Computer Sciences Department

Secure Programming as a Parity Game

William R. Harris
Benjamin Farley
Somesh Jha
Thomas Reps

Technical Report #1694

July 2011

Secure Programming as a Parity Game

William R. Harris
University of Wisconsin, Madison
wrharris@cs.wisc.edu

Benjamin Farley
University of Wisconsin, Madison
farleyb@cs.wisc.edu

Somesh Jha
University of Wisconsin, Madison
jha@cs.wisc.edu

Thomas Reps
University of Wisconsin, Madison
reps@cs.wisc.edu

Abstract

Traditionally, reference monitors have been used both to specify a policy of secure behaviors of an application, and to ensure that an application satisfies its specification. However, for recently proposed *privilege-aware systems*, applications satisfy a policy defined over a set of security-sensitive events by invoking explicitly a separate set of primitive operations provided by the system. To date, a programmer who writes an application for such a system both defines and implements a policy using the low-level primitives. While the programmer may have a high-level policy in mind, it is difficult for him to ensure that the application calls the primitives in such a way that it satisfies the policy. It is also difficult for him to ensure that he does not call the primitives in such a way that it cripples the program's original functionality.

In this paper, we formalize the problem of writing programs for privilege-aware systems, and make significant steps toward solving the problem, using an automata-theoretic approach. First, we distinguish between the high-level policies supported by a privilege-aware system and the low-level primitives provided to enforce policies. Second, we define the *policy-weaving problem*, which is to take as input (1) a declarative description of a privilege-aware system, (2) a program that makes no use of the primitives provided by the system, and (3) a high-level policy that describes the program's security and functionality requirements, and produce as output a program that uses the primitives of the privilege-aware system to satisfy the high-level policy. We reduce important subclasses of the weaving problem to finding strategies to *parity games*. Finally, we formalize the Capsicum capability system, demonstrating that the problem of writing secure programs for a practical privilege-aware system can be described as a policy-weaving problem.

1. Introduction

Developing practical programs to run securely on traditional operating systems is currently a nearly impossible task. In part, this is because traditional systems only weakly and implicitly associate a program module (e.g., a process) with the privileges that the module has for interacting with other objects. However, for many security-critical practical applications, the programmer must reason about the privileges of the modules that make up the application, as well as the modules with which the application interacts. Moreover, not all modules may be trusted to behave securely. An example of such an application is a server that receives requests from clients, and then services each request by interacting with a module that is either large and complicated—and thus is difficult to verify—or is loaded dynamically. Another example is a pipeline of applications, run with the privilege of a trusted user, that contains a commonplace, but complex and vulnerable, application. Well-tested and heavily used but complex programs, such as compression programs like `gzip`, still often have errors that may be exploited by an attacker to control a system [32]. Furthermore, many complex programs, such as `tcpdump` [31] or `WUFTPD` [9], contain large amounts of unverified code, but also contain small, trusted modules that must execute with high privilege to interact with, e.g., the file system or network services. Unfortunately, when such programs execute on traditional OS platforms, the untrusted modules often must be allowed to execute with the same high privileges required for the trusted modules.

To resolve this problem, the operating-systems community has developed a variety of *privilege-aware* operating systems that provide primitive operations that an application can invoke to manage the privileges granted to its modules [15, 21, 32, 33]. Such systems allow applications to satisfy security properties that would be difficult to satisfy on traditional OS platforms. The notion of privilege, and the mechanisms used to define privileges, differ from system to system. However, all privilege-aware systems place the burden on application developers to write their programs so that they correctly uses the primitives of the system to ensure secure behavior.

While privilege-aware systems allow programmers to write practical programs that satisfy strong security properties, such systems also present new challenges to the programmer. For example, consider the `tcpdump` program, a simplified version of which is presented as pseudocode in Fig. 1, along with its control-flow graph. For now, ignore the underlined code in Fig. 1. `Tcpdump` takes as input a pattern that describes network packets and an input device, and prints all packets read from the input device that match the pattern. `Tcpdump` performs its task by compiling the packet pattern into a Berkeley Packet Filter (BPF) [25], configuring the input device, and iteratively reading packets from the input device, printing each one that matches the BPF ([32]).

Historically, `tcpdump` has suffered from multiple security vulnerabilities because its module for matching packets against the BPF is complex. An input crafted maliciously by an attacker could

[Copyright notice will appear here once 'preprint' option is removed.]

