Free Willy: Prune System Calls to Enhance Software Security

Charlie Groh
Technical University of Munich
Germany

Sergej Proskurin
Technical University of Munich
Germany

Apostolis Zarras
University of Piraeus
Foundation for Research and Technology – Hellas
Greece

ABSTRACT

Many privilege escalation exploits on Linux abuse vulnerable system calls to threaten the system's security. Therefore, various *static* and *dynamic analysis* based seccomp policy generation frameworks emerged. Yet, they either focus on a subset of the available binaries or are constrained by the inherent properties of dynamic, testing-based analysis, which are prone to false negatives. In this paper, we present Jesse, a static-analysis-based framework for generating seccomp policies for ELF binaries. We design and implement an abstract-interpretation-based constant propagation that helps the analyst identify vital system calls for arbitrary, non-obfuscated binaries. Using the extracted results, Jesse allows producing effective seccomp policies, reducing the system's attack vector. To assess Jesse's effectiveness and accuracy, we have applied our system to over 1,000 ELF binaries for Debian 10, and show that—contrary to existing solutions—Jesse produces accurate and safely approximated results, *without* relying on any properties of the target binaries. In addition, we conduct a case study in which we combine Jesse's constant propagation strategy with container debloating techniques to produce seccomp policies that restrict up to five times more system calls than the Docker's default seccomp policy on average.

ACM Reference Format:
Charlie Groh, Sergej Proskurin, and Apostolis Zarras. 2023. Free Willy: Prune System Calls to Enhance Software Security. In *The 38th ACM/SIGAPP Symposium on Applied Computing (SAC '23), March 27-March 31, 2023, Tallinn, Estonia.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3555776.3577593

1 INTRODUCTION

The *principle of least privilege* mandates each entity to access only the necessary resources for its execution. Yet, modern Operating System (OS) architectures offer applications a uniform *system call interface* granting access to a high number of system calls, regardless of how many system calls the application requires. A vulnerability in one of those system calls has the potential to impair the underlying kernel, despite modern isolation and security mechanisms.

One way to mitigate this threat is to remove unnecessary system calls, available to (containerized) applications. For instance, recent advances in *library debloating* [1, 18, 22–24, 29] remove code regions in the process' address space that are irrelevant to the program's execution. While library debloating focuses on reducing Return-Oriented Programming (ROP) (or similar code-reuse) gadgets, it does not eliminate the threat; remaining gadgets can allow the attacker to still mount an attack that abuses a vulnerable system call handler. As a last line of defense, the Linux seccomp (Secure Computing) facility provides the necessary means to establish system call filters to restrain applications and containers. Yet, to this point, there is no easy way to automatically tailor seccomp policies.

Researchers propose *static* or *dynamic analysis* based techniques to identify the system calls required by an application (or container). Dynamic analysis based approaches [21, 27] employ automatic test generation that triggers different program behaviors, which 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 system call has not been identified by the dynamic analysis, it does not mean that it will not be called in production. Thus, *false negatives* can interrupt and terminate the execution of sandboxed environments. In contrast, static analysis [1, 7, 22, 29] either focuses on library or container debloating, or strongly rely on additional properties of the target binaries. These properties either comprise the availability of the source code, position-independent code, or other binary metadata, which is not always available.

Identifying a binary's authorized set of system calls can be reduced to *constant propagation*. Previous research has shown excellent results when applying constant folding and constant propagation to reconstruct Control-Flow Graphs (CFGs) from binaries [5, 10, 17]. Inspired by these works, we re-purpose the concepts behind constant propagation to automatically derive legitimate system call numbers from binaries to create tailored seccomp policies.

In this paper, we present Jesse, a framework that leverages static analysis to identify system calls for *arbitrary* non-obfuscated ELF binaries.[1] Jesse aims to complement existing tools [7, 11, 29] to bridge the gap to binaries that do not support position-independent code (PIC) and lack the necessary binary metadata. Given the provided set of system calls, it can be processed into strong seccomp policies that establish safe (sandboxed) environments. We implement a *constant propagation* technique based on *abstract interpretation* to derive the contextual semantics required to (safely) approximate the set of system calls a container should be authorized to use [6]. To assess Jesse, we picked a representative set of 1,064 ELF binaries of the Debian `buster` *base* image, and *manually* verified the results. Jesse did not introduce any false negatives and demanded human support in only 20 cases (i.e., 2%). Further, we employed Jesse to generate seccomp policies for five of the most prominent Docker

[1] Jesse is available online: https://gitlab.com/jesse-tool/jesse

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SAC '23, March 27-March 31, 2023, Tallinn, Estonia
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-9517-5/23/03…$15.00
https://doi.org/10.1145/3555776.3577593

containers available. On average, we block 54.7% of all system calls and 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.

In summary, we make the following main contributions:

- We introduce Jesse, a static analysis framework for generating Linux seccomp policies for ELF binaries.
- We develop an abstract interpretation based constant propagation connecting system calls to `syscall` instructions.
- We apply Jesse to create seccomp policies for five prominent Docker containers to thwart container escape vulnerabilities.

