted. This is because all successor instructions of I_i are mapped to n_a or n_a's successor nodes during the construction of PG. Thus, $\phi(PG, n_a)$ must contain all syscalls invoked behind I_i's execution. To compute $\phi(PG, n_a)$, we propose the Algorithm 1 which processes nodes in reverse topological order. Relying on the partial order property of PG, the topological order can be easily built. It is promised that the required syscall sets for the n_a's successors have been resolved and can be reused when calculating $\phi(PG, n_a)$.

Algorithm 1: Required Syscall Set Resolution.

Input : PG : Partially Ordered Graph of Program P.
Output: $\phi(PG, N)$: Required Syscall Set For Nodes of PG.

```
1  que ← Queue();
2  for n_a in PG do
3      if n_a.numberOfOutgoingEdges = 0 then
4          que.push(n_a);
5  while not que.empty() do
6      n_a ← que.front();
7      que.pop();
8      for n_x in n_a.getDirectPredecessors() do
9          n_x.numberOfOutgoingEdges ← n_x.numberOfOutgoingEdges − 1;
10         if n_x.numberOfOutgoingEdges = 0 then
11             que.push(n_x);
12     φ(PG, n_a) ← ∅;
13     for n_b in n_a.getDirectSuccessors() do
14         φ(PG, n_a) ← φ(PG, n_a) ∪ φ(PG, n_b);
15     for I_i in n_a.getInstructions() do
16         φ(PG, n_a) ← φ(PG, n_a) ∪ sys(I_i);
```

In detail, Algorithm 1 needs the partially ordered graph PG as input and calculates the required syscall set $\phi(PG, N)$ for nodes N of PG. The algorithm utilizes a queue to perform topological sorting and guarantee that all nodes are processed in a reverse topological order. In lines 1-4, the algorithm first pushes the nodes that have no outgoing edges into the queue. On PG of Figure 5, nodes n_{log} and n_6 are first pushed into the queue. For n_a at the head of the queue, from line 6 to 10, the algorithm finds each of its predecessors n_x and decreases the counter of n_x's outgoing edge. If the counter becomes zero after decrementing, n_x will be added to the end of the queue. Then, for each n_a's direct successor n_b, the algorithm merges n_b's required syscall set $\phi(PG, n_b)$ into $\phi(PG, n_a)$, as shown in line 13. Since n_b is sorted behind n_a in topological order, $\phi(PG, n_b)$ has been solved when calculating $\phi(PG, n_a)$. For example, in Figure 5, when processing node n_2, nodes n_3 and n_4 have been processed. Next, as shown in lines 14-15, by including syscalls invoked by instructions mapped to n_a, $\phi(PG, n_a)$ is resolved. After processing each node in the queue until it is empty, we obtain the required syscall set for each node. Finally, the required syscall set after each instruction can be obtained by finding the node that the instruction is mapped to.

Algorithm 2: Reduction Position Solving.

Input : CG, PG : Candidate Graph and Partially Ordered Graph of Program P.
Output : $\mathbb{R}$: Reduction Positions.

```
1 for s in CG.edges do
2     I_i ← s.from, I_j ← s.to;
3     n_a ← μ(CG, I_i), n_b ← μ(CG, I_j);
4     if φ(PG, n_b) ⊊ φ(PG, n_a) then
5         ℝ.add(r(I_i, I_j));
```

4.4.2 Reduction Position Solving. The analysis solves the reduction positions where the required syscall set reduces with Algorithm 2. The algorithm needs CG and PG as inputs and outputs reduction positions $\mathbb{R}$. In detail, as shown in line 1, the algorithm traverses each CG's edge s, which is a candidate reduction position. Then, in lines 2-3, for edge $s(I_i, I_j)$, the algorithm determines that instruction I_i and I_j are respectively mapped to nodes n_a and n_b on PG. Subsequently, from line 4 to 5, we use the required syscall sets for node n_a and n_b to over-approximate those of instruction I_i and i_j correspondingly. If $\phi(PG, n_b)$ is strictly smaller than $\phi(PG, n_a)$, then $r(I_i, I_j)$ should be a reduction position. For example, in Figure 4, for two nodes "I_4: `while`" and "I_6: `return_1`" connected by an edge, they are respectively mapped to nodes n_4 and n_6 on PG of Figure 5. Then, we find the required syscall set of n_6 is strictly smaller than that of n_4, due to the absence of `chmod`. Thus, $r(I_4, I_6)$ is the reduction position for restriction `chmod`. After solving reduction positions, we can dynamically adjust the sandbox policy at these positions to build DynBox.

4.4.3 Policy Enforcement. At reduction positions, the partial order analysis enforces new policies to restrict unnecessary syscall to build DynBox. Specifically, at each reduction position, we instrument a call instruction that invokes a function `addPolicy`, which we encapsulate in a dynamic library and link to the enhanced program. This function enforces the policy of Seccomp BPF using libseccomp [LibSeccomp 2023]. For example, as Figure 5 shows, between instruction "I_2: `if`" and "I_3: `call_getPriv`", the required syscall set varies, so we instrument a call targeting `addPolicy` to enforce the policy of restricting `chmod`. Finally, when the enhanced program executes to these reduction positions, it calls `addPolicy` and adds new policies to the BPF program to restrict unnecessary syscalls.

During the instrumentation, we need to handle three situations for a reduction position $r(I_i, I_j)$. (1). If I_i is a syscall invocation, like $r(I_7, I_8)$ in Figure 5, we can directly conduct instrumentation after I_i, so the program will enforce the policy when executing from I_i to I_j. For instance, between node "I_7: `call_setuid`" and "I_8: `call_log_2`", `setuid` is restricted after its last usage. (2). I_i may be a *jump instruction* except `call` and `return`, like $r(I_2, I_4)$ in Figure 5. In this situation, the analysis inserts a new basic block bb between node "I_2: `if`" and "I_4: `while`", and redirect the jump (I_2, I_4) into jumps (I_2, bb) and (bb, I_4). On bb, the analysis enforces the sandbox policy. It is because that node "I_2: `if`" has two jump targets, and "I_4: `while`" also has two predecessors, as shown in Figure 4. By adding bb, we can ensure that the policy will only be enforced when the program runs from statements "`if`" to "`while`". (3). I_i can be a function `call` or `ret`, which may also have multiple targets due to indirect calls. Assuming only when I_i invokes function f, the syscall sc should be restricted. For precise restriction of sc, we instrument the program to enforce different policies according to the different call targets of I_i. Specifically, we make instrumentation before the `call` instruction I_i to verify whether the target function is f. If so, we apply the syscall restriction for sc. As for `ret`, we instrument before `ret` and check where f should return to.

