Automated Policy Synthesis for System Call Sandboxing

SHANKARA PAILOOR, University of Texas at Austin, USA
XINYU WANG, University of Michigan, USA
HOVAV SHACHAM, University of Texas at Austin, USA
ISIL DILLIG, University of Texas at Austin, USA

System call whitelisting is a powerful sandboxing approach that can significantly reduce the capabilities of an attacker if an application is compromised. Given a *policy* that specifies which system calls can be invoked with what arguments, a sandboxing framework terminates any execution that violates the policy. While this mechanism greatly reduces the attack surface of a system, manually constructing these policies is time-consuming and error-prone. As a result, many applications —including those that take untrusted user input— opt not to use a system call sandbox.

Motivated by this problem, we propose a technique for automatically constructing system call whitelisting policies for a given application and policy DSL. Our method combines static code analysis and program synthesis to construct *sound and precise policies* that never erroneously terminate the application, while restricting the program's system call usage as much as possible. We have implemented our approach in a tool called Abhaya and experimentally evaluate it 674 Linux and OpenBSD applications by automatically synthesizing Seccomp-bpf and Pledge policies. Our experimental results indicate that Abhaya can efficiently generate useful and precise sandboxes for real-world applications.

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

Additional Key Words and Phrases: Security, Sandboxing, Abstract Interpretation, Program Synthesis

ACM Reference Format:
Shankara Pailoor, Xinyu Wang, Hovav Shacham, and Isil Dillig. 2020. Automated Policy Synthesis for System Call Sandboxing. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 135 (November 2020), 26 pages. https://doi.org/10.1145/3428203

1 INTRODUCTION

A "sandbox" is a security mechanism that separates applications from the rest of the system and prevents vulnerabilities in one component from compromising others. While there are different types of sandboxing mechanisms, a common approach is to restrict what system calls (syscalls) an application can make. Because attackers typically compromise a system by invoking syscalls in a way that normal applications do not [Provos 2003; Provos et al. 2003; Saltzer and Schroeder 1975], such *syscall sandboxing* mechanisms provide an effective way for mitigating the capability of an attacker [Krohn et al. 2005; Laurén et al. 2017]. For this reason, there are several frameworks, such as Seccomp-bpf [Edge 2015], Pledge [Pal 2018], and SELinux [Smalley 2002], that allow programmers to construct syscall sandboxes. Specifically, given a user-provided sandboxing *policy* that whitelists

Authors' addresses: Shankara Pailoor, University of Texas at Austin, Austin, Texas, USA, spailoor@cs.utexas.edu; Xinyu Wang, University of Michigan, Ann Arbor, Michigan, USA, xwangsd@umichigan.edu; Hovav Shacham, University of Texas at Austin, Austin, Texas, USA, hovav@cs.utexas.edu; Isil Dillig, University of Texas at Austin, Austin, Texas, USA, isil@cs.utexas.edu.

2475-1421/2020/11-ART135
https://doi.org/10.1145/3428203

certain syscall usage patterns, such frameworks enforce this policy at run-time by terminating executions that do not conform to the policy.

Despite their potential to substantially mitigate damage, programmers rarely write syscall sandboxing policies in practice. For example, consider the Unix utility `strings` which looks for printable sequences in binary files. As recently as 2014, the GNU version of this utility had a memory vulnerability that allowed attackers to take *complete control* of the user's account [Zalewski 2014]. Had this utility been equipped with a syscall sandbox that restricted access to the file system and network, the damage from this vulnerability would not have been nearly as severe. Unfortunately, this phenomenon is not restricted to just the `strings` utility: among the top *one thousand* debian packages that we inspected, we found that only *six* of them used a syscall sandbox.

In practice, developers do not leverage syscall sandboxing capabilities because manually constructing these policies is both difficult and error-prone. First, in order to write a useful policy, the developer needs to identify all the syscalls (and their argument values) that could be invoked by the program including code in third party libraries. Second, building a sandbox is not a one-time effort, as the policy may need to be updated whenever the application is modified. Third, different operating systems expose different policy languages that vary greatly in both syntax and semantics, making it especially difficult for cross-platform applications to build and maintain their sandboxes. Finally, since sandboxes are constructed manually, they inevitably contain bugs and may end up terminating legitimate executions of a program [chr 2013, 2015, 2016, 2017a,b].

In this paper, we present a technique for *automatically* synthesizing syscall sandboxing policies for C/C++ programs. Given a program P and a sandboxing policy language L, our technique automatically constructs a policy ψ in L that over-approximates the necessary syscall behavior in P as tightly as possible. At a high-level, our approach consists of two phases, namely, syscall analysis and policy synthesis. In the first syscall analysis phase, we perform abstract interpretation on P to compute a so-called *syscall invariant* Φ that *over-approximates* syscall usage patterns of P for well-defined executions. However, since this syscall invariant is, in general, not expressible in the given policy language L, the goal of the second policy synthesis phase is to find a best policy in L that over-approximates P as tightly as possible. Since the synthesizer is parametrized by a policy language L, this approach allows us to generate policies for multiple syscall sandboxing environments with different policy languages.

We have implemented our approach in a tool called Abhaya and used it to generate policies for two policy DSLs (i.e., Seccomp-bpf and Pledge) across 674 C/C++ programs. For the most-downloaded debian packages, Seccomp-bpf policies that are automatically synthesized by Abhaya can block nearly all known privilege escalation vulnerabilities for the Linux kernel in the past 5 years. Moreover, we compared Abhaya's automatically synthesized policies against hand-crafted ones for both Seccomp-bpf and Pledge and show that (a) Abhaya-generated policies are competitive with developer-written ones, and (b) in some cases, the policies inferred by Abhaya reveal developer-confirmed bugs in the original manually-written policies.

In summary, this paper makes the following contributions:

- We introduce the *policy synthesis* problem for automatically constructing syscall sandboxes.
- We present a precise and scalable interprocedural static analysis for automatically computing syscall invariants.
- We describe a new optimal program synthesis algorithm for finding a best policy that over-approximates the computed syscall invariants.
- We conduct an evaluation on hundreds of applications and show that Abhaya is able to generate useful policies that are both competitive with developer-written policies as well as effective at blocking real-world attacks.

```
#include <stdio.h>

int main(int argc, char ** argv)
{
    char c;
    const char *infile = argv[1];
    FILE *fp = fopen(infile, "re");
    if (fp == 0)
        return 0;
    while ((c = getc(fp)) != EOF) {
        if ((putchar(c)) == EOF)
            break;
    }
    return 0;
}
```

```
FILE * fopen(const char *file, const char *mode) {
    FILE *fp = __new_fp(); char c;
    int f, o, m = 0, i = 0;
    switch ((c = mode[i++])) {
      case 'r': m = 0x0; break;
      case 'w': m = 0x1; break;
      ...  }
    while ((c = mode[i++]) != '\0') {
        switch (c) {
          case 'e': o = 0x10000; break;
          ...
          default: o = 0; break; }
    }
    if ((f = open(file, m | o, 0x0)) < 0)  ...
    ...
}
```

Fig. 1. Motivating example

Pledge group	**Description**
inet	*Allows usage of select networks syscalls if they operate in the inet or inet6 domain.*
rpath	*Only allows read-only effects on the filesystem*
wpath	*Allows syscall usage patterns that may cause write effects on the filesystem*
stdio	*Allows syscall usage patterns that are required for libc stdio to work*
cpath	*Allows syscall usage patterns that may create new files or directories in the filesystem*
tmppath	*Allows system calls such as create, read, or write to do operations in /tmp directory*

Table 1. Sample Pledge group descriptions

Threat Model. We assume the existence of an attacker who wants to take over a target system by exploiting a benign but vulnerable application that runs on the system. In particular, the attacker first takes control of the application by crafting inputs that trigger some undefined behavior, such as buffer overflow or double free. Then, she tries to gain privileges through the invocation of system calls and compromises the rest of the system.

2 MOTIVATING EXAMPLE

In this section, we motivate the policy synthesis problem and provide a high-level overview of our solution with the aid of the code shown in Figure 1. This example involves a cat-like program that reads a file and then prints its contents to the console character by character. The programmer wants to write a policy in the Pledge sandboxing framework for OpenBSD to secure the application against potential exploits.

Specifically, a Pledge policy consists of a set of pre-defined groups where each group specifies which syscalls can be invoked with what arguments. Given a policy with groups $g_1, \cdot\cdot, g_n$, the Pledge framework terminates the application if it attempts to execute a syscall that is not allowed by any of these g_i's. To give the reader some intuition, Table 1 shows a subset of the 35 groups exposed by the Pledge framework along with a brief description. For instance, the inet group allows the program to invoke the socket syscall but only when its first argument is *AF_INET* or

AF_INET6. As another example, groups rpath and wpath both allow the open syscall, but rpath allows opening the file in read-only mode (indicated by the second argument), whereas wpath allows both reading from and writing to the file.

Going back to Figure 1, a suitable Pledge policy for this program would be {stdio, rpath} because both groups are necessary for the application to function correctly. To see why, let us focus on the main function. Here, stdio is necessary because the application needs to write to the console (putchar, line 11) and read from a file (getc, line 10). Likewise, rpath is necessary because the application needs to open a file (line 7). Observe that failing to include rpath in the policy would prevent the application from functioning correctly, whereas including additional groups (e.g., wpath) would allow the application to have access to more resources than it needs, thereby enlarging the program's attack surface.

2.1 Current Practice

Before explaining our solution, we briefly discuss how developers *currently* construct syscall policies. To manually construct a Pledge policy, the developer first needs to identify all the *resources* (i.e., syscalls and their arguments) required by the application; then, she needs to map them to a set of Pledge groups that whitelist the required functionality. Furthermore, this policy should be "tight" in that it should not include more groups than necessary.

However, manually constructing such a Pledge policy is non-trivial. First, since library functions are opaque to the developer, it is challenging to determine which syscalls the application makes, let alone their argument values. For instance, even the tiny program in Figure 1 calls several libc functions that may use syscalls in ways one may not expect, e.g., getc transitively invokes syscalls mmap, brk, and sysctl to improve performance. Furthermore, even if the developer can accurately perform this first task, it is non-trivial to map this information to the tightest Pledge policy. For instance, one may be tempted to think that the policy {stdio} is sufficient because stdio permits all the neccessary resouces required by the application (e.g open files). However, it is not sufficient because stdio only allows opening *specific* files whereas the application needs to be able to open any file.

2.2 Our Approach

We now explain how our approach generates the desired Pledge policy for this example. As mentioned in Section 1, our approach consists of two phases. In the first *syscall analysis* phase, we perform static analysis to compute an over-approximation of the resources required by the program. In the *policy synthesis* phase, we map the output of the analysis to an optimal Pledge policy. In the remainder of this section, we motivate various design choices behind our analysis and synthesis techniques in light of the running example.

2.2.1 Sycall Analysis Phase.

The goal of the syscall analysis phase is to compute the set of system calls the program can invoke, along with predicates that constrain their argument values (see Table 2). There are five key observations that guide our analysis design:

Observation #1: Integer-valued syscall arguments often serve as important flags and crucially affect policy choice. For example, the second argument of the open syscall is an integer indicating whether the file is opened in read or write mode. In order to choose the right Pledge groups that grant necessary permissions, we need to infer values of this argument.

Observation #2: While each individual argument of a syscall is important for determining a suitable policy, *relationships* between them are typically *not*. In particular, we have observed that argument values of syscalls are often independent of one another.

Syscall	Predicate
open	$arg_2 = 0x10000 \wedge arg_3 = 0$
brk	$arg_1 = 0$
mmap	$(arg_3 = 3 \vee arg_3 = 0) \wedge arg_4 = 4098 \ \wedge (arg_5 = -1) \wedge arg_6 = 0$
read, fstat	*true*
write	*true*
sysctl	$(arg_1[0] = 2 \wedge arg_1[1] = 12) \ \wedge (arg_2 = 2)$

Table 2. Syscall information produced by static analysis

Observation #3: It is important for the analysis to support some type of disjunctive reasoning. For instance, in our running example, the second argument of `open` (line 14 in `fopen` function) depends on variable `m`, whose value differs along different execution paths after the `switch` statement.

Observation #4: There are many cases where important syscall arguments depend on values of array elements. For instance, in our running example, the second argument of `open` is dependent on the `mode` array. Thus, it is important for our static analysis to reason about array elements.

Observation #5: Since most syscalls are invoked in library functions with complex implementations, it is crucial for the analysis to be both interprocedural and scalable.