2 THREAT MODEL

We assume an attacker who intends to escalate her privileges by abusing the Linux *system call interface*, the protection of which is the main target of this work. We expect the target Linux kernel to implement the Linux seccomp facility. We do not limit the adversarial capabilities to specific attacks, yet, we expect her to exploit memory corruption vulnerabilities to establish *read* and *write* primitives into the victim application's address space to hijack its control-flow, construct code gadgets to exploit kernel vulnerabilities through the system call interface, and finally take over the host system. We assume the application under attack is benign and that system employs state-of-the-art defenses against attacks in user and kernel space. We also assume that containerized applications are isolated from other applications via OS-level virtualization including Linux namespaces, control groups, and capabilities.

Assuming these defenses restrain the attacker inside the isolated environment, potential memory corruption vulnerabilities in system call handlers can allow establishing primitives that enable them to read/write to the kernel memory arbitrarily. Such primitives can (*i*) leak information to defeat Kernel Address Space Layout Randomization (KASLR), (*ii*) deactivate SMEP/SMAP, and hence (*iii*) inject and execute payload. To limit this attack vector, we concentrate on reducing the system call interface by creating seccomp policies tailored for benign user-space applications and containers.

3 SYSTEM CALL NUMBER ANALYSIS

Fig. 1 provides an overview on the system call number analysis, in which we put Jesse in perspective. The figure partitions the process of deriving seccomp profiles from applications into three phases. The first phase identifies all binaries that are relevant for the analysis. In other words, it is responsible for identifying the complete set of binaries (i.e., programs and shared libraries) on which the target program maintains a dependency. Since the program to be analyzed can be either a binary or a container image, resolving and extracting the dependent binaries can differ. For binaries, we can use the ELF header to identify their dependencies; for containers, we require container debloating techniques to dissect their images.

The second phase identifies all code regions associated with each exported function of the *standard C library*. This is done *only once* and allows Jesse to establish a map that associates each exported library function with a set of system calls. This way, we refer to the map to identify the system calls associated with an exported library function in the next phase (§ 4).

The third phase applies static analysis to derive the set of system calls a program requires and is the main focus of this section. We use this set to compile seccomp policies to filter unnecessary and potentially vulnerable system calls. The following presents the building blocks that allow us to harvest authorized system calls from ELF binaries. In essence, 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 `syscall` instructions. The second stage uses abstract interpretation to derive *system call numbers* passed as arguments to the identified `syscall` instructions.

3.1 Hypotheses for a Sound Analysis

Jesse's design and soundness bases on the following *key hypotheses* (**H1**–**H4**), derived by observing benign, unobfuscated, compiler-generated real-world binaries from official Debian mirrors (we have added the percentage of cases, in which a hypothesis applies in our evaluation (§ 6), to account for its applicability): **(H1)** most unobfuscated binaries comply with the x86-64 ABI calling conventions (100%) and can be linearly disassembled (99.34%) [2]; **(H2)** a `syscall` instruction invokes the same system call *in most cases* (98.98%); **(H3)** if the binary provides a system call number to a `syscall` instruction by crossing an indirect edge, the edge will not be part of the function holding the `syscall` instruction (100%); **(H4)** most binaries do not call system calls directly, instead they utilize wrappers in the standard C library.

(**H1**) presumes a sound and complete disassembly and adherence to the x86-64 ABI calling conventions. Contrary to (**H1**), malicious actors can obstruct disassemblers from correctly interpreting binaries or build binaries that use arbitrary calling conventions. The same applies to obfuscated binaries or code regions intertwined with data. Since binary analysis is undecidable [28], our analysis is limited in its ability to distinguish data from code. Yet, modern binaries strictly separate data from code regions by placing them into disjoint sections. In fact, prior research has shown that even though linear disassembly is an unsolved theoretical problem, most `gcc` and `clang` generated binaries can be linearly disassembled in practice [2]. Thus, we expect to recover a sound and complete disassembly from *unobfuscated* and *untampered* binaries.

(**H2**) assumes that distinct system calls differ with regard to behavior and preparations needed to call them; kernel developers prefer adding options to already existing system calls to support new features over introducing new system calls with similar semantics. Therefore, it is impractical to create dedicated functions that are merely responsible for setting up different arguments for a set of different system calls. If such wrapper functions are created nonetheless, they are often inlined, and thus become invisible in the target binary.

(**H3**) if we assume that every `syscall` instruction invokes the same system call handler (**H2**), the binary will hold constant system call numbers, hard-coded or resolved by the compiler. Most `syscall` instructions involve only a few preceding assembly instructions to acquire the associated system call numbers: our experimental findings confirm that (**H2**) holds in 98.98% of the analyzed system call invocations (§ 6). Out of the remaining 1.02%, we have observed that none of the system call numbers were provided through an

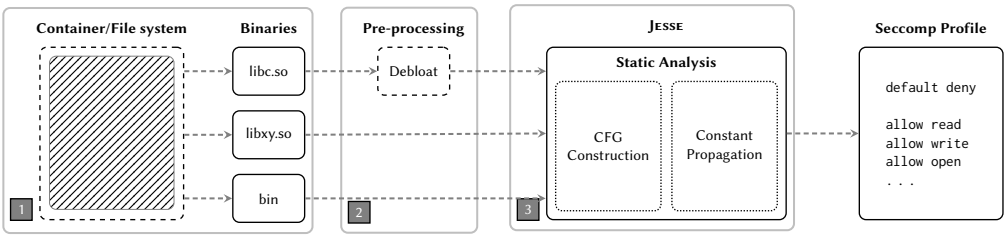

Figure 1: Three phases of the seccomp profile generation.

indirect edge *inside the same function* that holds the `syscall` instruction (**H3**)—all of these system call numbers depend on function arguments. Jesse does not consider indirect edges in functions to identify the system call numbers; if Jesse manages to associate a system call number with a `syscall` instruction, all potential indirect calls (inside the function) will invoke the same system call handler (**H2**). Indirect edges delivering system call numbers between functions *not* via function arguments would violate (**H1**). If such functions have been called indirectly, Jesse cannot be certain about the exact location of the caller that has initiated the function's invocation and it requests external expert knowledge (§ 6.2).

