Go With the Flow: Enforcing Program Behavior Through Syscall Sequences and Origins

Claudio Canella

claudio.canella@iaik.tugraz.at

Abstract

As the number of vulnerabilities continues to increase every year, we require more and more methods of constraining the applications that run on our systems. Control-Flow Integrity [1] (CFI) is a concept that constrains an application by limiting the possible control-flow transfers it can perform, i.e., control flow can only be re-directed to a set of previously determined locations within the application. However, CFI only applies within the same security domain, i.e., only within kernel or userspace. Linux seccomp [4], on the other hand, restricts an application's access to the syscall interface exposed by the operating system. However, seccomp can only restrict access based on the requested syscall, but not whether it is allowed in the context of the previous one.

This talk presents our concept of syscall-flow-integrity protection (SFIP), which addresses these shortcomings. SFIP is built upon three pillars: a state machine representing valid transitions between syscalls, a syscall-origin map that identifies locations from where each syscall can originate, and the subsequent enforcement by the kernel. We discuss these three pillars and how our automated toolchain extracts the necessary information. Finally, we evaluate the performance and security of SFIP. For the performance evaluation, we demonstrate that SFIP only has a marginal runtime overhead of less than 2 % in long-running applications like nginx or memcached. In the security evaluation, we first discuss the provided security of the first pillar, i.e., the syscall state machine. We show that SFIP reduces the number of possible syscall transitions significantly compared to Linux seccomp.. In nginx, each syscall can, on average, reach 39 % fewer syscalls than with seccomp-based protection. We also evaluate the provided security of the second pillar, i.e., the syscall-origin map. By enforcing the syscall origin, we eliminate shellcode entirely while constraining syscalls executed during a Return-Oriented Programming attack to legitimate locations.

1 Overview

This whitepaper covers our talk's topics and provides technical background. The whitepaper is a pre-print of our paper "SFIP: Coarse-Grained Syscall-Flow-Integrity Protection in Modern Systems" [2]. It presents our talk's content in more detail, such as the three pillars of SFIP and the challenges in automatically extracting the required information. It also provides detailed information on

how our implementation solves these challenges in our public proof-of-concept [3] as well as a more detailed evaluation. We also discuss how such systems can be further improved by extracting thread- or signal-specific syscall transitions and outlines the idea for a more fine-grained construction of the syscall transitions.

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

1. Protecting the syscall interface is important for security and requires more sophisticated approaches than currently available.
2. Automatically extracting the necessary information is challenging but feasible.
3. Enforcing the extracted information can be done with a minimal runtime overhead while significantly reducing the number of syscall transitions and origins.

References

[1] Abadi, M., Budiu, M., Erlingsson, U., and Ligatti, J. Control-Flow Integrity. In *CCS* (2005).

[2] Canella, C., Dorn, S., Gruss, D., and Schwarz, M. SFIP: Coarse-Grained Syscall-Flow-Integrity Protection in Modern Systems. *arXiv:2202.13716* (2022).

[3] Canella, C., Dorn, S., and Schwarz, M. SFIP/SFIP, `https://github.com/SFIP/SFIP` 2022.

[4] Edge, J. A seccomp overview, `https://lwn.net/Articles/656307/` 2015.

SFIP: Coarse-Grained Syscall-Flow-Integrity Protection in Modern Systems

Abstract

Control-Flow Integrity (CFI) is one promising mitigation that is more and more widely deployed and prevents numerous exploits. However, CFI focuses purely on one security domain, and transitions between user space and kernel space are not protected. Furthermore, if user-space CFI is bypassed, the system and kernel interfaces remain unprotected, and an attacker can run arbitrary transitions.

In this paper, we introduce the concept of syscall-flow-integrity protection (SFIP) that complements the concept of CFI with integrity for user-kernel transitions. Our proof-of-concept implementation relies on static analysis during compilation to automatically extract possible syscall transitions. An application can opt-in to SFIP by providing the extracted information to the kernel for runtime enforcement. The concept is built on three fully-automated pillars: First, a syscall state machine, representing possible transitions according to a syscall digraph model. Second, a syscall-origin mapping, which maps syscalls to the locations at which they can occur. Third, an efficient enforcement of syscall-flow integrity in a modified Linux kernel. In our evaluation, we show that SFIP can be applied to large scale applications with minimal slowdowns. In a micro- and a macrobenchmark, it only introduces an overhead of 13.1 % and 7.4 %, respectively. In terms of security, we discuss and demonstrate its effectiveness in preventing control-flow-hijacking attacks in real-world applications. Finally, to highlight the reduction in attack surface, we perform an analysis of the state machines and syscall-origin mappings of several real-world applications. On average, SFIP decreases the number of possible transitions by 41.5 % compared to seccomp and 91.3 % when no protection is applied.

1. Introduction

Vulnerablities in applications can be exploited by an attacker to gain arbitrary code execution within the application [62]. Subsequently, the attacker can exploit further vulnerabilities in the underlying system to elevate privileges [37]. Such attacks can be mitigated in either of these two stages: the stage where the attacker takes over control of a victim application [62, 13], or the stage where the attacker exploits a bug in the system to elevate privileges [36, 38]. Researchers and industry have focused on eliminating the first stage, where an attacker takes over control of a victim application, by reducing the density of vulnerabilities in software, e.g., by enforcing memory safety [62, 13]. The second line of defense, protecting the system, has also been studied extensively [36, 38, 22, 61]. For instance, sandboxing is a technique that tries to limit the available resources of an application, reducing the remaining attack surface. Ideally, an application only has the bare minimum of resources, e.g., syscalls, that are required to work correctly.

Control-flow integrity [1] (CFI) is a mitigation that limits control-flow transfers within an application to a set of pre-determined locations. While CFI has demonstrated that it can prevent attacks, it is not infallible [29]. Once it has been circumvented, the underlying system and its interfaces are once again exposed to an attacker as CFI does not apply protection across security domains.

In the early 2000s, Wagner and Dean [65] proposed an automatic, static analysis approach that generates syscall digraphs, *i.e.*, a k-sequence [19] of consecutive syscalls of length 2. A runtime monitor validates whether a transition is possible from the previous syscall to the current one and raises an alarm if it is not. The Secure Computing interface of Linux [18], seccomp, simplifies the concept by only validating whether a syscall is allowed, but not whether it is allowed in the context of the previous one. Recent work has explored hardware support for Linux seccomp to improve its performance [60]. In contrast to the work by Wagner and Dean [65] and other intrusion detection systems [21, 25, 32, 34, 44, 68, 47, 63, 69], seccomp acts as an enforcement tool instead of a simple monitoring system. Hence, false positives are not acceptable as they would terminate a benign application. Thus, we ask the following questions in this paper:

Can the concept of CFI be applied to the user-kernel boundary? Can prior syscall-transition-based intrusion detection models, e.g., digraph models [65], be transformed into an enforcement mechanism without breaking modern applications?

In this paper, we answer both questions in the affirmative. We introduce the concept of syscall-flow-integrity protection (SFIP), complementing the concept of CFI

with integrity for user-kernel transitions. Our proof-of-concept implementation relies on static analysis during compilation to automatically extract possible syscall transitions. An application can opt-in to SFIP by providing the extracted information to the kernel for runtime enforcement. SFIP builds on three fully-automated pillars, a syscall state machine, a syscall-origin mapping, and an efficient SFIP enforcement in the kernel.

The **syscall state machine** represents possible transitions according to a syscall digraph model. In contrast to Wagner and Dean's [65] runtime monitor, we rely on an efficient state machine expressed as an $N \times N$ matrix (N is the number of provided syscalls), that scales even to large and complex applications. We provide a compiler-based proof-of-concept implementation, called *SysFlow*[1], that generates such a state machine instead of individual sets of k-sequences. For every available syscall, the state machine indicates to which other syscalls a transition is possible. Our syscall state machine (*i.e.*, the modified digraph) has several advantages including faster lookups ($\mathcal{O}(1)$ instead of $\mathcal{O}(M)$ with M being the number of possible k-sequences), easier construction, and less and constant memory overhead.

The **syscall-origin mapping** maps syscalls to the locations at which they can occur. Syscall instructions in a program may be used to perform different syscalls, *i.e.*, a bijective mapping between code location and syscall number is not guaranteed. We resolve the challenge of these non-bijective mappings with a mechanism propagating syscall information from the compiler frontend and backend to the linker, enabling the precise enforcement of syscalls and their origin. During the state transition check, we additionally check whether the current syscall originates from a location at which it is allowed to occur. For this purpose, we extend our syscall state machine with a syscall-origin mapping that can be bijective or non-bijective, which we extract from the program. Consequently, our approach eliminates syscall-based shellcode attacks and imposes additional constraints on the construction of ROP chains.

