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

Claudio Canella
Graz University of Technology
claudio.canella@iaik.tugraz.at

Sebastian Dorn
Graz University of Technology
sebastian.dorn@student.tugraz.at

Daniel Gruss
Graz University of Technology
daniel.gruss@iaik.tugraz.at

Michael Schwarz
CISPA Helmholtz Center
for Information Security
michael.schwarz@cispa.de

***Abstract*—Growing code bases of modern applications have led to a steady increase in the number of vulnerabilities. Control-Flow Integrity (CFI) is one promising mitigation that is more and more widely deployed and prevents numerous exploits. CFI focuses purely on one security domain. That is, transitions between user space and kernel space are not protected by CFI. 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 $1.8\,\%$, 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 $38.6\,\%$ compared to seccomp and $90.9\,\%$ when no protection is applied.

I. Introduction

Vulnerablities in modern applications can be exploited by an attacker to gain arbitrary code execution within the application [64]. Subsequently, the attacker can exploit further vulnerabilities in the underlying system to elevate privileges [38]. Such attacks can be mitigated in either of these two stages: the stage where the attacker takes over control of a victim application [64], [14], or the stage where the attacker exploits a bug in the system to elevate privileges [37], [39]. Both 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 [64], [14]. The second line of defense, protecting the system, has also been studied extensively [37], [39], [24], [63]. 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 predetermined locations. While CFI has demonstrated that it can prevent attacks, it is not infallible [30]. 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 [67] proposed an automatic, static analysis approach that generates syscall digraphs, *i.e.*, a k-sequence [21] 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 [19], 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. In contrast to the work by Wagner and Dean [67], 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 [67], 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 [67] 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 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 [40], 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.

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 a macrobenchmark using nginx and ffmpeg, we observe an overhead of $1.5\,\%$ and $1.8\,\%$ compared to an unprotected version, respectively. 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 $38.6\,\%$ compared to seccomp and $90.9\,\%$ 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 $90.9\,\%$ on average in a set of 6 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 $1.8\,\%$ in a macrobenchmark.

Outline Section II provides background. Section III then provides our threat model, discusses the high-level concept of syscall-flow integrity, and discusses the challenges of extracting the information required for SFIP and enforcing it. Section IV provides implementation details, and Section V evaluates it with regard to performance and security. Section VI discusses limitations as well as future and related work. Section VII then concludes the work.

II. Background

This section discusses the necessary background for this work.

A. Sandboxing

Sandboxing is a technique that tries to constrain the resources of an application to the absolute minimum necessary for the application to still work correctly. For instance, a sandbox might limit an application's access to files, to the network, or the syscalls it can perform. As such, a sandbox is often the last line of defense in an already exploited application, trying to limit the post-exploitation impact. Nowadays, sandboxes are widely deployed in various applications, including in mobile operating systems [35], [3] and browsers [72], [55], [71]. Linux itself also provides various methods for sandboxing, including SELinux [73], AppArmor [4], or seccomp [19].

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

B. 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 [21]. A special case of k-sequences are sequences of length $k = 2$, which are commonly referred to as digraphs [67]. A model built upon these digraphs can allow for easier construction and more efficient checking while reducing the accuracy in the detection [67]. That is because only the previous syscall and the current one are considered, instead of a longer sequence.

C. Linux Seccomp

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

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 [38], [37], [39].

D. 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 [64]. 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 [64], [50]. 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 [52]. Shacham further generalized this attack technique as return-oriented programming (ROP) [62]. Similar to control-flow-hijacking attacks that overwrite pointers [62], [11], [44], [30], [58], memory safety violations can also be abused in data-only attacks [57], [36].

E. 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 [15].

III. Design of Syscall-Flow-Integrity Protection

In this section, we define the threat model for SFIP (Section III-A), the high-level design (Section III-B), and the challenges for such an approach (Section III-C).

A. Threat Model

SFIP is applied to userspace applications. We assume that the protected application is benign but potentially contains a vulnerability that allows an attacker to execute arbitrary code within the application. The post-exploitation then targets the operating system through the syscall interface to gain kernel privileges. SFIP restricts the execution of syscalls in two ways. First, a syscall is allowed if the state machine contains a valid transition from the previous syscall to the current one. Second, a syscall is allowed if it is a valid entry in the syscall-origin mapping, *i.e.*, it originates from a predetermined location. If either one is violated, the application is terminated by the kernel. Similar to prior work [9], [25], [17], [26], 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 [41], [74], [42], [47], [66], [59] on the state machine or syscall-origin mapping are out of scope.