Finally, (**H4**) assumes that developers prefer system call wrappers in `libc` over directly invoking system calls. This is because system call wrappers in `libc` are more convenient and increase the portability of the program. Hence, if a binary (other than `libc`) invokes a system call directly, it either (*i*) has highly specific needs regarding the system call's arguments or (*ii*) the `libc` does not support the desired use case. Consequently, even though binaries (outside `libc`) directly invoke only a few system calls, the system calls they require are of *key-value* for creating seccomp profiles.

Every time Jesse cannot deduce a system call number, it will inform and direct the analyst to the exact location of the affected *syscall* instruction to ease further analysis.

3.2 Control-Flow Graph Construction

First, we transform the binary into a CFG, which can be processed by the abstract interpretation [6]. 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 results for the abstract interpretation, the CFG must fulfill the following three requirements; violation of these can cause Jesse to misinterpret the target binaries: ① *Correctness of* N; ② *Completeness of* N; and ③ *Correctness of* E. Note that we do not require the set of edges to be complete; we compensate the lack of this property of the CFG through (**H2**) and (**H3**).

① ensures the correct contents of basic blocks; it is met by most CFG construction tools. Yet, not every CFG construction tool assures the requirements ② and ③. Given that binary analysis is undecidable [28], CFG construction remains an open problem. Often, such tools prefer *completeness* over *soundness* of the CFG, which can falsify the assumptions of analysis frameworks, e.g., through heuristics. For instance, both *angr* and *radare* can miss basic blocks and/or introduce invalid edges in certain situations. Angr's static analysis based CFG generation tool can miss code (i.e., violate ②) on indirect branches (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 ③), because it relies on symbolic execution to overapproximate the set of possible jump targets. Similar restrictions apply to other CFG reconstruction frameworks [10, 17]. One of the reasons for heuristics is the tools' generality in application: heuristics allow to predict the location of code regions to a certain degree (at the risk of potentially violating ② and ③), even for obfuscated binaries or binaries that do not comply with the ABI. Unfortunately, heuristics cannot provide correct results with 100% certainty, and thus would reduce Jesse's precision. On the other hand, by avoiding heuristics (and by following the assumptions (**H1**–**H4**)), we can exclude incorrect results with regard to CFG construction with certainty. Generally, any CFG construction tool that follows the aforementioned requirements (①–③) is an eligible candidate for Jesse. We have decided to apply Lightweight Disassembler (LWD) [8] for our purpose. LWD does not apply any heuristics and its small code base allows us to easily extend and integrate it into Jesse.

We modify LWD to not regard `syscall`/`call` as control-flow changing instructions but as instructions that can alter registers according to Linux calling conventions of the x86-64 ABI (**H1**). Further, we cause LWD to split the basic blocks holding a `syscall` into two basic blocks that we connect through an artificial edge. This way, we ensure that the second part of the split basic block begins with the `syscall` instruction. This strategy simplifies identifying the system call numbers passed as a parameter to `syscall`.

Contrary to conventional CFG construction frameworks, LWD only considers edges of *direct* branches; it avoids heuristics for indirect branches whose targets reside in general-purpose registers (other than `rip`) or memory that cannot be statically determined. Hence, even though the final CFG comprises only correct edges (③), these do not have to be necessarily *complete*. Granted, without considering all code regions, LWD would miss `syscall` instructions. Yet, under **H1**, this is not the case in our analysis: we modify LWD such that it first linearly disassembles the binary, and then extends the disassembly to a CFG by adding appropriate edges for all direct jumps. This allows us to achieve ① and ②. If we further assume that most `syscall` instructions invoke only *one specific system call* (**H2**), a missing edge would represent an alternative invocation of an already identified system call. Rarely, a `syscall` instruction is used to call different system calls. In such cases, it is impossible for Jesse to determine the system call number statically. As such, Jesse informs the analyst about its exact location. This setting allows Jesse to either correctly determine the system call number for the respective `syscall` instruction or request expert knowledge.

3.3 System Call Number Identification

Abstract interpretation operates on a user-provided set of symbols (S). It thus functions on abstract symbols instead of sets of concrete program states. S is extended to a complete lattice $L = (S, \leq, \sqcup, \bot, \top)$. The lattice requires a *partial order* relation ($\leq$) and a *join* operation ($\sqcup$) to work 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. Finally, to form a complete lattice, L 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, in the set of symbols S. These elements simulate the empty set and the universe of the sets of concrete program states.