The **efficient enforcement** of syscall-flow integrity is implemented in the Linux kernel. Instead of detection, *i.e.*, logging the intrusion and notifying a user as is the common task for intrusion detection systems [39], we focus on enforcement. Our proof-of-concept implementation places the syscall state machine and non-bijective syscall-origin mapping inside the Linux kernel. This puts our enforcement on the same level as seccomp, which is also used to enforce the correct behavior of an application. However, detecting the set of allowed syscalls for seccomp is easier. As such, our enforcement is an additional technique to sandbox an application, automatically limiting the post-exploitation impact of attacks. We refer to our enforcement as *coarse-grained syscall-flow-integrity protection*, effectively emulating the concept of control-flow integrity on the syscall level.

[1] https://github.com/SFIP/SFIP

We evaluate the performance of SFIP based on our reference implementation. In a microbenchmark, we only observe an overhead on the syscall execution of up to 13.1 %, outperforming seccomp-based protections. In real-world applications, we observe an average overhead of 7.4 %. In long-running applications, such as ffmpeg, nginx, and memcached, this overhead is even more negligible, with less than 1.8 % compared to an unprotected version. We evaluate the one-time overhead of extracting the information from a set of real-world applications. In the worst case, we observe an increase in compilation time by factor 28.

We evaluate the security of the concept of syscall-flow-integrity protection in a security analysis with special focus on control-flow hijacking attacks. We evaluate our approach on real-world applications in terms of number of states (*i.e.*, syscalls with at least one outgoing transition), number of average transitions per state, and other security-relevant metrics. Based on this analysis, SFIP, on average, decreases the number of possible transitions by 41.5 % compared to seccomp and 91.3 % when no protection is applied. Against control-flow hijacking attacks, we find that in nginx, a specific syscall can, on average, only be performed at the location of 3 syscall instructions instead of in 318 locations. We conclude that syscall-flow integrity increases system security substantially while only introducing acceptable overheads.

To summarize, we make the following contributions:

1. We introduce the concept of (coarse-grained) *syscall-flow-integrity protection* (SFIP) to enforce legitimate user-to-kernel transitions based on static analysis of applications.
2. Our proof-of-concept SFIP implementation is based on a syscall state machine and a mechanism to validate a syscall's origin.
3. We evaluate the security of SFIP quantitatively, showing that the number of possible syscall transitions is reduced by 91.3 % on average in a set of 8 real-world applications, and qualitatively by analyzing the implications of SFIP on a real-world exploit.
4. We evaluate the performance of our SFIP proof-of-concept implementation, showing an overhead of 13.1 % in a microbenchmark and 7.4 % in a macrobenchmark.

2. Background

2.1. Sandboxing

Sandboxing is a technique to constrain the resources of an application to the absolute minimum necessary for an

application to still work correctly. For instance, a sandbox might limit an application's access to files, network, or syscalls it can perform. A sandbox is often a last line of defense in an already exploited application, trying to limit the post-exploitation impact. Sandboxes are widely deployed in various applications, including in mobile operating systems [30, 3] and browsers [71, 54, 70]. Linux also provides various methods for sandboxing, including SELinux [72], AppArmor [4], or seccomp [18].

2.2. Digraph Model

The behavior of an application can be modeled by the sequence of syscalls it performs. In intrusion detection systems, windows of consecutive syscalls, so-called *k-sequences*, have been used [19]. k-sequences of length $k = 2$ are commonly referred to as digraphs [65]. A model built upon these digraphs allows easier construction and more efficient checking while reducing the accuracy in the detection [65] as only previous and current syscall are considered.

2.3. Linux Seccomp

The syscall interface is a security-critical interface that the Linux kernel exposes to userspace applications. Applications rely on syscalls to request the execution of privileged tasks from the kernel. Hence, securing this interface is crucial to improving the system's overall security.

To better secure this interface, the kernel provides Linux Secure Computing (seccomp). A benign application first creates a filter that contains all the syscalls it intends to perform over its lifetime and then passes this filter to the kernel. Upon a syscall, the kernel checks whether the executed syscall is part of the set of syscalls defined in the filter and either allows or denies it. As such, seccomp can be seen as a k-sequence of length 1. In addition to the syscall itself, seccomp can filter static syscall arguments. Hence, seccomp is an essential technique to limit the post-exploitation impact of an exploit, as unrestricted access to the syscall interface allows an attacker to arbitrarily read, write, and execute files. An even worse case is when the syscall interface itself is exploitable, as this can lead to privilege escalation [37, 36, 38].

2.4. Runtime Attacks

One of the root causes for successful exploits are memory safety violations. One typical variant of such a violation are buffer overflows, enabling an attacker to modify the application in a malicious way [62]. An attacker tries to use such a buffer overflow to overwrite a code pointer, such that the control flow can be diverted to an attacker-chosen location, e.g., to previously injected *shellcode*. Attacks relying on shellcode have become harder to execute on modern systems due to data normally not being executable [62, 49]. Therefore, attacks have to rely on already present, executable code parts, so-called *gadgets*. These gadgets are chained together to perform an arbitrary attacker-chosen task [51]. Shacham further generalized this attack technique as return-oriented programming (ROP) [59]. Similar to control-flow-hijacking attacks that overwrite pointers [59, 11, 43, 29, 56], memory safety violations can also be abused in data-only attacks [55, 35].

2.5. Control-Flow Integrity

Control-flow integrity [1] (CFI) is a concept that restricts an application's control flow to valid execution traces, *i.e.*, it restricts the targets of control-flow transfer instructions. This is enforced at runtime by comparing the current state of the application to a set of pre-computed states. Control-flow transfers can be divided into forward-edge and backward-edge transfers [7]. Forward-edge transfers transfer control flow to a new destination, such as the target of an (indirect) jump or call. Backward-edge transfers transfer the control flow back to a location that was previously used in a forward edge, e.g., a return from a call. Furthermore, CFI can be subdivided into coarse-grained and fine-grained CFI. In contrast to fine-grained CFI, coarse-grained CFI allows for a more relaxed control-flow graph, allowing more targets than necessary [14].

3. Design of Syscall-Flow-Integrity Protection

3.1. Threat Model

SFIP is applied to a benign userspace application that potentially contains a vulnerability allowing an attacker to execute arbitrary code within the application. The post-exploitation targets the operating system through the syscall interface to gain kernel privileges. With SFIP, a syscall is only allowed if the state machine contains a valid transition from the previous syscall to the current one and if it originates from a pre-determined location. If either one is violated, the application is terminated by the kernel. Similar to prior work [10, 24, 16, 23], our protection is orthogonal but fully compatible with defenses such as CFI, ASLR, NX, or canary-based protections. Therefore, the security it provides to the system remains even if these other protections have been circumvented. Side-channel and fault attacks [40, 73, 41, 46, 64, 57] on the state machine or syscall-origin mapping are out of scope.

3.2. High-Level Design

In this section, we discuss the high-level design behind SFIP. Our approach is based on three pillars: a digraph model for syscall sequences, a per-syscall model of syscall origin, and the strict enforcement of these models (cf. Figure 1).

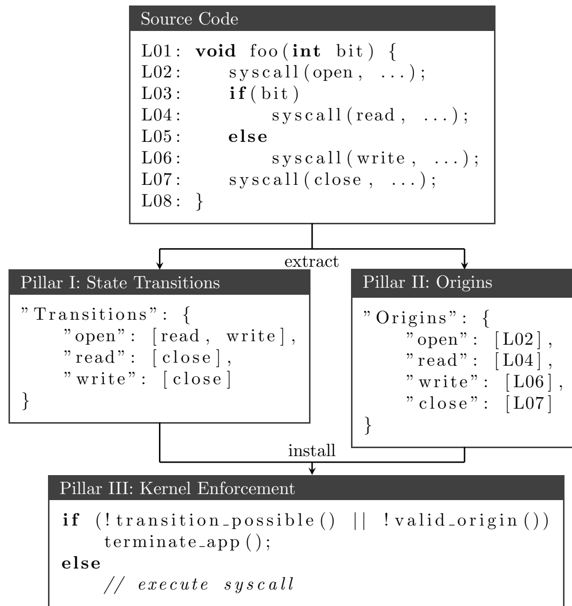

Figure 1: The three pillars of SFIP on the example of a function. The first pillar models possible syscall transitions, the second maps syscalls to their origin, and the third enforces them.