4.4.4 Soundness of DynBox. To construct DynBox, we utilize the required syscall set for PG's nodes to over-approximate the required syscall set after instructions. As Theorem 4.9 demonstrated, this over-approximation is sound and will not omit required syscalls, so DynBox will not interfere with the program's normal execution.

Theorem 4.9. *(**Soundness of Required Syscall Set** $\phi(P, I_i)$) For $I_i \in \psi$, if $\mu(CG, I_i) = n_a$ and $isInCluster(I_i) = false$, then $\phi(P, I_i) \subseteq \phi(PG, n_a)$.*

If the syscall $sc \in \phi(P, I_i)$, with Definition 4.2, we can find an instruction I_j satisfying $I_j \succcurlyeq I_i, sys(I_j) = sc$. According to Theorem 4.6, $I_j \succcurlyeq I_i$ indicates that $n_b \succcurlyeq n_a$ or $n_a = n_b$ where $n_a = \mu(CG, I_i), n_b = \mu(CG, I_j)$. Then, by Definition 4.8, the required syscall set $\phi(PG, n_a)$ must contain sc. Thus, the over-approximation is sound as any syscalls belonging to $\phi(P, I_i)$ are included in $\phi(PG, n_a)$. With a sound required syscall set, we ensure that the policy enforcement at each reduction position will not restrict potentially required syscalls, and thus DynBox will not interfere with the program's normal execution.

5 IMPLEMENTATION

In the implementation, DynBox is developed using the LLVM framework [Lattner and Adve 2004] and incorporates Seccomp BPF [McCanne and Jacobson 1993]. We compile a program using clang with Link-Time Optimization (LTO) enabled [LLVM 2022]. This allows us to obtain an intermediate representation (IR) file of the entire program, which assists us in conducting inter-procedural analysis. In the process of partial order analysis, we identify suitable reduction positions and carry out instrumentation. During the instrumentation, the analysis incorporates a callback function called `addPolicy`, which is implemented within a dynamic library. We develop the dynamic library based on `libseccomp` [LibSeccomp 2023], which can automatically generate the BPF program according to the specified policies and then load them into the Linux kernel. Finally, we compile the instrumented IR file into a sandbox-enhanced binary file and link it with the dynamic library. During the execution of the enhanced binary, when it reaches a reduction position, it will invoke the callback function `addPolicy`, utilizing `libseccomp` to enforce a new syscall policy into Seccomp BPF, thereby restricting unnecessary syscalls.

6 EVALUATION

The experiments aim to answer the following research questions.

- RQ1. How effective is DynBox in reducing the attack surface and protecting the operating system from malicious payloads?
- RQ2. Which syscalls are restricted by DynBox, and what is the impact of these syscalls in vulnerability exploitation?
- RQ3. What are the size expansion, analysis time, and runtime overhead of DynBox?

Applications. Seven complex applications are used as testbeds in our evaluation, including web servers (Httpd, Nginx), databases (Redis, Memcached, SQLite), a DNS system (Bind), and a compression tool (Tar). All of them are widely used and have complex architectures with an average codebase of 615 KLOC. They are chosen for two reasons. First, we choose different types of applications to validate DynBox's effectiveness. Second, these applications are also used as evaluation testbeds in previous works [Canella et al. 2021; Ghavamnia et al. 2020, 2022]. Considering the over-approximation characteristic of the partial order analysis, we ensure that DynBox will not interrupt the applications' normal execution. All isolated applications can pass the projects' test cases and run in a stable manner during stress tests.

Effectiveness Evaluation. First, we assess the number of permitted syscalls in effectiveness evaluation. As DynBox dynamically adjusts the syscall sandbox policy, it is necessary to evaluate DynBox at different positions during an application's execution. Therefore, we collect a set of vulnerabilities for each application and locate their potential exploitation positions. At each exploitation position of a vulnerability, we calculate the currently permitted syscalls, and then get an average number of permitted syscalls for an application among all exploitation positions. The vulnerabilities used in our evaluation are gathered from the Common Vulnerabilities and Exposures (CVE) library [MITRE 2022] and projects' mailing lists. We collect 43 critical vulnerabilities, which are recently discovered and exploitable. Meanwhile, we collect possible exploitation positions of vulnerabilities to evaluate DynBox at these positions.

Second, we collect a set of malicious payloads and evaluate DynBox's effectiveness in resisting malicious payloads. In this experiment, we adopt the setting of previous works [Canella et al. 2021; Ghavamnia et al. 2020], which compares the permitted syscalls with those required by a malicious payload. If any of the syscalls required by the payload are restricted, the payload can be blocked. Subsequently, for each exploitation position of the application, we can calculate a defense rate on all payloads with respect to the permitted syscalls at this position. Finally, an average defense rate for the application across all its vulnerability exploitation positions is computed to evaluate the overall effectiveness of DynBox on the application. For a fair comparison, our evaluation adopts the same setting of malicious payloads as that of Temp [Ghavamnia et al. 2020]. In detail, we first obtain 567 payloads from Metasploit [Rapid7 2022] and Shell-storm [Storm 2022], and then analyze the syscalls used by each payload. Moreover, as potential evasion attempts using alternative syscalls with the same functionalities to bypass syscall sandbox [Ghavamnia et al. 2020], like substituting `open` with `openat`, we also exhaustively enumerated all possible variants of each payload to test DynBox's performance against such bypass intents. Finally, we have a total of 1726 malicious payloads for effectiveness evaluation.

Overhead Evaluation. For overhead assessment, we utilize the Phoronix test suite [Suite 2023] as the evaluation framework. The performance stress test is conducted on an Ubuntu 20.04 with i7-10700K and 48GiB memory. By assigning an extensive workload to each application in a fixed period, we can measure the average throughput. For each application, we implement warm-up testing several times for a stable state and get the average value of five rounds to avoid randomness.

6.1 Evaluation of Effectiveness

This section presents the evaluation of DynBox's effectiveness. We assess the defense rate of DynBox when it resists the exploitation payloads of vulnerable applications. Meanwhile, the number of permitted syscalls is estimated for detailed analysis. We compare DynBox with three advanced methods, Chestnut [Canella et al. 2021], Temporal Specialization (Temp) [Ghavamnia et al. 2020], and C2C [Ghavamnia et al. 2022].

According to the observation from Temp [Ghavamnia et al. 2020], the lifecycle of an application can be manually divided into the initialization and serving phases. Since some syscalls are only used during initialization, Temp enforces the policy again after initialization to restrict these unnecessary syscalls. To compare with it, we evaluate the defense effectiveness on the serving phase and the whole lifecycle of applications separately. Normally, different vulnerabilities of an application are exploited at different exploitation positions, on which DynBox may permit different syscalls. To assess the effectiveness of DynBox, we calculate the average results across all exploitation positions within each application. Therefore, DynBox's permitted syscalls shown in Table 2 are decimal. As Temp also permits different syscalls in two phases, its whole lifecycle defense rate and permitted syscalls are also average values across all exploitation positions of each application.

