Save Willy: ...

Abstract

Most privilege escalation exploits targeting Docker containers on Linux abuse vulnerable system calls to threaten the container's isolation. Nevertheless, reducing the surface of this attack vector receives insufficient attention—despite highly effective system call filtering capabilities of the Linux kernel through *seccomp*. Generally, creating robust, yet thorough, seccomp policies by hand is a difficult task and hence calls for automated assistance. Various seccomp policy generation frameworks build upon dynamic analysis. Yet, inherent characteristics of dynamic, testing-based analysis are prone to false negatives, an intolerable inconvenience in production systems. On the other hand, static analysis has the power to produce conservatively overapproximated, but correct results.

In this paper, we present JESSE, a static analysis based framework for generating seccomp policies for Linux Docker containers. Specifically, we design and implement an abstract interpretation that assists the analyst in identifying the system calls that are vital for C/C++ binaries. By combining JESSE's abstract interpretation with container and library debloating techniques, we produce accurate and highly-effective seccomp policies that restrict up to 45% more system calls than the Docker's default seccomp policy on average. We have applied JESSE to construe seccomp policies for five of the most prominent Docker containers. Overall, our evaluation shows that (contrary to dynamic analysis) JESSE does not introduce any false negatives and offers effective protection against real-world exploits.

1 Introduction

Recent advances in Operating System (OS) level virtualization have successfully propagated this, once rarely used, technology to the masses. In general, OS-level virtualization techniques leverage services of the underlying OS to establish light-weight isolated execution environments, known as containers. Prominent container implementations are Linux Containers (LXC) [4], BSD jails [1], and Solaris Zones [5]; yet it was Docker [2] that gained the most popularity. Docker has become the de facto standard for containers in both private and industry sectors for shipping various applications in a convenient and platform-independent way. Unlike system virtualization techniques that implement a virtual hardware interface, i.e., the Virtual Machine (VM), that offers fully-fledged execution environments to OSes, containers share the same OS kernel with their host and solely abstract the view on global kernel resources. As such, potential kernel exploits originated from inside a container can impair the security of other containers on the same system and even the host itself.

The *principle of least privilege* mandates every entity to access only those resources that are necessary for its execution. Contrary to this concept, modern OS architectures offer applications a uniform interface, i.e., the *system call interface*, that grants access to an immense number of system calls, entirely independent of how many system calls the application requires. For instance, the Linux kernel v5.5 comprises 347 distinct system calls, excluding the number of compatibility system calls. Sadly, a vulnerability in one of those system calls has the potential to open the gate to the underlying kernel [22, 23, 30], despite modern isolation and security mechanisms, including Linux namespaces, control groups, capabilities, Mandatory Access Control (MAC), Kernel Address Space Layout Randomization (KASLR), and Supervisor Mode Execution and Access Prevention (SMEP/SMAP). Indeed, most privilege escalation exploits targeting containers on Linux abuse vulnerable system calls to overcome the isolation enforced by the container and Linux kernel [43].

One approach to mitigate this threat is to remove or filter out not needed system calls that are otherwise freely available to applications and containers. For instance, recent advances in *library debloating* [8, 37, 38, 48, 50, 51] remove code regions in the process' address space that is irrelevant to the program's execution. While library debloating mainly focuses on reducing Return-Oriented Programming (ROP) (or similar code-reuse) gadgets, it does not eliminate the threat; unavoidable remaining gadgets could allow the attacker to still mount an attack that, for instance, abuses a vulnerable system call

handler—not used by the program—to subvert the system. Contrary, as a last line of defense, the seccomp (secure computation) facility of the Linux kernel provides the necessary means to establish system call filters, which restrain applications and containers to use only white-listed system calls [17]. However, to this point, there is no easy way to automatically tailor seccomp policies that are suitable for general-purpose applications and containers.

So far, researchers have suggested different techniques that *dynamically* analyze applications to identify the system calls required by the application [47] or container [56] under test. Specifically, they employ automatic test generation techniques that aim to trigger different program behaviors, which, in turn, identify different system calls required for the particular functionality. This strategy allows collecting most of the system calls that are similarly accessed in production. Nevertheless, dynamic analysis suffers from *incompleteness*, as the triggered execution traces severely limit the analysis; if a particular system call has not been identified by the dynamic analysis, which bases its results on past execution traces, it does not necessarily mean that it will not be called in production. In fact, we have identified a set of system calls that were either completely missed or falsely interpreted by the dynamic analysis based policy generation techniques of Wan et al. [56] (in both cases, normally authorized system calls were blacklisted). Consequently, due to incomplete information, *false negatives* can interrupt and terminate the execution of sandboxed environments, which is highly impractical in production systems.

In this paper, we present JESSE, a system that leverages static analysis to create seccomp policies to establish a safe environment for Docker containers. Specifically, we statically analyze non-obfuscated programs and the associated libraries to identify all system call invocations. Since the invocations of system calls do not necessarily reveal the requested system call (a system call invocation is merely the execution of the `syscall` instruction),[1] we leverage *abstract interpretation* [21] to derive the contextual semantics required to approximate the set of system calls a container should be authorized to use. We consider the fact that essential general-purpose libraries, such as `libc` on Linux, provide user-space applications with wrappers to a vast number of system calls. To approach this, we further apply dead code elimination techniques to narrow down the set of all identified system calls in the considered binaries to the set of system calls that are necessary for the program's genuine execution. By applying abstract interpretation techniques, we ensure that the gathered results, by design, cannot introduce any false positives.

We employJESSE to generate seccomp policies for five of the most prominent Docker container images that are available on Docker Hub. On average, we manage to block 55.8% of all system calls and hence significantly increase the effectiveness of the default seccomp policy for Docker containers, which conservatively blocks around 10.6% completely, and up to 20.4% in combination with restrictions of additional Linux capabilities. Finally, we evaluate the precision and security of our system.

In summary, we make the following main contributions:

- We introduce JESSE, a static analysis based framework for generating last line of defense Linux seccomp policies for C/C++ binaries in Docker containers.
- We develop an abstract interpretation based static analysis for *conservatively* connecting system calls to identified `syscall` instructions in binaries.
- We apply JESSE to create seccomp policies for five of the most prominent Docker containers, to thwart entailed container escape vulnerabilities.
- We evaluate JESSE and show that we manage to block 55.8% of all system calls, without false negatives as opposed to dynamic analysis based systems.

2 Background

There exist two flavors of program analysis techniques: dynamic and static analysis. Due to their inherent characteristics, both flavors differ in accuracy of the gathered analysis results, as they might produce either *false negatives* or *false positives*. In this paper, we consider false negatives as system calls that were falsely missed during the analysis and hence are not considered in the set of authorized (i.e., whitelisted) system calls; execution attempts of these excluded system calls cause the program to crash. Dynamic analysis is known to produce false negatives as it solely relies on the given execution traces of a program and cannot consider all of its execution paths.

On the other hand, we consider false positives as system calls that were (conservatively) considered as vital during the analysis and hence, at the risk of being never executed, were added to the set of whitelisted system calls. Static analysis produces false positives as its assumptions abstract unnecessary details and rather overapproximate the results. Consequently, we resort to a specific static analysis technique, *abstract interpretation* [21], that we utilize to establish profiles holding whitelisted system calls, which we enforce with help of the Linux Secure Computing mode [17,19] for individual binaries executed in Docker containers.

2.1 Linux Secure Computing Mode

The Linux Secure Computing mode, known as *seccomp* [17, 19], is a feature of the Linux kernel that confines processes to a set of black- or white-listed system calls. Optionally, seccomp also allows filtering the system call's arguments

[1] While we assume the analyzed binaries use the modern 64-bit *fast system call* instruction, `syscall`, the analysis' concepts are by no means limited to it and can be extended to support different system call instruction variants.

We need some good publications here. Also, mention further research strategies if we have missed any so far. We should also probably incorporate all those debloating!!! papers at this point.

We need some results regarding the precision and security

TBD. It would be nice to have a CVE for each container type in the evaluation.

to restrain processes further. In the most restricted mode, `SECCOMP_SET_MODE_STRICT`, seccomp grants processes only to the four system calls `read()`, `write()`, `exit()`, and `sigreturn()`; the invocation of any other system call results in an immediate termination of the calling process. On the other hand, the mode `SECCOMP_SET_MODE_FILTER` delegates the filtering decision to a user-provided Berkely Packet Filter (BPF) program [] that can be installed per process; before every system call invocation, the in-kernel BPF bytecode Just In-Time (JIT) compiler executes the BPF program, with the invoked system call number and arguments as input, to enforce a given policy. This allows the BPF program to decide whether (i) to normally execute the system call, (ii) to kill the calling process, or (iii) to return an error.

citation

To improve the security of containers, Docker adopted seccomp profiles and highly simplified their usage. Instead of requiring the user to provide complex BPF programs, Docker expects a JSON-formatted profile that holds the black- and white-listed set of system calls. The profile further allows defining fine-grained policies per system call, for example, by restricting its allowed arguments or by binding the availability of the system call to a specific Linux capability (such as `CAP_SYS_ADMIN`). Docker (*v18.10.0, dev*) ships a default seccomp profile that unconditionally permits 276[2] (out of 347 available) system calls [6] on Linux kernel (*v5.5*), which is about 80.4% of all available system calls [6, 7]. By combining seccomp with Linux capabilities, the default seccomp profile may grant up to further 34 system calls. Thus, the default seccomp profile is very coarse-grained (it only forbids around 10.6% of all available system calls completely and up to 20.4% when combined with Linux capability restrictions) the refinement of which is the main focus of this work.