Defining the Lattice. The lattice $\dot{L}$ of our abstract interpretation is a quintuple $(\dot{S}, \dot{\leq}, \dot{\sqcup}, \dot{\bot}, \dot{\top})$. The system call numbers passed via `rax` to the system call dispatcher never result from arithmetic computations. Instead, `rax` is either ① initialized with an immutable immediate instruction operand, or ② assigned a value from a different register. While ① can be trivially determined by identifying the assignment of `rax`, pinpointing the exact value in ② must be accomplished through constant propagation techniques. Rarely, system call numbers are read from memory. Even though this property constraints static analysis, Jesse identifies the affected `syscall`'s location and requests the analyst's assistance. To address both cases, we define our lattice to keep track of the constant values that are propagated through the general-purpose registers.[2] Consequently, 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). 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:

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

Further, we define the *partial order* relation $\dot{\leq}$, with NX being the smallest element. Two vectors, x, y, fulfill $x \dot{\leq} y$ if they are identical or if elements of y differ only in their mapping to X (i.e., if the respective values are not known). We define formally for any lattice elements $a, b \in \dot{S}$:

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

The join operation $\dot{\sqcup}$ unifies two 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 other lattice element. The following defines $\dot{\sqcup}$:

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

We specify the bottom element of the lattice $\dot{L}$ ($\dot{\bot}$) to be the smallest (i.e., $\forall a \in \dot{S} : \dot{\bot} \dot{\leq} a$) and the top element ($\dot{\top}$) to be the largest (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 $\lambda r \in R.r \mapsto X$:

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

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 a program, or in our case, register states. To identify the exact system call number in `rax`, our interpretation function applies symbolic execution [25]. We leverage the symbolic execution framework *angr* 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. Utilizing a Satisfiability Modulo Theories (SMT) solver (e.g., Z3), the interpretation function checks whether the formula can be satisfied and if so, it determines the exact constant values assigned to the register state. Although angr uses Z3 to identify only one satisfying solution, instead of a complete set of all possible constant values, we ensure that the determined constant is indeed immutable via the process of elimination. Then, the interpretation function propagates the lattice elements to the main abstract interpretation algorithm, which incorporates the results by annotating the edges, to finally identify the system call a distinct `syscall` instruction refers to.

Alg. 1 describes the interpretation function $\dot{Int}(e, C)$ that receives an edge e and an annotated CFG C as parameters. This function identifies the basic block, $origin(e)$, from which the given edge e originates, and collects annotations of its origin's incoming edges. To propagate constants in the register state that precedes the basic block, the algorithm joins the annotations of its incoming edges in ll. 1 – 4. This defines the initial state of the basic block, which represents the precondition for the symbolic execution. In ll. 7 – 11, the interpretation function transforms the origin's initial state into its symbolic representation S and adds constraints for each register $r \in R$ which holds a constant value $c \in \{0, \ldots, 2^{64} - 1\}$ at the basic block's entry point (that expresses the constraint $r_{begin} = c$ at the basic block's entry point). At this point (ll. 11), we instruct angr to symbolically execute the basic block $origin(e)$, with the initial state S. Angr transforms the basic block into a logic formula F. Once angr finalizes the formula, we use angr's SMT solver to determine whether or not F can be satisfied. If F cannot be satisfied, the control-flow will never reach the end of the basic block, otherwise, we use angr's SMT solver to determine which registers hold a constant value after executing the basic block (ll. 12 – 19). For each register, first, we add a clause to F that prohibits the assignment of the identified value v to the register at the end of the basic block. Then,

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

Algorithm 1: The interpretation function $\dot{Int}(e, C)$ determines the register state for the edge e by propagating constants and determining constant assignments in the preceding basic block.

```
Input  : Edge e and an annotated control-flow graph C
Output : Lattice element post ∈ Ṡ
pre := NX;
foreach e' ∈ incoming(origin(e)) do
    a := get_annotation(e', C);
    pre := pre ⊔̇ a;
if pre = NX then
    return NX;
S := angr.blank_state();
foreach r ∈ R do
    if pre(r) ≠ X then
        angr.constrain_register_value(S, r, pre(r));
F := angr.symb_exec_basic_block(origin(e), S);
post := λr ∈ R.r ↦ X;
if ¬z3.is_satisfiable(F) then
    return NX;
foreach r ∈ R do
    v := z3.get_model(F).get_value(r);
    F' := F ∧ (r_end ≠ v);
    if ¬z3.is_satisfiable(F') then
        post(r) := v;
return post;
```

we consult the SMT solver to determine if the extended formula remains satisfiable. If the formula cannot be satisfied, we can assure that the basic block will always assign the same value to the register in question. At last, the interpretation function returns a lattice element, representing the register state after executing the basic block, to continue the abstract interpretation.

Applying Abstract Interpretation. 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 (§ 3.2). 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 analysis' incompleteness; she will have to complete the result before generating the seccomp filter.

4 REFINING SECCOMP POLICIES

We leverage Jesse to generate seccomp filters for (containerized) programs. We first determine the program's composing set of binaries by identifying the required libraries from the ELF file. Then, we apply the introduced abstract interpretation to the identified binaries to determine the system call numbers essential for their execution. Finally, we generate a seccomp policy that whitelists the identified system calls; and blacklists the remaining system calls.