B. 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. Figure 1 illustrates this high-level design.

For our first pillar, we rely on the idea of a digraph model from Wagner and Dean [67]. Digraphs are a special case of the *k-sequences*, *i.e.*, windowed sequences of consecutive syscalls, introduced by Forrest et al. [21]. They have been proposed in the early 2000s but neither the generation nor the runtime monitoring has been evaluated in the context of modern, large-scale applications. In a digraph model [67], the sequence length is fixed to 2. For our sycall-flow-integrity protection, we also use a sequence length of 2, but 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

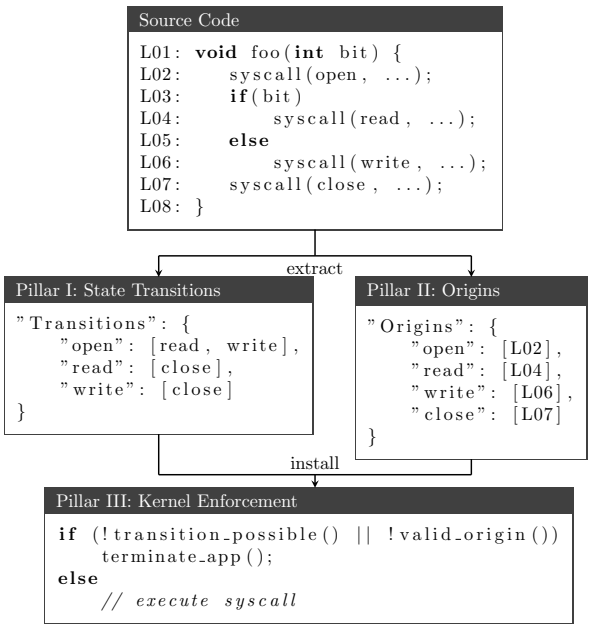

Fig. 1: The three pillars of SFIP on the example of a function. The first pillar models possible syscall transitions, the second maps syscalls to their origin, and the third enforces them.

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. The reason is that both are generated from the source code of the application; hence, any transition valid in one representation must be valid in the other. 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 security of the system (cf. Section V-B). However, the second pillar, the validation of the origin of a specific syscall, further improves the provided security guarantees by adding additional, enforcable information. The basis for this augmentation is the ability to precisely 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 located at address `0x7ffff7ecbc10` can only execute the syscalls *write* and *read*. By design, this concept alone eliminates shellcode attacks: 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.

```
1 void foo(int bit, int nr) {
2   syscall(open, ...);
3   if(bit)
4     syscall(read, ...);
5   else
6     syscall(nr, ...);
7   bar(...);
8   syscall(close, ...);
9 }
10
```

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

The third pillar of SFIP is the enforcement of the syscall state machine and the syscall-origin mapping. Wagner and Dean [67] proposed their runtime monitoring as a concept for intrusion-detection systems, which are designed to detect and log violations. There is still a domain expert involved to decide any further action [40]. The difference between monitoring and enforcement is that enforcement cannot afford false positives as this immediately leads to the termination of the application in benign scenarios. On the other hand, 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.

C. Challenges

Previous automation work outlined several challenges that need to be solved for automatically detecting the syscalls an application uses so that seccomp can correctly filter them [9]. We investigated several works [9], [17], [25] that propose automated seccomp-filter generation and find that each solves these challenges, but none provides the full information required for SFIP. Hence, we identify challenges that must be solved by each proposed approach to provide the information required for SFIP. Due to the switch from runtime monitoring to enforcement, the generation of our syscall state machine and syscall-origin mapping must be precise as false positives immediately lead to the termination of the application. Therefore, the challenges primarily focus on precise syscall information and inter- and intra-procedural control-flow transfer information. We illustrate the challenges using a simple dummy program shown in Listing 1.

a) $\mathcal{C}1$: 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, although for different reasons. 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. In our example, seccomp only requires the information that our function `foo` contains the syscalls *open*, *read*, and *close* and the one identified by the

function parameter *nr*. 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.

b) $\mathcal{C}2$: 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 be able to determine all possible syscalls that can be invoked at each syscall instruction.

c) $\mathcal{C}3$: 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 us with the information which syscalls occur at which syscall location, but does not provide us with information in which sequence they can be executed in. A trivial construction algorithm can simply assume that each syscall within a function can follow each other syscall within the function, 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 us with the information that the sequence of syscalls *open* → *read* → *close* is valid, while a sequence of *open* → *nr* → *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* → *read* → *close* is no longer valid. Hence, we additionally need to recover the precise location where control flow is transferred to another function, as well as 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 hard 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.