Motivated by these observations, we designed a (top-down) summary-based interprocedural static analysis that focuses on tracking values of integer-typed variables and single-level arrays. Specifically, observations #1 and #3 lead us to use a disjunctive numeric abstract domain, while observation #2 indicates that a *relational* abstract domain might be an overkill. In combination with observation #4, this leads us to use the *reduced product* [Cousot et al. 2011] of the array expansion [Feautrier 1988] and disjunctive interval domains [Sankaranarayanan et al. 2006] to achieve precise reasoning for both array contents and integer variables. Finally, to perform scalable interprocedural analysis (as motivated by observation #5), our method uses *procedure summaries* [Das et al. 2002; Mangal et al. 2014; Zhang et al. 2014] to summarize the relevant behavior of each procedure and avoids re-analyzing functions under similar calling contexts.

2.2.2 Policy Synthesis Phase. Given the syscall usage information produced by our static analysis, the goal of the policy synthesis phase is to construct a sandboxing policy that over-approximates this information as tightly as possible. For our running example in Figure 1, the output of the synthesizer should be a Pledge policy that minimizes the program's system call usage while granting access to all resources listed in Table 2. Furthermore, since our technique should not be specialized to Pledge, our synthesis algorithm should be able to generate policies for a broad class of syscall whitelisting policy DSLs.

In more detail, our synthesis algorithm takes as input a policy DSL L and a syscall invariant Φ which summarizes the program's syscall usage (e.g., information from Table 2) as a logical formula. Then, the goal of the synthesizer is to generate a program ψ in L such that $\Phi \Rightarrow [\![\psi]\!]$ is valid (i.e., ψ should allow all resources required by the program). Furthermore, since we want to find a "tightest" policy expressible in L, the synthesis output should guarantee that ψ is not weaker than any other policy ψ' satisfying $\Phi \Rightarrow [\![\psi']\!]$.

Going back to our example in Figure 1, the Pledge policy $\psi_1 \equiv \{\texttt{stdio}\}$ is not a valid synthesis result because ψ_1 does not allow some operations (e.g., opening files) needed by the program. On the other hand, the policy $\psi_2 \equiv \{\texttt{stdio}, \texttt{cpath}\}$ is also not a valid synthesis result because there exists a tighter policy $\psi \equiv \{\texttt{stdio}, \texttt{rpath}\}$ (such that $[\![\psi]\!] \Rightarrow [\![\psi_2]\!]$) and ψ does allow all legitimate executions of this program.

As illustrated by the discussion above, this is an *optimal synthesis* problem that requires finding a "best" program that satisfies the specification. However, since the search space over all possible policies in L is very large, solving this optimal synthesis problem is computationally challenging. For instance, in the case of Pledge, the search space consists of 2^{35} different policies. Our algorithm addresses this challenge by utilizing domain-specific automated reasoning techniques to prune the search space. In particular, instead of searching through all 2^{35} possible policies, we can solve this optimal synthesis problem by considering only 64 different Pledge policies for our running example.

3 SYSCALL WHITELISTING POLICIES

In order to formalize our problem, we define a *program state* σ to be a pair (pc, v) where pc is a program counter and v is a valuation that maps program variables to values. Given program P, we write $(pc, v) \Downarrow (pc', v')$ to indicate that P is in state (pc', v') after executing instruction $P[pc]$ on valuation v. Then, a *program trace* τ is a sequence of states, $\sigma_1, \cdot\cdot, \sigma_n$, such that $\sigma_i \Downarrow \sigma_{i+1}$ and σ_1 is an initial state of P. We also write $\eta(\tau)$ to denote the set of states in τ.

In this paper, we consider programs written in C-like languages whose semantics may be undefined in certain cases, such as when the program accesses an array out of bounds. Thus, given an input v of P, we say that v is a *valid input* for P if it does not result in undefined behavior. We also say that a trace τ of program P is *legitimate* if it corresponds to an execution of P on some valid input v.

3.1 Feasible Syscall States

Because this paper is concerned with system call whitelisting, we need a way to characterize the program's syscall usage. To simplify presentation, we assume that programs make system calls using a special instruction $syscall(f, x_1, \cdot\cdot, x_n)$.

Definition 3.1. **(Syscall projection)** Given a trace τ of program P, the *syscall projection* of τ, denoted $\pi(\tau)$, yields the set of all states $\sigma = (pc, v) \in \eta(\tau)$ such that $P[pc]$ is a syscall instruction (i.e., $P[pc] = syscall(f, x_1, \cdot\cdot, x_n)$).

Definition 3.2. **(Feasible syscall states)** Given program P, we say $(f, v_1, \cdot\cdot, v_n)$ is a *feasible syscall state* of P if there exists a *legitimate* execution τ of P such that $\pi(\tau)$ contains a state (pc, v) where $P[pc] = syscall(f, x_1, \cdot\cdot, x_n)$ and for all $i \in [1, n]$, we have $v(x_i) = v_i$.

Hence, *feasible syscall states* characterize all legitimate syscall usage patterns of a given program.

3.2 Syscall Policies

Syscall sandboxing frameworks provide a policy language that whitelists the application's feasible syscall states. While we do not fix a specific language in which such policies are written, we use the notation $[\![\psi]\!]$ to denote the set of all syscall states whitelisted (i.e., allowed) by ψ according to the semantics of the underlying policy DSL.

Example 3.3. Consider the policy $\psi = \{rpath\}$ in the Pledge framework. Based on the semantics of the *rpath* group, we have $\forall x, y.\ (open, x, 0, y) \in [\![\psi]\!]$ because *rpath* allows opening a file in read-only mode and value 0 for the second argument of *open* indicates read-only permission. However, $\forall x, y.(open, x, 1, y) \notin [\![\psi]\!]$ since *rpath* does not allow opening files in write mode.

Given a policy ψ and input program P, we need to reason about whether ψ is "correct" respect to P. Thus, we introduce the following notions of *soundness* and *completeness*.

Definition 3.4. **(Sound policy)** Given a program P and policy ψ, we say that ψ is *sound* with respect to P if $[\![\psi]\!]$ contains *all* feasible syscall states of P.

In other words, a sound policy for P does not terminate any legitimate executions of P.

$$
\begin{array}{rcl}
\texttt{Prog}\ P & := & D*, F+ \\
\texttt{Func}\ F & := & (\bar{v}_{out})\ \texttt{proc}\ \texttt{f}(\bar{v}_{in})\{D*, S+\} \\
\texttt{Decl}\ D & := & \tau\ x,\ \ \tau \in \{\texttt{int}, \texttt{ptr}\} \\
\texttt{Stmt}\ S & := & x = e \mid x[i] = y \mid x = y[i] \mid x = \texttt{alloc}(y) \mid S_1; S_2 \\
 & & \mid \texttt{if}(C)\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \mid \vec{r} = \texttt{call}\ (f, x_1, \cdots, x_k) \\
 & & \mid \vec{r} = \texttt{syscall}\ (f, x_1, \cdots, x_k) \\
\texttt{Pred}\ C & := & e \mid \neg p \mid p_1 \wedge p_2 \mid p_1 \vee p_2 \mid e_1 \oplus e_2 \\
\texttt{Expr}\ E & := & c \mid x \mid e_1 \otimes e_2
\end{array}
$$

Fig. 2. Programming language where $\oplus$ and $\otimes$ denote logical and arithmetic operators respectively.

Definition 3.5. **(Complete policy)** Given a program P and a policy ψ, we say that ψ is *complete* with respect to P if $[\![\psi]\!]$ contains *only* feasible syscall states of P.

That is, a complete policy only allows executions with legitimate syscall usage patterns. Ideally, the policy should be both sound *and* complete, so that is allows exactly those program executions with legitimate syscall usage patterns. However, policy languages *intentionally* restrict which syscall states can be whitelisted in order to minimize run-time overhead. Thus, in general, it is not possible to write policies that are *both* sound and complete.

As a result, we define a policy to be *correct* as long as it is sound according to Definition 3.4, because a policy that terminates legitimate executions would be disastrous in practice. However, not every correct policy is useful for mitigating attacks (e.g., consider a sound policy that allows *all* syscall states). Thus, in practice, we would like policies that are not only sound but also as complete as possible.

4 SYSCALL ANALYSIS

In this section, we describe our analysis for computing an over-approximation of a program's feasible syscall states. Specifically, our analysis computes a so-called *syscall invariant*, which maps each system call invoked by the application to a formula describing possible values of its arguments.

Definition 4.1. A *syscall invariant* Υ for a program P is a mapping from each syscall f to a formula Φ over variables $\vec{a}$ (denoting f's arguments) such that, for every feasible syscall state $\sigma = (f, v_1, \cdot\cdot, v_n)$ of P, we have $I_\sigma \models \Phi$ where I_σ is an interpretation that maps each a_i to v_i.

4.1 Programming Language

We describe our static analysis using the simple imperative language shown in Figure 2. Here, statements include standard constructs like assignments, loads, stores, conditionals, and memory allocation. Because our main focus is to reason about feasible syscall states, we differentiate `syscall` statements that invoke a system call from regular `call` statements that invoke other procedures. Note that we omit pointers and structs because they can be modeled using arrays [Gurfinkel and Navas 2017; Rakamarić and Hu 2009; Venet 2004]. Similarly, we also omit loops because they can be emulated using tail-recursion. Furthermore, to simplify presentation, we assume that functions are side-effect free: note that many existing static analysis infrastructures [Gurfinkel et al. 2015; Rakamarić and Emmi 2014] transform programs to such a form using a pre-processing step.

4.2 The ArrayVal Abstract Domain

As mentioned in Section 2, our static analysis is based on the reduced product [Cousot et al. 2011] of the disjunctive interval [Sankaranarayanan et al. 2006] and array expansion domains [Feautrier 1988]. In the remainder of this section, we refer to this abstract domain as ArrayVal, which uses array smashing [Blanchet et al. 2002] for unbounded arrays and precise, per-element reasoning for fixed-length arrays. However, because this domain doesn't have a standard implementation, we define both the abstract domain and its transformers in more detail.

Definition 4.2. **(ArrayVal Domain)** Abstract values in the ArrayVal domain include $\bot$, $\top$, and tuples (A, F) where $A = \langle v_1, \cdot\cdot, v_n \rangle$ is an array of disjunctive intervals v_i, and F keeps track of whether the array is smashed or expanded. Specifically, F can take one of three values, namely *collapsed* (C), *expanded* (E), or *scalar* (S).

For instance, consider the abstract value $(\langle [1,2] \vee [4,4] \rangle, C)$ for a variable x. This indicates that x is an array of unknown length where the value of each array element is $1, 2$ or 4. As another example, $(\langle v_1, \cdot\cdot, v_n \rangle, E)$ means that x is an array with n elements where the value of the i-th element belongs to interval v_i. As a final example, the value $(\langle [1, \infty] \rangle, S)$ for x indicates that x is a positive integer.

In order to describe our analysis, we lift the standard numeric operators from the disjunctive interval domain to our ArrayVal domain. In particular, let op_N and lop_N respectively denote the arithmetic and logical operators for the disjunctive interval domain. Then, the arithmetic operator op_A over α_1, α_2 is defined as $(\langle v_1 \rangle, S)\, op_A\, (\langle v_2 \rangle, S) = (\langle v_1\, op_N\, v_2 \rangle, S)$, and the logical operator lop_A is defined as $(\langle v_1 \rangle, S)\, lop_A\, (\langle v_2 \rangle, S) = (\langle v_1\, lop_N\, v_2 \rangle, S)$. Furthermore, as standard in abstract interpretation, we also define the join and widening operators:

Definition 4.3. Let $\sqcup_N$ and ∇_N denote the join and widening operators for the disjunctive interval domain, respectively. Then the join and widening operators, $\sqcup_A$, ∇_A, for the ArrayVal domain, are defined as follows:

- $(\langle v_1 \rangle, C) \otimes_A (\langle v_2 \rangle, C) = (\langle v_1 \otimes_N v_2 \rangle, C)$
- $(\vec{v}, E) \otimes_A (\vec{v}', E) = (\langle v_1 \otimes_N v_1', \cdot\cdot, v_n \otimes_N v_n' \rangle, E)$, $|\vec{v}| = |\vec{v}'|$
- $(\vec{v}, E) \otimes_A (\vec{v}', E) = (\langle v_1 \otimes_N \cdot\cdot \otimes_N v_n \otimes_N v_1' \otimes \cdot\cdot \otimes_N v_m' \rangle, C)$ where $|\vec{v}| = n$, $|\vec{v}'| = m$, and $n \neq m$
- $(\langle v_1, \cdot\cdot, v_n \rangle, E) \otimes_A (\langle v \rangle, C) = ([v_1 \otimes_N \cdot\cdot \otimes_N v_n \otimes_N v], C)$
- $(\langle v_1 \rangle, S) \otimes_A (\langle v_2 \rangle, S) = (\langle v_1 \otimes_N v_2 \rangle, S)$

where $\otimes_A \in \{\sqcup_A, \nabla_A\}$ and $\otimes_N \in \{\sqcup_N, \nabla_N\}$.