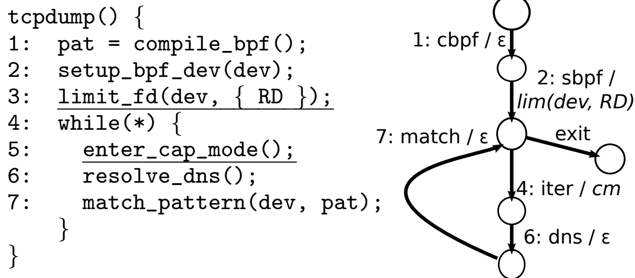

Figure 1. tcpdump instrumented to enforce a policy when run on Capsicum, given both as a pseudocode and as a transducer. In the pseudocode, the underlined statements are calls to Capsicum primitives. The structure of the transducer is based on the CFG of the original program. In particular, if the symbols to the right of each "/" are ignored, then the resulting automaton is the CFG.

compromise the packet-matching module, causing it to execute arbitrary code on behalf of the attacker, e.g., via a buffer-overflow attack [27]. However, `tcpdump` can be modified to run securely, without trusting its packet-matching module, by rewriting it to run on the Capsicum privilege-aware system [32].

For the discussion in this section, Capsicum can be thought of as a system that maintains, for each file descriptor opened by each process, the set of rights (e.g., "read," "write," "stat") that describe how the process may access each descriptor. Capsicum allows each process to call two primitive operations, which allow each process to manage its privileges: (1) for descriptor desc and set of rights R, the process may limit its rights for desc to R by calling `limit_fd`(desc, R); (2) the process may enter *capability mode*, after which it may no longer open descriptors to any other system resources, by calling `enter_cap_mode()`.

The Capsicum developers implemented a secure version of `tcpdump` for Capsicum by instrumenting `tcpdump` with the calls to primitives underlined in Fig. 1. After the instrumented `tcpdump` configures the input device, it limits itself to only be able to read from the input device by calling `limit_fd` (line 3), and before `tcpdump` executes its vulnerable packet-matching code, it disallows itself from opening descriptors to any new resources by calling `enter_cap_mode` (line 5).

While the resulting version of `tcpdump` instrumented for Capsicum is secure, the Capsicum developers later found, through testing, that the version no longer provided some of the core functionality of the original `tcpdump`. In particular, `tcpdump` invokes the `libc` DNS resolver, and the DNS resolver requires access to the file system to function correctly. Even though the resolver accesses the file system, it does not violate a developer's intuitive, high-level notion of security, even if `tcpdump` is compromised and the attacker can choose how to call the resolver. However, `tcpdump` instrumented for Capsicum could, in some safe executions, cause the resolver to fail by invoking `enter_cap_mode` before calling the resolver. Moreover, `tcpdump` could not be otherwise instrumented to call `enter_cap_mode` under different conditions: for each instrumentation of `tcpdump`, there is an execution of `tcpdump` that either calls `enter_cap_mode` before calling the resolver, or does not call `enter_cap_mode` before matching packets, which is insecure.

To resolve the tension between ensuring security during matching packets and preserving the functionality of the DNS resolver, the Capsicum developers leveraged the fact that Capsicum maintains a separate set of rights for each process that it hosts. In particular, the developers partitioned `tcpdump` to execute as two processes: the main process executes most of the `tcpdump` code, but when the main process needs to call the DNS resolver, it instead executes a *remote-procedure call* (RPC) to a separate process, which executes the resolver code. Even if the main `tcpdump` process calls the DNS resolver after calling `enter_cap_mode`, the resolver executes with the privilege to open descriptors for arbitrary files in the file system, and thus its functionality is preserved. After the developers partitioned `tcpdump` and instrumented it in this way, they found no more flaws in the instrumentation through testing, and thus tentatively determined that the new version of `tcpdump` was correct.

The final version of `tcpdump` for Capsicum seems to satisfy significantly stronger security properties than `tcpdump` for a traditional system. However, the Capsicum developer's experience rewriting `tcpdump` for Capsicum illustrates several fundamental problems that currently arise when writing a program for Capsicum, or other privilege-aware systems, such as Decentralized Information Flow Control (DIFC) systems (e.g., HiStar [33]).

First, the low-level primitives provided by a privilege-aware system are powerful mechanisms for enforcing secure behavior, but currently they are also the only mechanism provided for *specifying* secure behavior. As a result, it is difficult even to define what it means for a program rewritten to use the primitives to be "correct," because there is no separate first-class notion of a policy specification. For instance, in DIFC systems, programmers use labels as primitives to ensure the flow of information between system objects. While simple information-flow policies can be specified quite naturally using labels, a label-based system typically cannot express practical policies unless it is extended with capabilities for declassifying information by changing labels. Even if a desired high-level policy were stated, it would not be obvious whether such label manipulations satisfy a high-level policy for the flow of information among objects.