Table 2. The results of effectiveness evaluation. The "Rate" indicates the defense rate of malicious payloads, and the "Count" presents the average number of permitted syscalls. The results of the serving phase are calculated after the initialization of applications, and the "Whole Lifecycle" column shows the results throughout the complete execution of the application. The permitted syscalls of DynBox are decimal values because they are average values of permitted syscalls across different exploitation positions within an application.

Applications	Whole Lifecycle Defense Result						Serving Phase Defense Result					
	Chestnut		Temp		DynBox		Chestnut		Temp		DynBox	
	Rate	Count	Rate	Count	Rate	Count	Rate	Count	Rate	Count	Rate	Count
Nginx	53.48%	104.00	60.28%	102.76	**78.62%**	**76.52**	53.48%	104.00	72.36%	97.00	**82.87%**	**70.00**
Httpd	64.02%	99.00	78.84%	84.29	**83.20%**	**64.57**	64.02%	99.00	84.94%	79.00	**88.59%**	**60.20**
Redis	67.67%	86.00	75.72%	82.00	**83.59%**	**63.92**	67.67%	86.00	75.72%	82.00	**83.59%**	**63.92**
Sqlite	89.98%	74.00	89.98%	74.00	**90.96%**	**71.25**	89.98%	74.00	89.98%	74.00	**90.96%**	**71.25**
Memcached	75.03%	89.00	76.07%	85.45	**85.32%**	**58.00**	75.03%	89.00	76.42%	84.00	**86.15%**	**54.80**
Bind	77.58%	97.00	77.69%	85.00	**81.56%**	**65.00**	77.58%	97.00	77.69%	85.00	**81.56%**	**65.00**
Tar	76.07%	108.00	77.58%	97.00	**80.61%**	**93.24**	76.07%	108.00	77.58%	97.00	**80.61%**	**93.24**
Average	71.98%	93.86	76.59%	87.21	**83.42%**	**70.36**	71.98%	93.86	79.24%	85.43	**84.90%**	**68.34**

Comparison with Chestnut. The comparison result is shown in Table 2. As Chestnut only reduces the attack surface before an application starts, we mainly compare it on the whole lifecycle result, on which DynBox restricts 23.50 more syscalls and outperforms Chestnut by 15.88% in terms of defense rate. This result demonstrates that a fixed sandbox policy achieves limited defense performance. This is because many complex applications need critical syscalls like `execve`. Without dynamically adjusting the sandbox policy, Chestnut needs to permit these syscalls in the applications' entire lifecycle, leading to a loose sandbox policy.

Comparison with Temporal Specialization. Temp [Ghavamnia et al. 2020] mitigates the problem by restricting syscalls once more at a manually selected transition point of the initialization and serving phases in an application. As shown in Table 2, Temp has achieved encouraging results with 76.59% and 79.24% defense rates on the whole lifecycle and serving phase correspondingly. However, by dynamically adjusting the policy at automatically identified reduction positions, DynBox averagely achieves 8.90% and 7.15% higher defense rates respectively. This is because Temp needs to manually specify a function as the *entry* of the serving phase, but such manual intervention is tricky due to the following reasons. First, some applications like Sqlite do not have an obvious *entry* function. Thus, Temp can only set the policy at the beginning of the Sqlite and achieve the same rate as that of Chestnut.

Second, finding correct transition points is hard. For example, on Apache Httpd, Temp chooses the function `child_main`, the entry of the child process, as the beginning of the serving phase. In fact, Httpd has a master process, which remains alive and can be accessed by attackers, so its transition point of finishing initialization is located before the invocation of `child_main`. As CVE-2021-41773 [MITRE 2021] demonstrated, attackers can call `execve` in the master process outside the function `child_main`. Thus, on Httpd, Nginx, and Memcached, Temp cannot cover all vulnerabilities with an improper selection of transition points, leading to a lower whole lifecycle result than that of the serving phase.

Third, programs usually have a more complex lifecycle with several phases. On Redis and Bind, Temp finds proper transition points of initialization and serving phases, at which setting a policy can cover all vulnerabilities we collect. However, in this situation, DynBox inhibits 18.08 and 20.00 more syscalls respectively. The reason is that Temp's sandbox policy remains fixed after the serving phase starts, whereas DynBox can automatically find more reduction positions and build a more stringent sandbox for applications with complex lifecycles, providing better protection for OS.

Table 3. Comparison between C2C and DynBox on applications with different configurations.

Application		Httpd	Httpd	Nginx	Redis	Memcached	Tar	Tar	Tar	Average
Configuration		WordPress	Mediawiki	Zend Server	default	Ubuntu Default	-czvf	-xzvf	-test-label	
Permitted Syscalls	C2C	97.00	95.00	108.00	82.00	83.00	84.00	94.00	73.00	89.50
	DynBox	60.20	60.20	70.00	63.92	54.80	93.24	93.24	93.24	73.61
Defense Rate	C2C	74.62%	77.52%	70.51%	81.87%	76.59%	79.37%	77.58%	86.91%	78.12%
	DynBox	88.59%	88.59%	82.87%	83.59%	86.15%	80.61%	80.61%	80.61%	83.95%

Comparison with C2C. We also compare our work with the C2C [Ghavamnia et al. 2022], a configuration-aware syscall filter generation method. As C2C's defense result is highly related to the configuration of the application, we compare DynBox with it separately, and the results are illustrated in Table 3. The application we used is the same as those adopted in C2C. By removing the call edges that will not be activated under a specific configuration on call graphs, C2C can effectively restrict syscalls that are not needed by applications in specific modes. However, DynBox achieves better defense performance than C2C on all applications except for Tar with arguments `-test-label`. Since C2C still sets fixed policy after the program initializing rather than adjusting the policy according to the control flow, DynBox can reduce more attack surface on most applications.

C2C and DynBox make efforts to build tighter policy from two orthogonal aspects. C2C concentrates on resolving some conditional jump that is related to its configuration but does not automatically adjust the sandbox policy during execution. Thus, DynBox can be complementary to C2C. We build a configuration-aware dynamic sandbox to enhance the current version of C2C, and conduct a preliminary experiment on Tar. The result shows C2C gets a 4.93% higher defense rate when augmented with DynBox. It indicates that C2C and DynBox can complement each other to reduce more attack surfaces during program execution.

6.2 Syscall Analysis

In this section, we focus on the restrictions of critical syscalls to estimate their impact on vulnerability exploitation. Since Temp achieves the best performance in existing methods, we mainly compare DynBox with it in this section.