For our first pillar, we rely on the idea of a digraph model from Wagner and Dean [65]. For our sycall-flow-integrity protection, we rely on a more efficient construction and in-memory representation. In contrast to their approach, we express the set of possible transitions not as individual k-sequences, but as a global syscall matrix of size $N \times N$, with N being the number of available syscalls. We refer to the matrix as our *syscall state machine*. With this representation, verifying whether a transition is possible is a simple lookup in the row indicated by the previous syscall and the column indicated by the currently executing syscall. Even though the representation of the sequences differs, the set of valid transitions remains the same: every transition that is marked as valid in our syscall state machine must also be a valid transition if expressed in the way discussed by Wagner and Dean. Our representation has several advantages though, that we explore in this paper, namely faster lookups ($\mathcal{O}(1)$), less memory overhead, and easier construction.

Our syscall state machine can already be used for coarse-grained SFIP to improve the system's security (cf. Section 5.2). However, the second pillar, the validation of the origin of a specific syscall, further improves the provided security guarantees by adding additional, enforceable information. The basis for this augmentation is the ability to map syscalls to the location at which they can be invoked, independent of whether it is a bijective or non-bijective mapping. We refer to the resulting mapping as our *syscall-origin mapping*. For instance, our mapping might contain the information that the syscall instruction

```c
void foo(int bit, int nr) {
  syscall(open, ...);
  if(bit)
    syscall(read, ...);
  else
    syscall(nr, ...);
  bar(...);
  syscall(close, ...);
}

```

Listing 1: Example of a dummy program with multiple syscall-flow paths.

located at address `0x7ffff7ecbc10` can only execute the syscalls *write* and *read*. Neither unaligned execution (e.g., in a ROP chain) nor code inserted at runtime is in our syscall-origin mapping. Thus, syscalls can only be executed at already existing syscall instructions.

The third pillar is the enforcement of the syscall state machine and the syscall-origin mapping. Wagner and Dean [65] proposed their runtime monitoring as a concept for intrusion detection systems. There is still a domain expert involved to decide on any further action [39]. In contrast to monitoring, enforcement cannot afford false positives as this immediately leads to the termination of the application in benign scenarios. However, enforcement provides better security than monitoring as immediate action is undertaken, completely eliminating the time window for a possible exploit. Thus, by the use case of SFIP, namely enforcement of syscall-flow integrity, our concept is more closely related to seccomp but harder to realize than seccomp-based enforcement of syscalls.

3.3. Challenges

Previous automation work for seccomp filters outlined several challenges for automatically detecting an application's syscalls [10]. While several works [10, 16, 24] solve these challenges, none provides the full information required for SFIP. The challenges of getting this missing information focus on precise syscall information and inter- and intra-procedural control-flow transfer information. We illustrate the challenges using a simple dummy program in Listing 1.

$\mathcal{C}$1: Precise Per-Function Syscall Information The first challenge focuses on precise per-function syscall information. This challenge must be solved for the generation of the syscall state machine as well as the sycall-origin map. For seccomp-based approaches, *i.e.*, k-sequence of length 1, an automatic approach only needs to identify the set of syscalls within a function, *i.e.*, the exact location of the syscalls is irrelevant. This does not hold for SFIP, which requires precise information at which

location a specific syscall is executed. Thus, we have to detect that the first syscall instruction always executes the *open* syscall, the second executes *read*, and the third syscall instruction can execute any syscall that can be specified via *nr*. For the state machine generation, the precise information of syscall locations provides parts of the information required to correctly generate the sequence of syscalls. For the syscall-origin map, the precise information allows generating the mapping of syscall instructions to actual syscalls in the case where syscall numbers are specified as a constant at the time of invocation.

$\mathcal{C}2$: Argument-based Syscall Invocations The second challenge extends upon $\mathcal{C}1$ as it concerns syscall locations where the actual syscall executed cannot be easily determined at the time of compilation. When parsing the function `foo`, we can identify the syscall number for all invocations of the `syscall` function where the number is specified as a constant. The exception is the third invocation, as the number is provided by the caller of the `foo` function. As the call to the function, and hence the actual syscall number, is in a different translation unit than the actual syscall invocation, the possibility for a non-bijective mapping exists. Still, an automated approach must determine all possible syscalls that can be invoked at each syscall instruction.

$\mathcal{C}3$: Correct Inter- and Intra-Procedural Control-Flow Graph Precise per-function syscall information on its own is not sufficient to generate syscall state machines due to the non-linearity of typical code. Solving $\mathcal{C}1$ and $\mathcal{C}2$ provides the information which syscalls occur at which syscall location, but does not provide the information on the execution order. A trivial construction algorithm can assume that each syscall within a function can follow each other syscall, but this overapproximation leads to imprecise state machines. Such an approach accepts a transition from *read* to the syscall identified by *nr* as valid, even though it cannot occur within our example function.

Therefore, we need to determine the correct inter- and intra-procedural control-flow transfers in an application. The correct intra-procedural control-flow graph allows determining the possible sequences within a function. In our example, and if function `bar` does not contain any syscalls, it provides the information that the sequence of syscalls *open* $\rightarrow$ *read* $\rightarrow$ *close* is valid, while *open* $\rightarrow$ *nr* $\rightarrow$ *close* (where *nr* $\neq$ *read*) is not.

Even in the presence of a correct intra-procedural control-flow graph, we cannot reconstruct the syscall state machine of an application as information is missing on the sequence of syscalls from other called functions. For instance, if function `bar` contains at least one syscall, the sequence of *open* $\rightarrow$ *read* $\rightarrow$ *close* is no longer valid. Hence, we additionally need to recover the precise location where control flow is transferred to another function

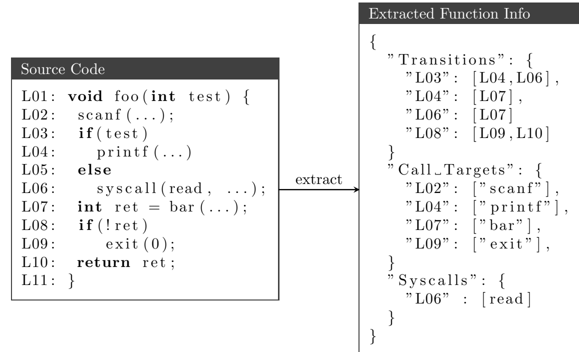

Figure 2: A simplified example of the information that is extracted from a function. *Transitions* identifies control-flow transfers between basic blocks, *Call Targets* the location of a call to another function and the targets name, *Syscalls* the location of the syscall and the corresponding syscall number.

and the target of this control-flow transfer. By combining the inter- and intra-procedural control-flow graph, the correct syscall sequences of an application can be modeled.

Constructing a precise control-flow graph is known to be a challenging task to solve efficiently [2, 31], especially in the presence of indirect control-flow transfers. These algorithms are often cubic in the size of the application, which makes them infeasible for large-scale applications. In the construction of the control-flow graph and, by extension, the generation of the syscall state machine and syscall-origin mapping, other factors, such as aliased and referenced functions, must be considered as well as functions that are passed as arguments to other functions, e.g., the entry function for a new thread created with `pthread_create`. Any form of imprecision can lead to the termination of the application by the runtime enforcement.

4. Implementation

In this section, we discuss our proof-of-concept implementation SysFlow and how we systematically solve the challenges outlined in Section 3.3 to provide fully-automated SFIP.

SysFlow SysFlow automatically generates the state machine and the syscall-origin mapping while compiling an application. As the basis of SysFlow we considered the works by Ghavamnia et al. [24] and Canella et al. [10].

4.1. State-Machine Extraction

In SysFlow, the linker is responsible for creating the final state machine. The construction works as follows: The linker starts at the main function, *i.e.*, the user-defined entry point of an application, and recursively follows the ordered set of control-flow transfers. Upon encountering a syscall location, the linker adds a transition from the previ-

ous syscall(s) to the newly encountered syscall. If control flow continues at a different function, the set of last valid syscall states is passed to the recursive visit of the encountered function. Upon returning from a recursive visit, the linker updates the set of last valid syscall states and continues processing the function. During the recursive processing, it also considers aliased and referenced functions. A special case, and source of overapproximation, are indirect calls, which we address with appropriate techniques from previous works [10, 16, 23]. The resulting syscall state machine and our support libarary are embedded in the static binary. We discuss the support library in more detail in Section 4.3.