Example 4.4. If $\alpha_1 = (\langle [1,2], [3,4] \rangle, E)$ and $\alpha_2 = (\langle [6,7] \rangle, C)$ then $\alpha_1 \sqcup_A \alpha_2 = (\langle [1,4] \vee [6,7] \rangle, C)$

We write $a[j]$ to denote the disjunctive interval at index j of a. If a is collapsed, then $a[j]$ is just a singleton value. If it is expanded but j is out-of-bounds, then $a[j]$ is $\bot$. We also use the notation $b = a\langle j \lhd \alpha \rangle$ to indicate that b is identical to a except that index j has been updated to α. Finally, given an abstract value v for variable x, we assume the existence of a function $Enc(x, v)$ that gives a logical encoding of the possible values of x.

4.3 Intraprocedural Transformers

In this section, we describe our *intraprocedural* rules shown in Figure 3 using judgments of the form $\Gamma, \Upsilon \vdash S \rightsquigarrow \Gamma', \Upsilon'$. Here S is a statement, $\Gamma : V \rightarrow A^{\#}$ is an abstract store mapping variables to abstract values, Υ is a syscall invariant, γ refers to the concretization function for *ArrayVal* domain, and Γ' and Υ' denote the new abstract store and syscall invariant after analyzing statement S. When describing our analysis, we assume the existence of a pointer analysis oracle that can answer aliasing queries. Specifically, given an expression e, *aliases*(e) returns all program variables v that *may* alias e.

In the remainder of this section we describe our abstract transformers in detail.

Assignments. We use three different abstract transformers to handle different types of assignment statements: *Var Assign*, *Const Assign* and *Arithmetic*. The key idea behind all three is to set the

$$\frac{\Gamma' = \Gamma[x \leftarrow \Gamma(y)]}{\Gamma, \Upsilon \vdash x = y \rightsquigarrow \Gamma', \Upsilon} \ \boxed{\textsc{Var Assign}} \qquad \frac{\Gamma' = \Gamma[x \leftarrow (\langle [c, c] \rangle, S)]}{\Gamma, \Upsilon \vdash x = c \rightsquigarrow \Gamma', \Upsilon} \ \boxed{\textsc{Const Assign}}$$

$$\frac{\begin{array}{c} x : \mathtt{int} \quad y : \mathtt{int} \\ \Gamma' = \Gamma[z \leftarrow \Gamma(x)\ \mathtt{op}_A\ \Gamma(y)] \end{array}}{\Gamma, \Upsilon \vdash z = x\ \mathtt{op}\ y \rightsquigarrow \Gamma', \Upsilon} \ \boxed{\textsc{Arithmetic}}$$

$$\frac{\begin{array}{c} p : \mathtt{ptr} \\ \alpha_p' = \bigsqcup_{j \in \gamma(\Gamma(i))} \Gamma(p)\langle j \triangleleft \alpha_x \rangle \end{array}}{\Gamma, \Upsilon \vdash \mathit{update}(p, i, \alpha_x) \rightsquigarrow \Gamma[p \leftarrow \alpha_p'], \Upsilon} \ \boxed{\textsc{Update}}$$

$$\frac{\begin{array}{c} x : \mathtt{int} \quad \{p_1, \cdots, p_n\} = \mathit{aliases}(p) \backslash \{p\} \\ \Gamma_0, \Upsilon \vdash \mathit{update}(p_1, i, \top) \rightsquigarrow \Gamma_1 \\ \cdots \\ \Gamma_0, \Upsilon \vdash \mathit{update}(p_n, i, \top) \rightsquigarrow \Gamma_n \\ \bigsqcup_{i=0}^{n} \Gamma_i, \Upsilon \vdash \mathit{update}(p, i, \top) \rightsquigarrow \Gamma' \end{array}}{\Gamma_0, \Upsilon \vdash p[i] = x \rightsquigarrow \Gamma', \Upsilon} \ \boxed{\textsc{Store (Ptr)}}$$

$$\frac{\begin{array}{c} x : \mathtt{int} \quad \{p_1, \cdots, p_n\} = \mathit{aliases}(p) \backslash \{p\} \\ \Gamma_0, \Upsilon \vdash \mathit{update}(p_1, i, \Gamma(x)) \rightsquigarrow \Gamma_1 \\ \cdots \\ \Gamma_0, \Upsilon \vdash \mathit{update}(p_n, i, \Gamma(x)) \rightsquigarrow \Gamma_n \\ \bigsqcup_{i=0}^{n} \Gamma_i, \Upsilon \vdash \mathit{update}(p, i, \Gamma(x)) \rightsquigarrow \Gamma' \end{array}}{\Gamma_0, \Upsilon \vdash p[i] = x \rightsquigarrow \Gamma', \Upsilon} \ \boxed{\textsc{Store (Int)}}$$

$$\frac{\begin{array}{c} \Phi = \bigwedge_{i=1}^{n} \mathit{Enc}(x_i, \Gamma(x_i))) \\ \Upsilon' = \Upsilon[f \leftarrow \Upsilon(f) \vee \Phi] \\ \Gamma' = \Gamma[r_i \leftarrow \top \mid i \in [1, |\vec{r}|]] \end{array}}{\Gamma, \Upsilon \vdash \vec{r} = \mathit{syscall}(f, x_1, \cdots x_n) \rightsquigarrow \Gamma', \Upsilon'} \ \boxed{\textsc{Syscall}}$$

$$\frac{x : \mathtt{int} \quad \alpha_y = \bigsqcup_{l \in \gamma(\Gamma(x))} (\overbrace{[\top, \cdots, \top]}^{l\text{-times}}, E)}{\Gamma, \Upsilon \vdash y = \mathit{alloc}(x) \rightsquigarrow \Gamma[y \leftarrow \alpha_y], \Upsilon} \ \boxed{\textsc{Alloc}} \qquad \frac{\alpha_x = \bigsqcup_{c \in \gamma(\Gamma(i))} \Gamma(p)[c]}{\Gamma, \Upsilon, \vdash x = p[i] \rightsquigarrow \Gamma[x \leftarrow \alpha_x], \Upsilon} \ \boxed{\textsc{Load}}$$

$$\frac{\begin{array}{c} \Gamma, \Upsilon \vdash S_1 \rightsquigarrow \Gamma_1, \Upsilon_1 \\ \Gamma_1, \Upsilon_1 \vdash S_2 \rightsquigarrow \Gamma_2, \Upsilon_2 \end{array}}{\Gamma, \Upsilon \vdash S_1; S_2 \rightsquigarrow \Gamma_2, \Upsilon_2} \ \boxed{\textsc{Composition}} \qquad \frac{\begin{array}{c} x : \mathit{int} \quad y : \mathit{int} \\ \Gamma' = \Gamma[x \leftarrow \Gamma(y)\ \mathtt{lop}_A\ \Gamma(x)] \end{array}}{\Gamma, \Upsilon \vdash \mathit{assume}(x\ \mathtt{lop}\ y) \rightsquigarrow, \Gamma', \Upsilon',} \ \boxed{\textsc{Assume}}$$

$$\frac{\begin{array}{c} \Gamma, \Upsilon \vdash \mathtt{assume}(C); S_1 \rightsquigarrow \Gamma_1, \Upsilon_1 \\ \Gamma, \Upsilon_1 \vdash \mathtt{assume}(\neg C); S_2 \rightsquigarrow \Gamma_2, \Upsilon_2 \\ \Gamma' = \Gamma_1 \sqcup \Gamma_2 \quad \Upsilon' = \Upsilon_1 \cup \Upsilon_2 \end{array}}{\Gamma, \Upsilon \vdash \mathtt{if}(C)\ S_1\ \mathtt{else}\ S_2 \rightsquigarrow, \Gamma', \Upsilon',} \ \boxed{\textsc{If-Else}}$$

Fig. 3. Intraprocedural transformers. We write $\Gamma_1 \sqcup \Gamma_2$ to denote $[x \leftarrow \Gamma_1(x) \sqcup_A \Gamma_2(x) \mid x \in \mathit{dom}(\Gamma_1) \cup \mathit{dom}(\Gamma_2))]$

abstract value of the variable on the left hand side to be the same as the abstract value of the expression on the right hand side.

Store operations. In our analysis, we treat array writes differently depending on whether it is a write to an integer or pointer array. In particular, because we leverage a separate pointer analysis to resolve aliasing queries, our analysis does not reason precisely about writes to pointer arrays and considers $p[i]$ to be $\top$ as described in the *Store (Ptr)* rule. On the other hand, since we do want to reason precisely about the contents of integer arrays, the *Store (Int)* rule computes a precise new abstract value for $p[i]$ (and the aliases of p). In particular, we first retrieve all aliases of p and invoke the *update* rule (see below) to obtain the abstract value for each alias p_i of p. However, because the pointer analysis provides *may-alias* information, we then join all the updated abstract states and finally invoke *update* on p to obtain the new abstract store Γ'.

Update operation. The *Update* rule describes how we update the abstract value for variables upon an array store. First, we obtain the concrete values for index i by calling $\gamma(\Gamma(i))$. Each $j \in \gamma(i)$ corresponds to an index in the abstract value that could be updated. Since any of these indexes *may* be updated, we perform a strong update for each index independently and then take the join of each update as our new abstract value.

Load. The *Load* rule first identifies all the concrete values for the index i (i.e., $\gamma(\Gamma(i))$). Each of these values corresponds to an index in the concrete array from which we *may* load. As such, we load the value from each index independently and set abstract value α_x to be the join of the loaded values.

Syscalls. Since the main point of our static analysis is to compute feasible syscall states of the program, the *Syscall* rule looks up the values of each syscall argument in the abstract store and lifts it to a logical formula Φ. In particular, note that Φ is expressed as a *conjunction* of the formulas built for each argument x_i since, for any feasible syscall state $(s, v_1, \cdots, v_n)$ produced at this callsite, each v_i will satisfy $Enc(x_i, \Gamma)$. Therefore, $(x_1 \to v_1, \ldots, x_n \to v_n)$ will be a model for Φ which means Φ over-approximates all feasible syscall states generated at this call site. After computing Φ, we update the syscall invariant for f to include both the new syscall state Φ as well as $\Upsilon(f)$. Finally, because we do not analyze the underlying implementation of syscalls in the operating system, our analysis sets the new value of the return value and each modified memory location to $\top$.

Allocation. The *Alloc* transformer creates the initial abstract value for an array variable y at its allocation site. Specifically, for each possible value l of x (i.e., allocation size), it creates an array of length l (with all elements initialized to $\top$) and sets the value of y in the abstract store to be the join of all of these values.

Assume. The *Assume* transformer generates the abstract value $\Gamma(y)\ \texttt{lop}_A\ \Gamma(x)$ corresponding to the concrete operation $y\ \texttt{lop}\ x$ and updates the abstract store to map x to this new abstract value.

Composition, IfElse. The Composition and If-Else rules recursively apply the other transformers. We handle a composite statement $S_1; S_2$ by first analyzing S_1 to produce a new abstract store Γ_1 and syscall invariant Υ_1. We then use our new store and invariant to analyze S_2. Likewise, we handle an If-Else statement by analyzing the bodies of the `if` and `else` branches independently. However, when analyzing the `if` body we first assume the conditional is true and when we analyze the `else` body we first assume the conditional is false.