Second, rewriting a program to be secure when executing on a privilege-aware system is often straightforward, but rewriting the program to be secure and to exhibit its required functionality is often non-trivial. To enforce desired policies, programs sometimes must be restructured or repartitioned before they can be instrumented with calls to system primitives. Just as `tcpdump` needed to be partitioned to execute correctly on Capsicum, the Apache webserver needed to be repartitioned before it could execute correctly on the DIFC operating system Flume [18].

Finally, an application programmer must reason about *two* models of his program: if the program satisfies some model of "good behavior" (e.g., a model of `tcpdump` in which the packet-matching code is not compromised), then the program should be functional. But even if the program follows a weaker "bad-behavior" model (e.g., a model of `tcpdump` in which the packet-matching code is compromised, and executes under the attacker's control), then the program must still be secure. This problem is particularly intricate because a compromised program can be used to conduct an API-level attack that invokes the privilege-aware system's primitives to attempt to induce an insecure behavior. Programmers for DIFC systems face this challenge when determining the labels for untrusted processes, and what capabilities each process should have for changing its label.

In this paper, we formalize the problem of instrumenting programs for privilege-aware systems as the *policy-weaving problem*, and present an algorithm for solving important subclasses of the problem. Our algorithm allows an application developer to define policies that specify what security-sensitive events the application should or should not be able to perform, and to obtain an instrumentation of the program that manipulates system primitives so that the policy will be enforced.

For an application developer to apply our algorithm, a system architect first provides to the algorithm a semantics of the privilege-aware system as a state machine. Often, system architects already

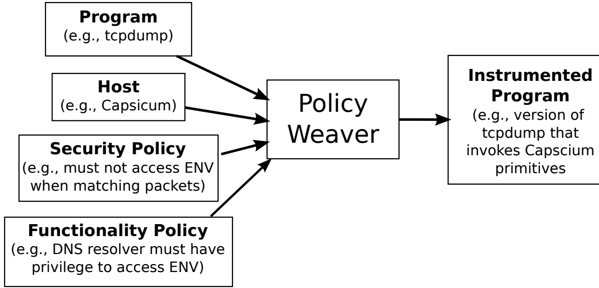

Figure 2. Flowchart of applying the policy-weaving algorithm.

describe their systems semi-formally through the transition relation of a state machine that the system implements [21, 33]. In a state machine for Capsicum, a state is a mapping from each process to the rights that the process holds for each descriptor, plus a Boolean value specifying whether the process is in capability mode. Capsicum defines how the state of the system changes when each application limits its rights, or enters capability mode.

The application developer provides to the algorithm a version of his program that does not manipulate system primitives, a description of when the program may be compromised (e.g., for `tcpdump`, any point after `tcpdump` reaches `match_pattern`), a policy that describes the set of all sequences of security-sensitive events that the program may be allowed to perform (e.g., for `tcpdump`, when an execution calls `match_pattern`, it should only be able to read from the BPF input device and write to STDOUT), and a policy that describes the set of all sequences of security-sensitive events that the program must be allowed to perform (e.g., for `tcpdump`), the DNS resolver should always be able to access the file system).

Given the inputs described above, the goal of our algorithm is to instrument the program to invoke the system primitives so that the instrumented program always satisfies the given policy. To do so, the algorithm reduces the problem of instrumenting the program to the problem of finding a *winning strategy* for an appropriate *two-player parity game* between an attacking player that corresponds to a program, and a defending player that responds to each step of the program's execution by invoking a host primitive. If the game has a winning strategy for the defender, then from the strategy, the algorithm instruments the program to call system primitives in such a way that the program satisfies the policy provided by the application developer. If the `tcpdump` developer provided the version of `tcpdump` partitioned to execute in two process spaces, along with the policy described above, then the algorithm would successfully instrument `tcpdump` to call `limit_fd` and `enter_cap_mode` so that it satisfies the policy, as in Fig. 1.

If the game has no winning strategy for the defender, then the algorithm produces as its final result an attacker strategy that describes program executions that foil any possible instrumentation. The application developer can use such a strategy as guidance for restructuring the program. If a `tcpdump` developer provided the original version of `tcpdump` and the policy described above, our algorithm would provide a strategy that establishes that the original version of `tcpdump` cannot be instrumented to satisfy the given policy, and the developer would use the strategy to partition `tcpdump` to execute in multiple processes.

This paper makes the following contributions. First, we define the policy-weaving problem, which formalizes, in automata-theoretic terms, the important, emerging practical problem of instrumenting a program to run on a privilege-aware system.

Second, we give algorithms for solving several important classes of the policy-weaving problem. In particular, we show that an instance of the problem can be solved soundly using suitable over-approximations of the given program and host. Furthermore, we show that many interesting classes of the policy-weaving problem are equivalent to classes of two-player parity games, a well-studied formalism for synthesizing reactive programs [3, 4].