IV. Implementation

In this section, we discuss our proof-of-concept implementation SysFlow and how we systematically solve the challenges outlined in Section III-C to provide fully automated SFIP. We discuss all three pillars of SFIP. First, we discuss our implementation to extract our syscall state machine. Second, we discuss our implementation to extract a syscall-origin mapping, which augments the concept of syscall-sequence checks. Third, we discuss our modified kernel that enforces the application's behavior instead of simply monitoring it. We also discuss our support library, which is required for setting up the enforcement, similar to *libseccomp*.

a) 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. [25] and Canella et al. [9]. Both tools are already capable of extracting most syscall numbers and a reasonably-precise call graph, although only for seccomp-based protection. However, neither of them solves the challenges we identified in Section III-C, making both equally well suited as a basis for our work. As Ghavamnia et al. [25] report significantly higher extraction times, we opted for the work by Canella et al. [9] as a basis. Consequently, SysFlow is also built on top of LLVM 10.

A. 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. Initialization functions, e.g., musl's *__libc_start_main*, are not analyzed, as the enforcement is not activated before the *main* function. Upon encountering a syscall location, the linker adds a transition from the previous 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

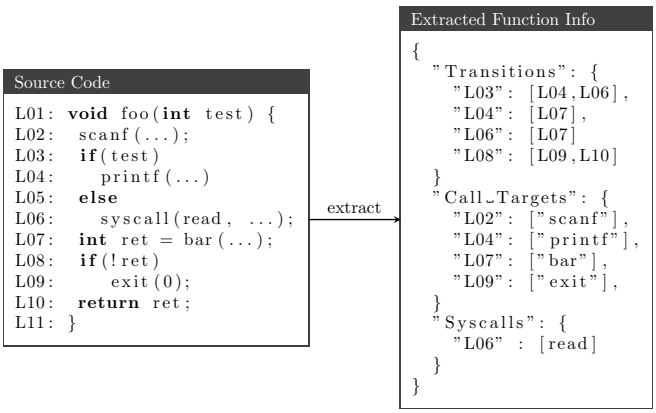

Fig. 2: A simplified example of the information that is extracted from a function. *Transitions* identifies control-flow transfers between basic blocks, *Call Targets* the location of a call to another function and the targets name, *Syscalls* the location of the syscall and the corresponding syscall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

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 [9], [17], [26]. At the site of the indirect call, we know the signature of the function that is indirectly called. We compare this signature with the signature of all functions that have their address taken. All functions that match are then processed in the same way as above.

When the linker reaches the end of the *main* function, it has processed all reachable functions. In the process, it has generated the set of valid transitions for each syscall. The resulting syscall state machine is then embedded in the static binary. Our support library is automatically included in the binary as well and is responsible for installing the state machine when the application is launched. We discuss the support library in more detail in Section IV-C.

The process of 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. To propagate the information to the linker, both the front- and backend store it in the resulting object file. Figure 2 illustrates the information that is extracted from a function.

a) Extracting Precise Syscall Information: In the frontend, we iterate over every IR instruction of a function and determine whether it is an inline assembly syscall instruction or a call to one of the libc syscall wrappers. In most cases, the syscall number is specified as a constant, allowing the compiler to easily associate it with the location of the syscall. In the backend, we iterate over every assembly instruction of a function specified in an assembly file. On x86, the syscall number is placed in the `RAX` register before the syscall instruction. Hence, we track the value that is moved into the register. Once we encounter a syscall instruction, we associate the instruction with the value in `RAX`. Extracting the information in the front- and backend successfully solves $\mathcal{C}\mathbf{1}$.

b) 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 instead of the IR level. Extracting this information in the front- and backend successfully solves $\mathcal{C}\mathbf{3}$.

B. Syscall-Origin Extraction

In SysFlow, the linker also generates the final syscall-origin mapping. The mapping maps all reachable syscalls to the locations at which they can occur, relative to the start of the function that encapsulates them. We extract the information as an offset instead of an absolute position to facilitate compatibility with ASLR. The linker identifies all functions reachable via the *main* function and adds their syscall map to the mapping. The resulting mapping is embedded in the final static binary from where our support library can extract it to make the final address computations (cf. Section IV-C).