2.2 Abstract Interpretation

Abstract interpretation [21] is a theory based on static analysis that allows approximating the semantics of programs. For instance, it has become a useful tool for detecting critical information leaks in programs [59]. In this paper, we leverage abstract interpretation to determine the set of system calls that is relevant for the program's genuine execution to tailor seccomp policies for containers.

find more/other examples (e.g., compiler optimization Check out this paper: https://www.usenix.org/conference/usenixsecurity19/presentation/wang-shuai Also, check out (at least) slides 96 and 97 in http://www2.informatik.uni-freiburg.de/~heizmann/Program_verification/slides/20111025-Tue.pdf

To identify the semantic properties of programs, abstract interpretation operates on the program's abstract representation. While the founders of abstract interpretation, Cousot et al. [21], leverage *finite flowcharts* to represent programs, in this paper, we resort to the more commonly used program representation through Control-Flow Graphs (CFGs). When applying abstract interpretation to a CFG, it annotates its edges (i.e., paths or transfers of the control-flow) with symbols representing (possibly infinite) sets of program states. Cousot et al. model a program's state as a mapping from variables to their values. Once the abstract interpretation terminates, the annotated symbols represent sets of states that can be reached at the associated edges; if a given program state is not part of the set of the computed program states at a particular edge in question, the program will never reach this state at that edge, independent of the program's input. At the same time, the computed sets may hold states that will never be reached. In most cases, an exact computation of all program properties is not feasible. Thus, abstract interpretation only provides an *overapproximation* of the exact program semantics.

Since it is not feasible to explicitly store all sets of possible program states, abstract interpretation operates on a user-provided, predefined set of symbols (S) instead. This allows the abstract interpretation to operate on the abstract symbols instead of the sets of concrete program states. Cousot et al. further extend the sets of symbols to a complete lattice L that comprises the following elements to assemble the quintuple $L = (S, \leq, \sqcup, \bot, \top)$. As such, the lattice requires a *less-than-or-equal-to* relation ($\leq$) and a *join* operation ($\sqcup$) that can be applied on the set of symbols. Both are necessary, as they simulate the *subset relation* ($\subseteq$) and the *union operation* ($\cup$) for the sets of concrete program states. This means, instead of unifying two sets of concrete program states, the abstract interpretation joins the symbols representing the sets; similarly, instead of applying the subset relation, it uses the *less-than-or-equal-to* relation from the lattice. Finally, to form a complete lattice, it must contain a *top* ($\top$) and a *bottom* ($\bot$) element. These represent the largest and the smallest element ($\forall s \in S : \bot \leq s \leq \top$), respectively, with regard to the *less-than-or-equal-to* relation in the set of symbols S. These elements simulate the empty set ($\bot$) and the universe on the sets ($\top$) of concrete program states.

Similar to concrete program states that are lifted to an abstract representation in the form of lattice elements ($s \in S$), abstract interpretation further requires lifting concrete operations on concrete program states to abstract operations on lattice elements. This is where the *interpretation function* ($Int(e, C)$) comes into play. It has to be tailored to the specific needs such that it specifies how the basic blocks in the CFG operate on the lattice elements. For example, in a scenario, in which the goal is to determine whether the values of variables in a given program are even or odd, it is the task of the interpretation function to infer whether the computation results in an even or odd value. Similar to Cousot et al., we assume that the interpretation function receives a specific edge of the CFG as a first argument and the complete CFG (including annotations of the previous rounds) as a second argument. The interpretation function computes the new annotation symbol for the particular edge and returns the symbol to the main abstract interpretation algorithm to update the CFG.

Algorithm 1 describes the abstract interpretation which statically analyzes programs by annotating the edges of the program's CFG with program states that can be reached at the

[2] Docker's default policy whitelists 315 system calls, out of which 41 either do not exist on x86 or are (redundant) compatibility system calls. Another 2 system calls are granted with argument restrictions.

Algorithm 1: The abstract interpretation algorithm which annotates the edges of the program's CFG with user-provided symbols. The annotations represent the sets of all reachable program states at the particular edge.

Input : A CFG $C = (N, E)$, a complete lattice $L = (S, \leq, \sqcup, \bot, \top)$, and an interpretation function $Int(e, C)$, with $e \in E$
Output : A CFG whose edges are annotated with symbols representing sets that contain all reachable program states at the particular edge

```
1  foreach e ∈ E do
2      annotate e with ⊥;
3  repeat
4      changes:=∅;
5      foreach e ∈ E do
6          oldAnnotation:=get annotation at e;
7          newAnnotation:=Int(e,(N,E));
8          if oldAnnotation ≠ newAnnotation then
9              changes:=changes ∪ {(e,newAnnotation)};
10     foreach (e,newAnnotation) ∈ changes do
11         annotate e with newAnnotation;
12 until changes = ∅;
13 return (N,E);
```

particular edge. The algorithm runs in rounds until it reaches a *fixpoint* (i.e., executing another round would not change the annotations). As input, the abstract interpretation receives a CFG C comprising the set of nodes N and set of edges E, a complete lattice L, and an interpretation function $Int(e, C)$. During initialization, the algorithm annotates all edges in the CFG with the bottom element of the lattice (lines 1 to 2). Then, the computation enters the main loop (lines 3 to 12), which is responsible for updating the annotations (lines 10 to 11). Specifically, the main loop calls the interpretation function in line 7 until the annotations of CFG do not change anymore (i.e., until the abstract interpretation reaches a fixpoint).

To demonstrate how to leverage abstract interpretation, let us consider a scenario in which we analyze a part of a function to determine whether or not it will always return even values in the general-purpose register `rcx`. In this context, we specify a lattice $\dot{L} = (\dot{S}, \dot{\leq}, \dot{\sqcup}, \dot{\bot}, \dot{\top})$ and an interpretation function $\dot{Int}(e, C)$. The lattice $\dot{L}$ comprises the set of symbols $\dot{S} := \{B, E, O, A\}$. The symbol B (*blank*) represents the empty set of states (i.e., we use B to represent unreachable edges); the symbols O and E describe that `rcx` can only hold *odd* (O) or *even* (E) values, respectively; and the symbol A (*all*) means that `rcx` can hold any (whether even or odd) natural number. Further, the *bottom* ($\dot{\bot}$) and the *top* ($\dot{\top}$) elements are represented by B and A. These abstract symbols over-approximate the concrete set of values. That is, if an edge is annotated with A, it means that `rcx` *can* hold any natural

Consider including the example program already here. It is not clear that you decrement rcx. Interpretation function is not clear without an example.

Algorithm 2: Interpretation function $\dot{Int}(e, C)$ that determines the new annotation in form of a lattice element for the provided edge e.

Input : Edge e and the annotated control-flow graph C
Output : Lattice element $s \in \dot{S}$

```
1  bb := origin(e);
2  pre := B;
3  foreach e' ∈ incoming(bb) do
4      a :=annotation at edge e';
5      pre := pre ⊔̇ a;
6  if pre = B then
7      return B;
8  instr :=extract instruction in bb;
9  if instr writes to rcx then
10     imm :=extract immediate in instr;
11     return { E  if imm mod 2 = 0
               { O  otherwise                ;
12 else if instr = loop rel then
13     rel :=extract jump target in instr;
14     if e targets rel then
15         return { A  if pre = A
                  { O  if pre = E
                  { E  otherwise             ;
16     else
17         return { B  if pre = E
                  { E  otherwise             ;
```

number at the particular edge, but it does not have to.