Critical Syscalls. We first show how DynBox restricts critical syscalls in Table 4. We select 13 critical syscalls in three types from Table 1, which are widely used by malicious payloads and critical to system functionalities. To demonstrate the security impact of abusing these syscalls, we also show the percentage of payloads using the corresponding syscall.

Specifically, more than 25% of payloads require `execve` to execute a malicious command. Both Temp and DynBox handle `execve` well and restrict it in almost all applications, except for Redis and Tar. These two applications require the `execve` during their main workflow, so Temp cannot block this syscall. However, DynBox can block it after applications use it for the last time and mitigate the vulnerability exploitation. We can see similar effectiveness on `clone` and `fork`.

As for the privilege control, though not many payloads abuse these syscalls, DynBox still tightly restricts the `setuid`, `chmod`, and `setgid` for prevention. On some syscalls like `chmod`, Temp totally inhibits it on Redis, while DynBox permits them on a few parts of exploitation positions of Redis. We explore the reason and find that `chmod` is actually required by Redis and should be permitted in certain execution phases. Therefore, Redis fails the test cases due to false-positive restrictions imposed by Temp. On the contrary, DynBox will not raise such false alarms and only restricts the syscall when it is determined to be unnecessary.

Another dangerous syscall set is related to the network. In most circumstances, attackers need to conduct remote attacks, thus requiring many network operations. More than half of the payloads

Table 4. Results of critical syscall analysis. The "Cmd" rows contain syscalls related to the command execution. The "Payloads" column shows the proportion of payloads including the corresponding syscall. The number under each application indicates the restriction level of the corresponding syscall at all vulnerabilities exploitation positions of the application. "1.00" indicates that syscall is forbidden on all vulnerability exploitation positions of an application, and "0.00" means none.

	Syscall \ App	Payloads	Nginx		Httpd		Redis		Sqlite		Memcached		Bind		Tar	
			Temp	Dyn	Temp	Dyn	Temp	Dyn	Temp	Dyn	Temp	Dyn	Temp	Dyn	Temp	Dyn
Cmd	clone	10.43%	0.36	1.00	0.00	1.00	0.00	1.00	1.00	1.00	0.00	1.00	0.00	1.00	0.00	1.00
	execve	27.81%	0.36	0.76	0.71	0.71	0.00	0.38	1.00	1.00	1.00	1.00	1.00	1.00	0.00	0.03
	fork	5.62%	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
Priviledge	chmod	4.17%	0.00	0.00	1.00	0.86	1.00	0.38	0.00	0.00	1.00	1.00	1.00	0.60	1.00	0.00
	mprotect	11.82%	0.00	0.00	0.00	0.00	0.00	0.38	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	setuid	3.48%	0.00	0.52	0.71	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.00	1.00	0.00	0.03
	setgid	0.93%	0.00	0.52	0.71	0.71	1.00	1.00	1.00	1.00	1.00	1.00	0.00	0.60	0.00	0.03
Network	accept	12.46%	0.00	0.00	1.00	0.43	0.00	0.38	1.00	1.00	0.00	0.45	0.00	0.40	1.00	1.00
	bind	25.96%	0.00	0.00	0.71	0.86	0.00	0.38	1.00	1.00	0.00	0.45	0.00	0.00	1.00	1.00
	listen	24.74%	0.00	0.64	0.71	0.86	1.00	0.38	1.00	1.00	0.00	0.45	0.00	0.40	1.00	1.00
	sendto	13.04%	0.00	0.00	0.00	0.00	0.00	0.00	1.00	1.00	0.00	0.45	0.00	0.00	1.00	1.00
	recvfrom	14.66%	0.00	1.00	1.00	0.86	1.00	1.00	1.00	1.00	0.00	0.45	0.00	1.00	1.00	1.00
	socket	52.84%	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.25	0.00	0.45	0.00	0.00	0.00	0.00

invoke `socket`, and the syscall `bind` is utilized by more than a quarter of payloads. Meanwhile, many applications heavily rely on network operations to achieve their basic functionality. Thus, it is hard to restrict them with fixed policies. By adjusting the policy along with the execution of an application, we can restrict syscalls according to its requirement dynamically and reduce the corresponding attack surface timely. For example, on Sqlite and Memcached, DynBox can even restrict `socket`, which effectively restricts the ability of attackers when exploiting.

In conclusion, by dynamically adjusting the sandbox policy with partial order analysis, DynBox can better restrict critical syscalls after their last usage and reduce exposed attack surfaces timely.

Payload Analysis. In addition to the critical syscall analysis, we also investigate the mitigation results of different types of payloads. According to their attack goal, the payloads can be categorized into four types. Specifically, among all payloads, 334 of them are extremely dangerous and want to escalate privilege through syscalls like `setuid` and `chmod`. 436 of them can be used to execute arbitrary commands via syscalls including `execve` and `fork`. 583 payloads intend to communicate with attackers through the network, requiring syscalls like `socket`, `connect`, and `bind`. 373 payloads only perform some basic system operations like accessing files. We calculate the average defense rates of all 7 applications in different categories, which are presented in Figure 6.

In all categories, DynBox has the best performance compared to the state-of-the-art tools. Among them, the most dangerous ones are the payloads of command execution and privilege escalation. To escalate privilege, attackers need vital syscalls like `setuid`, which are well-restricted by both tools. As for the command execution, since some applications like Redis and Tar need `execve` for their core functionalities, approaches like Temp have to permit this critical syscall. In contrast, by automatically identifying the reduction position for restricting `execve`, DynBox achieves an 8.04% higher defense rate than Temp on the category of command execution. As for the network operation, most of these applications' core functionalities heavily rely on the network, making it hard to totally restrict syscalls like `socket` and `connect`. However, DynBox improves by 13.83% in this category compared to Temp, indicating that dynamically restricting unnecessary syscalls can effectively reduce the attack surface of the operating system and limit the ability of attackers. In the category of basic operation, the performance of Temp is relatively limited, since payloads in this

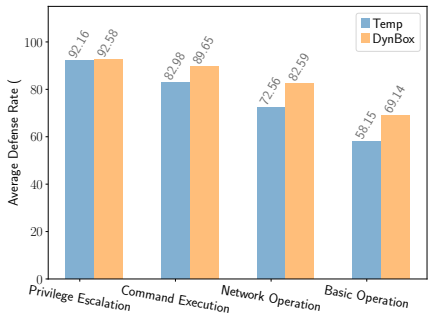


Fig. 6. Average defense rate of Temp and DynBox on 7 applications against malicious payloads from 4 different categories.