This naive approach tends to create coarse seccomp policies, as it does not consider that a system call may not be called; the corresponding system call invocation can be unreachable or not required by the binary. This presents an issue for the standard C library, `libc`. In fact, our evaluation shows that this approach creates seccomp profiles, which, in case of the GNU `libc` v2.28, would grant 273 system calls that stem only from the `libc`. Previous research has shown that 2016 Ubuntu applications use merely 5% of the `libc` on average [22]. We consider such libraries as security threats, as they can govern the access to potentially vulnerable system calls that the target program never intended to use.

The `libc` represents the main gatekeeper between the system's user-space applications and the kernel; it implements wrappers for most of the system calls. As part of the Linux Standard Base (LSB), `libc` establishes a level of portability between distributions and architectures, and can be used by proprietary, closed-source applications. We choose `libc` as the only library, in which we eliminate the unreachable code, as novel debloating techniques have requirements that are not necessarily met by closed-source binaries; they require recompilation [22], meta information in unstripped binaries [1], or must be position-independent [7, 29].

Recent advances in code debloating on source code [22], unstripped [1], and stripped binary level [29] locate unreachable code in libraries. We have implemented an approach similar to Nibbler [1], which we use to complement Jesse by identifying unreachable code in libraries via relocation details. We assume a given framework (e.g., Nibbler) that allows us to locate the code regions associated with each exported library function. This step is done only once, by the security analyst or the library maintainer. We use the code regions as input for our abstract interpretation to identify and create a (per function) map holding the involved system call numbers. This way, we can query the map to spot all system calls associated with the exported library functions, required by a given binary. Note that we do not regard this as an integral component of Jesse. Instead, we request this information from an external party only for one, central library, namely `libc`. Disregarding our decision for this work, code reachability information can be gathered from all involved libraries.

Once we identify the involved binaries, we apply Jesse to unfold their control-flows, leading to a set of `syscall` instructions and uncover the associated system calls. The unreachable code information allows Jesse to consider only viable paths in `libc`.

5 SOUNDNESS

Jesse comprises a *CFG construction* and an abstract interpretation based *constant propagation* technique. Our constant propagation leans on the work of Kildall, for which he provides a proof of soundness [16]. This proof assumes that the CFG is *correct* and *complete*. Yet, since the CFG constructed by Jesse is *correct*, but not necessarily *complete*, we have to pose requirements on the analyzed binaries to ensure the results' soundness.

First, we assume that the binary satisfies **H1**. **H1** considers that benign and untampered `gcc` and `clang` generated binaries can be linearly disassembled [2]. This lends Jesse the ability to (*i*) identify all `syscall` instructions in the binary, and (*ii*) ensure that the disassembled instructions in the basic blocks of the CFG correspond to valid basic blocks in the analyzed program. Due to our conservative CFG construction mechanism, we guarantee that all edges in the

CFG are valid. Also, **H1** assumes the analyzed binaries adhere to the calling conventions; violating this would lead to false results.

In addition, we assume that every system call invocation in the target binary adheres to **H2** or **H3**. If **H2** holds, the identified system call number is the only system call number that can be passed to the `syscall` instruction under analysis. If **H2** does not hold for this particular `syscall` instruction, yet, **H3** does, we can be sure that indirect jumps do not affect the computation of the system call number. In other words, the incomplete CFG does convey the necessary information for Jesse to derive that a `syscall` should in fact be able to call different system call handlers. In such cases, the analysis either directly assigns X to `rax` as soon as it identifes a second possible value, or it propagates X from one of the function arguments to `rax` at the `syscall` instruction. In both cases, Jesse informs the analyst, regardless whether the function has been called indirectly. If the provided CFG is incorrect, **H1** and **H2** will not suffice for sound results. As such, we have ruled out heuristic-based CFG construction tools; they can miss code or introduce illegal edges.

Finally, **H4** bases upon the observation that most binaries do not call system calls directly, but rather use wrappers in the standard C library. While this observation does not directly contribute to the soundness of Jesse, it does entail one corner case, which deals with *generic* system call wrappers, such as the `syscall()` function in `libc`. Jesse determines the system call number provided to such wrappers, only if they are called directly. In such cases, it considers calls to *generic* system call wrappers as hypothetical invocations of `syscall` instructions, which receive the system call number in another register instead of `rax`. Yet, if a binary calls *generic* system call wrappers indirectly, Jesse will not be able to identify these calls and hence to derive the system call number. We have not encountered a single case of this type.

6 EVALUATION

We applied Jesse to 1,064 binaries of the Debian `buster` *base* image to identify system calls and link them to their respective system call numbers. We have manually inspected all binaries and identified that 78 (out of 1,064) ELF binaries hold at least one `syscall` instruction. Finally, to compare our results, we applied the system call extraction framework, Confine [11], to the same set of binaries.

Table 1 shows the *manually* identified 984 distinct system call invocations (i.e., `syscall` instructions), to which we have matched the associated system call numbers. Jesse identified 100% of the system call invocations and linked the exact system call number in 964 cases (98.0%). We do not consider the remaining 2.0% as false negatives; in 20 cases, Jesse pointed to the exact location of the troubling `syscall` instructions and requested manual assistance. Granted, this is an involved, yet, realistic strategy: application developers or security analysts should create the seccomp policies.