Further, we define the *less-than-or-equal-to* ($\dot{\leq}$) relation and the join ($\dot{\sqcup}$) operation of the lattice $\dot{L}$ as follows. B is the smallest and A is the largest element (E and O are incomparable). That is, with $a, b \in \dot{S}$:

$$a \dot{\leq} b :\iff (a = B) \vee (b = A) \vee (a = b)$$

A join with B (the smallest element) always returns the original element. In all the other cases (i.e., $A \dot{\sqcup} E$, $A \dot{\sqcup} O$, $E \dot{\sqcup} A$, $O \dot{\sqcup} A$, $O \dot{\sqcup} E$, and $E \dot{\sqcup} O$), the result is A:

$$a \dot{\sqcup} b := \begin{cases} a & \text{if } b = B \\ b & \text{if } a = B \\ a & \text{if } a = b \\ A & \text{otherwise} \end{cases}$$

Algorithm 2 illustrates the interpretation function $\dot{Int}(e, C)$. This function takes an edge e of the CFG as first argument, the CFG C itself as second argument, and returns a lattice element $s \in \dot{S}$ as annotation for the provided edge e. To compute the annotation, the function first joins the annotations of

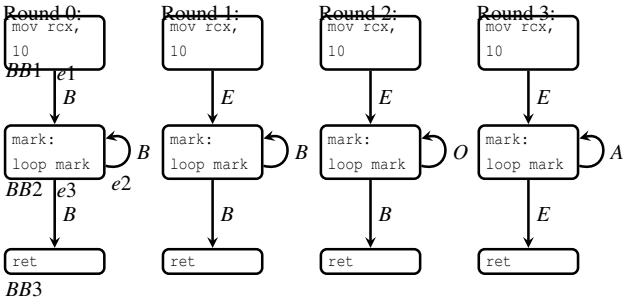

Figure 1: Abstract interpretation is applied to the program (in form or a CFG that decrements the value of `rcx` from 10 to 0 in a loop) to determine whether the returned value in `rcx` is *even* (E) or *odd* (O). The iterative annotations of the CFG added by the abstract interpretation are illustrated in four rounds.

incoming edges of the basic block from which the edge in the first argument originates ($origin(e)$ returns the basic block from which e originates; $incoming(b)$ returns all edges that target b) in lines 1 to 5. Assuming the basic block is reachable, the interpretation function extracts the instruction (for simplicity reasons, we assume that the basic block holds only one instruction) in line 8 and abstractly simulates its execution. In case the instruction writes to `rcx`, the function interprets the immediate value *imm* and returns E or O, depending on whether *imm* is even or odd (lines 9 to 11). If the extracted instruction is a branch to the beginning of the loop (`loop` *rel*), the function distinguishes between two cases depending on whether the edge represents a jump to the beginning of the loop or a fallthrough. In the latter, we return B if the join of the annotations on the incoming edges results in E, because `loop` decrements `rcx` and only terminates the loop if the result is zero and since the decrement of an even number can never be zero, this edge is never taken. Else, we return E, because `rcx` is guaranteed to be zero if it falls through and zero is even. In the case of the `jump`, we only have to consider that `rcx` is decremented by the `loop` instructions and that this flips it from even to odd and vice versa. In the case of A, this does not matter and we preserve it.

To collect the building blocks of the abstract interpretation, Figure 1 illustrates the iterative annotation results of applied to our code fragment example in the form of a CFG. We start in `Round 0` (on the left) with all edge annotations set to B. In `Round 1`, the edge leaving $BB1$ ($e1$) is annotated with E, because $BB1$ always sets `rcx` to 10 which is an even number. The other annotations do not change, because the union of the edges $e1$ and $e2$ in round zero is B which means that the control-flow never reaches $BB2$ in this round. Hence, it does not reach the edges $e2$ and $e3$, too. In `Round 2`, the same argumentation as in `Round 1` still applies for $e1$ and the union of $e1$ and $e2$ now returns E. Thus, the edges of $e2$ is annotated with O, because decrementing an even value always results in an odd value. The annotation of $e3$ remains B because this edge is not reachable since zero is an even value, and we currently deduce that `rcx` is odd when the control-flow reaches the end of $BB2$. In the last round, we annotate $e2$ with A, because the union of E and O is A and incrementing an arbitrary natural number results in another arbitrary number. In the case of $e3$, the same situation applies initially, but since the edge is only taken when `rcx` contains zero, the analysis recognizes that zero is even and annotates $e3$ with E.

Could you annotate all edges equally pls? Also, maybe use something like that: e1:bot in the figure

Adjust!

this still needs more restructuring and polishing but lets focus on the more important parts.

Which algorithms/tools are used for that? I mean the analysis needs to be quite good in understanding the program's semantics! Provide more info on that in the BG.

3 Threat Model

We assume an attacker inside of a Docker container, from which intends to escape; the attacker aims to take over the host system by abusing the system call interface to exploit kernel vulnerabilities. We assume the container to be isolated from other applications via state-of-the-art OS-level virtualization techniques including Linux namespaces [35], control groups [14], and capabilities [16, 26]. Further, we assume the system employs kernel and user space Address Space Layout Randomization (ASLR) [25, 42] and is protected against code-injection attacks through SMEP/SMAP [18, 32, 57] and effective W^X policy enforcement mechanisms. In addition, defenses against data-oriented attacks [15, 36, 44, 46] are orthogonal to our work and can be applied independently. The same applies to container [], configuration [37], and library debloating [] techniques that can further reduce the attack vector.

Refs!

Under the assumption that the upper state-of-the-art defenses restrain the attacker inside the isolated environment, potential memory corruption vulnerabilities in system call handlers can still allow establishing primitives that enable them to read or write to the kernel memory arbitrarily. Such primitives can be abused to (i) leak information to defeat KASLR, (ii) deactivate SMEP/SMAP, and hence (iii) pave the way for the attacker to inject and execute her payload. In order to limit this attack vector, we concentrate on reducing the system call interface that is available to the attacker.

4 System Call Number Analysis

In this work, we use static analysis to identify the set of system calls a program requires for its execution. This set of system calls allows us to automatically compile effective seccomp policies for Docker containers to filter out unnecessary and potentially vulnerable system calls (that would have been otherwise freely available) and hence reinforce the isolation capabilities of Docker containers on Linux. In the following, we present the building blocks that allow us to harvest the needed system calls from Executable and Linkable Format (ELF) binaries. Specifically, we statically analyze programs

and their required libraries to locate all invocations of the `syscall` instruction. Further, by leveraging abstract interpretation, we derive the contextual semantics so as to identify the system call. We divide our analysis into two stages. In the first stage, JESSE assembles a CFG for each function in the given binary and identifies the locations of the `syscall` instructions. The second stage leverages our abstract interpretation scheme to derive the *system call number* that is passed as argument to the identified `syscall` instruction.

4.1 Control-Flow Graph Construction

The CFG construction is the first step of our analysis. It transforms the code in the binary into a CFG, which can be further processed by the abstract interpretation (§ 2.2). A CFG is a tuple (N,E) comprising a set of nodes (i.e., basic blocks) N and a set of edges (i.e., control-flow transfers between basic blocks) E. Since inaccuracies in the CFG can produce incorrect assumptions for the abstract interpretation, it is crucial that the CFG fulfils the following three requirements: ❶ *Correctness of N*; ❷ *Completeness of N*; and ❸ *correctness of E*. These requirements are necessary as their violation could lead to misinterpretation of the target binaries.

Unfortunately, not every CFG construction framework assures the requirements ❷ or ❸. For instance, both *angr* [53, 55] and *radare* [49] can miss basic blocks and/or introduce invalid edges in certain situations. For instance, angr's static analysis based CFG generation tool, CFGFast, can miss code (i.e., violate ❷) on indirect branches [9] (i.e., the branch target addresses are computed at run-time). Respectively, angr's symbolic execution based CFG generation mechanism, CFGEmulated, can introduce illegal edges (i.e., violate ❸), because it relies on symbolic execution to overapproximate the set of possible jump targets [9]. Consequently, instead of relying on (potentially insufficient) properties of CFG generation tools, we extend the minimalistic Lightweight Disassembler (LWD) [27] to provide the upper guarantees for control-flows across program and library binaries.

Correctness of N. Generally, all basic blocks in a CFG must have a single entry and exit point (and correspond to a code sequence in the binary). Further, `branch` instructions that transfer the program's control-flow to a location other than the immediately following instruction conclude the basic block. Most CFG construction tools provide this guaranty.

Correctness of E. Contrary to conventional CFG construction frameworks, LWD only considers edges of *direct* branches (including direct relative, absolute, and (un)conditional branches); it avoids heuristics for indirect branches whose targets depend on general-purpose register contents (other than `rip`) that cannot be statically determined. For instance, the instructions `ret` and `jmp reg` always conclude a basic block, yet they do not introduce any edges.

Completeness of N. Note that our guarantees concerning the *correctness* of E can lead to undiscovered code sequences that can hold `syscall` instructions; without considering all code regions, LWD would miss (in this way) hidden `syscall` instructions during the CFG construction. To counteract this issue, we extend LWD to maintain an interval tree that facilitates coverage monitoring of the binary's code section. LWD updates the interval tree with the address of every disassembled instruction that is placed into a basic block of the CFG until it fully covers the binary's code section.