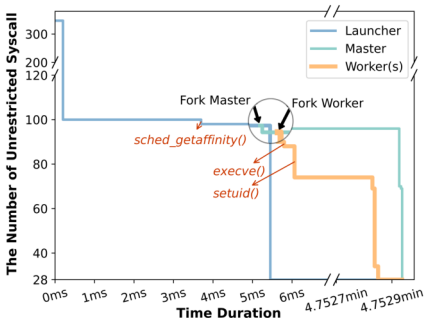


Fig. 7. The variation in the number of permitted syscalls during the execution of Nginx. Each line represents one process.

category usually require some basic operations with syscalls like open. Although extremely critical syscalls like execve are not abused in these payloads, attackers can still cause serious consequences, such as trojan injection or information theft. Thus, these syscalls need to be carefully handled, and DynBox mitigates 18.90% more payloads compared with Temp.

With the detailed analysis of syscalls and payloads, it is demonstrated that critical syscalls play a crucial role in vulnerability exploitation. Meanwhile, DynBox can effectively restrict critical syscalls with dynamical adjustment, and thus resists more malicious payloads.

6.3 Overhead of DynBox

We measure the overhead of DynBox from three aspects, runtime overhead, binary size expansion, and the analysis time. The results are shown in Table 5.

Table 5. The overhead of DynBox. "Binary Expansion" shows the increase in the size of the applications' executables. The "Analysis Time" row shows the seconds needed for the partial order analysis of applications.

	Nginx	Httpd	Redis	Sqlite	Memcached	Bind	Tar
Runtime Overhead (%)	1.83%	0.14%	17.44%	0.75%	5.15%	5.22%	4.96%
Binary Expansion (%)	5.53%	4.31%	0.72%	5.07%	1.72%	2.44%	9.83%
Analysis Time (s)	5.27	3.31	3.67	6.92	1.00	11.52	1.87
Binary Size (MiB)	1.22	1.78	1.18	2.11	0.34	3.96	1.16

The runtime overhead was evaluated with a stress testing benchmark named Phoronix [Suite 2023]. During the stress testing, Phoronix sent extensive workloads to applications, keeping the CPU saturated. On average, DynBox leads to 5.07% more execution time, which is similar to other works [Canella et al. 2021; Ghavamnia et al. 2022]. Such runtime overhead comes from two parts. The first part relates to the instrumentation used for enforcing new policies. Based on our experiment, it takes 0.03ms to add a policy for one syscall on average, which accounts for only a small proportion of the overall overhead. The major overhead is due to the privilege checks, since OS has to run BPF programs on each syscall. Depending on the design and implementation of applications, the overhead varies in our experiment. The runtime overhead of DynBox ranges from 0.14% to 5.22% except for Redis. Redis needs to handle extensive requests in concurrency under Phoronix's stress test, thus invoking a large number of syscalls to handle I/O accesses. As a result, it invokes 19,303 syscalls per second, resulting in a 17.44% runtime overhead. In contrast, DynBox

only introduces 1.83% runtime overhead on Nginx, since it only makes 7,248 syscalls per second. In addition, we evaluate the runtime overhead of Temp and Chestnut, which result in average runtime delays of 5.04% and 4.91% respectively. Given that DynBox introduces 5.07% runtime overhead, which is similar to that of Temp and Chestnut, adjusting the policy multiple times does not incur heavy additional overhead. Therefore, it is reasonable to build DynBox for a stricter restriction.

Table 5 also shows the size of the executable file after instrumentation. Among seven applications, the binary files are slightly increased by 4.23% on average. Such little overhead demonstrates DynBox's high practicability in the real world.

The column "Analysis Time" in Table 5 exhibits the time consumption of the analysis. Overall, DynBox efficiently builds the sandbox within around 10 seconds. Theoretically, the overall complexity of the partial order analysis is $O(m + n)$, where n is the total number of program instructions, and m represents the jumps on the control flow. During *CG* construction, the analysis traverses all instructions and jumps for once. Then, *PG* is produced with in $O(m + n)$, as discussed in Section 4.3.2. Finally, DynBox is built with one traversal of *PG*. Thus, the overall complexity of analysis is $O(m + n)$. The experiments also demonstrate that the analysis time is proportional to the binary size. For example, Bind consumes a relatively longer analysis time than others, since it has the largest binary size as presented in Table 5.

In conclusion, DynBox efficiently isolates complex vulnerable programs and mitigates potential exploitation impact on OS with negligible runtime overhead and binary size expansion.

6.4 Real-World Case Study

In this section, we present how DynBox adjusts the sandbox policy along with the execution of Nginx. The syscall requirement changes during Nginx's lifecycle are presented in Figure 7.

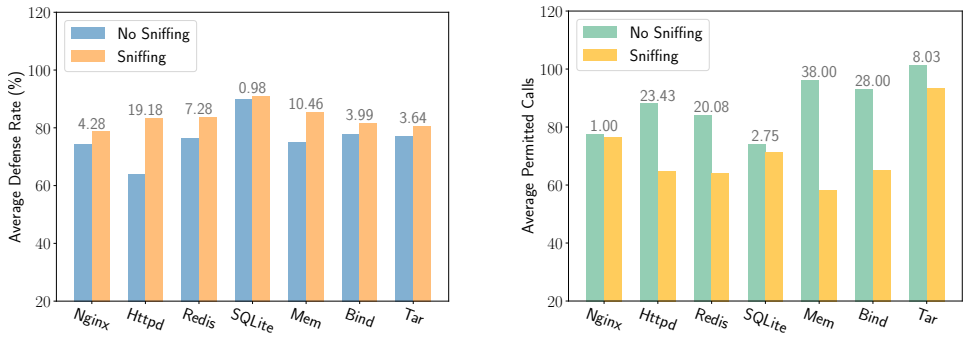
Initially, DynBox restricted 247 syscalls and set an initial sandbox policy with 100 permitted syscalls. Soon, at 3.6ms, DynBox restricted two more syscalls, including `sched_getaffinity`, which is widely used to exploit concurrent vulnerabilities. Then, the launcher process spawned a master process and exited. This operation can make Nginx a daemon process in the system. Then, the master process started with the sandbox policy inherited from the launcher process. Soon, DynBox restricted two more unnecessary syscalls, and Nginx began spawning worker processes.

We only present syscall changes for one of the worker processes spawned by the master, since all the other workers behave similarly. Once a worker process started, many crucial syscalls required by the master process became unnecessary. Therefore, at 5.8ms, syscall `execve` was restricted. Then, after the initialization of the worker process at 6.0ms, syscalls like `setuid` were further confined. After 6.0ms, both the worker and master ran in a stable state for handling requests. As the master needs to manage worker processes, it requires syscalls like `execve` and `setuid`. At about 4.7 minutes, we stopped Nginx, which entered the exiting phase. During this phase, DynBox still gradually restricted unnecessary syscalls until Nginx exited.