Although the analyzed binaries adhere to the x86-64 calling conventions, we did not linearly disassemble 7 (0.66%) binaries; these binaries intertwine *data* with *code* in their `.text` sections, and thus violate **H1**. We have encountered 10 (1.02%) system call invocations which violate **H2**. We have not observed any violations of **H3**. Jesse has informed us about each of these violations; once we have resolved the violation, we were able to resume Jesse to complete the analysis.

Table 1: Identified system call numbers of 1,064 binaries of the Debian buster base image.

	Manual	Confine	Jesse
Raw numbers:			
◦ Analyzed binaries	1,064	1,064	1,064
◦ Identified system call invocations	984	1142	984
System call number identification:			
◦ System calls flagged for manual inspection		183	20
◦ Falsely identified system call numbers	0	32	0
◦ Missed system call numbers	0	76	0
Overall result:			
◦ Correctly identified system call numbers	100%	89.7%	98.0%

Surprisingly, Confine reported 158 system call invocations in addition to the (manually) identified invocations and flagged 183 for manual inspection. We found that it falsely interprets byte sequences in the `.data` section as `syscall` instructions, and considers calls to functions, whose symbols contain the string *"syscall"*, as system call invocations. Even though its approximation is conservative, it can miss system calls in case of tail-call optimizations. Confine will miss system calls, if the compiler replaces a `call` instruction to a generic system call wrapper with a `jmp` instruction; its implementation only considers `call`s, yet, disregards `jmp`s to such wrappers. Finally, we discovered that it has misidentified (32) and missed available system call numbers (76), which introduces false negatives. Overall, in 89.7% Confine links the correct system call number to one of the 984 available `syscall` instructions.

Our results show that Jesse has automatically identified the correct system call number in 98%. Jesse did neither over- nor under-approximate any system call invocations in the given binaries. Assuming that the analyst would always be able to identify the system call number of the flagged `syscall` instructions, Jesse has not produced any false negatives. Combined with Confine's ability in analyzing `libc`, Jesse can significantly improve seccomp policies for (containerized) applications.

6.1 Enhancing Docker Container Security

We applied Jesse to five popular Docker containers on Docker Hub (these are `httpd`, `mysql`, `nginx`, `postgress`, and `redis` Docker containers). Jesse dissected the containers to identify only necessary programs and libraries (§ 4). Jesse linked the exact system call number to 719 of 725 `syscall` instructions in containers' latest version. Jesse did not determine the system call numbers of the remaining 6 `syscall`s: 3 in `libc`, 2 in `libpthread`, and 1 in `mysqld`. Hence, in 99.17%, our analysis determined the correct system call (verified by manual inspection). We do not consider the remaining 0.83% as false negatives as Jesse pointed to the location of the troubling `syscall` instructions and requested manual assistance.

Further, we applied the discussed optimization that allows Jesse to identify unreachable code and hence to focus only on viable paths in the `libc` (§ 4). The optimization drops the `syscall` instructions in the above container binaries to a subset that is reached by the analyzed program. To accommodate closed-source binaries, we apply the optimization only to `libc`; system calls of other dissected container binaries are accumulated through the general system call number analysis without the additional optimization. On average, the generated policies restricted 54.7% of all system calls

(55.3% for `httpd v2.4.41`; 41.2% for `mysql v8.0.19`; 61.1% for `nginx v1.17.9`; 42.9% for `postgres v12.2`; and 58.5% for `redis v5.0.2`). This way, we have significantly improved the granularity of Docker's default seccomp policy, which prohibits, depending on the container configuration, 10.6% to 20.4% of the system calls.

To rule out false negatives, we applied these policies to the above Docker containers and ran benchmarks, testing the container's main functionality. Finally, to quantify the impact of this pre-processing step, we have applied Jesse to the GNU `libc` v2.28 that is part of the Debian base installation. It turns out that the library holds 404 `syscall` instructions, which call 273 distinct system calls. Without applying debloating techniques, the resulting seccomp profile would grant 273 distinct system calls in addition to those that are not part of `libc`. As such, without this additional pre-processing step the restrictions would be less effective. For instance, the generated seccomp policies for the Docker containers Httpd, MySQL, and Nginx would restrict only 17.3%, 14.1%, and 17.3% of all system calls without debloating the `libc`, instead of 55.3%, 41.2%, and 61.1% when debloating the `libc` respectively.

6.2 Reasons for Incomplete Mappings

Table 1 shows that Jesse requested human support in 20 cases. The reason for the incomplete mapping is threefold and in line with our hypotheses **H1** – **H4**. First, there exist functions, e.g., `syscall()` in `libc`, which allow to select arbitrary system calls by specifying the system call number in the function parameters. Since the exact system call number depends on the caller, an analysis of `syscall()` alone cannot determine the provided value (violation of **H2**). We can address this issue by generalizing our definition of the abstract interpretation. Instead of identifying the value in `rax` before the `syscall` instruction, we cause the abstract interpretation to identify the value in `rdi` (i.e., the first function parameter) immediately before calling `syscall()`. The exact register depends on the called function. The second class refers to the incompleteness of the `Capstone` disassembly framework. For instance, `Capstone` failed to disassemble rare floating-point instructions and to determine data in the `.text` section (violation of **H1**). This has impeded further analysis. Jesse encountered 7 cases, in which `Capstone` did not disassemble parts of the binary. We have extended Jesse to mark such binaries for manual inspection, in which the analyst can assist `Capstone` by explicitly stating whether the troubling section belongs to the `.text` or `.data` section. To solve this issue, one has to extend the capabilities of `Capstone`. Finally, a small class of functions reads the system call number from memory. Albeit this strategy impedes static analysis, in this and all upper cases Jesse narrows down the `syscall` instruction's exact location and lets analysts incorporate their knowledge to satisfy the policy.