C	alphabet of program commands
D	alphabet of host primitives
E	alphabet of privilege events
$\mathsf{C} \times \mathsf{D}$	alphabet of turns
$\mathsf{C} \times \mathsf{E}$	alphabet of security-sensitive events
$(\mathsf{C} \times \mathsf{D})^*$	language of all plays
$(\mathsf{C} \times \mathsf{E})^*$	language of all system traces

Table 1. Glossary of policy-weaving alphabets and languages.

Finally, we formalize the Capsicum, HiStar, Asbestos, and Flume privilege-aware systems. The formalization demonstrates that our policy-weaving problem is sufficiently powerful to describe the practical problem of instrumenting programs for a real privilege aware system. We formalize an abstraction of each system useful for weaving algorithms, programs and policies for each system.

Organization §2 uses the running example of `tcpdump` to describe informally the policy-weaving problem and how to solve it. §3 defines the policy-weaving problem and algorithms for solving important subclasses of the problem. §4 formalizes the Capsicum privilege-aware system. §5 discusses related work, and §6 concludes.

2. Overview

In this section, we informally present our policy-weaving algorithm by demonstrating how it instruments `tcpdump` for Capsicum. The algorithm reduces a policy-weaving problem for instrumenting `tcpdump` to finding a winning strategy for a parity game. The program, policies, and host that define a policy-weaving problem are defined as languages over appropriate alphabets. In particular, a program will be treated as a language of executions defined over an alphabet C of program commands. All alphabets and languages used to define a policy weaving problem are summarized in Tab. 1.

Example 1. *Let the alphabet of program commands of* `tcpdump`*,* C_{Tcp} *be the abbreviated commands listed to the left of the "/" symbols on edge labels in* `tcpdump`*'s control-flow graph in Fig. 1. The language* $\mathsf{P}_{Tcp} \subseteq \mathsf{C}^*_{Tcp}$ *of* `tcpdump` *is all sequences of program commands in* C_{Tcp} *on runs of* `tcpdump`*'s CFG.*

Along with a program, a weaving problem also consists of two policies. Each policy is a language of *system traces*, where a system trace is a string over an alphabet of *security events* $\mathsf{C} \times \mathsf{E}$, which is the product of C and an alphabet of *privilege events* E.

This definition of a policy for a program on a privilege-aware host has a richer structure than some policies for programs that execute on traditional systems; the latter are sometimes defined as languages of allowed or disallowed strings of program commands. However, a policy for a program that executes on a privilege-aware host cannot naturally be defined as a language over program commands: a program command only induces a security event if, when the command is executed, the host determines that the program has sufficient privileges to induce the event.

Example 2. *For a program executing on Capsicum, a security event is (i) a program command that attempts to access a resource, paired with (ii) the program's right to access the resource. In language-theoretic terms, the alphabet of security events for* `tcpdump` *on Capsicum is the product of the alphabet of program commands* $\mathsf{C}Tcp$ *(Ex. 1) and an alphabet of privilege events* E_{Tcp}*,*

where each privilege event corresponds to a right that `tcpdump` *has to access a particular file descriptor or the environment. Privilege event* r(desc) *denotes that* `tcpdump` *may access descriptor* desc *with right* r*. The privilege event* ENV *denotes that* `tcpdump` *may open new descriptors to resources in its environment. The privilege event* null*, which is paired with a program command to denote that the command executes, but does not say anything about what privileges the program has when it executes the command.*

The second component of a policy-weaving problem is a *security policy*, which defines the language of all system traces that the instrumented program may induce.

Example 3. *The security policy for* `tcpdump` *discussed in §1 specifies that when* `tcpdump` *matches inputs against a BPF, the instrumented* `tcpdump` *may only induce security events in which* `tcpdump` *may read from the BPF device, but not open descriptors to resources in its environment. For example, in language-theoretic terms, the following system trace over the alphabet of security events* $\mathsf{C}_{Tcp} \times \mathsf{E}_{Tcp}$ *should not be allowed:*

$$(\text{cbpf}, \mathsf{null}), (\text{sbpf}, \mathsf{ENV}), (\text{iter}, \mathsf{ENV}), (\text{dns}, \mathsf{ENV}), (\text{match}, \mathsf{ENV}).$$

In general, the only system traces that should be allowed are those in the regular language:

$$\mathsf{S}_{Tcp} = (((\mathsf{C}_{Tcp} \backslash \{\text{match}\}) \times \mathsf{E}_{Tcp}) \mid (\text{match}, \mathsf{rd}(\mathsf{dev})))^*$$

The third component of a policy-weaving problem is a *functionality policy*, which describes the required functionality of the instrumented program. In language-theoretic terms, the functionality policy is a language over system traces. For each system trace in the functionality policy, if the program executes the sequence of program commands that act as first components of each security event in the trace, then the host must allow the security events in the trace.