In this case, we can observe that programs tend to have complex lifecycles, requiring extensive efforts and expert knowledge to identify different execution phases manually. Therefore, DynBox is urgently needed to automatically and precisely identify reduction positions and dynamically restrict unnecessary syscalls during the program's execution.

7 DISCUSSION

Tradeoff of Critical Syscall Sniffing. In node mapping, we utilize the critical syscall sniffing strategy to remove illusory traces. It may delay the restriction of some uncritical syscalls and degrade the defense rate. However, as Figure **??** shows, it can improve the overall analysis precision. In the experiment, we try to disable the critical syscall sniffing strategy while building DynBox, as the blue and green bars in Figure **??** show. Without the strategy, the partial order analysis cannot



(a) Average Defense Rate (b) Average Permitted Syscalls

Fig. 8. The defense rate and permitted syscalls of DynBox without and with critical syscall sniffing strategy. The number on each of the two bars is the difference between them.

create a robust sandbox for applications. On average, 19.02% more syscalls are permitted to the applications, which causes a 7.12% drop in the average defense rate. Therefore, critical syscall sniffing can eliminate many illusory traces and improve the overall precision of sandbox policy, helping DynBox resist attacks with the highest defense rate among existing methods.

Imprecision of Third-Party Static Analyzers. Although the entire analysis has been proven to be sound theoretically, we utilized two third-party static analyzers - SVF [Sui and Xue 2016] for indirect call resolving and angr [Shoshitaishvili et al. 2016] for dynamic library analysis. These two analyzers are highly effective and convenient but may be imprecise in corner cases, potentially compromising the soundness of the analysis and resulting in false positives in syscall restriction. Nevertheless, we conduct extensive experiments to ensure that DynBox will not interrupt applications' normal execution. In experiments, applications augmented with DynBox pass the unit tests in their repositories and execute stable under the stress tests. Therefore, these two methods will not compromise the over-approximation property of the partial order analysis.

Limitations. Partial order analysis, being an inter-procedural analysis, necessitates the construction of the entire program's intermediate representation (IR) through link-time optimization (LTO) of LLVM [LLVM 2022]. Though LTO may not work optimally on all programs, it is worth noting that many complex applications, such as Chrome and Firefox, have started providing support for LTO in recent years. This support enables DynBox to scale effectively to a wide range of complex programs. Furthermore, some applications may load new dynamic libraries during execution, which is not supported by DynBox currently. We plan to investigate suitable methods to handle such scenarios in future work. In addition, DynBox may not support programs with self-modifying code as it is hard to analyze the possible control flow of such programs. Nonetheless, the majority of programs do not possess this feature.

8 RELATED WORK

Attack Surface Reduction. To protect the operating system, lots of researchers make efforts to reduce its attack surface by removing unnecessary features or code of programs. Early works focus on removing the unnecessary codes from processes' address space so as to prevent attackers from using high-privilege code in functions or libraries that are not needed [Agadakos et al. 2019; Mulliner and Neugschwandtner 2015]. After such removal, attackers' exploitation payloads may fail to execute, and some unknown vulnerabilities can be eliminated. Apart from libraries and

code, researchers also reduce attack surfaces by limiting the accessible resources of a program. Many researchers concentrate on intra-process memory isolation [Bensoussan et al. 1972; Park et al. 2019]. They usually restrict memory access from vulnerable and malicious dynamic libraries with the assistance of hardware mechanisms. Hu et al, [Hu et al. 2018] attempt to lower privilege during the program's execution with temporal reasoning, so that it is difficult for attackers to obtain high-privilege access (e.g., root authority). The above studies reduce attack surfaces from different aspects and can mitigate different attacks. They are orthogonal to DynBox as it reduces attack surfaces by restricting unnecessary syscalls.

Syscall Sandbox. Some early works concentrating on intrusion detection usually use syscall sequences as patterns. They first learn from the benign pattern and then check the syscall sequence pattern [Goldberg et al. 1996] or validate arguments [Jachner and Agarwal 1984; Wagner and Dean 2001] to mitigate attacks. These methods aim to separate the benign and abnormal patterns, which usually suffer from false positives, while the syscall sandbox wants to establish isolation without interrupting the programs' normal execution. In addition to intrusion detection, a few studies also made exploration in syscall sandbox building. Goldberg et al.[Goldberg et al. 1996] are the first to mitigate vulnerability exploitation by restricting the programs' usable syscalls. Then, many following works [Jain and Sekar 2000; Provos 2003; Rajagopalan et al. 2005] are proposed. However, these researchers mainly focus on designing a dependable and efficient isolation mechanism to ensure the integrity of policy rather than generating a suitable policy.

Recently, researchers have started to leverage static analysis to construct a tighter sandbox policy. Zeng et al. [Zeng et al. 2013] utilize binary analysis to build the call graph, on which they collect the reachable functions and get the required syscalls. Sysfilter [DeMarinis et al. 2020] further improves the precision of the analysis. As these mechanisms rely on binary-level analysis, which suffers from severe precision degradation [Canella et al. 2021], we do not use them as the comparison baseline in evaluation. Recent works have started to analyze the source code, such as Chestnut [Canella et al. 2021], which builds a call graph from the source code to collect the required syscalls. By taking program configuration into consideration, C2C [Ghavamnia et al. 2022] can better finetune the sandbox policy. Temporal Specialization (Temp) [Ghavamnia et al. 2020] notices that the execution of programs can be manually divided into initialization and server phases. After the initialization, some syscalls become unnecessary, so Temp enforces the sandbox policy once more to restrict them. Abhaya [Pailoor et al. 2020] leverages abstract interpretation to compute possible syscall arguments and restrict abnormal invocations. We plan to incorporate it in our work to further restrict syscall arguments during dynamic restriction in future work. All these methods only enforce the policy at the early running stage of the program's lifecycle. After that, the sandbox policy remains fixed during the program's subsequent execution. However, many programs need critical syscalls to achieve their functionality in certain execution phases, and therefore existing methods cannot restrict these syscalls. In contrast, using partial order analysis, DynBox automatically identifies reduction positions for restricting unnecessary syscalls throughout the entire lifecycle of the program, producing a dynamic syscall sandbox policy that offers higher security.

9 CONCLUSION

In this paper, we propose DynBox, which can dynamically reduce the attack surface via restricting syscalls along the program's execution. Different from existing methods which set the sandbox policy only at the beginning of the program, we notice that the required syscall set of a program monotonically reduces during the program's entire lifecycle. Thus, we design DynBox to automatically adjust the sandbox policy and only permit minimal necessary syscall set on each instruction. Specifically, we propose the partial order analysis to estimate the execution order of any two instructions. With determined orders, the required syscall set and the reduction positions can

be determined. By tightening the syscall sandbox policy at these reduction positions, DynBox is customized to sandbox a program. We demonstrate the effectiveness of DynBox by evaluating it with exploitable vulnerabilities and payloads. On average, by restricting 23.50, 16.86, and 15.89 more syscalls, DynBox resists 15.88%, 8.90%, and 7.46% more malicious payloads compared with three state-of-the-art tools. In summary, DynBox can isolate the vulnerable program effectively from affecting the operating system with a dynamically adjusted sandbox policy.