Building the state machine requires that precise information of the syscalls a function executes ($\mathcal{C}\mathbf{1}$) and a control-flow graph of the application ($\mathcal{C}\mathbf{3}$) is available to the linker. Both the front- and backend are involved in collecting this information. The frontend extracts the information from the LLVM IR generated from C source code, while the backend extracts the information from assembly files. Figure 2 illustrates the information that is extracted from a function.

Extracting Precise Syscall Information In the frontend, we iterate over every IR instruction of a function and determine the used syscalls. In the backend, we iterate over every assembly instruction to extract the syscalls. Extracting the information in the front- and backend successfully solves $\mathcal{C}\mathbf{1}$.

Extracting Precise Control-Flow Information Recovering the control-flow graph ($\mathcal{C}\mathbf{3}$) in the frontend requires two different sources of information: IR call instructions and successors of basic blocks. The former allows tracking inter-procedural control-flow transfers while the latter allows tracking intra-procedural transfers. For inter-procedural transfers, we iterate over every IR instruction and determine whether it is a call to an external function. For direct calls, we store the target of the call; for indirect calls, we store the function signature of the target function. In addition, we also gather information on referenced and aliased functions, as well as functions that are passed as arguments to other functions. For the intra-procedural transfers, we track the successors of each basic block. In the backend, we perform similar steps, although on a platform-specific assembly level. Extracting this information in the front- and backend successfully solves $\mathcal{C}\mathbf{3}$.

4.2. Syscall-Origin Extraction

In SysFlow, the linker also generates the final syscall-origin mapping. The mapping maps all reachable syscalls to the locations where they can occur. We extract the information as an offset instead of an absolute position to facilitate compatibility with ASLR. The linker requires precise information of syscalls, *i.e.*, their offset relative to the start of the encapsulating function, and a precise call graph of the application. Both the front- and backend are responsible for providing this information. Figure 3 illustrates the extraction. From the frontend, the syscall information generated by the state machine extraction is re-used ($\mathcal{C}\mathbf{1}$). A challenge is the possibility of non-bijective syscall mappings ($\mathcal{C}\mathbf{2}$).

Non-Bijective Syscall Mappings If the syscall number cannot be determined at the location of a syscall instruction, a non-bijective mapping exists for the instruction, *i.e.*, multiple syscalls can be executed through it. An example of such a case is shown in Listing 1. In such cases, the backend itself cannot create a mapping of a syscall to the syscall instruction. Hence, it must propagate the syscall number and the syscall offset from their respective translation unit to the linker, which can then merge it, solving $\mathcal{C}\mathbf{2}$.

4.3. Installation

For each syscall, the binary contains a list of all other reachable syscalls as an $N \times N$ matrix, *i.e.*, the state machine, with N being the number of syscalls available. Valid transitions are indicated by a 1 in the matrix, invalid ones with a 0 to allow fast checks and constant memory overhead. If a function contains a syscall, the offset of the syscall is added to the load address of the function. The state machine and the syscall-origin mapping are sent to the kernel and installed.

4.4. Kernel Enforcement

In this section, we discuss the third and final pillar of SFIP: enforcement of the syscall flow and origin where every violation leads to immediate process termination.

Our Linux kernel is based on version 5.13 configured for Ubuntu 21.04 with the following modifications.

First, a new syscall, *SYS_syscall_sequence*, which takes as arguments the state machine, the syscall-origin mapping, and a flag that identifies the requested mode, *i.e.*, is state-machine enforcement requested, syscall-origin enforcement, or both. The kernel rejects updates to already installed syscall-flow information. Consequently, an unprivileged process cannot apply a malicious state machine or syscall origins before invoking a setuid binary or other privileged programs using the *exec* syscall [17].

Second, our syscall-flow-integrity checks are executed before every syscall. We create a new *syscall_work_bit* entry, which determines whether or not the kernel uses the slow syscall entry path, like in seccomp, to ensure that our checks are executed. Upon installation, we set the respective bit in the *syscall_work* flag in the *thread_info* struct of the requesting task.

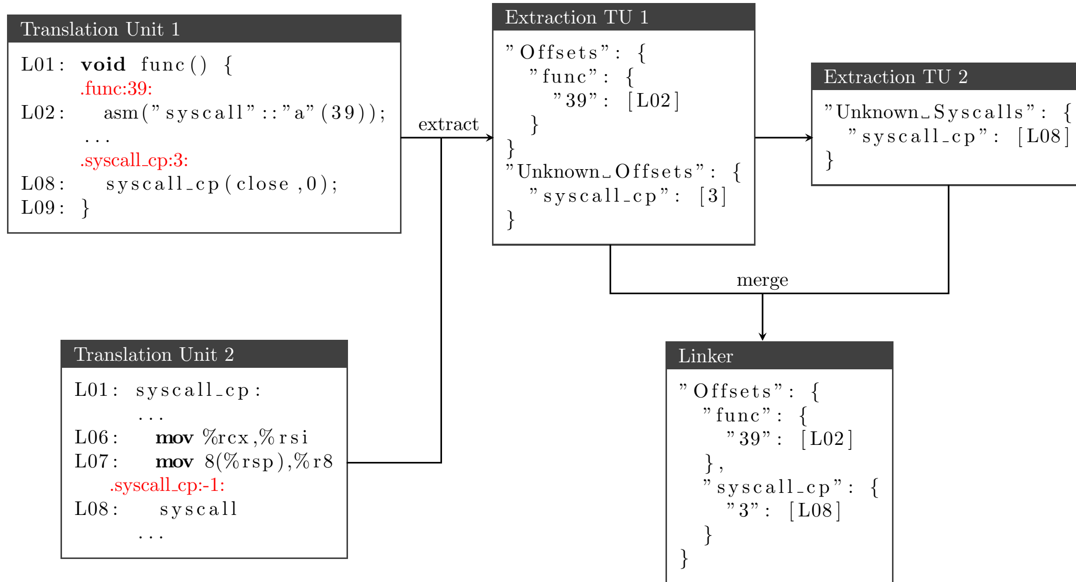

Figure 3: A simplified example of the syscall-origin extraction. Inserted red labels mark the location of a syscall and encode available information. The extraction deconstructs the label and calculates the offset using the label's address from the symbol table. The linker combines the information from each translation unit and generates the final syscall-origin mapping.

Third, the syscall-flow information has to be stored and cleaned up properly. As it is never modified after installation, it can be shared between the parent and child processes and threads. Upon task cleanup, we decrease the reference counter, and if it reaches 0, we free the respective memory. The current state, *i.e.*, the previously executed syscall, is not shared between threads or processes and is thus part of every thread.

Enforcing State Machine Transitions Each thread and process tracks its own current state in the state machine. As we enforce sequence lengths of size 2, storing the previously executed syscall as the current state is sufficient for the enforcement. Due to the design of our state machine, verifying whether a syscall is allowed is a single lookup in the matrix at the location indicated by the previous and current syscall. If the entry indicates a valid transition, we update our current state to the currently executing syscall and continue with the syscall execution. Otherwise, the kernel immediately terminates the offending application. The simple state machine lookup, with a complexity of $\mathcal{O}(1)$, ensures that only a small overhead is introduced to the syscall (cf. Sections 5.1.2 and 5.1.3).

Enforcing Syscall Origins The enforcement of the syscall origins is very efficient due to its design. Our modified kernel uses the current syscall to retrieve the set of possible locations from the mapping to check whether the current RIP, minus the size of the syscall instruction itself, is a part of the retrieved set. If not, the application requested the syscall from an unknown location, which results in the kernel immediately terminating it. By design, the complexity of this lookup is $\mathcal{O}(N)$, with N being the number of valid offsets for that syscall. We evaluate typical values of N in Section 5.2.6.

5. Evaluation

In this section, we evaluate the general idea of SFIP and our proof-of-concept implementation SysFlow. In the evaluation, we focus on the performance and security of the syscall state machines and syscall-origins individually and combined. We evaluate the overhead introduced on syscall executions in both a micro- and macrobenchmark. We also evaluate the time required to extract the required information from a selection of real-world applications.

Our second focus is the security provided by SFIP. We first consider the protection SFIP provides against control-flow hijacking attacks. We evaluate the security of pure syscall-flow protection, pure syscall-origin protection, and combined protection. We discuss mimicry attacks and how SFIP makes such attacks harder. We also consider the security of the stored information in the kernel and discuss the possibility of an attacker manipulating it. Finally, we extract the state machines and syscall origins from several real-world applications and analyze them. We evaluate several security-relevant metrics such as the number of states in the state machine, average possible transitions per state, and the average number of allowed syscalls per syscall location.