Example 4. *§1 describes an informal functionality policy for* `tcpdump`*: the instrumented* `tcpdump` *must have the right to access its environment when setting up the BPF device, it must have the right to read from the BPF device when matching input packets against the filter, and it must have the right to access the environment when executing the DNS resolver.*

For example, in language-theoretic terms, if the program executes the sequence of program commands

$$\text{cbpf sbpf iter dns match}$$

then the host should allow the system trace

$$(\text{cbpf}, \mathsf{null})(\text{sbpf}, \mathsf{ENV})(\text{iter}, \mathsf{ENV})(\text{dns}, \mathsf{ENV})(\text{match}, \mathsf{rd}(\mathsf{dev}))$$

In general, for each system trace in the following regular language F_{Tcp}*, if* `tcpdump` *executes the sequence of program commands that form the first components of security events in the trace, then Capsicum must allow all security events in the trace:*

$$((\mathsf{C}_{Tcp} \times \{\mathsf{null}\}) \mid (\text{sbpf}, \mathsf{ENV}) \mid (\text{dns}, \mathsf{ENV}) \mid (\text{match}, \mathsf{rd}(\mathsf{dev})))^*$$

The fourth component of a policy-weaving is a host language, which relates each execution of a program executing on the host to the system traces induced by the execution. The language is defined over the alphabet $(\mathsf{C} \times \mathsf{D}) \times \mathsf{E}$, where D is an alphabet of *host primitives* D. Let a *turn* be a program command paired with a host primitive (i.e., an letter in the alphabet $\mathsf{C} \times \mathsf{D}$), and let a *play* be a sequence of turns. The language of the host is defined over the alphabet of all turns paired with privilege events $(\mathsf{C} \times \mathsf{D}) \times \mathsf{E}$, and the language describes the system traces allowed by the host, as follows. If $[((c_i, d_i), (e_i)]_i$ with each $c_i \in \mathsf{C}$, $d_i \in \mathsf{D}$, and $e_i \in \mathsf{E}$ is a string in the language of the host, then if a program running on the host executes each program command c_i followed immediately by each host primitive d_i in sequence, then the host allows the system trace $[(c_i, e_i)]$.

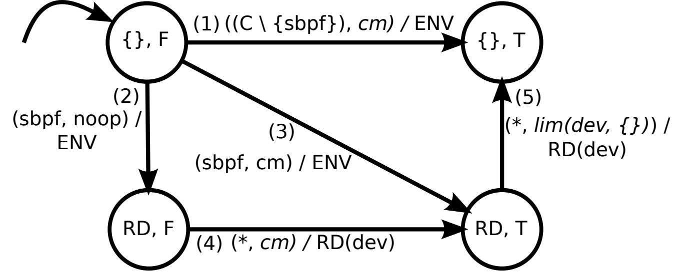

Figure 3. A fragment of the transducer model of the Capsicum-state of `tcpdump`. Each transition is labeled with a program command paired with a host command, followed by a "/", followed by an allowed privilege event.

Example 5. *The Capsicum host provides primitive operations to* `tcpdump` *that allow* `tcpdump` *to limit the rights that it has to access each file descriptor, enter capability mode, or not change its Capsicum state. In language-theoretic terms, the alphabet of Capsicum host primitives* D_{Tcp} *contains, for each descriptor* desc*, and set of rights* R*, a primitive* lim(desc, R) *that limits the rights of* `tcpdump` *for* desc *to* R*, a primitive* cm *that transitions* `tcpdump` *into capability mode, and a primitive* noop *that has no effect.*

Fig. 3 shows a fragment of a finite-state machine H_{Tcp} *that represents the Capsicum state of* `tcpdump`*. The language of the state machine is its set of runs, and the language relates each sequence of turns constructed from* `tcpdump` *program commands* C_{Tcp} *and Capsicum host primitives* D_{Tcp} *to the system traces allowed by Capsicum in response to the sequence of turns.*

Each state in H_{Tcp} *corresponds to a map from each descriptor to the set of rights that* `tcpdump` *has for the descriptor, paired with a Boolean value that denotes whether or not* `tcpdump` *is in capability mode.*

Each transition in Fig. 3 is labeled with an index and a turn in $\mathsf{C}_{Tcp} \times \mathsf{D}_{Tcp}$*, followed by a privilege event in* E_{Tcp}*. For clarity, Fig. 3 does not contain all of the transitions of a complete model of the Capsicum system, but the transitions included in Fig. 3 illustrate the salient points of how Capsicum relates plays to system traces:*