Further, we modify LWD such that it does not regard `syscalls` as control-flow changing instructions. Instead, they are regarded as instructions that can alter registers according to the Linux calling conventions of the x86-64 Application Binary Interface (ABI) [31]. In addition, we cause LWD to split the basic blocks holding a `syscall` instruction into two basic blocks that we connect through an artificial edge. In this way, we ensure that the second part of the split basic block begins with the `syscall` instruction. This strategy simplifies the identification of the system call number that is passed as a parameter for `syscall`.

4.2 System Call Number Identification

The modern x86-64 architecture implements the *fast system call* instruction, `syscall` [32], that allows less-privileged user-space applications (`ring 3`) to make use of services provided by the high-privileged kernel (`ring 0`). Upon the execution of the `syscall` instruction, the system switches the context and executes the registered system call dispatcher in kernel space. The system call dispatcher then inspects the number passed in the `rax` register to determine which system call handler has been requested [31]. As such, we apply abstract interpretation (§ 2.2) to determine the system call number (that is passed in `rax`) for every identified `syscall` instruction in the CFG of the given binary. In the following, we define a complete lattice $\dot{L}$ and the interpretation function $\dot{Int}(e,C)$ that form our abstract interpretation.

4.2.1 Defining the Lattice

The lattice $\dot{L}$ of our abstract interpretation is a quintuple $(\dot{S},\dot{\leq},\dot{\sqcup},\dot{\bot},\dot{\top})$. Note that the system call numbers that are passed through the `rax` register to the system call dispatcher are neither read from memory nor are they the result of arithmetic computations. Instead, the `rax` register is either (i) initialized with an immutable immediate instruction operand or (ii) assigned a value from a different register. While (i) can be trivially determined by identifying the assignment of `rax`, pinpointing the exact value in (ii) must be accomplished through constant propagation techniques. To address both cases, we define our lattice to keep track of the constant values that are propagated through the general-purpose registers.[3] Con-

[3] We exclude `rsp` (stack pointer) and consider only 15 general-purpose registers. These include `rbp` as the compiler can reuse the frame pointer.

sequently, we define every element $s \in \dot{S}$ to hold either the symbol NX (i.e., the neutral element; the associated edge gets *Never eXecuted*) or a 15-dimensional vector (i.e., the register state). This vector maps each register to an element in the set $\{X, 0, \ldots, 2^{64}-1\}$ (i.e., to a constant or to an unknown value X). Note that we did not limit our analysis to merely considering the set of existing system call numbers; instead we decided to consider the complete range of numbers. This decision does not affect the performance and allows us to apply the analysis to system calls that could be potentially added in the future. Formally, $\dot{S}$ adheres to the following definition:

$$\dot{S} := \{NX\} \cup f : R \to \{X, 0, \ldots, 2^{64}-1\})$$

with R representing the set of general-purpose registers:

$$R := \{\texttt{r[a|b|c|d]x, r[s|d]i, rbp, r[8-15]}\}$$

Further, we define the *less-than-or-equal-to* relation $\dot{\leq}$, with NX as the smallest element. Two vectors $a, b \in S \backslash \{NX\}$ fulfill $a \dot{\leq} b$ if they are identical or if elements of b differ only in their mapping to X. Hence, the following holds for $\dot{\leq}$:

$$a \dot{\leq} b :\iff (a = NX) \vee \bigwedge_{i \in R}((a(i) = b(i)) \vee (b(i) = X))$$

The join operation $\dot{\sqcup}$ unifies two lattice element vectors $a, b \in \dot{S}$. By unifying two vectors, for each element (i.e., register) of the new vector, we preserve the old element's value (if both vector elements are identical). Otherwise, we assign the symbol X to the new element, as the associated register can hold multiple values. Further, if either a or b is assigned the value of the neutral element, NX, the join operation takes the assignment of the respective other lattice element. Thus, the following defines $\dot{\sqcup}$:

$$a \dot{\sqcup} b := \begin{cases} a & \text{if } b = NX \\ b & \text{if } a = NX \\ \lambda r \in R.r \mapsto \begin{cases} a(r) & \text{if } a(r) = b(r) \\ X & \text{if } a(r) \neq b(r) \end{cases} & \text{else} \end{cases}$$

Finally, we specify the bottom element of the lattice $\dot{L}$ ($\dot{\bot}$) to be the smallest element (i.e., $\forall a \in \dot{S} : \dot{\bot} \dot{\leq} a$) and the top element ($\dot{\top}$) of the lattice to be the largest element (i.e., $\forall a \in \dot{S} : a \dot{\leq} \dot{\top}$). Therefore, $\dot{\bot}$ inevitably represents NX. Also, since the vector that assigns all registers to X stands for all possible program states, $\dot{\top}$ represents $\lambda r \in R.r \mapsto X$:

$$\dot{\bot} := NX$$
$$\dot{\top} := \lambda r \in R.r \mapsto X$$

4.2.2 Defining the Interpretation Function

Abstract interpretation leverages an interpretation function to project how the basic blocks in the CFG operate on abstract lattice elements that represent program, or in our case, register states (§ 2.2). To identify the exact system call number in `rax`, our interpretation function applies symbolic execution [53–55] . Specifically, we leverage the symbolic execution framework *angr* [9] to propagate already assigned constant values (in the register state) and to determine the register assignment in basic blocks. Angr converts a loop-free sequence of x86-64 instructions (i.e., a basic block) into an equivalent *logic formula* that represents all possible execution traces of the basic block. By means of a Satisfiability Modulo Theories (SMT) solver (e.g., Z3 [58]), the interpretation function checks whether the formula can be satisfied, and if so, it determines the exact constant values assigned to the register state. Subsequently, the interpretation function propagates the lattice elements to the main abstract interpretation algorithm (which, in turn, incorporates the results by annotating the edges) to eventually identify the system call a particular `syscall` instruction refers to.

many many refs

To further clarify the details, Algorithm 3 describes the interpretation function $\dot{Int}(e, C)$ that receives an edge e and an annotated CFG C as parameters. The interpretation function identifies the basic block, *origin*, out of which the given

Algorithm 3: Interpretation function $\dot{Int}(e, C)$ that determines the register state for the given edge e by propagating constants and determining constant assignments to registers in the preceding basic block.

Input : Edge e and an annotated control-flow graph C
Output : Lattice element $post \in \dot{S}$

```
1  pre := NX;
2  foreach e' ∈ incoming(origin(e)) do
3      a := get annotation at edge e' in C;
4      pre := pre ⊔̇ a;
5  if pre = NX then
6      return NX;
7  S:=(initial) start state of symbolic interpreter;
8  foreach r ∈ R do
9      if pre(r) ≠ X then
10         set register r to pre(r) in S;
11 Symbolically execute origin(e) starting with start state S;
12 F:=the logic formula resulting from the symbolic
     execution;
13 post:=λr ∈ R.r ↦ X;
14 if F is satisfiable then
15     foreach r ∈ R do
16         if there is only one value for r such that F is
             satisfiable then
17             post(r):=the unique value for r that
                 preserves the satisfiability of F;
18 return post;
```

edge e originates, and collects annotations of origin's incoming edges. To propagate constants in the register state that precedes the basic block origin, the algorithm joins the annotations of its incomming edges in lines 1 to 4. This defines the initial state of origin. Then, in lines 7 to 11, the interpretation function transforms the origin's initial (register) state into its symbolic representation and uses *angr* to symbolically execute the basic block. At this point, the function instructs *angr* to transform the basic block into a logic formula (that expresses the constraints given by the basic block). For each register $r \in R$, *angr* extends the logic formula by a clause stating the registers that hold a constant value at the basic block's entry point. Once angr finalizes the logic formula, we use angr's SMT solver to determine which of the registers in R hold a constant value after executing the basic block (lines 13 to 17). This concludes the interpretation function, which then returns a lattice element, representing the register state after executing the basic block origin (i.e., the annotation for the edge e), to continue the abstract interpretation (Algorithm 1).

4.2.3 Applying Abstract Interpretation

Given the right tools at hand (CFG C, Lattice $\dot{L}$, and $\dot{Int}(e,C)$), we are able to apply abstract interpretation to identify the system call number that is passed to an identified `syscall` instruction in a binary. Once, the abstract interpretation has made a pass through the CFG, every edge in the CFG will be annotated with a lattice element representing the register state after each basic block. This includes the artificially included edge e' that immediately precedes the identified `syscall` instruction (§ 4.1). That is, to determine the system call number for a particular `syscall` instruction, we have to examine the annotation of e' and extract the final value from the `rax` register. In case `rax` maps to a constant value ($\texttt{rax} \in \{0,...,2^{64}-1\}$), we can be sure that the `syscall` instruction will always receive the same system call number. Otherwise, if `rax` maps to X, we will not be able to deduce the system call number. In this case, we inform the analyst by stating the incompleteness of the analysis; she will have to complete the result before generating the seccomp filter.