4.4 Interprocedural Transformers

To achieve a good balance between precision and scalability, our interprocedural analysis adopts a *top-down summary-based* approach [Mangal et al. 2014] to achieve context-sensitivity. In particular, each summary maps the initial abstraction on function entry to the resulting abstraction after analyzing the function body. Thus, we can reuse summaries in calling contexts where the initial abstract state matches an existing one.

$$\frac{\begin{array}{c} C_f = ctx(\Gamma, x_1, \cdots, x_n) \quad (f, C_f) \in Dom(\Sigma) \\ \Gamma^{'} = \Gamma[r_i \leftarrow \Sigma(f, C_f)[out_i] \mid i \in [1, |\vec{r}|]] \end{array}}{\Sigma, \Gamma, \Upsilon \vdash \vec{r} = \texttt{call}\ f(x_1, \cdots x_n) \rightsquigarrow \Sigma, \Gamma', \Upsilon}$$

$$\boxed{\textsc{Summary Match}}$$

$$\frac{\begin{array}{c} C_f = ctx(\Gamma, x_1, \cdots, x_n) \quad (f, C_f) \notin Dom(\Sigma) \\ MaxSumm(\Sigma, f) \quad C_k = closestMatch(\Sigma, f, C_f) \quad C_k \neq \bot \\ \Gamma^{'} = \Gamma[r_i \leftarrow \Sigma(f, C_k)[out_i] \mid i \in [1, |\vec{r}|]] \end{array}}{\Sigma, \Gamma, \Upsilon \vdash \vec{r} = \texttt{call}\ f(x_1, \cdots x_n) \rightsquigarrow \Sigma, \Gamma', \Upsilon}$$

$$\boxed{\textsc{Max Summaries - Closest Match}}$$

$$\frac{\begin{array}{c} C_f = ctx(\Gamma, x_1, \cdots, x_n) \quad (f, C_f) \notin Dom(\Sigma) \quad MaxSumm(\Sigma, f) \\ closestMatch(\Sigma, f, C_f) = \bot \quad C_k = chooseSummary(\Sigma, f) \\ C' = C_k \nabla C_f \quad \Sigma^{'} = \Sigma[(f, C') \leftarrow \Gamma_\bot] \quad \Gamma^{'} = \Gamma[r_i \leftarrow \bot \mid i \in [1, |\vec{r}|]] \end{array}}{\Sigma, \Gamma, \Upsilon \vdash \vec{r} = \texttt{call}\ f(x_1, \cdots, x_n) \rightsquigarrow \Sigma', \Gamma'}$$

$$\boxed{\textsc{Max Summaries - No Closest Match}}$$

$$\frac{\begin{array}{c} C_f = ctx(\Gamma, x_1, \cdots, x_n) \\ (f, C_f) \notin Dom(\Sigma) \quad \neg MaxSumm(\Sigma, f) \\ \Sigma^{'} = \Sigma[(f, C_f) \leftarrow \Gamma_\bot] \quad \Gamma^{'} = \Gamma[r_i \leftarrow \bot \mid i \in [1, |\vec{r}|]] \end{array}}{\Sigma, \Gamma, \Upsilon \vdash \vec{r} = \texttt{call}\ f(x_1, \cdots, x_n) \rightsquigarrow \Sigma', \Gamma'}$$

$$\boxed{\textsc{New Summary}}$$

$$\frac{\begin{array}{c} S = \{C_i \mid (f, C_i) \in Dom(\Sigma)\} \\ \exists C_k \in S.\ C \sqsubseteq C_k \\ \forall C_j \neq C_k.\ C \sqsubseteq C_j \implies C_j \not\sqsubseteq C_k \end{array}}{\vdash closestMatch(\Sigma, f, C) \rightsquigarrow C_k}$$

$$\boxed{\textsc{ClosestMatch-1}}$$

$$\frac{\begin{array}{c} S = \{C_i \mid (f, C_i) \in Dom(\Sigma)\} \\ \forall C_k \in S. C \not\sqsubseteq C_k \end{array}}{\vdash closestMatch(\Sigma, f, C) \rightsquigarrow \bot}$$

$$\boxed{\textsc{ClosestMatch-2}}$$

$$\frac{C_{new} = \{((in_i, \Gamma(x_i)) \mid i \in [1, n]\}}{\vdash ctx(\Gamma, x_1, \cdots, x_n) \rightsquigarrow C_{new}}$$

$$\boxed{\textsc{Ctx}}$$

Fig. 4. Interprocedural transformers. Given two contexts C_1, C_2 we say $C_1 \sqsubseteq C_2$ if for every $x \in dom(C_1)$, $C_1(x) \sqsubseteq C_2(x)$ $MaxSumm(\Sigma, f)$ returns true if we have created the the maximum number of summaries allowed for f. $\Gamma_\bot$ is an abstract state that maps all variables to $\bot$, and $chooseSummary(\Sigma, f)$ selects an existing summary for f. The notation $C_1 = C_2 \nabla C_3$ means $C_1(x) = C_2(x) \nabla C_3(x)$ for all $x \in dom(C_2) \cap dom(C_3)$

Figure 4 summarizes our inter-procedural transformers, which belong to one of two classes: (a) those that reuse summaries and (b) those that create new summaries. However, because the lattice associated with our abstract domain has infinite height, we cannot create new summaries for every calling context. To guarantee termination, our analysis is parametrized by a value N that controls when we apply widening. The idea is that, if we have already created N summaries for a procedure f, we apply widening to generate a coarser summary that can be reused in more calling contexts.

Our interprocedural transformers are formalized using judgments of the form $\Sigma, \Gamma, \Upsilon \vdash S \rightsquigarrow \Sigma^{'}, \Gamma', \Upsilon$ where Σ is a *summary environment* that maps each tuple (f, C) to abstract states. Here, f denotes a procedure, and C is a context that maps formals to abstract values. The idea is to reuse summary $\Sigma(f, C)$ at f's call sites with matching abstract state C. The first two rules, labeled *Summary Match* and *Max Summaries-Closest Match* allow us to reuse existing summaries and we now describe them in more detail.

Summary Match. Whenever we analyze a callsite, we first generate a context $C_f = ctx(\Gamma, x_1, \cdots, x_n)$ for the site and check whether that context is already in that environment. If it is (i.e., $(f, C_f) \in Dom(\Sigma)$), then we reuse the summary $\Sigma(f, C_f)$ to update the abstract store Γ at the callsite.

Max Summaries - Closest Match. If context C_f is not in the domain, we then check whether we have exceeded the maximum number of summaries for this function by calling $MaxSummaries(\Sigma, f)$. If we have exceeded the threshold, we try to find a summary that over-approximates this context by using a helper rule called *ClosestMatch*, which finds the most precise context C_k over-approximating C_f. If such a summary exists (i.e. $C_k \neq \bot$), we use $\Sigma(f, C_k)$ to update the abstract store at this call site.

The next two rules, *New Summary* and *Max Summaries, No Closest Match*, create "placeholder" summaries (with the output abstraction initialized to $\bot$) when we cannot find a suitable summary for callee f. We describe them below in more detail:

New Summary. If context C_f is not in the environment (i.e., $(f, C_f) \notin Dom(\Sigma)$), and we have not exceeded the maximum number of summaries for f (i.e., $\neg MaxSummaries(\Sigma, f)$) then we (1) create a placeholder summary $\Gamma_\bot$ with all values initialized to $\bot$, (2) update the environment to map (f, C_f) to the placeholder summary i.e. $\Sigma^{'} = \Sigma[(f, C_f) \leftarrow \Gamma_\bot]$ and (3) use the placeholder summary to set the abstract values for the return variables to $\bot$ and proceed with the analysis.

Max Summaries, No Closest Match. If (1) the C_f is not in the environment, (2) we have already generated the maximum number of summaries allowed (i.e., $\neg MaxSummaries(\Sigma, f)$) and (3) there is no summary that over-approximates C_f (i.e., $C_k = \bot$), then we arbitrarily choose an existing summary C_k and generate a new summary $C' = C_k \nabla C_f$. This guarantees that (a) there will be a re-usable summary for f when we analyze this call site next time, and (b) the analysis will eventually terminate.

4.5 Analysis Algorithm

To ensure the correctness of our analysis, we perform a fixed-point computation as shown in Algorithm 1. This algorithm starts by constructing an initial summary Γ_{main} for *main* (line 2). Then, in each iteration of the loop (lines 5–12), we remove a single entry from the worklist (line 6) and call a subroutine named *analyze* (line 7) that produces a new environment Σ' and syscall invariant Υ' using the abstract transformers described earlier. If *analyze* changes Σ (line 9), we then add to the worklist all contexts that need to be re-analyzed, including all summaries that changed as well as all of their dependencies (i.e., immediate callers).

Theorem 4.5 (Soundness). *Let P be the input program and Υ be a set of syscall formulas produced by* Analyze*(P). Then Υ is a syscall invariant for P.*

5 POLICY SYNTHESIS

In this section, we describe our synthesis algorithm for converting a syscall invariant to an optimal policy in a given domain-specific policy language.

```
procedure ANALYZE($P$)
input: A program $P$ in our input language
output: A syscall invariant $\Upsilon$
    $C_{main} = (argc \rightarrow \top, argv \rightarrow \top)$
    $\Upsilon = \{\};\ \ \Sigma = \{(main, C_{main}) \rightarrow \Gamma_\perp\}$
    $worklist := \{(main, C_{main})\}$
    while $worklist \neq \emptyset$ do
        $(f, C_f) :=$ worklist.remove();
        $\Sigma', \Upsilon' = analyze(f_{body}, C_f, \Sigma, \{\})$
        $\Upsilon = \Upsilon \cup \Upsilon'$
        if $\Sigma' \neq \Sigma$ then
            $newSumms = \Sigma' \setminus \Sigma$
            $worklist = worklist \cup newSumms \cup deps(newSumms)$
            $\Sigma = \Sigma'$
    return $\Upsilon$;
```

Algorithm 1. Fixed-point algorithm.

5.1 Optimal Policy Synthesis Problem

Despite differences among syscall sandboxing frameworks, almost all policy languages that we know of (e.g., AppArmor [Murray 2019], SeLinux[McCarty 2004], Seccomp-bpf [Edge 2015], Pledge [Pal 2018]) express a policy as a set of *capabilities* $\{C_1, \cdot\cdot, C_n\}$, where each capability belongs to one of the following two classes:

- A *group* g, which denotes a *pre-defined* set of whitelisted syscall states.
- A *guarded system call*, (f, b), where f is a syscall and b is a *user-defined* predicate over the syscall arguments.

For instance, as discussed in Section 2, a Pledge policy consists of a set of groups and is therefore a more restricted version of this model. On the other hand, the Seccomp-bpf framework does not define any groups, but allows users to write their own predicates over integer-valued arguments. Finally, policy languages of SeLinux and AppArmor allow a combination of groups and custom predicates.

Based on this observation, we assume a *family* of syscall whitelisting DSLs that can be expressed using the meta-grammar of Figure 5. Note that Figure 5 does not restrict the pre-defined set of groups (i.e., DSL constants) or the concrete operators. However, inspired by existing policy DSLs, we restrict atomic predicates to be of the form $x \oplus c$, where x is a syscall argument, c is a constant, and $\oplus$ is a comparator.

Given policy language L and semantics $[\![g]\!]$ [1] for each group in L, we assume the following semantics of the whitelisting framework: For a policy ψ consisting of groups $\mathcal{G}$ and guarded syscalls $\mathcal{S}$, the framework allows invoking a syscall f' with arguments $v_1, \cdot\cdot, v_n$ if one of these conditions holds:

(1) $\exists\, g \in \mathcal{G}.\ I_g \models [\![g]\!]$ where $I_g = [f \mapsto f', x_1 \mapsto v_1, \cdot\cdot, x_n \mapsto v_n]$
(2) $\exists\, (f, b) \in \mathcal{S}.\ I_v \models b$ where $I_v = [x_1 \mapsto v_1, \cdot\cdot, x_n \mapsto v_n]$

In the remainder of this section, we write $\Phi(\psi)$ to represent a formula whose satisfying assignments correspond to all syscall states allowed by policy ψ. Similarly, given syscall invariants Υ, we

[1] Here, $[\![g]\!]$ is a formula whose satisfying assignments correspond to syscall states allowed by group g.

$$\begin{array}{rcl} \psi & := & \{\, C, \cdot\cdot, C \,\} \\ C & := & (f, b) \mid g \\ b & := & \text{false} \mid \text{true} \mid x \oplus c \mid \Box(b, \ldots, b) \end{array}$$

Fig. 5. Meta-grammar of policy languages. Here, f is a system call, g is a group, x is a syscall argument, and c is an integer constant. $\oplus$ denotes a comparison operator (e.g., equality, less than), and $\Box$ denotes an n-ary boolean operator (such as $\neg, \wedge, \vee$).

write $\Phi(\Upsilon)$ to denote the formula:

$$\Phi(\Upsilon) = \bigvee_{(f_i, \mathcal{I}_i) \in \Upsilon} \Big((f = f_i) \wedge \mathcal{I}_i \Big)$$

Definition 5.1. **(Optimal policy synthesis)** Given a policy language L and syscall invariant $\Upsilon = \{(f_1, \mathcal{I}_1), \cdot\cdot, (f_n, \mathcal{I}_n)\}$, the optimal policy synthesis problem is to find a policy $\psi \in L$ such that (1) $\Phi(\Upsilon) \Rightarrow \Phi(\psi)$, and (2) there is no other policy $\psi' \in L$ such that $\Phi(\psi') \not\Leftrightarrow \Phi(\psi)$ and $\Phi(\psi') \Rightarrow \Phi(\psi)$.