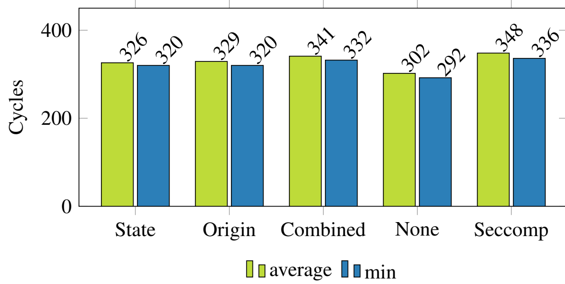

Figure 4: Microbenchmark of the *getppid* syscall over 100 million executions. We evaluate SFIP with only state machine, only syscall origin, both, and no enforcement active. For comparison, we also benchmark the overhead of seccomp.

5.1. Performance

5.1.1. Setup All performance evaluations are performed on an i7-4790K running Ubuntu 21.04 and our modified Linux 5.13 kernel. For all evaluations, we ensure a stable frequency.

5.1.2. Microbenchmark We perform a microbenchmark to determine the overhead our protection introduces on syscall executions. Our benchmark evaluates the latency of the *getppid* syscall, a syscall without side effects that is also used by kernel developers and previous works [6, 10, 33]. SysFlow first extracts the state machine and the syscall-origin information from our benchmark program, which we then execute once for every mode of SFIP, *i.e.*, state machine, syscall origins, and combined. Each execution measures the latency of 100 million syscall invocations. For comparison, we also benchmark the execution with no active protection. As with seccomp, syscalls performed while our protection is active require the slow syscall enter path to be taken. As the slow path introduces part of the overhead, we additionally measure the performance of seccomp in the same experiment setup.

Results Figure 4 shows the results of the microbenchmark. Our results indicate a low overhead for the syscall execution for all SFIP modes. Transition checks show an overhead of 8.15 %, syscall origin 9.13 %, and combined 13.1 %. Seccomp introduces an overhead of 15.23 %. The improved seccomp has a complexity of $\mathcal{O}(1)$ for simple allow/deny filters [12], the same as our state machine. The syscall-origin check has a complexity of $\mathcal{O}(N)$, with typically small numbers for N, *i.e.*, $N = 1$ for the *getppid* syscall in the microbenchmark. Section 5.2.6 provides a more thorough evaluation of N in real-world applications. The additional overhead in seccomp is due to its filters being written in cBPF and converted to and executed as eBPF.

5.1.3. Macrobenchmark To demonstrate that SFIP can be applied to large-scale, real-world applications with a minimal performance overhead, we perform a macrobenchmark using applications used in previous work [10, 24, 60]. We measure the performance over 100 executions with only state machine, only syscall origin, both, and no enforcement active. For nginx, we measure the time it takes to process 100 000 requests. For ffmpeg, we convert a video (21 MB) from one file format to another. With pwgen, we generate a set of passwords while coreutils and memcached are benchmarked using their respective testsuites. In all cases, we verified that syscalls are being executed, e.g., each request for nginx executes at least 13 syscalls.

Table 1: The results of our extraction time evaluation in real world applications. We present both the compilation time of the respective application with and without our extraction active.

Application	Unmodified Average / SEM	Modified Average / SEM
ffmpeg	162.12 s / 0.78	1783.15 s / 10.61
mupdf	58.01 s / 0.71	489.85 s / 0.68
nginx	8.22 s / 0.03	226.64 s / 1.67
busybox	16.09 s / 0.08	81.33 s / 0.14
coreutils	5.50 s / 0.02	14.39 s / 0.41
memcached	2.90 s / 0.03	4.59 s / 0.01
pwgen	0.07 s / 0.00	0.12 s / 0.00

Results Figure 5 shows the results of the macrobenchmark. In nginx, we observe a small increase in execution time when any mode of SFIP is active. If both checks are performed, the average increase from 24.96 s to 25.34 s (+1.52 %) is negligible. We observe similar overheads in the ffmpeg benchmark. For the combined checks, we only observe an increase from 9.41 s to 9.58 s (+1.52 %). pwgen and coreutils show the highest overhead. pwgen is a small application that performs its task in under a second; hence any increase appears large. The absolute change in runtime is an increase of 0.05 s. For the coreutils benchmark, we execute the testsuite that involves all 103 utilities. Each utility requires that the SFIP information is copied to the kernel, which introduces a majority of the overhead. As the long-running applications show, the actual runtime overhead is less than 1.8 %. Our results demonstrate that SFIP is a feasible concept for modern, large-scale applications.

5.1.4. Extraction-Time Benchmark We evaluate the time it takes to extract the information required for the state machine and syscall origins. As targets, we use several real-world applications (cf. Table 1) used in previous works on automated seccomp sandboxing [10, 24, 16]. These range from smaller utility applications such as busy-

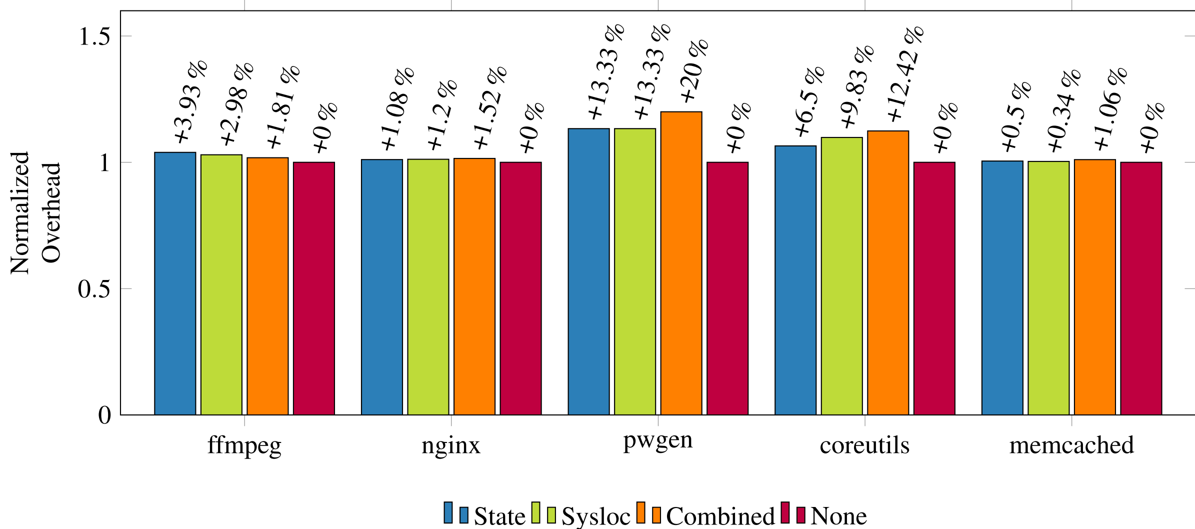

Figure 5: We perform a macrobenchmark using 5 real-world applications. For nginx, we measure the time it takes to handle 100 000 requests using ab. For ffmpeg, we convert a video (21 MB) from one file format to another. pwgen generates a set of passwords while coreutils and memcached are benchmarked using their respective testsuites. Each benchmark measures the average execution time over 100 repetitions of each mode of SFIP.

box and coreutils to applications with a larger and more complex codebase such as ffmpeg, mupdf, and nginx. For the benchmark, we compile each application 10 times using our modified compiler with and without our extraction active.

Results Table 1 shows the result of the extraction-time benchmark. We present the average compilation time and the standard error for compiling each application 10 times. The results indicate that the extraction introduces a significant overhead. For instance, for the coreutils applications, we observe an increase in compilation time from approximately 6 s to 15 s. We observe the largest increase in nginx from approximately 8 s to 227 s. Most of the overhead is in the linker, while the extraction in the frontend and backend is fast. We expect that a full implementation can significantly improve upon the extraction time by employing more efficient caching and by potentially applying other construction algorithms.

Similar to previous work [24], we consider the increase in compilation time not to be prohibitive as it is a one-time cost. Hence, the security improvement outweighs the increase in compilation time.

5.2. Security

In this section, we evaluate the security provided by SFIP. We discuss the theoretical security benefit of each mode of SFIP in the context of control-flow-hijacking attacks. We then evaluate a real vulnerability in BusyBox version 1.4.0 and later[2]. We also consider mimicry attacks [65, 66] and perform an analysis of real-world state machines and syscall origins.