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 ($\mathcal{C}\mathbf{1}$); hence, we do not discuss it again but focus on the additional changes in the backend. Extracting the offset is a complex process that requires adding and propagating partial information through various parts of the backend. The main reason for this complex process is the non-fixed instruction size on x86. An additional problem is the possibility of non-bijective syscall mappings, which must also be resolved ($\mathcal{C}\mathbf{2}$).

a) Non-Fixed Instruction Size: On architectures with fixed instruction sizes, computing the offset of an instruction relative to the start of the function is trivial. On x86, with its non-fixed instruction size, the final instruction size is only known once relocations have been decided, but at this point, no information on syscalls is available. Hence, earlier stages must add and propagate syscall information to make it available.

b) Non-Bijective Syscall Mappings: If the syscall number cannot be determined at the location of a syscall instruction, a

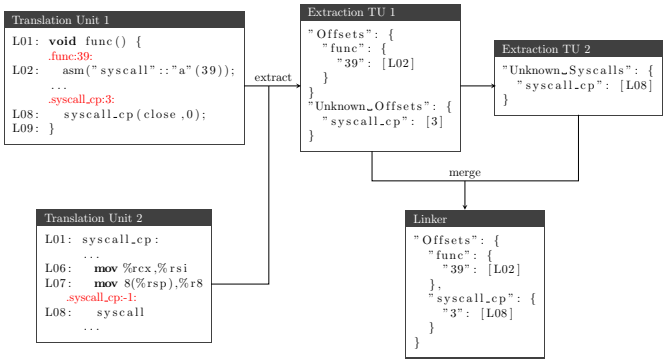

Fig. 3: A simplified example of the syscall-origin extraction. We insert labels (red) that mark the location of each syscall and encode available information for it. In the extraction, we deconstruct the label and calculate the offset using the label's address from the symbol table. The linker combines the information from each translation unit and generates the final syscall-origin mapping.

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. Instances of this are non-inlined calls to the syscall wrapper functions provided by libc, *i.e.*, `syscall()` or `syscall_cp()`, as syscall number and syscall instruction are in different translation units. In such cases, the backend itself is not able to 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. This allows the linker to finally solve $\boldsymbol{C2}$.

c) Propagating Syscall Information: A new pass in the backend iterates over all functions in the translation unit and the instructions that comprise them. For every syscall that is encountered, the pass emits a label before it. The label is emitted independent of whether the syscall is performed using an inlined syscall instruction or a call to one of the wrappers. The label serves two purposes: it is used to propagate necessary information through the backend, and enables the computation of the syscall offset. For an inlined syscall, we embed the name of the encapsulating function and, if available, the syscall number in the label. For a call to one of the syscall wrappers, we embed the name of the wrapper and, if available, the syscall number. If the syscall number is not available ($\boldsymbol{C2}$), we embed the value -1. This identifies a syscall where the linker is responsible for constructing the correct function-to-offset-to-syscall mapping.

In addition to the new backend pass, we extend the AsmParser to emit labels for syscalls originating in assembly files. This is necessary since our MachineFunctionPass cannot add these labels as it iterates over the machine IR generated from the LLVM IR.

d) Offset Calculation: Before the object file is generated, the backend determines relocations, which determines final instruction sizes, and computes the symbol table. The latter assigns an address to the injected labels. With the symbol table computed, all information required to create the translation unit's syscall mapping is available. The symbol table provides the offset, and the label's content provides the remaining information required to create the mapping. If the label contains the syscall number, we can directly create the mapping of function to syscall number to syscall offset, e.g., `malloc` performs a *futex* syscall at offset $0x209$. If the label contains -1, we only create a mapping of function to offset, e.g., `syscall_cp` performs an unknown syscall at offset $0x1c$ If the label contains the name of a syscall wrapper, we create a mapping of wrapper function to syscall number, e.g., `syscall_cp` performs syscall *read* at an unknown offset. The linker merges the latter two once it has extracted the information from all object files and solves $\boldsymbol{C2}$. This finally allows the linker to create the complete syscall-origin mapping of the application.

C. Installation

In this section, we discuss the implementation of our support library, which is responsible for extracting the generated information from the binary and installing it in the kernel. SysFlow automatically adds the library to the static binary during compilation. It contains a constructor that is run before the execution starts *main*, similar to what has been done in previous work [9], [17].

a) State Machine: For each syscall, the binary contains a list of all other reachable syscalls. The library converts this information into 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. This design allows for fast checks and constant memory overhead, independent of the number of possible transitions. The resulting state machine is sent to the kernel and installed.

b) Syscall Origins: Installing the syscall-origin information requires additional pre-processing to determine the final location of every syscall. The support library extracts the symbol table of the static binary to retrieve the load address of every function. If a function contains a syscall, the offset of the syscall is added to the load address of the function. The corresponding entry in the syscall-origin mapping is updated with the result. The final syscall-origin mapping is sent to the kernel and installed.

D. Kernel Enforcement

In this section, we discuss the third and final pillar of SFIP: enforcement of the syscall flow and origin. As we previously discussed, SFIP does not perform simple runtime monitoring as is common in intrusion-detection systems, but runtime enforcement. Hence, every violation leads to immediate process termination.