1. *If* `tcpdump` *executes any program command and invokes* cm*, then the resulting Capsicum state is in capability mode (e.g., transitions (1) and (3)).*
2. *If* `tcpdump` *executes the program command* sbpf*, (which opens* dev*) when its Capsicum state is not in capability mode, then in the resulting Capsicum state,* `tcpdump` *has all rights for* dev *(e.g., transitions (2) and (3)). In this section, we model Capsicum as maintaining for each descriptor only a single right "read" (*rd*), but the full implementation of Capsicum supports 63 rights for each descriptor [32].*
3. *If* `tcpdump` *executes a program command from a Capsicum state in which it has the right to read* dev*, then when it executes the command, it has the privilege to read from* dev *(e.g., transitions (4) and (5)).*
4. *If* `tcpdump` *executes any program command, and then invokes the primitive* lim(dev, {})*, then in the resulting Capsicum state,* `tcpdump` *does not have the right to read from* dev *(e.g., transition (5)).*

A host as defined above relates each play issued by an instrumented program to the system traces that the play induces. A program P, security policy S, functionality policy F, and host H define

a *policy-weaving problem*, which is satisfied by an instrumented program P' that, when run on H, enforces S and F. In language-theoretic terms, an instrumented program $\mathsf{P}' \subseteq (\mathsf{C} \times \mathsf{D})^*$ is a language of plays that when composed with H, induce system traces that enforce S and F. We may view P' itself as a composition $\mathsf{P} \circ \mathsf{Q}$ of P and a *solution* Q, a function from strings in C^* to strings in D^* of equal length, where the plays of P' are constructed by pairing each command string of P element-wise with its image under Q. The policy-weaving problem then amounts to finding a Q such that $\mathsf{P} \circ \mathsf{Q}$ enforces S and F when run on H. There is no harm in only searching for instrumented programs of the form $\mathsf{P} \circ \mathsf{Q}$, because one can show that a weaving problem is satisfied by some language of plays if and only if it is satisfied by the language of $\mathsf{P} \circ \mathsf{Q}$ for some Q. Moreover, there are multiple advantages to searching for solutions as functions from program commands to host primitives. First, because the solution is separate from the program that it instruments, a solution to a weaving problem defined over an abstract program directly solves analogous weaving problems defined over refinements of the abstract program (see §3.4.1).

We may view P' itself as a composition $\mathsf{Ins}(\mathsf{P}, \mathsf{Q})$ of P and a *solution* Q, a function from strings in C^* to strings in D^* of equal length, where the plays of P' are constructed by pairing each command string of P element-wise with its image under Q. The policy-weaving problem then amounts to finding a sufficient Q that $\mathsf{Ins}(\mathsf{P}, \mathsf{Q})$ satisfies enforces S and F when run on H. There is no harm in only searching for languages of plays of the form $\mathsf{Ins}(\mathsf{P}, \mathsf{Q})$, because one can show that a weaving problem is satisfied by some language of plays if and only if it is satisfied by the language of $\mathsf{Ins}(\mathsf{P}, \mathsf{Q})$ for some Q. Moreover, there are multiple advantages to searching for solutions as functions from program commands to host primitives. First, because the solution is separate from the program that it instruments, a solution to a weaving problem defined over an abstract program directly solves analogous weaving problems defined over refinements of the abstract program (see §3.4.1).

The second, and more fundamental, reason to view a solution to a weaving problem as a function is that if a program may be approximated by a suitably simple language, such as a regular language, then we may find a solution as a winning strategy to an appropriate two-player parity game. The algorithm that constructs the parity game constructs from the languages of the host and the security policy the language of all plays that, when issued on the host, induce a system trace that violates the security policy (see §3):

Example 6. *To instrument* `tcpdump` *for Capsicum, the weaving algorithm constructs the language of plays that violate the security policy* S_{Tcp} *(Ex. 3) under the model of Capsicum* H_{Tcp} *(Ex. 5). The plays that violate security do not call the host primitive* `enter_cap_mode` *before executing the program command* match*. This language of plays may be represented by the following regular expression over the alphabet of turns* $\mathsf{C}_{Tcp} \times \mathsf{D}_{Tcp}$*, where "." is any character in* $\mathsf{C}_{Tcp} \times \mathsf{D}_{Tcp}$*:*

$$(\mathsf{C}_{Tcp} \times (\mathsf{D}_{Tcp} \backslash \{\mathsf{cm}\}))^* \ (\mathrm{match}, \mathsf{D}_{Tcp}) \ .^*$$

Just as the weaving algorithm constructs from the host and security policy the language of plays that violate security, the algorithm also constructs from the host and functionality policy the language of plays that violate functionality (see §3). Constructing the language of plays that violate functionality is less direct than constructing the language of plays that violate security, so for now we only present the language of functionality-violating plays for `tcpdump` and Capsicum, and leave a precise description of the steps of the construction for §3.