We refer to any policy satisfying (1) as a *sound* policy, but only those policies that satisfy both (1) and (2) are *optimal.*

5.2 Synthesis Algorithm

Our algorithm for synthesizing an optimal policy is described in Algorithm 2. The Synthesize procedure takes as input a syscall invariant Υ and policy language L, and returns a policy ψ satisfying the two conditions from Definition 5.1. [2]

At a high level, Algorithm 2 performs top-down search over *policy templates*, which are policies with unknown groups (denoted $?^g$), unknown predicates (denoted $?^b$), or unknown (integer) constants (indicated as $?^c$). Here, a hole $?^g$ can be instantiated with any single group g_i provided by the policy DSL; hole $?^b$ can be instantiated with any guard b in L (up to some size), and a hole $?^c$ can be instantiated with any integer constant c (also up to some bound).

In more detail, Algorithm 2 maintains a worklist W, which initially contains all guarded syscalls of the form $(f_i, ?^b_i)$ as well as k groups $?^g_1, \cdot\cdot, ?^g_k$ for each possible value of k. Thus, templates in the initial worklist (from line 3) can generate any policy allowed by the DSL. The main loop (lines 4-16) expands the worklist W and populates set Π until W becomes empty, at which point Π is guaranteed to contain all optimal policies. During each iteration, we dequeue a template χ (line 5) and consider the following four cases:

No unknowns: If χ has no unknowns (i.e., χ is a concrete policy, line 6) and is sound, we add χ to set Π.

Unknown groups: If χ has at least one hole $?^g$ representing an unknown group, we replace it with a concrete group $g \in \mathcal{G}$ (lines 8-10) if $\chi[g/?^g]$ is not suboptimal. In particular, we only consider group g if it adds new capabilities required by Υ that are not captured by existing capabilities ψ. This is illustrated schematically in Figure 6a.

Unknown predicate: If χ has at least one hole $?^b$ (line 11), we create new policy templates by replacing an unknown predicate in χ with new predicates (lines 12-14). Specifically, lines 12 and 13 add to the worklist all expansions of χ where $?^b$ is concretized using an atomic predicate. On the other hand, line 14 adds all expansions of χ where $?^b$ is replaced with boolean combinations of other (fresh) unknowns.

[2] As standard in synthesis literature [Polozov and Gulwani 2015; Solar-Lezama 2008], we consider policies up to a certain size (by bounding the set of predicates and constants). Thus, the technique described here solves a *bounded* optimal synthesis problem, where the goal is to find an optimal policy within a *finite* search space.

```
1:  procedure SYNTHESIZE($\Upsilon, L$)
    input: a syscall invariant $\Upsilon = \{(f_1, \mathcal{I}_1), \cdot\cdot, (f_n, \mathcal{I}_n)\}$.
    input: a policy DSL $L$ with groups $\mathcal{G}$, a set of comparison operators $\mathbb{O}$, and a set of logical
    operators $\mathbb{B}$
    output: an optimal policy $\psi$.
2:      $\Pi := \emptyset$;  $\chi_0 := \{(f_i, ?^b) \mid (f_i, \mathcal{I}_i) \in \Upsilon\}$;
3:      $W := \{\chi_0\} \cup \{\chi_0 \cup \{?^g_1, \cdot\cdot, ?^g_k\} \mid 0 \le k \le |\mathcal{G}|\}$;
4:      while $W \neq \emptyset$ do
5:          $\chi := W.remove()$;
6:          if $\chi$ has no unknowns then                                  ▷ $\chi$ is a concrete policy.
7:              if $\Phi(\Upsilon) \Rightarrow \Phi(\chi)$ then $\Pi := \Pi \cup \{\chi\}$;
8:          else if $\chi$ has unknown group $?^g$ then
9:              $\psi := \{C_i \in \chi \mid$ capability $C_i$ does not have unknowns$\}$;
10:             $W := W \cup \{\chi[g/?^g] \mid g \in \mathcal{G}, SAT\ (\llbracket g \rrbracket \land \neg\Phi(\psi) \land \Phi(\Upsilon))\}$;
11:         else if $\chi$ has unknown predicate $?^b$ then
12:             $W := W \cup \{\chi[\text{true}/?^b]\} \cup \{\chi[\text{false}/?^b]\}$
13:             $W := W \cup \{\chi[x_i \oplus ?^c/?^b] \mid \oplus \in \mathbb{O},\ x_i$ is a syscall arg$\}$;
14:             $W := W \cup \{\chi[e/?^b] \mid e \in \{\Box(?^b_1, \cdots, ?^b_{n-1}), \Box \in \mathbb{B}\}\}$;
15:         else
16:             $\Pi := \Pi \cup$ SOLVE($\chi, \Upsilon$);
17:     return SelectOptimal($\Pi$);
```

Algorithm 2. Optimal policy synthesis algorithm.

Unknown constant: Finally, if χ contains only unknown constants (line 15), we invoke the Solve procedure (discussed later) to instantiate these unknowns with concrete integers.

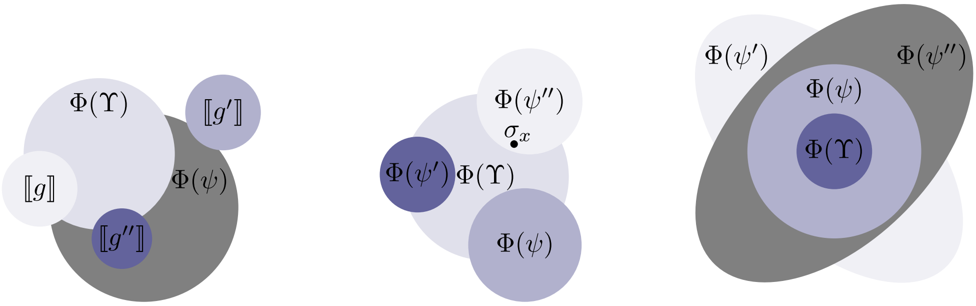
At the end of the main loop, Π is guaranteed to contain *all* optimal policies; thus, line 17 selects one optimal policy (which may not be unique) and returns it as the solution.

5.2.1 Solving Policy Templates. In this section, we describe our Solve procedure for finding completions of a policy template with (only) unknown constants. Given a template χ and a syscall invariant Υ, Solve returns a set of policies Π such that every optimal completion of χ is in Π. From a high-level, this is a typical $\exists\forall$ problem where the goal is to find values of the unknowns that make the following formula valid:

$$\exists ?^c_1, \cdot\cdot, ?^c_n. \forall \vec{x}.\ \Phi(\Upsilon(\vec{x})) \Rightarrow \Phi(\chi(\vec{x}, ?^c_1, \cdot\cdot, ?^c_n))$$

That is, we want an instantiation $c_1, \cdot\cdot, c_n$ of the unknowns $?^c_1, \cdot\cdot, ?^c_n$ such that the resulting policy $\psi[\vec{?^c}\backslash\vec{c}]$ allows all feasible syscall states allowed by Υ (i.e., models of $\Phi(\Upsilon)$).

At a high-level, the Solve algorithm (see Algorithm 3) follows the *counterexample-guided inductive synthesis (CEGIS)* principle and consists of induction and validation steps. Given a set of "test cases" (each of which is a value assignment σ_x for $\vec{x}$), we first induce a candidate solution σ_c (i.e., values of all unknown integers), such that $\Phi(\Upsilon[\sigma_x]) \Rightarrow \Phi(\chi[\sigma_x, \sigma_c])$ holds for each σ_x in the test set. Then, we validate if $\chi[\sigma_c]$ is sound by checking whether $\forall \vec{x}.\ \Phi(\Upsilon) \Rightarrow \Phi(\chi[\sigma_c])$ holds. If $\chi[\sigma_c]$ is not sound, we obtain a *counterexample* σ_x that is added as a new "test case". On the other hand, if $\chi[\sigma_c]$ is sound, we cannot return $\chi[\sigma_c]$ as a solution because we want to find an *optimal* policy. Thus,



(a) Pruning groups (b) Pruning unsound policies with Cegis (c) Pruning suboptimal policies

Fig. 6. In all three diagrams, circles/ellipses represent formulas and points represent interpretations. (a) Illustration of the satisfiability check at line 10 of Algorithm 2. Here, only g passes the satisfiability check. (b) Illustration of line 6 in Algorithm 3. Given unsound policy ψ and test case σ_x, we prune policy ψ' since σ_x is not a model of ψ'. However, we cannot prune the unsound policy ψ'' as σ_x is a model of ψ''. (c) Illustration of how Algorithm 3 prunes suboptimal policies ψ' and ψ'' given a sound policy ψ.

```
1:  procedure SOLVE($\chi, \Upsilon$)
    input: a policy template $\chi$ with only unknown integers.
    input: a syscall invariant $\Upsilon$.
    output: a set $\Pi$ containing all optimal completions of $\chi$.
2:    $\theta := \bigwedge_{?^c \in \mathsf{Unknowns}(\chi)} (\mathtt{MIN} \leq ?^c \leq \mathtt{MAX})$;    $\Pi := \emptyset$;
3:    while $SAT(\theta)$ do
4:      $\psi := \chi[\mathsf{Model}(\theta)]$;
5:      if $\Phi(\Upsilon) \not\Rightarrow \Phi(\psi)$ then                              ▷ $\psi$ is not sound.
6:        $\sigma_x := \mathsf{Model}\big(\Phi(\Upsilon) \land \neg\Phi(\psi)\big)$;
7:        $\theta := \theta \land \Phi\big(\chi[\sigma_x]\big)$;
8:      else                                                             ▷ $\psi$ is sound.
9:        $\Pi := \Pi \cup \{\psi\}$;
10:       $\theta := \theta \land \exists\vec{x}.\big(\Phi(\psi) \land \neg\Phi(\chi)\big)$;
11:   return $\Pi$;
```

Algorithm 3. Algorithm to solve unknown constants.

our SOLVE algorithm exhaustively explores the search space but uses a novel pruning techinuqe that avoids instantiating the template with constants guaranteed to produce suboptimal policies.

In more detail, Algorithm 3 encodes the search space as an SMT formula θ, which initially includes all possible integers (line 2). In each loop iteration (lines 3-10), we sample a model σ_c of θ and check if this model corresponds to a sound policy ψ (line 5). If this is not the case, we find values of σ_x of $\vec{x}$ that prove that ψ is not sound (line 6) and strengthen θ by insisting that any model of θ should satisfy $\chi[\sigma_x]$ (line 7). This is illustrated schematically in Figure 6b.

If ψ is a sound policy, we not only add ψ to Π but also strengthen θ to prune policies that are guaranteed to be suboptimal. In particular, any instantiation σ_c of χ such that $\Phi(\psi) \Rightarrow \Phi(\chi[\sigma_c])$ (i.e., *UNSAT*$(\Phi(\psi) \land \neg\Phi(\chi[\sigma_c]))$ is guaranteed to be sub-optimal. Thus, we only want to search for assignments σ_c such that $\Phi(\psi) \land \neg\Phi(\chi)$ is satisfiable. To restrict our search space to such assignments, we then strengthen θ by conjoining it with $\exists\vec{x}.\ \Phi(\psi) \land \neg\Phi(\chi)$. Figure 6c provides a schematic illustration of this discussion.

Example 5.2. Consider the syscall invariant:

$$\Upsilon = \{(\texttt{pipe}, x_1[0] = 0 \land x_1[1] = 0 \land 50 \leq x_2 \leq 52)\}$$

and policy template $\chi = \{(\texttt{pipe}, x_2 < ?^c\}$. Here, the only optimal policy is obtained by instantiating $?^c$ with 53. Now, we illustrate how the Solve procedure works on this example. Let us assume the integer constant $?^c$ is chosen from $[0, 100]$; thus, Solve starts with the constraint $\theta_0 \equiv 0 \leq ?^c \leq 100$.