Our kernel is based on Linux kernel version 5.13 configured for Ubuntu 21.04. We first discuss the common parts of both the state machine as well as the syscall origins, namely the following three modifications of the kernel:

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. This is necessary to make the kernel aware of our syscall-flow integrity information. The kernel copies the corresponding data and stores it in the *task_struct* of the current process. It rejects updates to already installed syscall-flow information. The kernel also sets the *NO_NEW_PRIVS* flag, similar to seccomp. Consequently, an unprivileged process cannot apply a malicious state machine or syscall origins before invoking a setuid binary or other privileged programs using one of the *exec* syscalls [18].

Second, we follow the example of seccomp and modify the kernel so that our syscall-flow integrity checks are executed before every syscall if the process has requested them. For this purpose, 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.

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. Hence, only a reference count for the stored information is required. The current state, *i.e.*, the previously executed syscall, is not shared between threads or processes. Thus, it is necessary to modify the *copy_process* function to copy the reference and the initial current state into every new process and thread if the parent has it installed. For cleanup, we modify the *release_task* function that is called upon task cleanup. There, we decrease the reference counter, and if it reaches 0, we free the respective memory.

a) Enforcing State Machine Transitions: Overall, the process of enforcing our state machine is very efficient. Each thread and process tracks its own current state in the state machine, which is laid out as a flattened $N \times N$ matrix. 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. If the entry does not indicate a valid transition, the application tries to perform a syscall that it should not be executing. 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 V-A2 and V-A3).

b) Enforcing Syscall Origins: The enforcement of the syscall origins is also very efficient due to its design. As previously discussed, our syscall-origin mapping maps syscall numbers to their respective virtual addresses. Hence, our modified kernel uses the current syscall to retrieve the set of possible locations from the mapping. It then checks whether the current RIP, minus the size of the syscall instruction itself, is a part of the retrieved set. If so, the syscall originates from a valid location, and we continue its execution. Otherwise, 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 V-B6.

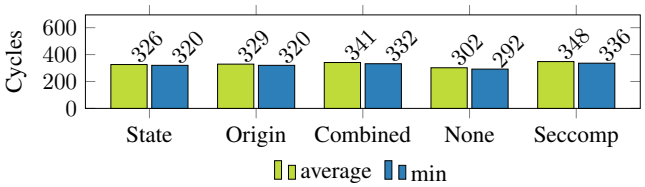

Fig. 4: Microbenchmark of the *getppid* syscall over 100 million executions. We evaluate SFIP with only state machine, only syscall origin, both, and no enforcement active. For comparison, we also show the overhead seccomp introduces on the syscall execution. Latency is given in cycles.

V. 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 then 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, the number of average possible transitions per state, and the average number of allowed syscalls per syscall location.

A. Performance

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.

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], [9], [33]. SysFlow first extracts the state machine and the syscall-origin information from our benchmark program. We execute the benchmark program once for every mode of SFIP, *i.e.*, state machine,

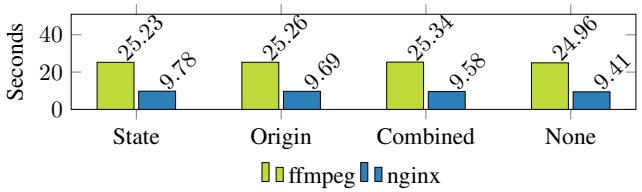

Fig. 5: We perform a macrobenchmark of the two largest applications in our set of real-world applications. For nginx, we use the `ab` tool to make 100 000 requests and time how long it takes to process them. For ffmpeg, we convert a video (21 MB) from one file format to another. In both cases, we perform the test 100 times for each mode of SFIP.

syscall origins, and combined. Each execution measures the latency of 100 million syscall invocations. For comparison, we also benchmark the execution with no protection active. As with seccomp, syscalls performed while our protection is active require the slow syscall enter path to be taken due to *_TIF_WORK_SYSCALL_ENTRY* being set. This is not the case for the benchmark without protection. As the slow path introduces part of the overhead, we additionally measure the performance of seccomp in the same experiment setup.

a) 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 %. We expect that large parts of the overhead are due to the slow and complex syscall enter path instead of the checks itself. The improved seccomp has a complexity of $\mathcal{O}(1)$ for simple allow/deny filters [12], 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 V-B6 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.