DATA-AVAILABILITY STATEMENT

The relevant evaluation presented in this paper was conducted using the prototype which has been packaged and made available on Zenodo [Zhang et al. 2023b] and GitHub [Zhang 2023].

ACKNOWLEDGMENTS

This research is sponsored in part by the National Key Research and Development Project (No. 2022YFB3104000, No2021QY0604) and NSFC Program (No. 62022046, 92167101, U1911401, 62021002, U20A6003).

REFERENCES

Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. 2009. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC)* 13, 1 (2009), 1–40.

Ioannis Agadakos, Di Jin, David Williams-King, Vasileios P. Kemerlis, and Georgios Portokalidis. 2019. Nibbler: debloating binary shared libraries. In *Proceedings of the 35th Annual Computer Security Applications Conference, ACSAC 2019, San Juan, PR, USA, December 09-13, 2019*, David Balenson (Ed.). ACM, 70–83. https://doi.org/10.1145/3359789.3359823

A. Bensoussan, C. T. Clingen, and Robert C. Daley. 1972. The Multics Virtual Memory: Concepts and Design. *Commun. ACM* 15, 5 (1972), 308–318. https://doi.org/10.1145/355602.361306

Claudio Canella, Mario Werner, Daniel Gruss, and Michael Schwarz. 2021. Automating Seccomp Filter Generation for Linux Applications. In *CCSW@CCS '21: Proceedings of the 2021 on Cloud Computing Security Workshop, Virtual Event, Republic of Korea, 15 November 2021*, Yinqian Zhang and Marten van Dijk (Eds.). ACM, 139–151. https://doi.org/10.1145/3474123.3486762

Nicholas DeMarinis, Kent Williams-King, Di Jin, Rodrigo Fonseca, and Vasileios P. Kemerlis. 2020. Sysfilter: Automated system call filtering for commodity software. *RAID 2020 Proceedings - 23rd International Symposium on Research in Attacks, Intrusions and Defenses* (2020), 459–474.

Seyedhamed Ghavamnia, Tapti Palit, Shachee Mishra, and Michalis Polychronakis. 2020. Temporal System Call Specialization for Attack Surface Reduction. In *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, Srdjan Capkun and Franziska Roesner (Eds.). USENIX Association, 1749–1766. https://www.usenix.org/conference/usenixsecurity20/presentation/ghavamnia

Seyedhamed Ghavamnia, Tapti Palit, and Michalis Polychronakis. 2022. C2C: Fine-grained Configuration-driven System Call Filtering. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*, Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi (Eds.). ACM, 1243–1257. https://doi.org/10.1145/3548606.3559366

Ian Goldberg, David A. Wagner, Randi Thomas, and Eric A. Brewer. 1996. A Secure Environment for Untrusted Helper Applications. In *Proceedings of the 6th USENIX Security Symposium, San Jose, CA, USA, July 22-25, 1996*. USENIX Association. https://www.usenix.org/conference/6th-usenix-security-symposium/secure-environment-untrusted-helper-applications

Xiaoyu Hu, Jie Zhou, Spyridoula Gravani, and John Criswell. 2018. Transforming Code to Drop Dead Privileges. In *2018 IEEE Cybersecurity Development, SecDev 2018, Cambridge, MA, USA, September 30 - October 2, 2018*. IEEE Computer Society, 45–52. https://doi.org/10.1109/SecDev.2018.00014

Jacek Jachner and Vinod K. Agarwal. 1984. Data Flow Anomaly Detection. *IEEE Transactions on Software Engineering* SE-10, 4 (1984), 432–437. https://doi.org/10.1109/TSE.1984.5010256

K. Jain and R. Sekar. 2000. User-Level Infrastructure for System Call Interposition: A Platform for Intrusion Detection and Confinement. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2000, San Diego, California, USA*. The Internet Society. https://www.ndss-symposium.org/ndss2000/user-level-infrastructure-system-call-interposition-platform-intrusion-detection-and-confinement/

Vasileios P. Kemerlis, Georgios Portokalidis, and Angelos D. Keromytis. 2012. kGuard: Lightweight Kernel Protection against Return-to-User Attacks. In *Proceedings of the 21th USENIX Security Symposium, Bellevue, WA, USA, August 8-10, 2012*,

Tadayoshi Kohno (Ed.). USENIX Association, 459–474. https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/kemerlis

Linux Kernel. 2022. Seccomp BPF. https://www.kernel.org/doc/html/v4.16/userspace-api/seccomp_filter.html.

Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. San Jose, CA, USA, 75–88. https://doi.org/10.1109/CGO.2004.1281665

Yiwen Li, Brendan Dolan-Gavitt, Sam Weber, and Justin Cappos. 2017. Lock-in-Pop: Securing Privileged Operating System Kernels by Keeping on the Beaten Path. In *2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017*, Dilma Da Silva and Bryan Ford (Eds.). USENIX Association, 1–13. https://www.usenix.org/conference/atc17/technical-sessions/presentation/li-yiwen

LibSeccomp. 2023. libSeccomp. https://github.com/seccomp/libseccomp

LLVM. 2022. Link Time Optimization. https://llvm.org/docs/LinkTimeOptimization.html.

Steven McCanne and Van Jacobson. 1993. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *Proceedings of the Usenix Winter 1993 Technical Conference, San Diego, California, USA, January 1993*. USENIX Association, 259–270. https://www.usenix.org/conference/usenix-winter-1993-conference/bsd-packet-filter-new-architecture-user-level-packet

Dirk Merkel. 2014. Docker: lightweight linux containers for consistent development and deployment. *Linux journal* 2014, 239 (2014), 2.

MITRE. 2016. CVE-2016-0746. https://nvd.nist.gov/vuln/detail/CVE-2016-0746

MITRE. 2021. CVE-2021-41773. https://nvd.nist.gov/vuln/detail/CVE-2021-41773

MITRE. 2022. CVE. https://www.cve.org/.