5 Refining the Seccomp Policy for Docker

We demonstrate the effectiveness of JESSE by generating seccomp filters for Docker containers. Specifically, for each involved program in the container, we leverage the introduced abstract interpretation to identify the system calls that are essential for its legitimate execution (§ 4). This set of system calls allows us to compile and apply an effective seccomp policy that forbids applications to execute unneeded and potentially vulnerable system calls inside the Docker container.

5.1 Dissecting Docker Containers

Docker uses a *Dockerfile* (i.e., a structured document holding instructions) to build containers. Amongst other things, this file specifies, the entry point, `ENTRYPOINT` (or `CMD`), representing the main program that is to be executed inside the isolated environment. Although, it is advised to provide only one service per container, it is not strictly prohibited to share one container among multiple services. Also, even if the container comprises only one service, its setup can require additional programs, e.g., to prepare the environment (i.e., switch to another user, initialize a database, etc.). In other words, the specified executable can engage multiple programs in the Docker image, with each requiring an own set of libraries. Unfortunately, Docker images are known to incorporate a significant amount of unneeded programs and libraries [12,24,39,50,51]. Consequently, before JESSE can determine the inevitable set of system calls required for the container, it identifies all programs that are necessary to prepare and provide the intended service(s). To achieve this, we leverage existing container debloating techniques [12,24,39,50,51] to dissect the container image and hence precisely narrow down the set of programs to receive accurate analysis results in the following steps.

5.2 Avoiding Unreachable Code

One of the inherent properties of shared libraries is their general-purpose character. Similar to a Swiss army-knife, they are built to provide functionality for all intents and purposes. At the same time, due to this characteristic, libraries are often considered as *bloated* [8,48], especially in the face of individual programs that often resort to only a very small fraction of the entire library. For instance, recent research has shown that 2016 Ubuntu applications make use of merely 5% of the standard C library (i.e., `libc`) on average [48]. Similar to library debloating frameworks [8,48], we consider such fully-blown libraries as security threats, as they can govern the access to unneeded (and potentially vulnerable) system calls that the target program binaries never intended to use (in a benign setting). Consequently, before determining the system call numbers for identified `syscall` instructions, we have to locate all library code regions that are unreachable by the benign program binaries, and exclude these from the subsequent system call number identification.

more refs!

Recent advances in code debloating on source code [48] and (unstripped) binary level [8] have demonstrated their effectiveness and precision in locating unreachable code in libraries. Given sufficient ground of research, we have independently implemented an approach very similar to Nibbler [8], which identifies unreachable code in libraries on binary level by using relocation information (as we do not claim any novelty results, we refer the reader to Nibbler [8]). Instead, in our scenario, we assume a given (blackbox) framework (e.g., Nibbler) that allows us to identify the unreachable code regions

in a library for a given program binary (i.e., the dissected container service), which we use as input for our abstract interpretation based system call number analysis in the next step. Further, we do not intend to perform this fine-grained code reachability analysis for all libraries. Instead, we request this code reachability information only for one, central library, namely `libc`. This decision simplifies the information gathering (e.g., closed-source libraries impede even binary-level analysis as they usually avoid shipping unstripped binaries). At the same time, by focusing on the central OS interface (i.e., `libc`) that is employed by most C applications, we substantially reduce the number of all system calls. Note that disregarding to our decision for this work, code reachability information can be gathered from all involved libraries; we leave this for future work.

5.3 Identifying System Calls

Once we have identified the involved binaries (including programs and their libraries, and optionally the libraries' unreachable code regions), we apply JESSE to (*i*) unfold their control-flows leading to a set of `syscall` instructions and (*ii*) uncover the associated system calls (§ 4). Considering that most C programs employ the functionality of the standard C library (`libc`), the net effect of our system call number analysis results in a highly coarse-grained collection of system calls. Generally, `libc` implements an essential part of the OS interface that simplifies the bootstrapping of programs and assists them with a solid foundation of general-purpose functionality including a high number of system call wrappers that simplify the communication with the OS kernel. For instance, the `libc` v2.24 holds 502 `syscall` instructions that can call up to 264 (fixed) system calls, disregarding the generic `syscall()` function that allows to call any of the available system calls. Consequently, the naive accumulation of all system calls in the target binaries results in a heavy overapproximation of the system calls that are truly used and hence calls for an optimization that avoids unreachable code. Yet, by considering the unreachable code information gathered in § 5.2, JESSE considers only viable paths in `libc` in its system call number analysis.

6 Evaluation

6.1 Precision Evaluation

The quality and hence strength of the generated seccomp policies highly depends on (*i*) the precision and coverage of the abstract interpretation (§ 4) and (*ii*) the accuracy of the optimization to disregard unused code sequences (§ 5.2).

6.1.1 Abstract Interpretation

We demonstrate the effectiveness of JESSE by applying its abstract interpretation based system call number identification to the standard C library (`libc`). The `libc` v2.24 holds 502 `syscall` instructions that refer to 264 unique system calls. By applying the abstract interpretation of JESSE, we were able to link the exact system call number to 484 (96.4%) `syscall` instructions. JESSE was not able to determine the system call numbers of the remaining 18 `syscalls` in the `libc`. To generalize the precision of JESSE's abstract interpretation we have applied the system call number analysis to other 13 common libraries. We have selected these by accumulating all libraries that are used by the binaries in the `/bin` directory of a vanilla Debian Buster (base) image. These formed additional 292 `syscall` instructions, for which JESSE was not able identify 21 system call numbers: 16 in `libpthread` and 5 in `librt`. Therefore, in 95% of all cases, our analysis determined the right system calls (which we have verified by manually inspecting the binaries). For the the remaining 5%, JESSE informed us about the non-identified system calls, such that we were able to identify the exact cause by manually analyzing the binaries.

Exact Linux Distribution/Version?

The reason for the incomplete mapping is fourfold. First, there exist functions (e.g., the `syscall()` function in `libc`) that allow to select arbitrary system calls by specifying the system call number in one of the function parameters. Since the exact system call number depends on the calling function, an analysis of the `syscall()` function alone, will not be able to determine the provided value. We were able to address this issue by generalizing our definition of the abstract interpretation: in such cases, instead of identifying the value in the `rax` register immediately before the `syscall` instruction, we cause the abstract interpretation to identify the value in the `rdi` register (i.e., the first function parameter) immediately before calling the `syscall()` function. Note that the exact register depends on the called function. The second class refers to the incompleteness of `angr`. For instance, `angr` cannot symbolically execute certain AVX instructions (such as `vbroadcast`) and hence impedes further analysis. This engineering issue can be solved by extending the capabilities of `angr`. The third class builds upon our simplified structure of the CFG. Our current implementation ignores branch conditions as this strategy significantly simplifies the analysis. Yet, since branch conditions can influence the value in `rax`, ignoring this information makes it impossible to analyze the system call number of some `syscall` instructions. We leave the implementation of an abstract interpretation that considers the conditions of branches for future work. Finally, a very small class of functions reads the exact system call number from memory. Although, this strategy breaks static analysis, contrary to dynamic analysis that cannot be sure whether the dynamic tests have covered all possible paths of the program [56], in this and all upper cases, JESSE is capable of

narrowing down the function's exact location and let the analyst incorporate her expert knowledge to nevertheless satisfy the policy.

syscalls in memory: we cannot analyze that (have a look at the LWN article!)

6.1.2 Avoiding Unreachable Code

We evaluated the accuracy of the discussed optimization that allows JESSE to avoid unreachable code in libraries (§ 5.2). Specifically, we have applied JESSE to five popular Docker containers from Docker Hub [3]. To also compare our results with related work [56] in the next section, we have selected the same set of containers as used by Wan et al. (Table 2). The applied optimization allows us to narrow down the set of all identified `syscall` instructions used in container binaries to a subset that is in fact reached by the analyzed program. This way, we establish the foundation for accurate and effective seccomp policies. To accommodate closed-source libraries and to consider that most C program libraries (excluding `libc`) will not dramatically increase the number system calls, we apply the optimization only to `libc`. The system calls of other dissected container binaries are accumulated through the general system call number analysis without the additional optimization (§ 4). In other words, we trade off accuracy for compatibility, and still achieve more accurate results than Docker's default seccomp policy. Note that compared to library code debloating strategies [8, 48], JESSE does neither need to modify the libraries' layout in the target process' address space nor individual code pages to remove the remaining, yet unneeded, code at load-time, and hence does not suffer from increased load-time performance, or the additional memory overhead, respectively.

Think about removing this part. Ipmrecise. Yet, if leaving this, add the % of the averaged syscall increase by other libraries

Table 2 shows for each analyzed Docker container image the percentage of system calls that we were able to restrict by means of the generated seccomp policy. On average, the generated policies restricted 55.8% of the available system calls and thus significantly improved the granularity of Docker's default seccomp policy, which prohibits, depending on the container configuration, between 10.6% and 20.4% of the system calls (§ 2.1). To rule out false negatives (i.e., falsely restricted system calls) we applied the generated seccomp policies to the Docker containers in Table 2. Once we have fortified the containers, we run the following benchmarks that achieve a good coverage of the containers. We have selected the same set of benchmarks that are also used by Wan et al. to dynamically mine seccomp policies for Docker containers [56]. Note that instead of collecting performance results (as we did not modify the applications themselves), we focus on stressing the containers to achieve a high coverage of the tested container applications.