5.2.1. Syscall-Flow Integrity in the Context of Control-flow Hijacking In the threat model of SFIP (cf. Section 3.1), an attacker has control over the program-counter value of an unprivileged application. In such a situation, an attacker can either inject code, so-called shellcode, that is then executed, or reuse existing code in a so-called code-reuse attack. In a shellcode attack, an attacker manages to inject their own custom code. With control over the program-counter value, an attacker can redirect the control flow to the injected code. On modern systems, these types of attacks are by now harder to execute due to data execution prevention [62, 49], *i.e.*, data is no longer executable. As a result, an attacker must first make the injected code executable, which requires syscalls, e.g., the *mprotect* syscall. For this, an attacker has to rely on existing code (gadgets) in the exploited application to execute such a syscall. An attacker might be lucky, and the correct parameters are already present in the respective registers, resulting in a straightforward code-reuse attack commonly known as ret2libc [51]. Realistically, however, an attacker first has to get the location and size of the shellcode area into the corresponding registers using existing code gadgets. Depending on the type of gadgets, such attacks are known as return-oriented-programming [59] or jump-oriented-programming attacks [5].

On an unprotected system, every application can execute the *mprotect* syscall. Depending on the application, the *mprotect* syscall cannot be blocked by seccomp if the respective application requires it. With SFIP, attacks that rely on *mprotect* can potentially be prevented even

[2] https://ssd-disclosure.com/ssd-advisory-busybox-local-cmdline-stack-buffer-overwrite/

if the application requires the syscall. First, we consider a system where only the state machine is verified on every syscall execution. *mprotect* is mainly used in the initialization phase of an application [24, 10]. Hence, we expect very few other syscalls to have a transition to it, if any. This leaves a tiny window for an attacker to execute the syscall to make the shellcode executable, *i.e.*, it is unlikely that the attempt succeeds in the presence of state-machine SFIP. Still, with only state-machine checks, the syscall can originate from any syscall instruction within the application.

Contrary, if only the syscall origin is enforced, the *mprotect* syscall is only allowed at certain syscall instructions. Hence, an attacker needs to construct a ROP chain that sets up the necessary registers for the syscall and then returns to such a location. In most cases, the only instance where *mprotect* is allowed is within the libc `mprotect` function. If executed from there, the syscall succeeds. If the syscall originates from another location, the check fails, and the application is terminated. Still, with only syscall origins being enforced, the previous syscall is not considered, allowing an attacker to perform the attack at any point in time.

With both active, *i.e.*, full SFIP, several restrictions are applied to a potential attack. The attacker must construct a ROP chain that either starts after a syscall with a valid transition to *mprotect* was executed, or the ROP chain must contain a valid sequence of syscalls that lead to such a state, *i.e.*, a mimicry attack (cf. Section 5.2.3). Additionally, all syscalls must originate from a location where they can legally occur. These additional constraints significantly increase the security of the system.

5.2.2. Real-world Exploit For a real-world application, we evaluate a stack-based buffer overflow in the BusyBox arp applet from version 1.4.0 to version 1.23.1. In line with our threat model, we assume that all software-based security mechanisms, such as ASLR and stack protector, have already been circumvented. The vulnerable code is in the `arp_getdevhw` function, which copies a user-provided command-line parameter to a stack-allocated structure using `strcpy`. By providing a device name longer than `IFNAMSIZ` (default 16 characters), this overflow overwrites the stack content, including the stored program counter.

The simplest exploit we found is to mount a return2libc attack using a *one gadget RCE*, *i.e.*, a gadget that directly spawns a shell. In libc version 2.23, we discovered such a gadget at offset `0xf0897`, with the only requirement that offset `0x70` on the stack is zero, which is luckily the case. Hence, by overwriting the stored program counter with that offset, we can successfully replace the application with an interactive shell. With SFIP, this exploit is prevented. Running the exploit executes the *socket* syscall right before the *execve* syscall that opens the shell. While the *execve* syscall is at the correct location, the state machine does not allow a transition from the *socket* to the *execve* syscall. Hence, exploits that directly open a shell are prevented. We also verified that there is no possible transition from *socket* to *mprotect*,; hence loaded shellcode cannot be marked as executable. There are only 21 syscalls after a *socket* syscall allowed by the state machine. Especially as neither the *mprotect* nor the *execve* syscall are available, possible exploits are drastically reduced. To circumvent the protection, an attacker would need to find gadgets allowing a valid transition chain from the *socket* to the *execve* (or *mprotect*) syscall. We also note that the buffer overflow itself is also a limiting factor. As the overflow is caused by a `strcpy` function, the exploit payload, *i.e.*, the ROP chain, cannot contain any null byte. Thus, given that user-space addresses on 64-bit systems always have the 2 most-significant address bits set to 0, a longer chain is extremely difficult to craft.

5.2.3. Syscall-Flow-Integrity Protection and Mimicry Attacks We consider the possibility of mimicry attacks [65, 66] where an attacker tries to circumvent a detection system by evading the policy. For instance, if an intrusion detection system is trained to detect a specific sequence of syscalls as malicious, an attacker can add arbitrary, for the attack unneeded, syscalls that hide the actual attack. With SFIP, such attacks become significantly more complicated. An attacker needs to identify the last executed syscall and knowledge of the valid transitions for all syscalls. With this knowledge, the attacker needs to perform a sequence of syscalls that forces the state machine into a state where the malicious syscall is a valid transition. Additionally, as syscall origins are enforced, the attacker has to do this in a ROP attack and is limited to syscall locations where the specific syscalls are valid. While this does not make mimicry attacks impossible, it adds several constraints that make the attack significantly harder.

5.2.4. Security of Syscall-Flow Information in the Kernel The security of the syscall-flow information stored in the kernel is crucial for effective enforcement. Once the application has sent the information to the kernel for enforcement, it is the responsibility of the kernel to prevent malicious changes to the information. The case where the initial information sent to the kernel is malicious is outside of the threat model (cf. Section 3.1).

The kernel stores the information in kernel memory; hence direct access and manipulation is not possible. The only way to modify the information is through our new syscall. Our implementation currently does not allow for any changes to the installed information, *i.e.*, no updates are allowed. An attacker using our syscall and a ROP attack to manipulate the information is also not possible as the syscall itself needs to pass SFIP checks before being executed. As the application contains no valid transition

nor location for the syscall, the kernel terminates the application.

Still, as allowing no updates is a design decision, another implementation might consider allowing updates. In this case, the application needs to perform our new syscall to update the filters. Before our syscall is executed, SFIP is applied to the syscall, *i.e.*, it is verified whether there is a valid transition to it and whether it originates at the correct location. If not, the kernel terminates the application; otherwise, the update is applied. In this case, if timed correctly, an attacker is able to maliciously modify the stored information.

5.2.5. State Machine Reachability Anaysis We analyse the state machine of several real-world applications in more detail. We define a state in our state machine as a syscall with at least one outgoing transition. While Wagner and Dean [65] only provide information on the *average branching factor*, *i.e.*, the number of average transitions per state, we extend upon this to provide additional insights into automatically generated syscall state machines. We focus on several key factors: the overall number of states in the application and the minimum, maximum, and average number of transitions across these states. These are key factors that determine the effectiveness of SFIP. We do not consider additional protection provided by enforcing syscall origins. We again rely on real-world applications that have been used in previous work [10, 16, 24, 60]. For busybox and coreutils, we do not provide the data for every utility individually, but instead present the average of all contained utilities, *i.e.*, 398 and 103, respectively. To determine the improvement in security, we consider an unprotected version of the respective application, *i.e.*, every syscall can follow the previously executed syscall. Additionally, we compare our results to a seccomp-based version.

Results Table 2 shows the results of this evaluation. nginx shows the highest number of states with 108, followed by memcached, mutool, and ffmpeg with 87, 61, and 56 states, respectively. coreutils and busybox also provide multiple functionalities but split across various utilities. Hence, their number of states is comparatively low.

Interestingly, each application has at least one state with only one valid transition. We manually verified this transition, and in every case, it is a transition from the *exit_group* syscall to the *exit* syscall, which is indeed the only valid transition for this syscall.

The combination of the average and maximum number of transitions together with the number of states provides some interesting insight. We observe that in most cases, the number of average transitions is relatively close to the maximum number of transitions, while the difference to the number of states can be larger. This indicates that our state machine is heavily interconnected. Modern applications delegate many tasks via syscalls to the kernel, such as allocating memory, sending data over the network, or writing to a file. As syscalls can fail, they are often followed by error checking code that performs application-specific error handling, logs the error, or terminates the application. Hence, a potential transition to these syscalls is automatically detected, leading to larger state machines. Another source is locking, as the involved syscalls can be preceded and followed by a wide variety of other syscalls. Additionally, the overapproximation of indirect calls also increases the number of transitions.