3) Macrobenchmark: To demonstrate that SFIP can be applied to large scale, real-world applications with a minimal performance overhead, we perform a macrobenchmark. We benchmark two of the larger applications used in previous work [9], [25], *i.e.*, nginx and ffmpeg. We extract the syscall state machine and syscall-origin information for each of the two applications. We measure the performance over 100 executions with only state machine, only syscall origin, both, and no enforcement active. For nginx, we start the server and measure the time it takes to process 100 000 requests. For ffmpeg, we convert a video (21 MB) from one file format to another. In both cases, we verified that syscalls are being executed, e.g., each request for nginx executes at least 13 syscalls.

a) 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. On average, we saw an increase from 24.96 s to 25.34 s in case both checks are performed. Hence, the overhead is negligible and unnoticable for most users. We observe similar overheads in the case of our ffmpeg benchmark. For the combined checks, we only observe an increase from 9.41 s to 9.58 s. ffmpeg demonstrates one interesting result: the overhead for the combined checks is less than the overhead for each individual component. Across several repetitions of the benchmark, this effect was always present. Independent of the application, our results demonstrate that SFIP is a feasible concept for modern, large scale applications.

TABLE I: 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

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 I) used in previous works on automated seccomp sandboxing [9], [25], [17]. These range from smaller utility applications such as busybox 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. The resulting applications all use musl's implementation of libc and are static binaries.

a) Results: Table I 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, in the case of the coreutils applications, we observe an increase in compilation time from approximately 6 s to 15 s. We observe the largest increase in nginx with an increase from approximately 8 s to 227 s. Naturally, the larger the application, the longer our extraction takes as more functions must be visited. Most of the overhead materializes 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 [25], we consider the increase in compilation time not to be prohibitive as it is a one-time cost when releasing a new version of the application. Hence, the

improvement in security outweighs the increase in compilation time.

B. 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 [67], [68] and perform an analysis of real-world state machines and syscall origins.

1) Syscall-Flow Integrity in the Context of Control-flow Hijacking: In the threat model of SFIP (cf. Section III-A), an attacker has gained 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 into the address space of a running application. 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 [64], [50], *i.e.*, data is no longer executable. An attacker must not only be able to inject their own code and gain control over the program counter, but also make the injected code executable. This process of making data executable 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. Then, the control flow only has to be diverted to a code sequence that executes the syscall, resulting in a straightforward code-reuse attack commonly known as return2libc [52]. Realistically, however, an attacker first has to get the location and size of the shellcode area into the corresponding registers. Achieving that is possible by relying on small code gadgets executing one or more instructions which an attacker can chain together to achieve arbitrary code execution. Depending on the type of gadgets, such attacks are commonly known as return-oriented-programming [62] or jump-oriented-programming attacks [5]. While such attacks relying purely on existing code are Turing complete [56], writing the entire payload in such a way is very tedious. Hence, attackers often use code-reuse attacks to spawn a shell or make shellcode executable [53]. As these workflows are common, they are also integrated into, e.g., Ropper, a tool to help with such attacks [60].

On an unprotected system, every application can execute the *mprotect* syscall. Depending on the application, the *mprotect* syscall can also not be blocked by seccomp if the respective application requires it. With SFIP, attacks that rely on *mprotect* can potentially be prevented even if the application requires the syscall. First, we consider a system where only the state machine is verified on every syscall execution. *mprotect* is a rare syscall that is mainly used in the initialization phase of an application [25], [9]. Hence, we expect very few other syscalls to have a transition to it, if any. This leaves a very small 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 in place, 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 V-B3). Additionally, all syscalls must originate from a location where they can legally occur. The attack succeeds only in this special case. These additional constraints significantly increase the security of the system.

2) Real-world Exploit: For a real-world application, we evaluate a stack-based buffer overflow present 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 the loaded shellcode cannot be marked as executable. There are

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

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.

3) Syscall-Flow-Integrity Protection and Mimicry Attacks: We consider the possibility of mimicry attacks [67], [68]. In a mimicry attack, 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 harder. An attacker needs to identify the last executed syscall and knowledge of the valid transitions for all syscalls. With this knowledge, the attacker then 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.

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 III-A).

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.

TABLE II: We evaluate various properties of applications station machines. These metrics include the average number of transitions per state, the number of states in the state machine, min and max transitions. The numbers for busybox and coreutils are the averages over all individual utilites (398 and 103 utilities, respectively).

Application	Average Transitions	#States	Min Transitions	Max Transitions
busybox	15.73	24.51	1	21.09
muraster	17.51	41.00	1	33.00
nginx	65.55	108.00	1	80.00
coreutils	15.75	27.11	1	23.00
ffmpeg	48.48	56.00	1	51.00
mutool	32.00	61.00	1	46.00

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 [67] 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 [9], [17], [25]. 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.

a) Results: We show the results of this evaluation in Table II. nginx shows the highest number of states with 108, followed by mutool and ffmpeg with 61 and 56 states, respectively. This is to be expected as they have the largest code base and provide many different functionalities. 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. Based on the source code of musl, this 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, which is to be expected when we consider the design of modern applications and what syscalls are used for. Consider a normal application written in a high-level language. The application must delegate certain 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. As these tasks additionally require syscalls, 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. Canella et al. [9] reported 105 syscalls in nginx and 63 in ffmpeg. Ghavamnia et al. [25] 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 security of the system.