Specifically, we employ `httperf` [45] to test the genuine execution of the `nginx` and `httpd` web servers. Similarly to Wan et al. (who leverage these test suites to dynamically mine system calls), we create 10 times 100 connections, each with an increasing connection rate from 5 to 50 requests per second (req/sec), with steps of $5req/sec$ and 2 seconds sleep time whenever the connection rate is increased. Further, we test the *mysql* containers by applying the benchmark `sysbench`. In more detail, we use `OLTP` test with 8 parallel threads and the maximum number of requests capped to 800. For the `postgress` containers we applied the `tpc-b-like` and the `simple-update` test of the `pgbench` test suite for 60 seconds each. Finally, we applied *redis-benchmark* to the `reddis` containers. In all cases, the containers ran through and were not interrupted by a falsely prohibited system call.

Add refs to the remaining benchmarks

6.2 Benefits and Pitfalls of Dynamic Analysis

In comparison to JESSE that prohibits around 55.8% of the available system calls, the dynamically gathered results of Wan et al. [56] demonstrate a more fine-grained restriction policy of around 75.2% on average (Table 2). Generally, dynamic approaches generate more rigorous policies. Yet, they highly depend on the achieved program coverage by the employed test suites, and hence merely underapproximate the set of system calls that is vital for the analyzed program. In other words, in case a benchmark misses an execution path (that leads to a system call) in the target application, a potential execution of the hereby introduced false negative (a falsely unauthorized system call in the seccomp policy) will eventually falsely crash the process. In the following, we present pitfalls of the incompleteness of dynamic analysis based related work [56].[4]

By closely analyzing the Docker container versions and the respective seccomp policies that were dynamically generated by Wan et al., we discovered that a set of essential system calls was overlooked by their analysis. For instance, the dynamically generated seccomp policy for the `redis` v3.2.3 container missed the system calls `rename()` and `fsync()`; both are used for the `redis`' background saving functionality. Another example is that the generated policy for the `nginx` v1.11.1 container missed the system calls `rename()` and `chmod()`; the system calls that define the web server's capabilities with regard to UNIX-domain sockets. These findings are incomplete, yet, they question the credibility of the dynamically gathered results and, at the same time, underline the need for a more complete, static analysis.

Further, dynamic analysis builds upon that the system call gathering framework uses a correct representation of occured system calls. By analyzing the generated policies of Wan et al. we have located another set of system calls that was identified and authorized by the dynamically generated policy, yet, prohibited in the statically generated policy of JESSE. In theory, the static analysis produces an overapproximated set of the authorized system calls and hence raised our attention. It turned

[4] We do not intend to harm our colleague's work but rather use their results to make a point against dynamic analysis with respect to generating system call policies.

Table 1: TBD!

	Binary	Httpd	MySQL	Nginx	Postgress	Redis	# of `syscall` Instructions	# of Assigned System Calls
Libraries	`libc.so`	✓	✓	✓	✓	✓	502	498 (99.20%)
	`libpthread-2.24.so`	✓	✓	✓	✓	✓	169	153 (90.53%)
	`ld-2.24.so`	✓	✓	✓	✓	✓	43	43 (100.00%)
	`librt-2.24.so`	✓	✓		✓	✓	30	25 (83.33%)
	`libnuma.so.1.0.0`		✓				9	9 (100.00%)
	`libstdc++.so.6.0.22`		✓				6	6 (100.00%)
	`libsystemd.so.0.17.0`				✓		6	6 (100.00%)
	`libaio.so.1.0.1`		✓				5	5 (100.00%)
	`libcrypt-2.24.so`	✓		✓			3	2 (66.67%)
	`libuuid.so.1.3.0`	✓					2	2 (100.00%)
	`libgcrypt.so.20.1.6`				✓		1	1 (100.00%)
	`libk5crypto.so.3.1`				✓		1	1 (100.00%)
	`libselinux.so.1`		✓		✓		1	1 (100.00%)
Programs	`httpd`	✓					1	1 (100.00%)
	`mysqld`		✓				17	17 (100.00%)
	`nginx`			✓			7	7 (100.00%)

Table 2: Restricted system calls through seccomp policies tailored for five popular Docker container images. The seccomp policies were generated (statically) by JESSE and (dynamically) through *mining* by Wan et al. [56].

Container	Mining [56]	JESSE
`httpd` v2.4.23	78.9%	57.3%
`httpd` v2.4.37	-	56.7%
`mysql` v5.7.13	69.7%	49.5%
`mysql` v8.0.14	-	45.2%
`nginx` v1.11.1	77.8%	65.1%
`nginx` v1.15.8	-	63.6%
`postgres` v9.5.4	71.4%	50.7%
`postgres` v11.1	-	43.8%
`redis` v3.2.3	78.6%	65.4%
`redis` v5.0.2	-	60.8%

Ref!

out, that the dynamic analysis blindly trusted the output of the tool `sysdig` that was used to accumulate the set of called system calls. In this way, the analysis added a set of falsely named system calls (that do not exist in the Linux kernel) into the seccomp policy. Table 3 summarizes the set of system calls that were explicitly authorized by the seccomp policies of Wan et al., but do not exist in the Linux kernel. By examining the source code of `sysdig`, we find out that the tool uses wrong names for certain system calls. For instance, `sysdig` returns `sigreturn()` upon being notified of the execution of the `rt_sigreturn()` system call. Similarly, `sysdig` falsely uses the output of `eventfd()` upon the occurrence of both `eventfd()` and `eventfd2()`, even though they represent two different system calls.

Table 3

Container	Misinterpreted System Calls
`httpd` v2.4.23	`procexit()`, `signaldeliver()`, `sigreturn()`
`mysql` v5.7.13	`set_tls()`, `getresgid32()`, `getresuid32()`, `pread()`, `pwrite()`, `procexit()`, `signaldeliver()`, `sigreturn()`
`nginx` v1.11.1	`pread()`, `pwrite()`, `procexit()`, `signaldeliver()`
`postgres` v9.5.4	`procexit()`, `signaldeliver()`, `sigreturn()`
`redis` v3.2.3	`procexit()`, `signaldeliver()`, `sigreturn()`

6.3 Security Evaluation

We evaluated the added security of the generated `seccomp` policy by using real-world exploits against (*i*) MySQL server v5.7.14 running inside a Docker container and (*ii*) the Linux kernel v4.13. The combined exploits transformed a vulnerable system call into an effective *write* primitive that allowed the subverted container to directly modify the kernel memory for malicious purposes. In this context, we leveraged CVE-2016-6662 (i.e., SQL injection vulnerability of the MySQL server) to gain arbitrary code execution capabilities inside the container. Generally, once the attacker gains control over the container, we assume she will attack the Linux kernel to attempt to escape the sandboxed environment, as this will grant her the capability to control other containers and even the kernel itself. To evaluate this scenario, we have used the exploit for CVE-2017-5123 that was introduced into the Linux kernel v4.13 to similate an attacker that attempts to escape the container. In this context, first, we have reproduced the exploit of Chris Salls, the discoverer of CVE-2017-5123, to bypass KASLR [52]. As our container was initially

not bound by `seccomp` (and hence did not block unauthorized system call invocations), we have abused the vulnerable `waitid()` system call to establish an *arbitrary write* primitive into the kernel memory; by abusing the fact that the vulnerable `waitid()` system call handler missed the necessary `access_ok()` checks (that prevent the user space argument `siginfo_t *infop` from pointing to unauthorized memory), we were able to write-access arbitrary kernel addresses. Since the `unsafe_put_user()` (and other) kernel helper does not crash the kernel when accessing invalid memory regions, we were able to fingerprint the kernel's address space to identify its exact mapping [52], despite KASLR.

In the next step, similar to the original exploit, the gained *write* primitive lent us the capability to perform the *ret2dir* attack [34]. To achieve this, we identify a page within the kernel's *physmap* (i.e., a contiguous memory region that directly maps a part of, or even all, physically available memory into the kernel space) that is aliased with a user space page that is controlled by us. Then, by injecting a *fake* data structure (e.g., `struct file`) with function pointers that we can use to initiate the execution a ROP gadgets inside the kernel, e.g., to disable the systems Supervisor Mode Execution Prevention (SMEP) and Supervisor Mode Access Prevention (SMAP) protection and to grant sufficient privileges of the executing container process. Once we receive sufficient privileges, we become able, e.g., to request `CAP_SYS_MODULE` capabilities which allow us to load kernel modules and thus fully control the entire system.