Example 7. *To instrument* `tcpdump` *for Capsicum, the weaving algorithm constructs the language of plays that violate the functionality policy* F_{Tcp} *(Ex. 4) under the model of Capsicum (Ex. 5). Intuitively, a play violates* F_{Tcp} *if it causes* `tcpdump` *to enter capability mode before executing* sbpf *or* dns*, or if it limits* `tcpdump` *not to have the right to read from* desc *when executing* match*. The language of such plays is described by the following regular expression over the alphabet of turns:*

$$.^*(\mathsf{C}_{Tcp} \times \{\mathsf{cm}\}).^*(\{\mathrm{sbpf}, \mathrm{dns}\} \times \mathsf{D}_{Tcp}).^*$$
$$\cup \ .^*(\mathsf{C}_{Tcp} \times \{\mathsf{lim}(\mathsf{dev}, \emptyset)\}).^*(\{\mathrm{match}\} \times \mathsf{D}_{Tcp}).^*$$

where "." denotes any letter in $\mathsf{C} \times \mathsf{D}$*.*

Once the algorithm constructs a language V of all plays that violate security or functionality, it constructs a parity game such that from a winning strategy to the game, the algorithm may construct a solution to the weaving problem. The game is played in turns by an attacker, who chooses a program command on each turn, and a defender who responds to each program command by invoking a host primitive. Thus, each play of the game corresponds to a play in the context of the weaving problem. The attacker wins a play of the game if it corresponds to a play in V; otherwise the play is won by the defender. A defender strategy for a game is a function that maps each sequence of program commands to the sequence of host primitives that the defender invokes in response. Thus, a defender strategy that always wins—i.e., a strategy that never allows a play in V—corresponds directly to a solution to the weaving problem. On the other hand, an attacker strategy that always wins—i.e., a strategy that always produces a play in V in a finite number of turns no matter what host primitives are invoked by the defender—provides a proof that the original weaving problem has no solution.

Example 8. *The weaving algorithm reduces the weaving problem for* `tcpdump` *and Capsicum to the two-player parity game defined by the languages of plays that violate security (Ex. 6) or functionality (Ex. 7). Both languages of violating plays are regular, and thus the parity game resulting from the reduction is a parity game over a finite graph. Such games are well-studied, and known algorithms solve them efficiently [3, 4]. The game has a winning attacker strategy: if the attacker chooses the sequence of program commands*

$$\mathrm{cbpf\ sbpf\ iter\ dns\ match\ iter\ dns}$$

then any response by the defender produces a play that violates security or functionality. To enforce the security policy, the defender must invoke cm *before the attacker executes* match*, but if the defender invokes* cm *and the attacker then executes* dns*, then the resulting play violates functionality.*

Thus, if a `tcpdump` *developer provides* `tcpdump`*, the security and functionality policies of Exs. 3 and 4, and the model of Capsicum from Ex. 5 to the weaving algorithm, the algorithm will fail to instrument* `tcpdump`*. However, the algorithm provides to the developer an attacker strategy that explains why the program cannot be instrumented.*

The developer may be able to use the strategy to understand how to restructure `tcpdump` *so that it may be instrumented with host primitives to enforce his desired policies. Suppose that a* `tcpdump` *developer uses the attacker strategy to manually partition* `tcpdump` *into two processes, where one process executes the bulk of the* `tcpdump` *code, and the other process executes* dns*. The Capsicum state of each process is represented by a distinct state machine analogous to the one in Fig. 3, and the Capsicum state of* `tcpdump` *is represented as the product of the two per-process state machines. If the developer provides the partitioned* `tcpdump`*, security and functionality policies, and the model of Capsicum, then the weaving algorithm instruments the partitioned* `tcpdump` *to enforce the two policies. We do not discuss each step of how the algorithm in-*

struments the partitioned `tcpdump`*, but only note that the weaving problem defined by such a program has a solution that waits for the* `tcpdump` *process to setup the BPF device, limits the* `tcpdump` *process to only have the right to read from the BPF device, and then enters capability mode. The strategy may be represented as the following regular language over the alphabet $\mathsf{C}_{Tcp} \times \mathsf{D}_{Tcp}$:*

$$((\mathrm{sbpf}, \mathsf{lim}(\mathsf{dev}, \{\mathsf{rd}\}))|(\mathrm{iter}, \mathsf{cm})|((\mathsf{C}_{Tcp}\backslash\{\mathrm{sbpf}, \mathrm{iter}\}), \mathsf{noop}))^*$$