Even with such interconnected state machines, the security improvement is still large compared to an unprotected version of the application or even a seccomp-based version. In the case of an unprotected version, all syscalls are valid successors to a previously executed syscall. An unmodified Linux kernel 5.13 provides 357 syscalls. Compared to nginx, which has the highest number of average transitions with 66, this is an increase of factor 5.4 in terms of available transitions. In our state machine, the number of states corresponds to the number of syscalls an automated approach needs to allow for seccomp-based protection. These numbers also match the numbers provided in previous work on automated seccomp filter generation. For instance, Canella et al. [10] reported 105 syscalls in nginx and 63 in ffmpeg. Ghavamnia et al. [24] reported 104 in nginx. Each such syscall can follow any of the other syscalls that are part of the set. In the case of nginx, this is around factor 1.6 more than in the average state when SFIP is applied. Hence, we conclude that even coarse-grained SFIP can drastically increase the system's security.

5.2.6. Syscall Origins Analysis We perform a similar analysis for our syscall origins in real-world applications. We focus on analyzing the number of syscall locations per application and for each such location, the number of syscalls that can be executed. Special focus is put on the number of syscalls that can be invoked through the syscall wrapper functions as they can allow a wide variety of syscalls. Hence, the fewer syscalls are available through these functions, the better the security of the system.

Results We show the results of this evaluation in Table 3. The average number of offsets per syscall indicates that many syscalls are available at multiple locations. This is most likely due to the inlining of the syscall. This number is largely driven by the *futex* syscall, as locking is required in many places of applications. Error handling is a less driving factor in this case as these are predominantly printed using dedicated, non-inlined functions.

The last two columns analyze the number of syscalls that can be invoked by the respective syscall wrapper function and demonstrate a non-bijective mapping of syscalls to syscall locations. Relatively few syscalls are available through the `syscall()` function as it can be more easily

Table 2: **We evaluate various properties of applications state machines, including the average number of transitions per state, number of states in the state machine, min and max transitions. Busybox and coreutils show the averages over all contained utilites (398 and 103 utilities, respectively).**

Application	Average Transitions	#States	Min Transitions	Max Transitions
busybox	15.73	24.51	1.0	21.09
pwgen	12.42	19	1	16
muraster	17.51	41	1	33
nginx	65.55	108	1	80
coreutils	15.75	27.11	1.0	23.0
ffmpeg	48.48	56	1	51
memcached	40.6	87	1	71
mutool	32.0	61	1	46

Table 3: We evaluate various metrics for our syscall location enforcement, including the total number of functions containing syscalls, min, max and average number of syscalls per function, total syscall offsets found, average offsets per syscall, and the number of syscalls in the used musl syscall wrapper functions. Busybox and coreutils show the averages over all contained utilites (398 and 103 utilities, respectively).

Application	#Functions	Min Syscalls	Max Syscalls	Avg. Syscalls per Function	Total #Offsets	Avg #Offsets	#syscall()	#syscall_cp()	#syscall_cp_asm()
busybox	30.57	1.0	9.83	1.48	102.64	3.75	1.71	9.79	0
pwgen	28	1	3	1.25	84	4.42	0	2	0
muraster	55	1	12	1.62	193	4.6	0	4	0
nginx	105	1	24	1.53	318	3.0	7	24	0
coreutils	36.86	1.0	4.21	1.38	116.71	4.42	1.0	3.41	0
ffmpeg	89	1	13	1.55	279	4.98	0	13	13
memcached	101	1	20	1.5	317	3.69	0	20	0
mutool	81	1	14	1.67	278	4.15	6	14	0

inlined, *i.e.*, it is almost always inlined within libc itself. On the other hand, `syscall_cp()` cannot be inlined as it is a wrapper around an aliased function that performs the actual syscall.

Our results also indicate that, on average, every function that contains a syscall contains more than one syscall. nginx contains the most functions with a syscall and the highest number of total syscall offsets. Without syscall-origin enforcement, an attacker can choose from 318 syscall locations to execute any of the 357 syscalls provided by Linux 5.13 during a ROP attack. With our enforcement, the number is drastically reduced as each one of these locations can, on average, perform only 3 syscalls instead of 357.

6. Discussion

Limitations and Future Work Our proof-of-concept implementation currently does not handle signals and syscalls invoked in a signal handler. However, this is not a conceptual limitation. The compiler can identify all functions that serve as a signal handler and the functions that are reachable through it. Hence, it can extract a per-signal state machine to which the kernel switches when it sets up the signal stack frame. This allows for small per-signal state machines, which further improve security. As this requires significant engineering work, we leave the implementation and evaluation for future work.

Our state-machine construction leads to coarse-grained state machines, which can be improved by the fact that we can statically identify syscall origins. Future work can intertwine this information on a deeper level with the generated state machine. By doing so, a transition to another state is then not only dependent on the previous and the current syscall number but also on the virtual address of the previous and current syscall instruction. This allows to better represent the syscall-flow graph of the

application without relying on context-sensitivity or call stack information [65, 28, 58]. As this requires significant changes to the compiler and the enforcement in the kernel and thorough evaluation, we leave this for future work.

Recent work has proposed hardware support for seccomp [60]. In future work, we intend to investigate whether similar approaches are possible to improve the performance of SFIP.

Related Work In 2001, the seminal work by Wagner and Dean [65] introduced automatically-generated syscall NDFAs, NDPDAs, and digraphs for sequence checks in intrusion detection systems. SFIP builds upon digraphs but modifies their construction and representation to increase performance. We further extend upon their work by additionally verifying the origin of a syscall. The accuracy and performance of SFIP allows real-time enforcement in large-scale applications.

Several papers have focused on extracting and modeling an applications control flow based on the work by Forrest et al. [19]. Frequently, such approaches rely on dynamic analysis [21, 25, 32, 34, 44, 68, 47, 63, 69]. Other approaches rely on machine-learning techniques to learn syscall sequences or detect intrusions [74, 53, 48, 8, 67, 26]. Giffin et al. [27] proposed incorporating environment information in the static analysis to generate more precise models. The Dyck model [28] is a prominent approach for learning syscall sequences that rely on stack information and context-sensitive models. Other works disregard control flow and focus instead on detecting intrusions based on syscall arguments [42, 50]. Forrest et al. [20] provide an analysis on the evolution of system-call monitoring. Our work differs as we do not require stack information, context-sensitive models, dynamic tracing of an application, or code instrumentation. The only additional information we consider is the mapping of syscalls to syscall instructions.

Recent work has investigated the possibility of automatically generating seccomp filters from source or existing binaries [16, 10, 24, 23, 52]. SysFlow can be extended to generate the required information from binaries as well. More recent work proposed a faster alternative to seccomp while also enabling complex argument checks [9]. In contrast to these works, we consider syscall sequences and origins, which requires additional challenges to be solved (cf. Section 3.3).

A similar approach to our syscall-origin enforcement has been proposed by Linn et al. [45] and de Raadt [15]. The former extracts the syscall locations and numbers from a binary and enforces them on the kernel level but fails in the presence of ASLR. The latter restricts the execution of syscalls to entire regions, but not precise locations, *i.e.*, the entire text segment of a static binary is a valid origin. Additionally, in the entire region, any syscall is valid at any syscall location. Our work improves upon them in several ways as we (1) present a way to enforce syscall origins in the presence of ASLR, (2) limit the execution of specific syscalls to precise locations, (3) combine syscall origins with state machines which lead to a significant increase in security.

7. Conclusion

In this paper, we introduced the concept of syscall-flow-integrity protection (SFIP), complementing the concept of CFI with integrity for user-kernel transitions. In our evaluation, we showed that SFIP can be applied to large-scale applications with minimal slowdowns. In a micro- and a macrobenchmark, we observed an overhead of only 13.1 % and 7.4 %, respectively. In terms of security, we discussed and demonstrated its effectiveness in preventing control-flow-hijacking attacks in real-world applications. Finally, to highlight the reduction in attack surface, we performed an analysis of the state machines and syscall-origin mappings of several real-world applications. On average, we showed that SFIP decreases the number of possible transitions by 41.5 % compared to seccomp and 91.3 % when no protection is applied.

References

[1] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-Flow Integrity. In *CCS*, 2005.

[2] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, 1994.

[3] Android. Application Sandbox, 2021.

[4] AppArmor. AppArmor: Linux kernel security module, 2021.

[5] Tyler K. Bletsch, Xuxian Jiang, Vincent W. Freeh, and Zhenkai Liang. Jump-oriented programming: a new class of code-reuse attack. In *AsiaCCS*, 2011.

[6] Davidlohr Bueso. tools/perf-bench: Add basic syscall benchmark, 2019.