6.3 Comparison with the State-of-the-Art

We position Jesse among three existing frameworks, which employ static [7, 11] and dynamic analysis [27] to extract the set of system calls for (containerized) applications. We show that the inherent properties of dynamic analysis approaches can lead to false negatives. While static analysis frameworks are great candidates, we amplify upon their limitations and reveal in what way Jesse can complement their functionality.

Dynamic Analysis. Dynamic approaches generate rigorous policies. Yet, they highly depend on the achieved program coverage of the test suites, and hence merely underapproximate the set of system calls that is vital for the analyzed program. If a benchmark misses an execution path that leads to a system call, a potential execution of the hereby introduced false negative (a falsely unauthorized system call in the seccomp policy) will eventually crash the process. We discovered that the analyzed dynamic analysis framework overlooked a set of essential system calls. 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 underline the need for a robust static analysis.

Static Analysis. Confine [11] identifies system call numbers for containerized applications to create seccomp policies. For this, it requires the source code of `libc`. Confine's analysis accurately maps system calls of exported functions in `libc` to binaries using the respective functions. Unfortunately, the analysis of the remaining system calls (outside `libc`) lacks precision. We showed that Confine can produce false negatives, which are intolerable in production systems. In addition, it uses a backpropagation-only technique, which, by design, cannot deal with loops. Jesse is not limited by loops and does not require external assistance to identify the system call number in such cases. On the other hand, `sysfilter` [7] provides highly precise results, as it identifies and eliminates system calls in libraries linked against binaries, yet, not needed at run-time. Unfortunately, `sysfilter` only works with PIC binaries, which is a limiting factor of the analysis. For instance, 43.9% of all binaries in the Debian base installation comprise dynamically-linked, non-PIC binaries, and hence should not be disregarded. Consequently, we regard the techniques applied in Jesse complementary to both Confine and `sysfilter`.

7 RELATED WORK

Janus [13] is one of the first sandboxes for untrusted applications. It uses the Solaris' tracing capabilities to interpose and confine system calls. Boxify [4] establishes a framework that uses Android's process isolation to execute apps in the context of a trusted process with restricted permissions. It limits the capabilities of untrusted apps by intercepting system calls and calls to the Android API. DroidSafe [14] models the Android API and uses static analysis to find information leaks in Android apps. Similarly, FlowDroid [3] is a static taint analysis that offers higher precision. Sadly, these approaches cannot determine permissions and system calls that should be whitelisted. As such, Jamrozik et al. [15] present boxmate, a framework that executes tests against target applications on Android to extract the set of resources required during the tests. DroidBot [20] generates custom tests used by the framework of Le et al. [19], to also consider parameters of calls to the Android API. Wan et al. [27] mine sandboxes for Linux containers. They apply dynamic analysis to determine authorized system calls and filter all the others with seccomp filters. Similarly, Confine [11] constructs

seccomp profiles based on the results of a static source code analysis. However, this does not work reliably if a container contains closed-source software which directly invokes system calls, e.g., via by either calling the `syscall()` function or executing the `syscall` instruction. A novel mechanism is the temporal system call specialization by Ghavamnia et al. [12] that divides a program's execution into phases in which it applies different seccomp filters.

Another direction is to apply container debloating. Rastogi et al. [23] develop Cimplifier partitions containers into a set of minimal containers. Their research applies either dynamic analysis or static analysis [24] to remove unused programs from the container's image. Other debloating techniques target program libraries. Quach et al. [22] use static analysis to discover unneeded instructions in libraries. The authors base their analysis on the LLVM IR, and hence they have to recompile binaries with `clang`. Agadakos et al. [1] develop Nibbler, which analyses binaries to identify and erase unused functions in libraries. Yet, the presented tools require ELF symbols, and hence unstripped binaries. Egalito [29] is a framework for recompiling binaries to enhance their security (e.g., by reordering instructions, introducing retpolines, Just In-Time (JIT)-shuffling, etc.). In addition, Egalito applies techniques to debloat programs and libraries and is extended by DeMarinis et al. [7] to also support the construction of seccomp profiles for individual programs. Although this also allows the construction of seccomp for Linux containers, their approach focuses more on general binary hardening and trades safety for precision. Contrary to Jesse, Egalito requires PIC to gather metadata. Although this approach significantly reduces the code size, it cannot remove unneeded code that is disabled in the configuration. To overcome this, Koo et al. [18] develop a configuration-driven software debloating technique.