Even though the attacker received strong capabilities that allowed writing to arbitrary kernel memory, our seccomp policy was able to eliminate the vulnerability. Note that the Docker container did not require the vulnerable system call for its genuine execution. As such, after applying the generated `seccomp` policy to the container, `seccomp` successfully suppressed the unauthorized invocation of the vulnerable system call and immediately terminated the container.

7 Discussion

Add the following:
- syscall() system call
- self-modifying code etc. loading of new code
- assembly vs C-code vs calling conventions issues
- Function padding with zero vs nop instructions
- Discuss the benefit/disadvantages our your paper and how this can be increased through additional library debloating mechanisms
- Discuss that the static analysis could have been performed, e.g., via LLVM (see SLAKE paper (CCS'19))
- Discuss the Nibbler paper (ACSAC'19)

Identifying semantic properties of programs are generally *undecidable*. Consequently, the employed abstract interpretation of JESSE only approximates the results and, in some cases, encounters its limits. These limits stem from both the dynamic nature of program behavior and the implementation deficits of our prototype, which we outline in the following.

Conditional branches: The implementation of the abstract interpretation of JESSE does not differentiate between the different branch targets to avoid having to determine the conditions that are only determined at run time. Consequently, if the `rax` content depended on a condition, our prototype would not be able to determine its value. Interestingly, we have not encountered any conditional assignments of the `rax` register that preceeded a `syscall` instruction.

Generic system call wrappers: Further, the abstract interpretation is currently limited to function-level analysis. While this is sufficient for most system call invocations, there exist functions that act as generic wrappers (e.g., the `syscall()` function in `libc`) for arbitrary system calls, as they allow to specify the system call number in one of the function parameters. Consequently, since the exact system call number depends on the calling function, analyzing the `syscall()` function alone, is insufficient to determine the provided system call number. We counter this issue by, first, letting JESSE inform us about the missing system call number identification, and second, instead of identifying the value in the `rax` register just before the `syscall` instruction, adjust the abstract interpretation to identify (in the case of the `syscall()` function) the value of the `rdi` register just before calling the `syscall()` function. As this solution targets the `syscall()` function, a generic solution would need to consider the register state that is transferred between function calls.

System call numbers in memory: The introduced abstract interpretation that associates system call numbers to identified `syscall` instructions (§ 4) hinges on the premise that system call numbers are passed as constants via the `rax` register, and that they are neither arithmetically computed nor read from memory. Instead, our abstract interpretation assumes that the system call numbers are assigned and propagated as constants, and never read from memory. The system call number analysis becomes ineffective, as soon as this assumption breaks. Fortunately, most system call numbers adhere to our assumption. Yet, there exist exceptions that prove our assumption. Specifically, certain functions receive the system call numbers through memory. For instance, the signal handler `sighandler_setxid()` in `libpthread` determines whether to change the *user* or *group id* by reading the system call number from a data structure in memory. In such cases, JESSE informs and requests the analyst's input to make an informed decision.

Virtual Dynamic Shared Object: On Linux, the virtual Dynamic Shared Object (vDSO) is an architecture dependent, shared library that is mapped into the address space of every user-space process [13]. Generally, the vDSO increases the performance of selected system calls by emulating their functionality in user space; virtual system calls do not have to switch into the kernel space to perform their task. While the vDSO is practical, its architecture dependencies can introduce difficulties to dynamically generated seccomp policies [20]. For instance, the time keeping system calls highly depend

on the program's choice of the hardware's clock source and the system's configuration [20]. In case, the vDSO (or rather its virtual system call `gettimeofday()`) should not be in the position to meet these demands, it will fall back to calling the system call in kernel space. (Note that before Linux kernel v5.3, the fall back called the 32-bit `clock_gettime[64]()` system call [20].) This introduces an issue for both static and dynamic approaches for generating seccomp policies. Static approaches cannot analyze the vDSO (and hence miss system calls that are invoked in the respective fall back paths of virtual system calls). That is, even though the static analysis is able to extract the called virtual system call (e.g., `gettimeofday()`), it will not identify the system call number that will be potentially called on another system (e.g., `clock_gettime[64]()`). Similarly, dynamic approaches can simply miss the system call numbers used in the fall back path as the system could be differently configured on a different machine. To address this issue, we have completely analyzed the vDSO and by default authorize the execution of all system calls that can be called by the vDSO.

Attack vector reduction: Even though seccomp effectively contributes to reducing the system's attack vectors, it will not be able to completely diminish future attacks. As such, JESSE (as the last line of defense) is most effective in combination with further security hardening measures. For instance, when combined with accurate library code debloating mechanisms [8, 48], JESSE would establish a highly restricted environment, deprived of unnecessary and potentially threatening gadgets and system calls.

8 Related Work

Mention Capsicum: `https://www.usenix.org/legacy/event/sec10/tech/full_papers/Watson.pdf`

References: (1) Do we need something like Flowdroid [10] and Droidscope and Droid*? (i) Look at CHEX, and COPES (and Flowdroid) (all referenced in the sandbox mining paper) as they use static analysis techniques to establish rules for sandboxes.
(2) mention container security [43] (i) Use [43] to find more related work!
(3) Reference "Improving host security with system call policies"
(4) Reference: [8] (5) Reference:

Goldberg et al. [28] inroduce Janus, one of the first sandboxed environments for untrusted applications. Janus leverages the Solaris' tracing capabilities to interpose and confine system calls. With Boxify [11], Backes et al. establish a user-space only framework that leverages the Android's process isolation features to execute apps in the context of a trusted process with restricted permissions. Boxify further limits the capabilities of untrusted applications by intercepting calls to the Android API and system calls.

Gordon et al. [29] propose DroidSafe which models the Android Application Programming Interface (API) and uses a static analysis technique to find information leaks in Android application. Similarily, Arzt et al. [10] develop FlowDroid, a static taint analysis that does the same with a higher precision. While these approaches implement the necessary means for isolating and restraining individual applications, they lack the ability to determine relevant permissions and system calls that should be white-listed. As such, Jamrozik et al. [33] introduce a novel technique, coined sandbox mining, to determine the resources that are required by applications at run-time. In this context, Jamrozik et al. present BOXMATE, a framework that executes tests against target applications an Android to extract the set of resources (including system calls) required during the tests. This allows BOXMATE to limit the applications to the gathered resources. Instead of relying on general benchmark-based tests, DroidBot [41] generates customized test-cases that are leveraged by the framework of Le et a. [40], such that the sandbox also consider parameters of calls to the Android API. In a similar way, Wan et al. [56] mine sandboxes for Linux containers. Specifically, the authors leverage dynamic analysis techniques to determine which system calls are called by programs inside of the container. After that, they leverage seccomp to blacklist all system calls not found during the analysis phase. Although they do not find any false positives in their evaluation, their analysis misses some system calls XXXXXX.

Another approach is to debloat containers. Rastogi et al. [50, 51] develop Cimplifier, a tool that partitions an input container into multiple, minimal output containers. When the input container only runs one main program, this reduces to the removal of all unused programs in the container's filesystem. The difference between the two papers is the analysis method. In the first paper, Rastogi et al. use dynamic analysis techniques to find out which programs run in the container and in the second paper, they switch to static analysis procedures.

In addition to debloating containers, it is also possible to debloat programs. In this case the goal is to remove dead code that is introduced through loaded libraries. Quach et al. [48] implement a static analysis technique that finds out which instructions in the libraries are to dead code. After that, they prevent these parts of the binaries from loading. However, this approach requires the recompiliation of the analyzed binaries with `clang`, because they run their analysis on the LLVM IR representation of the programs and libraries. Agadakos et al. [8] extend the work of Quach et al. by developing Nibbler, a library debloating tool that is able to debloat libraries on a binary level, i.e., a recompilation of the analyzed binaries is not necessary. But, they require all ELF symbols tables and thus, unstripped binaries.

Although, this reduces the code size significantly, it cannot remove code that is not executed due to disabled features in the configuration, because such code is only dead in the current configuration but not in general. As such, Koo et al. [38] develop a configuration-driven software debloating technique that overcomes this limitation by analyzing which included libraries are only used to provide one specific feature and removing these libraries when the feature is disabled.

The problem with debloating approaches is that they cannot

defend against injected, malicious code. Once an adversary is able to run her own programs inside of the container, the defences are useless, because she is not required to reuse existing programs anymore.

- shradder paper

9 Conclusion

References

[1] BSD Jails. https://www.freebsd.org/doc/handbook/jails.html.

[2] Docker. [Online; accessed 12-September-2018].

[3] docker hub. [Online; accessed 19-May-2019].

[4] Infrastructure for Container Projects – LXC. https://linuxcontainers.org/lxc/introduction/.

[5] Oracle Solaris Zones. https://docs.oracle.com/cd/E18440_01/doc.111/e18415/chapter_zones.htm.