Iteration 1: Because θ_0 is satisfiable, we obtain a model of θ_0 where $?^c$ is assigned to 0. This corresponds to the policy $\psi_1 \equiv \{(\texttt{pipe}, x_2 < 0)\}$, which is clearly not sound (i.e., $\Phi(\Upsilon) \Rightarrow \Phi(\psi_1)$ is not valid). Therefore, we enter the branch at line 5 and obtain a model σ_x where $\sigma_x(x_2) = 52$. Then, we obtain a stronger constraint $\theta_1 \equiv 0 \leq ?^c \leq 100 \land 52 < ?^c$ at line 7. Note that this strengthening prunes 53 policies.

Iteration 2: Since θ_1 is still satisfiable, Solve enters the loop again and obtains a model of θ_1 where $?^c$ is assigned 55. This model corresponds to $\psi_2 \equiv \{(\texttt{pipe}, x_2 < 55)\}$, which is a sound policy. Thus, we enter the branch at line 8. We include ψ_2 in the set Π and then strengthen θ_1 according to line 10. In particular, this strengthening gives us a stronger formula $\theta_2 \equiv 0 \leq ?^c \leq 100 \land 52 < ?^c \land \exists x_2 . ?^c \leq x_2 < 55$, which simplifies to $\theta_2 \equiv 0 \leq ?^c \leq 100 \land 52 < ?^c \land ?^c \leq 54$. This constraint further eliminates 46 suboptimal policies weaker than ψ_2 from the search space.

Iteration 3: Since θ_2 is still satisfiable, we obtain another model of θ_2, for instance, one that assigns 53 to $?^c$. This gives us a policy $\psi_3 \equiv \{(\texttt{pipe}, x_2 < 53)\}$, which is again sound. We first add ψ_3 into Π (now $\Pi \equiv \{\psi_2, \psi_3\}$) and then perform optimality-guided pruning, which gives us a stronger formula $\theta_3 \equiv 0 \leq ?^c \leq 100 \land 52 < ?^c \leq 54 \land \exists x_2 . ?^c \leq x_2 < 53$.

Termination: At this point, θ_3 is no longer satisfiable, so the algorithm terminates. Thus, Solve returns two policies, namely, $\{(\texttt{pipe}, x_2 < 55)\}$ and $\{(\texttt{pipe}, x_2 < 53)\}$. Both of these policies are sound, and the second one is optimal.

Theorem 5.3. *If Synthesize(Υ) returns a policy ψ, then ψ is sound and optimal with respect to Υ.*

6 IMPLEMENTATION

We have implemented our proposed approach in a tool called Abhaya which takes as input an application's source code and a target policy language, and returns an optimal policy.

Tools. Abhaya is implemented as an LLVM pass [Lattner and Adve 2004] and leverages a number of existing tools. First, Abhaya's syscall analysis is built on top of the Crab framework [Gange et al. 2016], a popular abstract interpretation engine for analyzing LLVM bitcode. We extend Crab to fully support the ArrayVal domain as well as our summary-based interprocedural analysis. To resolve virtual calls and aliasing queries in the syscall analysis, we use LLVM-DSA [Lattner et al. 2007], a highly scalable, summary-based, and flow-insensitive pointer analysis. Finally, Abhaya also makes extensive use of the Z3 SMT solver [De Moura and Bjørner 2008].

Tunable parameters. Our implementation has two important tunable parameters. For the analysis phase, we used a widening threshold of 2 for the number of summaries. For the synthesis phase, we limit the number of atomic predicates in a syscall guard to 3. We found these hyper-parameters to achieve a good trade-off between precision and efficiency.

Synthesis optimizations. Abhaya incorporates a few optimizations to improve synthesis efficiency. For example, one optimization is to prune policy templates based on *policy template weakening*. Specifically, given a policy template χ, we construct a concrete policy ψ that is *weaker* than any completion of χ. If ψ is not sound, χ can be safely discarded.

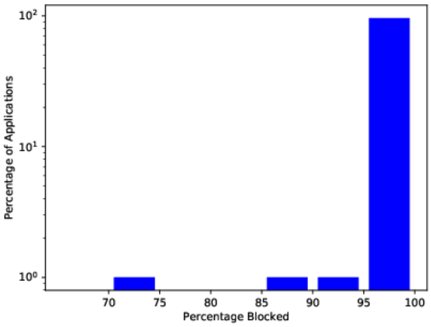


Fig. 7. % of apps vs. percentile of exploits blocked.

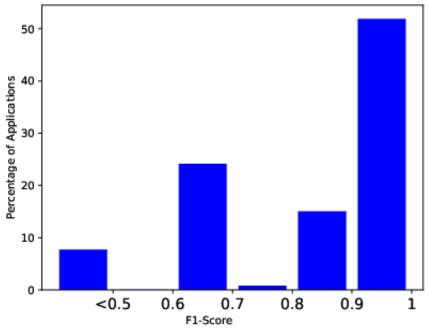


Fig. 8. F1 score of synthesized Pledge policies.

7 EVALUATION

In this section, we present our evaluation results. Specifically, we aim to answer the following research questions:

RQ1. Can Abhaya's policies block real-world exploits? (Section 7.1)
RQ2. How do Abhaya's policies compare with manual ones? (Section 7.2)
RQ3. How long does Abhaya take to generate policies? (Section 7.3)
RQ4. How important are Abhaya's design choices (i.e., abstract domain in program analysis phase and pruning strategy in program synthesis phase)? (Sections 7.4, 7.5)

Benchmarks. To answer these questions, we collect two sets of C/C++ applications running on Linux and OpenBSD. For Linux, we search the top 1000 most popular Debian packages according to the Debian Popularity Contest [deb 2011]. 146 of these 1000 packages are for C/C++ applications while the rest are for standalone libraries or applications in other languages. Among the 146 C/C++ packages, Abhaya cannot analyze 30 as these can only be built with GCC, but our analysis is based on LLVM, which requires Clang. Of the remaining 116 packages, we extract 491 total applications and compile each program, along with their library dependencies, to LLVM. We also link each application against musl-libc (compiled to LLVM) as glibc cannot be compiled with Clang.

For OpenBSD, we collect all C/C++ programs in the OpenBSD base system that come with developer-written policies (183 in total). Since the most common syscall sandboxing frameworks for Linux and OpenBSD are Seccomp-bpf and Pledge respectively, we use Abhaya to generate policies targeting these two environments.

Setup. All experiments are conducted on the Google Compute Engine [gce 2013] using a 16 vcpu instance with 120GB memory running Ubuntu 18.04 and with a time limit of 24 hours.

7.1 Blocking Privilege Escalation Attacks on Linux

In this section, we evaluate whether Abhaya's policies can meaningfully restrict the capabilities of an attacker for a broad class of applications. Our threat model is that an attacker has taken control over the application, and they try to gain full privileges on the system by exploiting vulnerabilities of the underlying OS kernel. The goal of our evaluation is to assess whether Abhaya-generated policies can prevent the attacker from gaining full privileges.

To perform this evaluation, we first collected proof-of-conept (PoC) privilege escalation exploits for the Linux kernel by searching for all Common Vulnerabilities and Exposures (CVEs) reported in the past 5 years[3]. We found 33 such CVEs, 23 of which have published PoCs. Then, we measured how many of these 23 exploits the Abhaya-generated policy can block for each application.

Our main result is that, for *all* 491 Linux programs, the Abhaya-generated policies were able to block *every single one* of the 23 PoC exploits. However, we found that, in many cases, the proof-of-concept exploits are too specific in two ways: (1) they often contain many auxiliary syscalls that make it *more likely* for the exploit to succeed but are not strictly required, and (2) they invoke system calls with specific values even though the kernel vulnerability can be triggered using a broader class of values. Thus, to understand how robust Abhaya-generated policies are, we further generalized each PoC exploit by (a) getting rid of the auxiliary syscalls, and (b) identifying a broader class of syscall argument values that will trigger the kernel vulnerability. Using this methodology, we obtained a more general class of 23 exploits that capture *necessary conditions* for exploiting the kernel vulnerability.

The results of this experiment (for the generalized version of the 23 exploits) are summarized in Figure 7. In particular, Figure 7 shows the percentage of applications for which a given percentile of the exploits are blocked. The take-away message is that for 96% of the applications, Abhaya-generated policies can block over 95% of the exploits. In fact, for 90% of the 491 applications, Abhaya is able to synthesize policies that block *all* 23 exploits.

Outlier analysis. We manually analyzed the cases where our policies permitted the generalized version of the exploits. In 53% of these cases, the application *needs* to use syscalls in the same way that is required for triggering the OS-level vulnerability. On the other hand, in 47% of the cases, there exists a policy that could block the exploit without interfering with the application's functionality; however, Abhaya does not generate the more restrictive policy due to imprecision in the underlying pointer analysis.

Bugs in manually-written Seccomp-bpf policies. Among the 491 Linux applications, six of them actually come with developer-written Seccomp-bpf policies. We manually compared the Abhaya-generated policies against existing policies for these 6 applications and found that our synthesized policies identified 4 developer-confirmed bugs in two of them. In particular, the existing policy for *sshd* unnecessarily allows the `shutdown` syscall while erroneously disallowing `writev`, which is needed when the application is linked against `musl-libc`. Similarly, the manual policy for *file* also *unconditionally* allows `ioctl`, whereas Abhaya only allows it when its second argument is one of two values. Since there are known exploits that invoke `ioctl` with specific values for this argument, Abhaya's more precise policy is preferable. The *file* developer adopted the policy synthesized by Abhaya and the *sshd* developers plan to update their policy.

Result 1: 96% of the Abhaya-sandboxed applications are resistant to >95% of known privilege escalation attacks.

7.2 Comparison with Handcrafted Pledge Policies

In this section, we compare the quality of Abhaya-generated policies with Pledge policies written by developers. To conduct this comparison, we treat developer-written policies as the ground truth and measure similarity between a pair of policies (i.e., Abhaya-generated policy and developer-written policy) using the F1 score, which takes into account both precision and recall. In the context of Pledge, *precision* is the percentage of groups in the synthesized policy that are also in the ground truth. Conversely, *recall* is the percentage of ground truth groups that are also in the synthesized policy. For example, if the ground truth is $\{stdio, tmppath\}$ and the Abhaya-synthesized policy is $\{stdio, rpath, wpath, cpath\}$, then precision would be 0.25 since we include three additional groups

[3] Security vulnerabilities in Linux typically have a lifetime between 3-5 years [Cook 2016] so we restricted our search to the last 5 years. We also focus on Linux here because we were not able to find PoC exploits for OpenBSD.

Application	Manual Policy	Synthesized Policy
calendar	$\{stdio, rpath, tmppath, fattr, getpw, \ldots\}$	$\{stdio, rpath, wpath, cpath, fattr, getpw, \ldots\}$
diff	$\{stdio, rpath, tmppath\}$	$\{stdio, rpath, wpath, cpath\}$
mandoc	$\{stdio, rpath, tmppath, tty, proc, exec\}$	$\{stdio, rpath, wpath, cpath, tty, proc, exec\}$
lndir	$\{stdio, rpath, wpath, cpath\}$	$\{stdio, rpath, cpath\}$
tset	$\{stdio, rpath, wpath, tty\}$	$\{stdio, rpath, tty\}$

Table 3. Synthesized policies with imperfect recall. The omitted groups for the calendar app in both the manual and synthesized policies are *id*, *proc*, *exec*

(namely, *rpath*, *wpath*, *cpath*) and the recall would be 0.5 since *stdio* is included in the synthesized policy but *tmppath* is not. This gives us an F1 score of 0.33.

The result of this experiment is summarized in Figure 8, which shows the percentage of applications for a given F1 score range. As we can see, 63.3% of the synthesized policies *exactly* match the ground truth (with an F1 score of 1), and over 75% of them have an F1 score above 0.8. [4] The average F1 score across all 183 OpenBSD applications is 0.94.

Outlier analysis. For 8.1% of applications, Abhaya's policies have an F1 score less than 0.5. In all of these cases, Abhaya reports there is no sound policy due to imprecision in our static analysis. More specifically, for all these cases, the application computes the first argument of the `socket` syscall by traversing a (dynamically allocated) data structure which our analysis is not able to reason about precisely. As such, our analysis generates an invariant that leaves the first argument of `socket` unconstrained. However, since every Pledge group constrains the first argument of `socket` in some way, Abhaya cannot synthesize a sound policy that over-approximates the invariant.