Generating system call whitelists has been subject to Host-based Intrusion Detection Systems (HIDSs). Wagner et al. [26] and Feng et al. [9] use system call traces to detect malicious behavior. They apply static analysis to build automatons representing the benign system call traces and use them to detect malicious behavior at run-time. Yet, both approaches rely on hand-crafted models to identify the set of system calls of library functions. Finally, constant propagation is a well-studied direction. Kinder et al. [17], Fleury et al. [10], and Bardin et al. [5] employ constant propagation to recover CFGs from binaries. While Bardin et al. rely on symbolic execution to underapproximate the CFG, Kinder et al. and Fleury et al. overapproximate the CFG by using abstract interpretation identifying possible targets of indirect branches.

8 CONCLUSION

In this paper, we proposed Jesse, a tool that leverages static analysis to tailor seccomp policies to ELF binaries and Linux containers. Specifically, we utilized abstract interpretation to identify system calls in ELF binaries. We further combined our analysis with library debloating techniques to exclude unreachable code. We demonstrated the effectiveness of our work by applying Jesse to all binaries in the Debian *base* image and popular Docker containers, and by reinforcing seccomp against container escalation exploits. The outcomes of this work confirm that Jesse can be the last line of defense against real-world exploits.

ACKNOWLEDGMENT

This work has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No. 883275 (HEIR). Any opinions, findings, conclusions, or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the aforementioned supporters.

REFERENCES

[1] Ioannis Agadakos et al. 2019. Nibbler: Debloating Binary Shared Libraries. In *Annual Computer Security Applications Conference (ACSAC)*.
[2] Dennis Andriesse et al. 2016. An In-Depth Analysis of Disassembly on Full-Scale x86/x64 Binaries. In *USENIX Security Symposium*.
[3] Steven Arzt et al. 2014. Flowdroid: Precise Context, Flow, Field, Object-Sensitive and Lifecycle-Aware Taint Analysis for Android Apps. *ACM SIGPLAN Notices*.
[4] Michael Backes et al. 2015. Boxify: Full-Fledged App Sandboxing for Stock Android. In *USENIX Security Symposium*.
[5] Sébastien Bardin et al. 2011. Refinement-based CFG Reconstruction from Unstructured Programs. In *International Workshop on Verification, Model Checking, and Abstract Interpretation (VMCAI)*.
[6] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Symposium on Principles of Programming Languages*.
[7] Nicholas DeMarinis et al. 2020. `sysfilter`: Automated System Call Filtering for Commodity Software. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*.
[8] Alexis Engelke. 2022. Lightweight Disassembler. https://github.com/aengelke/lwdpy.
[9] Henry Hanping Feng et al. 2004. Formalizing Sensitivity in Static Analysis for Intrusion Detection. In *IEEE Symposium on Security and Privacy (S&P)*.
[10] Emmanuel Fleury et al. 2015. Insight: An Open Binary Analysis Framework. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*.
[11] Seyedhamed Ghavamnia et al. 2020. Confine: Automated System Call Policy Generation for Container Attack Surface Reduction. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*.
[12] Seyedhamed Ghavamnia et al. 2020. Temporal System Call Specialization for Attack Surface Reduction. In *USENIX Security Symposium*.
[13] Ian Goldberg et al. 1996. A Secure Environment for Untrusted Helper Applications: Confining the Wily Hacker. In *USENIX Security Symposium*.
[14] Michael I Gordon et al. 2015. Information Flow Analysis of Android Applications in Droidsafe. In *ISOC Network and Distributed System Security Symposium (NDSS)*.
[15] Konrad Jamrozik et al. 2016. Mining Sandboxes. In *International Conference on Software Engineering (ICSE)*.
[16] Gary A Kildall. 1973. A Unified Approach to Global Program Optimization. In *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*.
[17] Johannes Kinder and Helmut Veith. 2008. Jakstab: A static analysis platform for binaries. In *International Conference on Computer-Aided Verification (CAV)*.
[18] Hyungjoon Koo et al. 2019. Configuration-Driven Software Debloating. In *European Workshop on System Security (EuroSec)*.
[19] Tien-Duy B Le et al. 2018. Towards Mining Comprehensive Android Sandboxes. In *International Conference on Engineering of Complex Computer Systems*.
[20] Yuanchun Li et al. 2017. DroidBot: A Lightweight UI-guided Test Input Generator for Android. In *International Conference on Software Engineering Companion*.
[21] Niels Provos. 2003. Improving Host Security With System Call Policies. In *USENIX Security Symposium*.
[22] Anh Quach et al. 2018. Debloating Software Through Piece-Wise Compilation and Loading. In *USENIX Security Symposium*.
[23] Vaibhav Rastogi et al. 2017. Cimplifier: Automatically Debloating Containers. In *Joint Meeting on Foundations of Software Engineering*.
[24] Vaibhav Rastogi et al. 2017. New Directions for Container Debloating. In *Workshop on Forming an Ecosystem Around Software Transformation*.
[25] Yan Shoshitaishvili et al. 2016. SoK: (State Of) the Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy (S&P)*.
[26] David Wagner and Drew Dean. 2001. Intrusion Detection via Static Analysis. In *IEEE Symposium on Security and Privacy (S&P)*.
[27] Zhiyuan Wan et al. 2017. Mining Sandboxes for Linux Containers. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*.
[28] Richard Wartell et al. 2011. Differentiating Code from Data in x86 Binaries. In *European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML-PKDD)*.
[29] David Williams-King et al. 2020. Egalito: Layout-Agnostic Binary Recompilation. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.