[6] Source code of docker. [Online; accessed 18-October-2019].

[7] Source code of the linux kernel. [Online; accessed 18-October-2019].

[8] Ioannis Agadakos, Di Jin, David Williams-King, Vasileios P. Kemerlis, and Georgios Portokalidis. Nibbler: Debloating Binary Shared Libraries. In *Annual Computer Security Applications Conference (ACSAC)*, 2019.

[9] Angr. Angr documentation. https://docs.angr.io, 2020.

[10] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. *Acm Sigplan Notices*, 49(6):259–269, 2014.

[11] Michael Backes, Sven Bugiel, Christian Hammer, Oliver Schranz, and Philipp von Styp-Rekowsky. Boxify: Full-fledged App Sandboxing for Stock Android. In *USENIX Security Symposium*, 2015.

[12] Jean-Tiare Le Bigot. How I shrunk a Docker image by 98.8% – featuring fanotify. https://blog.yadutaf.fr/2015/04/25/how-i-shrunk-a-docker-image-by-98-8-featuring-fanotify/, 2015. [Online; accessed 12-January-2020].

[13] Daniel Pierre Bovet. Implementing Virtual System Calls, 2014.

[14] Neil Brown. Control groups series by Neil Brown. https://lwn.net/Articles/604609/, July 2014.

[15] Quan Chen, Ahmed M. Azab, Guruprasad Ganesh, and Peng Ning. PrivWatcher: Non-bypassable Monitoring and Protection of Process Credentials from Memory Corruption Attacks. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2017.

[16] Corbet. A Bid to Resurrect Linux Capabilities. https://lwn.net/Articles/199004/, September 2006.

[17] Jonathan Corbet. Seccomp and Sandboxing, 2009. [Online; accessed 17-October-2019].

[18] Jonathan Corbet. Supervisor Mode Access Prevention. https://lwn.net/Articles/517475/, October 2012.

[19] Jonathan Corbet. Yet another new approach to seccomp, 2012. [Online; accessed 17-October-2019].

[20] Jonathan Corbet. vDSO, 32-bit Time, and Seccomp, 2019.

[21] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM, 1977.

[22] National Vulnerability Database. CVE-2016-8655. https://nvd.nist.gov/vuln/detail/CVE-2016-8655, 2016.

[23] National Vulnerability Database. CVE-2017-7308. https://nvd.nist.gov/vuln/detail/CVE-2017-7308, 2017.

[24] DockerSlim. Minify and Secure Docker containers (free and open source!). https://github.com/docker-slim/docker-slim. [Online; accessed 12-January-2020].

[25] Jake Edge. Kernel Address Space Layout Randomization. https://lwn.net/Articles/569635/, October 2013.

[26] Jake Edge. Inheriting Capabilities. https://lwn.net/Articles/632520/, Februar 2015.

[27] Alexis Engelke. Lightweight disassembler. https://github.com/aengelke/lwdpy, 2017. [Online; accessed 05-August-2018].

[28] Ian Goldberg, David Wagner, Randi Thomas, Eric A Brewer, et al. A Secure Environment for Untrusted Helper Applications: Confining the Wily Hacker. In *USENIX Security Symposium*, 1996.

[29] Michael I Gordon, Deokhwan Kim, Jeff H Perkins, Limei Gilham, Nguyen Nguyen, and Martin C Rinard. Information flow analysis of android applications in droidsafe. In *ISOC Network and Distributed System Security Symposium (NDSS)*, 2015.

[30] Red Hat. CVE-2017-5123. `https://access.redhat.com/security/cve/cve-2017-5123`, 2017.

[31] Hubicka, Jan and Jaeger, Andreas and Matz, Michael and Mitchell, Mark. *System V Application Binary Interface AMD64 Architecture Processor Supplement*. 2013.

[32] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual, Combined Volumes: 1, 2A, 2B, 2C, 3A, 3B, 3C, 3D, and 4*. 2019.

[33] Konrad Jamrozik, Philipp von Styp-Rekowsky, and Andreas Zeller. Mining sandboxes. In *International Conference on Software Engineering (ICSE)*, 2016.

[34] Vasileios P. Kemerlis, Michalis Polychronakis, and Angelos D. Keromytis. ret2dir: Rethinking Kernel Isolation. In *USENIX Security Symposium*, 2014.

[35] Michael Kerrisk. Namespaces in Operation, Part 1: Namespaces Overview. `https://lwn.net/Articles/531114/`, January 2013.

[36] Koen Koning, Xi Chen, Herbert Bos, Cristiano Giuffrida, and Elias Athanasopoulos. No Need to Hide: Protecting Safe Regions on Commodity Hardware. In *ACM European Conference on Computer Systems (EuroSys)*, 2017.

[37] Hyungjoon Koo, Seyedhamed Ghavamnia, and Michalis Polychronakis. Configuration-Driven Software Debloating. In *European Workshop on System Security (EuroSec)*, 2019.

[38] Hyungjoon Koo, Seyedhamed Ghavamnia, and Michalis Polychronakis. Configuration-driven software debloating. In *Proceedings of the 12th European Workshop on Systems Security*, page 9. ACM, 2019.

[39] Aneesh Kumar. Auditing your Docker instance down to a 'Bare Necessity' footprint. `https://medium.com/codefish/working-with-dockers-64c8bc4b5f92#.f3i10qkyt`, 2015. [Online; accessed 12-January-2020].

[40] Tien-Duy B Le, Lingfeng Bao, David Lo, Debin Gao, and Li Li. Towards Mining Comprehensive Android Sandboxes. In *International Conference on Engineering of Complex Computer Systems (ICECCS)*, 2018.

[41] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. DroidBot: A Lightweight UI-guided Test Input Generator for Android. In *International Conference on Software Engineering Companion (ICSE-C)*, 2017.

[42] Siarhei Liakh. NX Protection for Kernel Data. `https://lwn.net/Articles/342266/`, July 2009.

[43] Xin Lin, Lingguang Lei, Yuewu Wang, Jiwu Jing, Kun Sun, and Quan Zhou. A Measurement Study on Linux Container Security: Attacks and Countermeasures. In *Annual Computer Security Applications Conference (ACSAC)*, 2018.

[44] Yutao Liu, Tianyu Zhou, Kexin Chen, Haibo Chen, and Yubin Xia. Thwarting Memory Disclosure with Efficient Hypervisor-Enforced Intra-Domain Isolation. In *ACM Conference on Computer and Communications Security (CCS)*, 2015.

[45] David Mosberger and Tai Jin. httperf — A Tool for Measuring Web Server Performance. In *ACM SIGMETRICS Performance Evaluation Review*, 1998.

[46] Sergej Proskurin, Marius Momeu, Seyedhamed Ghavamnia, Vasileios P. Kemerlis, and Michalis Polychronakis. xMP: Selective Memory Protection for Kernel and User Space. In *IEEE Symposium on Security and Privacy (S&P)*, 2020.

[47] Niels Provos. Improving Host Security with System Call Policies. In *USENIX Security Symposium*, 2003.

[48] Anh Quach, Aravind Prakash, and Lok Yan. Debloating Software through Piece-wise Compilation and Loading. In *USENIX Security Symposium*, 2018.

[49] Radare2. Libre and Portable Reverse Engineering Framework. `https://rada.re/n/`, 2020.

[50] Vaibhav Rastogi, Drew Davidson, Lorenzo De Carli, Somesh Jha, and Patrick McDaniel. Cimplifier: automatically debloating containers. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 476–486. ACM, 2017.

[51] Vaibhav Rastogi, Chaitra Niddodi, Sibin Mohan, and Somesh Jha. New directions for container debloating. In *Proceedings of the 2017 Workshop on Forming an Ecosystem Around Software Transformation*, pages 51–56. ACM, 2017.

[52] Chris Salls. Exploiting CVE-2017-5123 with full protections. SMEP, SMAP, and the Chrome Sandbox! https://salls.github.io/Linux-Kernel-CVE-2017-5123, 2017.

[53] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Firmalice — Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware. In *ISOC Network and Distributed System Security Symposium (NDSS)*, 2015.

[54] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. 2016.

[55] Nick Stephens, John Grosen, Christopher Salls, Audrey Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *ISOC Network and Distributed System Security Symposium (NDSS)*, 2016.

[56] Zhiyuan Wan, David Lo, Xin Xia, Liang Cai, and Shanping Li. Mining sandboxes for Linux Containers. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2017.

[57] Fenghua Yu. Enable/Disable Supervisor Mode Execution Protection. https://goo.gl/utKHno, May 2011.

[58] Z3. Z3 documentation. https://github.com/Z3Prover/z3/wiki/Documentation, 2020.

[59] Matteo Zanioli and Agostino Cortesi. Information leakage analysis by abstract interpretation. In *International Conference on Current Trends in Theory and Practice of Computer Science*, pages 545–557. Springer, 2011.