Cases with imperfect recall. Because our method is sound, we would expect the recall to always be 1 across all cases; however, 5 of the synthesized policies have a recall below 1. Upon manual inspection, we found that Abhaya indeed synthesizes semantically sound policies, but they are syntactically different from the developer-written ones. Table 3 shows the 5 applications where our synthesized policy had imperfect recall. As shown in this table, the developer-written policies for `calendar`, `diff`, and `mandoc` include a group called *tmppath* which is not included in the corresponding Abhaya-generated policies. Instead, Abhaya synthesizes groups *wpath*, *cpath*, *rpath* which collectively cover the syscall states in the *tmppath* group. Therefore, the synthesized policy is still sound in these cases.

On the other, for the remaining two applications (`lndir` and `tset`), the policy synthesized by Abhaya is indeed more restrictive than the manually written policy, which unnecessarily contains the *wpath* group that is *not* required for the application to work correctly. After we reported this issue to the developers, they removed the *wpath* path group from their policy.

Bugs in manually-written Pledge policies. By inspecting the application where the Abhaya-synthesized policy differs from the manually written policy, we were also able to identify a real bug in the existing policy for the `csplit` application. In particular, the synthesized policy includes the *fattr* group, which is missing from the existing policy. As confirmed by developers, the *fattr* group is actually necessary for the application to work correctly, and the manual policy would terminate legitimate executions in which there is a call to the `tmpfile` libc function.

Result 2: Abhaya's policies match manually-written policies with an average F1-score of 0.94 and *exactly* match the manual policy for the majority of the benchmarks.

[4] One may wonder why our synthesized optimal policy would be weaker than ground truth. The reason is due to imprecision from the syscall analysis, i.e., the synthesized policy is an optimal policy that over-approximates the syscall invariant inferred by the static analysis (whereas the ground truth does not over-approximate the invariant).

7.3 Running Time of Abhaya

In terms of running time, Abhaya generates policies in an average time of 8.7 minutes across 674 applications in Linux and OpenBSD. We now provide more detailed statistics about the two phases.

Syscall analysis phase. Across all 674 Linux and OpenBSD applications, Abhaya requires an average *7.3 minutes* (standard deviation: 4.4) to analyze an application with 122,350 lines of LLVM code (LoC) on average. The largest application has 2.1 million LoC and the maximum runtime is 2 hours 30 minutes. Unsurprisingly, analysis time strongly correlates with program size, and Abhaya takes less than 2 minutes to analyze applications with under 35,000 LoC.

Policy synthesis phase. For the 491 Linux applications, it takes Abhaya an average of *2.3 minutes* (SD: 0.3) to synthesize a Seccomp-bpf policy, whereas for the 183 OpenBSD applications, Abhaya takes on average *20.1 seconds* (SD: 0.1) to synthesize a Pledge policy. Synthesizing Pledge policies is six times faster than Seccomp-bpf because Pledge does not allow arbitrary predicates, making the search space smaller. The average size of the synthesized policy for Seccomp-bpf and Pledge is 90.5 and 3.4 AST nodes respectively, whereas the maximum size is 130 and 11 respectively.

Result 3: Abhaya can synthesize policies for applications with >100K LoC within a few minutes on average.

7.4 Ablation Study for Syscall Analysis

We now describe an ablation study to evaluate the design decisions underlying our syscall analysis from Section 4. Specifically, to justify the necessity of using the ArrayVal abstract domain, we compare our analysis against three of its own variants:

- Abhaya-NoVal: This variant only tracks which system calls are invoked but does not track anything about their arguments.
- Abhaya-NoArray: This variant does not reason about contents of arrays, but uses the disjunctive interval domain to reason about integer variables.
- Abhaya-NoDisj: This variant uses the reduced product of the array expansion and interval domains. In other words, it differs from our ArrayVal domain in that it does not use disjunctions.

Impact on Pledge policies. Table 4 summarizes the results from re-running the experiment from Section 7.2 using each of the three variants as well as the full Abhaya system. There are three take-aways from this evaluation:

(1) Abhaya is significantly most precise compared to each of the three variants, In particular, it synthesizes policies that are (on average) nearly 20% more precise than Abhaya-NoDisj and 10x more precise than Abhaya-NoArray. Moreover, Abhaya's policies exactly match the developer policies for 113 of the applications compared to only 53 for Abhaya-NoDisj and 15 for Abhaya-NoArray.
(2) Tracking array contents is crucial for synthesizing precise policies. In particular, Abhaya-NoArray can only synthesize policies for 8% of the applications. This is because many groups ($stdio$, ps, . . .) restrict the array arguments of commonly used syscalls
(3) Using a disjunctive domain is important for increasing precision. In particular, Abhaya's policies are on average 20% more precise than Abhaya-NoDisj.
(4) Tracking array contents introduces a noticeable overhead (4-5x over Abhaya-NoArray) in the analysis; however, it is essential for successfully synthesizing Pledge policies.

Baseline	Exact Match	% Synthesized	Avg. Precision	Avg. Time (sec)	Max Time	Min Time
ABHAYA	116	91.8	0.92	415.5 ± 322.4	5542.5	44.2
ABHAYA-NODISJ	51	91.2	0.76	372.2 ± 301.1	4833.3	25.2
ABHAYA-NOARRAY	15	8.2	0.09	84.4 ± 55.5	332.2	13.1
ABHAYA-NOVAL	0	0.0	0.00	48.4 ± 18.2	78.2	13.1

Table 4. Results from the ablation study on the 183 OpenBSD benchmarks. The Avg. Time column indicates the average run time along with the standard deviation.

Baseline	Blocks All	Blocks >90%	>75%	>50%	Avg. Time (sec)	Max Time	Min Time
ABHAYA	90.2	97.2	100.0	100.0	499.5 ± 367.4	9325.5	32.2
ABHAYA-NODISJ	75.2	85.2	96.3	100.0	401.2 ± 322.1	8842.3	22.2
ABHAYA-NOARRAY	83.3	90.3	98.2	100.0	93.4 ± 64.5	442.2	11.1
ABHAYA-NOVAL	13.3	25.2	35.5	66.2	55.4 ± 22.2	90.2	10.1

Table 5. Results from the ablation study for the syscall analysis on all 491 benchmarks Linux.

Impact on Seccomp-bpf policies. We perform another ablation study by repeating the Seccomp-bpf experiment from Section 7.1. As summarized in Table 5, there are three main takeaways from this experiment:

(1) Using disjunctions is extremely important for generating useful Seccomp-bpf policies. In particular, if we do not use a disjunctive domain, the percentage of policies that block all exploits drops from 90% to 75%.

(2) Even though tracking array contents leads to a significant increase in running time, it is important for the precision of the analysis. In particular, if we do not reason about individual array elements, the percentage of benchmarks for which all exploits can be blocked drops down to 83%.

Result 4: Our ablation study shows the importance of reasoning precisely about the arguments of system calls using the ArrayVal domain.

7.5 Ablation Study for Policy Synthesis

In this section, we perform another ablation study to justify the pruning strategies in our policy synthesis algorithm from Section 5. Towards this goal, we consider the following two variants of our proposed synthesis algorithm:

(1) ABHAYA-NOGEN replaces lines 6-7 in Algorithm 3 by blocking the current assignment of constants. Intuitively, this variant blocks a single assignment but does not perform any generalization (CEGIS) based on the current counterexample.

(2) ABHAYA-NOSUBOPT replaces lines 9-10 in Algorithm 3 by blocking the current assignment of constants. Intuitively, this variant does not prune other sound sub-optimal policies.

To perform this ablation study, we repeat the experiment from Section 7.1 using the two variants described above. [5] For this experiment, we set a timeout of two hours for each benchmark.

The results for this experiment are presented in Table 6. Note that ABHAYA is able to synthesize policies for all benchmarks in 2.3 minutes on average whereas ABHAYA-NOSUBOPT and ABHAYA-NOGEN can only solve 6 and 11 benchmarks within the time limit. Even among the benchmarks that ABHAYA-NOGEN and ABHAYA-NOSUBOPT can solve, they take take 20-25x longer than ABHAYA. These results highlight the importance of both pruning strategies used in our synthesis algorithm.

Result 5: This ablation study highlights the importance of the pruning and generalization strategies used by our policy synthesis algorithm from Section 5.

[5] The Pledge experiments from Section 7.2 are not relevant in this case since Pledge policies do not involve constants.

	Policies Synthesized	Avg. Time (m)	Max Time (m)	Timouts
ABHAYA	491	2.3	8.2	0
ABHAYA-NOSUBOPT	6	55.1	110.2	485
ABHAYA-NOGEN	11	40.1	118.3	480

Table 6. Impact of pruning strategies when instantiating policy templates

8 RELATED WORK

In this section, we survey prior work related to this paper.

Syscall analysis. There has been a long line of work in the area of intrusion detection that uses static analysis to detect sequences of syscalls invoked by an application. For example, several papers [Giffin et al. 2004; Lam and Chiueh 2004; Lam et al. 2006; Wagner and Dean 2001], starting with Wagner and Dean [Wagner and Dean 2001], use static analysis to infer the set of syscalls invoked by the program, but they do not reason precisely about syscall arguments. In addition, all of these papers propose custom sandboxing frameworks tailored to their analysis results, whereas our approach targets existing syscall whitelisting frameworks.

There have also been proposals for using dynamic analysis to automate syscall sandboxing [Mutz et al. 2006; Wan et al. 2019; Wan et al. 2017]. While these approaches use testing to collect the set of invoked syscalls and their arguments, they do not automatically synthesize a policy. In addition, they cannot be used to generate sound policies and typically suffer from low code coverage.

Permission Analysis. There has been a long line of work related to permission analysis for Java and Android applications [Bartel et al. 2012, 2014; Felt et al. 2011; Geay et al. 2009; Jamrozik et al. 2016; Koved et al. 2002; Naumovich and Centonze 2004; Pistoia et al. 2005; Pottier et al. 2001]. For example, Koved et al. use lightweight data-flow analysis to determine all possible objects that could be passed to a checkPermission method [Koved et al. 2002], and Felt et al. also use (intra-procedural) data-flow analysis to identify all possible Android API calls invoked by the program [Felt et al. 2011]. Compared to these techniques, our approach addresses a different problem, namely that of automating system call whitelisting. In addition, there are two important technical differences. First, our method performs fine-grained inter-procedural static analysis to precisely reason about values of system calls. As we show experimentally in Section 7.4, this level of precision is crucial for successfully synthesizing system call whitelisting policies. Second, our method decomposes the problem into two separate syscall analysis and policy synthesis phases. Such a decomposition makes it much easier to re-target our approach to other policy enforcement frameworks. As we demonstrate experimentally, our uniform approach can be used to synthesize both Pledge and Seccomp-bpf policies.

CEGIS. In the CEGIS paradigm[Alur et al. 2015; Solar-Lezama et al. 2007, 2006], an inductive synthesizer generates a candidate program P that is consistent with a set of examples, and the verifier checks the correctness of P or provides counterexamples. Our technique for solving policy templates can be viewed as an instance of the CEGIS paradigm; however, it looks for *optimal* solutions.

Optimal Synthesis. There has also been recent work on synthesizing programs such that an objective function is optimized. For example, Bornholt and Torlak intoduce the the concept of a *metasketch* and use the gradient of the cost function to direct the search [Bornholt et al. 2016]. However, this technique is not easily applicable to our domain as they mainly consider optimality with respect to syntax (e.g, number of instructions or branches) rather than optimality in a logical sense.

9 CONCLUSION

We presented a new technique, and its implementation in a tool called Abhaya, to automatically synthesize a syscall sandboxing policy for a given application and policy DSL. Our approach uses abstract interpretation to compute a sound over-approximation of the resources required by the program and generates a policy in the given DSL from the analysis results using a custom program synthesis technique. We used Abhaya to synthesize Pledge and Seccomp-bpf policies for a total of 674 Linux and OpenBSD applications. Our evaluation demonstrates that Abhaya's policies (a) prevent over 95% of known privilege escalation exploits for 96% of the programs, and (b) they match manually written policies in the majority of cases.

ACKNOWLEDGMENTS

We thanks our anonymous reviewers and members of the UToPiA group for their helpful feedback. We also thank Jorge Navas from SRI for his very helpful and prompt assistance with Crab-LLVM. This material is based upon work supported by the National Science Foundation under Grant Nos. CNS-1908304, CCF-1811865, and CNS-1514435, the US Air Force, AFRL/RIKE and DARPA under Contract No. FA8750-20-C-0208, along with gifts from Google, Mozilla, and Qualcomm. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation, US Air Force, AFRL/RIKE, DARPA, Google, Mozilla, and Qualcomm.