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.

a) Results: We show the results of this evaluation in Table III. The average number of offsets per syscall indicates that most 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 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. Hence, without syscall-origin enforcement, an attacker can choose from 318 syscall locations to execute any syscall 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.

VI. Discussion

a) Limitations and Future Work: Our proof-of-concept implementation currently does not handle signals and syscalls invoked in a signal handler. A full implementation can efficiently solve this limitation. Wagner and Dean [67] propose to additionally monitor signal events and add a pre-guard and post-guard event to the control-flow graph. With that, certain transitions in the control-flow graph are then only possible upon the reception of such an event. Our proposal deviates from that, *i.e.*, does not require pre- and post-guard events in the control-flow graph, makes checks easier, and reduces the size of the main state machine. 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. Note that a similar approach might be feasible for generating per-thread state machines.

The state-machine construction we proposed in this paper leads to coarse-grained state machines. We propose an improvement that leads to fine-grained state machines with improved security. The improvement is based on 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 location of the previous and current syscall instruction in the virtual address space. This allows to better represent the syscall-flow graph of the application without relying on context-sensitivity or call stack information [67], [29], [61]. As this requires significant changes to the compiler and the enforcement in the kernel, as well as a thorough evaluation, we leave this for future work.

b) Related Work: In 2001, the seminal work by Wagner and Dean [67] introduced automatically generated syscall NDFAs, NDPDAs, and digraphs for sequence checks in intrusion-detection systems. We build upon the concept of a syscall digraph that they introduced but modify its construction and representation to allow for better performance. Our work then further extends upon theirs by additionally verifying whether a syscall originates from a valid location. The accuracy and

TABLE III: We evaluate various metrics for our syscall location enforcement. The metrics include the total number of functions containing syscalls, min and max and average number of syscalls per function, total syscall offsets found, average offsets per syscall, and the number of syscalls in the musl syscall wrapper functions used by the application. The numbers for busybox and coreutils are the averages over all contained utilites (398 and 103 utilities, respectively).

Application	#Functions	Min Syscalls	Max Syscalls	Avg Syscalls / Function	Total #Offsets	Avg #Offsets	#syscall()	#syscall_cp()
busybox	30.57	1	9.83	1.48	102.64	3.75	1.71	9.79
muraster	55.00	1	12.00	1.62	193.00	4.60	0.00	4.00
nginx	105.00	1	24.00	1.53	318.00	3.00	7.00	24.00
coreutils	36.86	1	4.21	1.38	116.71	4.42	1.00	3.41
ffmpeg	89.00	1	13.00	1.55	279.00	4.98	0.00	13.00
mutool	81.00	1	14.00	1.67	278.00	4.15	6.00	14.00

performance of our syscall-flow-integrity protection allows for real-time enforcement in large-scale applications.

Several papers have focused on extracting and modeling the control flow of an application, based on the work by Forrest et al. [21]. Frequently, such approaches rely on dynamic analysis of repeated runs of the application [23], [27], [32], [34], [45], [69], [48], [65], [70]. Other approaches rely on machine-learning techniques to learn the sequence of syscalls [49] or to detect intrusions [75], [54]. Giffin et al. [28] proposed incorporating environment information in the static analysis to generate more precise models. Other works have disregarded the control flow and instead focused on detecting intrusions based on syscall arguments alone [43], [51]. Forrest et al. [22] provide an analysis on the evolution of system-call monitoring.

Two prominent approaches to learn the sequence of syscalls are VtPath [20] and the Dyck model [29]. Both consider additional stack information and rely on context-sensitive models. 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.

A recent study provides a modern implementation of a syscall-sequence-based intrusion-detection system that relies on hidden markov models [8]. The approach relies on eBPF to hook the syscall exit handler. Since the discovery of Spectre [42], kernel developers have restricted access to eBPF. New, unprivileged use cases have faced considerable pushback from the kernel developers, making such an approach infeasible [13].

Recent, orthogonal work has investigated the possibility of automatically generating filters for seccomp [17], [9], [25], [26], either from the source code [9], [25] or from a binary [17], [9]. We expect that SysFlow can be extended to generate the required information from binaries. More recent work proposed an alternative to seccomp that maintains the speed of seccomp while also providing a secure way to perform complex argument checks [10]. In contrast to these works, we consider sequences of syscalls and the origin of a syscall, which requires additional challenges to be solved (cf. Section III-C).