[7] Nathan Burow, Scott A. Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. Control-Flow Integrity: Precision, Security, and Performance. *ACM Computing Surveys*, 2017.

[8] Jeffrey Byrnes, Thomas Hoang, Nihal Nitin Mehta, and Yuan Cheng. A Modern Implementation of System Call Sequence Based Host-based Intrusion Detection Systems. In *TPS-ISA*, 2020.

[9] Claudio Canella, Andreas Kogler, Lukas Giner, Daniel Gruss, and Michael Schwarz. Domain Page-Table Isolation. *arXiv:2111.10876*, 2021.

[10] Claudio Canella, Mario Werner, Daniel Gruss, and Michael Schwarz. Automating Seccomp Filter Generation for Linux Applications. In *CCSW*, 2021.

[11] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. Return-oriented programming without returns. In *CCS*, 2010.

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

[13] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX Security*, 1998.

[14] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *USENIX Security Symposium*, August 2014.

[15] Theo de Raadt. syscall call-from verification, 2019.

[16] Nicholas DeMarinis, Kent Williams-King, Di Jin, Rodrigo Fonseca, and Vasileios P. Kemerlis. sysfilter: Automated System Call Filtering for Commodity Software. In *RAID*, 2020.

[17] Jake Edge. System call filtering and no_new_privs, 2012.

[18] Jake Edge. A seccomp overview, 2015.

[19] S. Forrest, S.A. Hofmeyr, A. Somayaji, and T.A. Longstaff. A sense of self for Unix processes. In *S&P*, 1996.

[20] Stephanie Forrest, Steven Hofmeyr, and Anil Somayaji. The Evolution of System-Call Monitoring. In *ACSAC*, 2008.

[21] Thomas D. Garvey and Teresa F. Lunt. Model-based intrusion detection. In *NCSC*, 1991.

[22] Xinyang Ge, Nirupama Talele, Mathias Payer, and Trent Jaeger. Fine-Grained Control-Flow Integrity for Kernel Software. In *Euro S&P*, 2016.

[23] Seyedhamed Ghavamnia, Tapti Palit, Shachee Mishra, and Michalis Polychronakis. Confine: Automated System Call Policy Generation for Container Attack Surface Reduction. In *RAID*, 2020.

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

[25] Anup Ghosh, Aaron Schwartzbard, and Michael Schatz. Learning Program Behavior Profiles for Intrusion Detection. In *ID*, 1999.

[26] Anup K. Ghosh and Aaron Schwartzbard. A Study in Using Neural Networks for Anomaly and Misuse Detection. In *USENIX Security Symposium*, 1999.

[27] Jonathon Giffin, David Dagon, Somesh Jha, Wenke Lee, and Barton Miller. Environment-Sensitive Intrusion Detection. In *RAID*, 2005.

[28] Jonathon T Giffin, Somesh Jha, and Barton P Miller. Efficient Context-Sensitive Intrusion Detection. In *NDSS*, 2004.

[29] Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. Out of control: Overcoming control-flow integrity. In *S&P*, 2014.

[30] Google. Seccomp filter in Android O, 2017.

[31] Michael Hind. Pointer analysis: Haven't we solved this problem yet? In *PASTE*, 2001.

[32] Steven A. Hofmeyr, Stephanie Forrest, and Anil Somayaji. Intrusion Detection Using Sequences of System Calls. *J. Comput. Secur.*, 1998.

[33] Tom Hromatka. seccomp and libseccomp performance improvements, 2018.

[34] K. Ilgun, R.A. Kemmerer, and P.A. Porras. State transition analysis: a rule-based intrusion detection approach. *TSE*, 1995.

[35] Kyriakos K. Ispoglou, Bader AlBassam, Trent Jaeger, and Mathias Payer. Block Oriented Programming: Automating Data-Only Attacks. In *CCS*, 2018.

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

[37] Vasileios P Kemerlis, Michalis Polychronakis, and Angelos D Keromytis. ret2dir: Rethinking kernel isolation. In *USENIX Security Symposium*, 2014.

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

[39] Richard A Kemmerer and Giovanni Vigna. Intrusion detection: a brief history and overview. *Computer*, 2002.

[40] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors. In *ISCA*, 2014.

[41] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. In *S&P*, 2019.

[42] Christopher Kruegel, Darren Mutz, Fredrik Valeur, and Giovanni Vigna. On the Detection of Anomalous System Call Arguments. In *ESORICS*, 2003.

[43] Bingchen Lan, Yan Li, Hao Sun, Chao Su, Yao Liu, and Qingkai Zeng. Loop-oriented programming: a new code reuse attack to bypass modern defenses. In *IEEE Trustcom/BigDataSE/ISPA*, 2015.

[44] Terran Lane and Carla E. Brodley. Temporal Sequence Learning and Data Reduction for Anomaly Detection. *TOPS*, 1999.

[45] C. M. Linn, M. Rajagopalan, S. Baker, C. Collberg, S. K. Debray, and J. H. Hartman. Protecting Against Unexpected System Calls. In *USENIX Security Symposium*, 2005.

[46] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading Kernel Memory from User Space. In *USENIX Security Symposium*, 2018.

[47] Teresa F. Lunt. Automated Audit Trail Analysis and Intrusion Detection: A Survey. In *NCSC*, 1988.

[48] Shaohua Lv, Jian Wang, Yinqi Yang, and Jiqiang Liu. Intrusion Prediction With System-Call Sequence-to-Sequence Model. *IEEE Access*, 2018.

[49] Microsoft. Data Execution Prevention, 2021.

[50] Darren Mutz, Fredrik Valeur, Giovanni Vigna, and Christopher Kruegel. Anomalous System Call Detection. *TOPS*, 2006.

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

[52] Shankara Pailoor, Xinyu Wang, Hovav Shacham, and Isil Dillig. Automated Policy Synthesis for System Call Sandboxing. *PACMPL*, 2020.

[53] Y. Qiao, X.W. Xin, Y. Bin, and S. Ge. Anomaly intrusion detection method based on HMM. *Electronics Letters*, 2002.

[54] Charles Reis, Alexander Moshchuk, and Nasko Oskov. Site Isolation: Process Separation for Web Sites within the Browser. In *USENIX Security Symposium*, 2019.

[55] Roman Rogowski, Micah Morton, Forrest Li, Fabian Monrose, Kevin Z. Snow, and Michalis Polychronakis. Revisiting Browser Security in the Modern Era: New Data-Only Attacks and Defenses. In *EuroS&P*, 2017.

[56] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications. In *S&P*, 2015.

[57] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In *CCS*, 2019.

[58] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni. A fast automaton-based method for detecting anomalous program behaviors. In *S&P*, 2001.

[59] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *CCS*, 2007.

[60] Dimitrios Skarlatos, Qingrong Chen, Jianyan Chen, Tianyin Xu, and Josep Torrellas. Draco: Architectural and Operating System Support for System Call Security. In *MICRO*, 2020.

[61] Brad Spengler. Recent ARM Security Improvements, 2013.

[62] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. SoK: Eternal War in Memory. In *S&P*, 2013.

[63] H.S. Teng, K. Chen, and S.C. Lu. Adaptive real-time anomaly detection using inductively generated sequential patterns. In *S&P*, 1990.

[64] Stephan van Schaik, Alyssa Milburn, Sebastian österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: Rogue In-flight Data Load. In *S&P*, 2019.

[65] D. Wagner and R. Dean. Intrusion detection via static analysis. In *S&P*, 2001.

[66] David Wagner and Paolo Soto. Mimicry Attacks on Host-Based Intrusion Detection Systems. In *CCS*, 2002.

[67] C. Warrender, S. Forrest, and B. Pearlmutter. Detecting intrusions using system calls: alternative data models. In *S&P*, 1999.

[68] Lee Wenke, S.J. Stolfo, and K.W. Mok. A data mining framework for building intrusion detection models. In *S&P*, 1999.

[69] Andreas Wespi, Marc Dacier, and Hervé Debar. Intrusion Detection Using Variable-Length Audit Trail Patterns. In *RAID*, 2000.

[70] Mozilla Wiki. Project Fission, 2019.

[71] Mozilla Wiki. Security/Sandbox, 2019.

[72] SELinux Wiki. FAQ — SELinux Wiki, 2009.

[73] Yuval Yarom and Katrina Falkner. Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security Symposium*, 2014.

[74] Zhang Zhengdao, Peng Zhumiao, and Zhou Zhiping. The Study of Intrusion Prediction Based on HsMM. In *APSCC*, 2008.