REFERENCES

2011. Debian Popularity Contest. https://popcon.debian.org/

2013. Google Cloud Platform (GCP). https://cloud.google.com/.

2013. Issue 329053. https://bugs.chromium.org/p/chromium/issues/detail?id=329053

2015. Issue 546204. https://bugs.chromium.org/p/chromium/issues/detail?id=546204

2016. Issue 662450. https://bugs.chromium.org/p/chromium/issues/detail?id=662450

2017a. Issue 682488. https://bugs.chromium.org/p/chromium/issues/detail?id=682488

2017b. Issue 766245. https://bugs.chromium.org/p/chromium/issues/detail?id=766245

Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo MK Martin, Mukund Raghothaman, Sanjit A Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2015. Syntax-guided synthesis. *Dependable Software Systems Engineering* 40 (2015), 1–25.

A. Bartel, J. Klein, Y. Le Traon, and M. Monperrus. 2012. Automatically securing permission-based software by reducing the attack surface: an application to Android. In *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. 274–277. https://doi.org/10.1145/2351676.2351722

A. Bartel, J. Klein, M. Monperrus, and Y. Le Traon. 2014. Static Analysis for Extracting Permission Checks of a Large Scale Framework: The Challenges and Solutions for Analyzing Android. *IEEE Transactions on Software Engineering* 40, 6 (June 2014), 617–632. https://doi.org/10.1109/TSE.2014.2322867

Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. 2002. The Essence of Computation. Springer-Verlag New York, Inc., New York, NY, USA, Chapter Design and Implementation of a Special-purpose Static Program Analyzer for Safety-critical Real-time Embedded Software, 85–108. http://dl.acm.org/citation.cfm?id=860256.860262

James Bornholt, Emina Torlak, Dan Grossman, and Luis Ceze. 2016. Optimizing Synthesis with Metasketches *(POPL)*. ACM, 775–788.

Kees Cook. 2016. https://outflux.net/blog/archives/2016/10/18/security-bug-lifetime/

Patrick Cousot, Radhia Cousot, and Laurent Mauborgne. 2011. The Reduced Product of Abstract Domains and the Combination of Decision Procedures. In *Proceedings of the 14th International Conference on Foundations of Software Science and Computational Structures: Part of the Joint European Conferences on Theory and Practice of Software* (Saarbr&#252;cken, Germany) *(FOSSACS'11/ETAPS'11)*. Springer-Verlag, Berlin, Heidelberg, 456–472. http://dl.acm.org/citation.cfm?id=1987171.1987210

Manuvir Das, Sorin Lerner, and Mark Seigle. 2002. ESP: Path-sensitive Program Verification in Polynomial Time. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation* (Berlin, Germany) *(PLDI '02)*. ACM, New York, NY, USA, 57–68. https://doi.org/10.1145/512529.512538

Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Budapest, Hungary) *(TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340. http://dl.acm.org/citation.cfm?id=1792734.1792766

Jake Edge. 2015. A seccomp overview. Online: https://lwn.net/Articles/738694/.

P. Feautrier. 1988. Array Expansion. In *Proceedings of the 2Nd International Conference on Supercomputing* (St. Malo, France) *(ICS '88)*. ACM, New York, NY, USA, 429–441. https://doi.org/10.1145/55364.55406

Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. 2011. Android Permissions Demystified. In *Proceedings of the 18th ACM Conference on Computer and Communications Security* (Chicago, Illinois, USA) *(CCS '11)*. Association for Computing Machinery, New York, NY, USA, 627–638. https://doi.org/10.1145/2046707.2046779

Graeme Gange, Jorge A. Navas, Peter Schachte, Harald SOndergaard, and Peter J. Stuckey. 2016. An Abstract Domain of Uninterpreted Functions. In *Proceedings of the 17th International Conference on Verification, Model Checking, and Abstract Interpretation - Volume 9583* (St. Petersburg, FL, USA) *(VMCAI 2016)*. Springer-Verlag New York, Inc., New York, NY, USA, 85–103. https://doi.org/10.1007/978-3-662-49122-5_4

E. Geay, M. Pistoia, Takaaki Tateishi, B. G. Ryder, and J. Dolby. 2009. Modular string-sensitive permission analysis with demand-driven precision. In *2009 IEEE 31st International Conference on Software Engineering*. 177–187. https://doi.org/10.1109/ICSE.2009.5070519

Jonathon T. Giffin, Somesh Jha, and Barton P. Miller. 2004. Efficient Context-Sensitive Intrusion Detection. In *Proceedings of NDSS 2004*, Mike Reiter and Dan Boneh (Eds.).

Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A. Navas. 2015. The SeaHorn Verification Framework. In *Computer Aided Verification*, Daniel Kroening and Corina S. Păsăreanu (Eds.). Springer International Publishing, Cham, 343–361.

Arie Gurfinkel and Jorge Navas. 2017. A Context-Sensitive Memory Model for Verification of C/C++ Programs. 148–168. https://doi.org/10.1007/978-3-319-66706-5_8

Konrad Jamrozik, Philipp von Styp-Rekowsky, and Andreas Zeller. 2016. Mining Sandboxes. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 37–48. https://doi.org/10.1145/2884781.2884782

Larry Koved, Marco Pistoia, and Aaron Kershenbaum. 2002. Access rights analysis for Java. In *In ACM OOPSLA*.

Maxwell Krohn, Petros Efstathopoulos, Cliff Frey, Frans Kaashoek, Eddie Kohler, David Mazières, Robert Morris, Michelle Osborne, Steve VanDeBogart, and David Ziegler. 2005. Make Least Privilege a Right (Not a Privilege). In *Proceedings of the 10th Conference on Hot Topics in Operating Systems - Volume 10* (Santa Fe, NM) *(HOTOS'05)*. USENIX Association, Berkeley, CA, USA, 21–21. http://dl.acm.org/citation.cfm?id=1251123.1251144

Lap Chung Lam and Tzi-cker Chiueh. 2004. Automatic extraction of accurate application-specific sandboxing policy. In *Proceedings of RAID 2004 (LNCS, Vol. 3224)*, Erland Jonsson and Alfonso Valdes (Eds.). Springer, 1–20.

Lap Chung Lam, Wei Li, and Tzi-cker Chiueh. 2006. Accurate and Automated System Call Policy-Based Intrusion Prevention. In *Proceedings of DSN 2006)*, Lorenzo Alvisi (Ed.). IEEE Computer Society, 413–24.

Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization* (Palo Alto, California) *(CGO '04)*. IEEE Computer Society, Washington, DC, USA, 75–. http://dl.acm.org/citation.cfm?id=977395.977673

Chris Lattner, Andrew Lenharth, and Vikram Adve. 2007. Making Context-sensitive Points-to Analysis with Heap Cloning Practical for the Real World. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Diego, California, USA) *(PLDI '07)*. ACM, New York, NY, USA, 278–289. https://doi.org/10.1145/1250734.1250766

Samuel Laurén, Sampsa Rauti, and Ville Leppänen. 2017. A Survey on Application Sandboxing Techniques. In *Proceedings of the 18th International Conference on Computer Systems and Technologies* (Ruse, Bulgaria) *(CompSysTech'17)*. ACM, New York, NY, USA, 141–148. https://doi.org/10.1145/3134302.3134312

Ravi Mangal, Mayur Naik, and Hongseok Yang. 2014. A Correspondence Between Two Approaches to Interprocedural Analysis in the Presence of Join. In *Proceedings of the 23rd European Symposium on Programming Languages and Systems - Volume 8410*. Springer-Verlag New York, Inc., New York, NY, USA, 513–533. https://doi.org/10.1007/978-3-642-54833-8_27

Bill McCarty. 2004. *SELinux: NSA's Open Source Security Enhanced Linux*. O'Reilly Media, Inc.

Alex Murray. 2019. AppArmor. Online: https://wiki.ubuntu.com/AppArmor.

Darren Mutz, Fredrik Valeur, Giovanni Vigna, and Christopher Kruegel. 2006. Anomalous System Call Detection. *ACM Trans. Inf. Syst. Secur.* 9, 1 (Feb. 2006), 61–93. https://doi.org/10.1145/1127345.1127348

Gleb Naumovich and Paolina Centonze. 2004. Static analysis of role-based access control in J2EE applications. *ACM SIGSOFT Software Engineering Notes* 29 (09 2004), 1–10. https://doi.org/10.1145/1022494.1022530

Neeraj Pal. 2018. Pledge: OpenBSDs defensive approach to OS Security.

Marco Pistoia, Robert Flynn, Larry Koved, and Vugranam Sreedhar. 2005. Interprocedural Analysis for Privileged Code Placement and Tainted Variable Detection. *Lecture Notes in Computer Science* 3586, 362–386. https://doi.org/10.1007/11531142_16

Oleksandr Polozov and Sumit Gulwani. 2015. FlashMeta: A Framework for Inductive Program Synthesis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, 107–126.

François Pottier, Christian Skalka, and Scott Smith. 2001. A Systematic Approach to Static Access Control. In *Programming Languages and Systems*, David Sands (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 30–45.

Niels Provos. 2003. Improving Host Security with System Call Policies. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12* (Washington, DC) *(SSYM'03)*. USENIX Association, Berkeley, CA, USA, 18–18. http://dl.acm.org/citation.cfm?id=1251353.1251371

Niels Provos, Markus Friedl, and Peter Honeyman. 2003. Preventing Privilege Escalation. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12* (Washington, DC) *(SSYM'03)*. USENIX Association, Berkeley, CA, USA, 16–16. http://dl.acm.org/citation.cfm?id=1251353.1251369

Zvonimir Rakamarić and Michael Emmi. 2014. SMACK: Decoupling Source Language Details from Verifier Implementations. In *Computer Aided Verification*, Armin Biere and Roderick Bloem (Eds.). Springer International Publishing, Cham, 106–113.

Zvonimir Rakamarić and Alan J. Hu. 2009. A Scalable Memory Model for Low-Level Code. In *Proceedings of the 10th International Conference on Verification, Model Checking, and Abstract Interpretation* (Savannah, GA) *(VMCAI '09)*. Springer-Verlag, Berlin, Heidelberg, 290–304. https://doi.org/10.1007/978-3-540-93900-9_24

J. H. Saltzer and M. D. Schroeder. 1975. The protection of information in computer systems. *Proc. IEEE* 63, 9 (Sep. 1975), 1278–1308. https://doi.org/10.1109/PROC.1975.9939

Sriram Sankaranarayanan, Franjo Ivančić, Ilya Shlyakhter, and Aarti Gupta. 2006. Static Analysis in Disjunctive Numerical Domains. In *Proceedings of the 13th International Conference on Static Analysis* (Seoul, Korea) *(SAS'06)*. Springer-Verlag, Berlin, Heidelberg, 3–17. https://doi.org/10.1007/11823230_2

Steven Smalley. 2002. Configuring the SELinux Policy. *NAI Labs Report* (Feb. 2002), 7–22.

Armando Solar-Lezama. 2008. *Program synthesis by sketching.* Ph.D. Dissertation.

Armando Solar-Lezama, Gilad Arnold, Liviu Tancau, Rastislav Bodik, Vijay Saraswat, and Sanjit Seshia. 2007. Sketching Stencils *(PLDI)*. ACM, 167–178.

Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial Sketching for Finite Programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 404–415.

Arnaud Venet. 2004. A Scalable Nonuniform Pointer Analysis for Embedded Programs. In *Static Analysis*, Roberto Giacobazzi (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 149–164.

David Wagner and Drew Dean. 2001. Intrusion Detection via Static Analysis. In *Proceedings of IEEE Security and Privacy ("Oakland") 2001*, Roger Needham and Martin Abadi (Eds.). IEEE Computer Society, 156–68.

Zhiyuan Wan, David Lo, Xin Xia, and Liang Cai. 2019. Practical and effective sandboxing for Linux containers. *Empirical Software Engineering* (04 Jul 2019). https://doi.org/10.1007/s10664-019-09737-2

Z. Wan, D. Lo, X. Xia, L. Cai, and S. Li. 2017. Mining Sandboxes for Linux Containers. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. 92–102. https://doi.org/10.1109/ICST.2017.16

Michal Zalewski. 2014. PSA: don't run 'strings' on untrusted files (CVE-2014-8485). Online: https://lcamtuf.blogspot.com/2014/10/psa-dont-run-strings-on-untrusted-files.html.

Xin Zhang, Ravi Mangal, Mayur Naik, and Hongseok Yang. 2014. Hybrid Top-down and Bottom-up Interprocedural Analysis. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. ACM, New York, NY, USA, 249–258. https://doi.org/10.1145/2594291.2594328