A similar approach to our syscall-origin enforcement has been proposed by Linn et al. [46] and de Raadt [16]. The former extracts the syscall locations and numbers from a binary and enforces them on the kernel level, but their technique fails in the presence of ASLR. The latter is only able to restrict the execution of syscalls to entire regions of a binary, but not a precise location, *i.e.*, the entire text segment of a static binary is a valid origin. Additionally, in the entire region, any syscall is valid. Our work improves upon these works in several ways: First, we present a way to enforce the syscall location in the presence of ASLR, improving upon the former. Second, our approach limits the execution of specific syscalls to precise locations, improving upon the latter. Third, we improve upon both by considering the security benefits when combining such an approach with syscall state machines.

VII. 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. We showed that SFIP can be implemented based on three pillars: a syscall state machine, representing possible syscall transitions; a syscall-origin mapping, which maps syscalls to the locations at which they can occur; an efficient enforcement machanism, implemented within the Linux kernel. Based on these pillars, we demonstrated that SFIP can be fully automated on the compiler and operating-system level. Similar to seccomp, SFIP is opt-in, and thus fully backward compatible with legacy applications and operating systems. 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 $1.8\,\%$, 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 $38.6\,\%$ compared to seccomp and $90.9\,\%$ when no protection is applied.

ACKNOWLEDGMENTS

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No 681402). Additional funding was provided by generous gifts from ARM and from Amazon. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding parties.

REFERENCES

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

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

[3] Android, "Application Sandbox," 2021. [Online]. Available: https://source.android.com/security/app-sandbox

[4] AppArmor, "AppArmor: Linux kernel security module," 2021. [Online]. Available: https://apparmor.net/

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

[6] D. Bueso, "tools/perf-bench: Add basic syscall benchmark," 2019. [Online]. Available: https://lore.kernel.org/patchwork/patch/1048777/

[7] N. Burow, S. A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler, and M. Payer, "Control-Flow Integrity: Precision, Security, and Performance," *ACM Computing Surveys*, 2017.

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

[9] C. Canella, M. Werner, D. Gruss, and M. Schwarz, "Automating Seccomp Filter Generation for Linux Applications," in *CCSW*, 2021.

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

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

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

[13] ——, "eBPF seccomp() filters," 2021. [Online]. Available: https://lwn.net/Articles/857228/

[14] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton, "Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks." in *USENIX Security*, 1998.

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

[16] T. de Raadt, "syscall call-from verification," 2019. [Online]. Available: https://lwn.net/Articles/806863/

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

[18] J. Edge, "System call filtering and no_new_privs," 2012. [Online]. Available: https://lwn.net/Articles/475678/

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

[20] Feng, H.H. and Kolesnikov, O.M. and Fogla, P. and Lee, W. and Weibo Gong, "Anomaly detection using call stack information," in *S&P*, 2003.

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

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

[23] Garvey, Thomas D. and Lunt, Teresa F., "Model-based intrusion detection," in *NCSC*, 1991.

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

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

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

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

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

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

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

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

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

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

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

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

[36] K. K. Ispoglou, B. AlBassam, T. Jaeger, and M. Payer, "Block Oriented Programming: Automating Data-Only Attacks," in *CCS*, 2018.

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

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

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

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

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

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

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

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

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

[46] 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.

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

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

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

[50] Microsoft, "Data Execution Prevention," 2021. [Online]. Available: https://docs.microsoft.com/en-us/windows/win32/memory/data-execution-prevention

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

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

[53] One, Aleph, "Smashing the Stack for Fun and Profit," *Phrack*, 1996.

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

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

[56] Roemer, Ryan and Buchanan, Erik and Shacham, Hovav and Savage, Stefan, "Return-Oriented Programming: Systems, Languages, and Applications," *TISSEC*, 2012.

[57] R. Rogowski, M. Morton, F. Li, F. Monrose, K. Z. Snow, and M. Polychronakis, "Revisiting Browser Security in the Modern Era: New Data-Only Attacks and Defenses," in *EuroS&P*, 2017.

[58] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A. Sadeghi, and T. Holz, "Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications," in *S&P*, 2015.

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

[60] scoding.de, "Ropper - rop gadget finder and binary information tool," 2013. [Online]. Available: https://scoding.de/ropper/

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

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

[63] Spengler, Brad, "Recent ARM Security Improvements," 2013.

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

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

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

[67] Wagner, D. and Dean, R., "Intrusion detection via static analysis," in *S&P*, 2001.

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

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

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

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

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

[73] S. Wiki, "FAQ — SELinux Wiki," 2009. [Online]. Available: http://selinuxproject.org/w/?title=FAQ&oldid=729

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

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