Collin Mulliner and Matthias Neugschwandtner. 2015. Breaking Payloads with Runtime Code Stripping and Image Freezing. (2015). http://www.mulliner.org/security/codefreeze/1

Santosh Nagarakatte, Jianzhou Zhao, Milo M. K. Martin, and Steve Zdancewic. 2009. SoftBound: highly compatible and complete spatial memory safety for c. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, Michael Hind and Amer Diwan (Eds.). ACM, 245–258. https://doi.org/10.1145/1542476.1542504

Shravan Narayan, Craig Disselkoen, Tal Garfinkel, Nathan Froyd, Eric Rahm, Sorin Lerner, Hovav Shacham, and Deian Stefan. 2020. Retrofitting Fine Grain Isolation in the Firefox Renderer. In *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, Srdjan Capkun and Franziska Roesner (Eds.). USENIX Association, 699–716. https://www.usenix.org/conference/usenixsecurity20/presentation/narayan

Shankara Pailoor, Xinyu Wang, Hovav Shacham, and Isil Dillig. 2020. Automated policy synthesis for system call sandboxing. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020). https://doi.org/10.1145/3428203

Neeraj Pal. 2018. Pledge: OpenBSD's defensive approach to OS Security. https://medium.com/@_neerajpal/pledge-openbsds-defensive-approach-for-os-security-86629ef779ce

Soyeon Park, Sangho Lee, Wen Xu, Hyungon Moon, and Taesoo Kim. 2019. libmpk: Software Abstraction for Intel Memory Protection Keys (Intel MPK). In *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*. USENIX Association, 241–254. https://www.usenix.org/conference/atc19/presentation/park-soyeon

The Linux Documentation Project. 2023. Dynamic Loaded (DL) Libraries. https://tldp.org/HOWTO/Program-Library-HOWTO/dl-libraries.html

Niels Provos. 2003. Improving Host Security with System Call Policies. In *Proceedings of the 12th USENIX Security Symposium, Washington, D.C., USA, August 4-8, 2003*. USENIX Association. https://www.usenix.org/conference/12th-usenix-security-symposium/improving-host-security-system-call-policies

Anh Quach, Aravind Prakash, and Lok Yan. 2018. Debloating Software through Piece-Wise Compilation and Loading. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 869–886. https://www.usenix.org/conference/usenixsecurity18/presentation/quach

Mohan Rajagopalan, Matti A. Hiltunen, Trevor Jim, and Richard D. Schlichting. 2005. Authenticated System Calls. In *2005 International Conference on Dependable Systems and Networks (DSN 2005), 28 June - 1 July 2005, Yokohama, Japan, Proceedings*. IEEE Computer Society, 358–367. https://doi.org/10.1109/DSN.2005.23

Rapid7. 2022. Metasploit. https://www.metasploit.com/.

Charles Reis, Alexander Moshchuk, and Nasko Oskov. 2019. Site Isolation: Process Separation for Web Sites within the Browser. In *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, Nadia Heninger and Patrick Traynor (Eds.). USENIX Association, 1661–1678. https://www.usenix.org/conference/usenixsecurity19/presentation/reis

Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. 2015. Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*. IEEE Computer Society, 745–762. https://doi.org/10.1109/SP.2015.51

Hovav Shacham. 2007. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *Proceedings of the 2007 ACM Conference on Computer and Communications Security, CCS 2007, Alexandria, Virginia, USA, October 28-31, 2007*, Peng Ning, Sabrina De Capitani di Vimercati, and Paul F. Syverson (Eds.). ACM, 552–561. https://doi.org/10.1145/1315245.1315313

Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Krügel, and Giovanni Vigna. 2016. SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*. IEEE Computer Society, 138–157. https://doi.org/10.1109/SP.2016.17

Shell Storm. 2022. Shell-storm. http://www.shell-storm.org/.

Yulei Sui and Jingling Xue. 2016. SVF: Interprocedural Static Value-Flow Analysis in LLVM. In *Proceedings of the 25th International Conference on Compiler Construction* (Barcelona, Spain) *(CC 2016)*. Association for Computing Machinery, New York, NY, USA, 265–266. https://doi.org/10.1145/2892208.2892235

Phoronix Test Suite. 2023. Phoronix Test Suite. https://github.com/phoronix-test-suite/phoronix-test-suite.

Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. SoK: Eternal War in Memory. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*. IEEE Computer Society, 48–62. https://doi.org/10.1109/SP.2013.13

D. Wagner and R. Dean. 2001. Intrusion detection via static analysis. In *Proceedings 2001 IEEE Symposium on Security and Privacy. S&P 2001*. 156–168. https://doi.org/10.1109/SECPRI.2001.924296

Yves Younan, Wouter Joosen, and Frank Piessens. 2012. Runtime countermeasures for code injection attacks against C and C++ programs. *ACM Comput. Surv.* 44, 3 (2012), 17:1–17:28. https://doi.org/10.1145/2187671.2187679

Michał Zalewski. 2016. American Fuzzy Lop Whitepaper. https://lcamtuf.coredump.cx/afl/technical_details.txt

Qiang Zeng, Zhi Xin, Dinghao Wu, Peng Liu, and Bing Mao. 2013. Tailored Application-specific System Call Tables.

Quan Zhang. 2023. DynBox. https://github.com/ZQ-Struggle/DynBox.git

Quan Zhang, Yifeng Ding, Yongqiang Tian, Jianmin Guo, Min Yuan, and Yu Jiang. 2021. AdvDoor: adversarial backdoor attack of deep learning system. In *ISSTA '21: 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, Denmark, July 11-17, 2021*, Cristian Cadar and Xiangyu Zhang (Eds.). ACM, 127–138. https://doi.org/10.1145/3460319.3464809

Quan Zhang, Yongqiang Tian, Yifeng Ding, Shanshan Li, Chengnian Sun, Yu Jiang, and Jiaguang Sun. 2023a. CoopHance: Cooperative Enhancement for Robustness of Deep Learning Systems. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17-21, 2023*, René Just and Gordon Fraser (Eds.). ACM, 753–765. https://doi.org/10.1145/3597926.3598093

Quan Zhang, Chijin Zhou, Yiwen xu, Zijing Yin, Mingzhe Wang, Zhuo Su, Chengnian Sun, Yu Jiang, and Jiaguang Sun. 2023b. Building Dynamic System Call Sandbox With Partial Order Analysis. Zenodo. https://doi.org/10.5281/zenodo.8328524

Chijin Zhou, Lihua Guo, Yiwei Hou, Zhenya Ma, Quan Zhang, Mingzhe Wang, Zhe Liu, and Yu Jiang. 2023. Limits of I/O Based Ransomware Detection: An Imitation Based Attack. In *44th IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, May 21-25, 2023*. IEEE, 2584–2601. https://doi.org/10.1109/SP46215.2023.10179372

Chijin Zhou, Quan Zhang, Mingzhe Wang, Lihua Guo, Jie Liang, Zhe Liu, Mathias Payer, and Yu Jiang. 2022. Minerva: browser API fuzzing with dynamic mod-ref analysis. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*. ACM, 1135–1147. https://doi.org/10.1145/3540250.3549107

Received 2023-04-14; accepted 2023-08-27