In Fig. 1, the full pseudocode and the fully-labeled control-flow graph represent the `tcpdump` *instrumented according to this strategy.*

Note that the simple strategy in Fig. 1 is inefficient in that it invokes the primitive `enter_cap_mode` *on each iteration through the loop, while it need only call* `enter_cap_mode` *once before entering the loop. We leave the problem of optimally instrumenting programs as future work.*

3. The Policy-Weaving Problem

In this section, we define the policy-weaving problem, which is to take an uninstrumented program, a host, and policies, and construct a solution that describes how the program should invoke the primitives of the host to enforce the policies. We solve the policy-weaving problem by reducing it to a parity game. We present a symbolic technique for solving a class of weaving problems partially defined by languages given as symbolic automata. Finally, we motivate and introduce the problem of weaving over partially-trusted programs, and sketch a solution to a class of such problems using a symbolic technique.

3.1 The Policy-Weaving Problem

The policy-weaving problem is a language-theoretic problem defined using a set of operations that build languages and relations over strings from other languages and relations. The problem is defined using the following relational operators. In each definition, let Σ_1, Σ_2, and Σ_3 be arbitrary alphabets of symbols.

Definition 1. Replace: *For $l_i \in \Sigma_1$ and $A_i \subseteq \Sigma_2$, let the* replace *binary relation $\mathsf{Repl}[\{A_i/l_i\}_i] \subseteq \Sigma_1^* \times \Sigma_2^*$ be such that $\mathsf{Repl}[\{A_i/l_i\}_i]$ $([a_1, a_2, \ldots, a_n], [b_1, b_2, \ldots, b_n])$ if and only if for each $a_i = l_j$, $b_i \in A_j$.*

Zip: *For $T \subseteq \Sigma_1^* \times \Sigma_2^*$, let $\mathsf{Zip}[T] \subseteq (\Sigma_1 \times \Sigma_2)^*$ be such that $\mathsf{Zip}[T]([(a_1, b_1), \ldots, (a_n, b_n)]$ if and only if $T([a_1, \ldots, a_n]$, $[b_1, \ldots, b_n])$.*

Unzip: *For $L \subseteq (\Sigma_1 \times \Sigma_2)^*$, let $\mathsf{Unzip}[L] \subseteq \Sigma_1^* \times \Sigma_2^*$ be such that $\mathsf{Unzip}[L]([a_1, \ldots, a_n], [b_1, \ldots, b_n])$ if and only if $L([(a_1, b_1), \ldots, (a_n, b_n)])$.*

Compose: *For $T_1 \subseteq \Sigma_1^* \times \Sigma_2^*$ and $T_2 \subseteq \Sigma_2^* \times \Sigma_3^*$, let the composition $T_1 \circ T_2 \subseteq \Sigma_1^* \times \Sigma_3^*$ be $\{(u, w)|\exists v \in \Sigma_2^* : T_1(u, v) \wedge T_2(v, w)\}$.*

Apply: *For $T \subseteq \Sigma_1^* \times \Sigma_2^*$ and $L \subseteq \Sigma_1^*$, let the* application *$T(L) \subseteq \Sigma_2^*$, be the set $\{\tau \in \Sigma_2^* | \exists \sigma \in \Sigma_1^* : T(\sigma, \tau)\}$.*

Restrict: *For $T \subseteq \Sigma_1^* \times \Sigma_2^*$ and $L \subseteq \Sigma_1^*$, let the* restriction *of T to L, denoted as $T|_L \subseteq \Sigma_1^* \times \Sigma_2^*$, be such that $T|_L([a_1, \ldots, a_n], [b_1, \ldots, b_n])$ if and only if $T([a_1, \ldots, a_n]$, $[b_1, \ldots, b_n])$ and $[a_1, \ldots, a_n] \in L$.*

Project: *Let the* projection *$\pi_{\Sigma_1 \times \Sigma_2, 1} \subseteq (\Sigma_1 \times \Sigma_2)^* \times \Sigma_1^*$ be $\{(\sigma, \tau)|\sigma = [(a_0, b_0), (a_1, b_1), \ldots, (a_n, b_n)] \wedge \tau = [a_0, a_1, \ldots, a_n]\}$.*

The policy-weaving problem is defined as follows.

Definition 2. *For an alphabet of program commands C, alphabet of host primitives D, and alphabet of privilege events E, let the* program *be $\mathsf{P} \subseteq \mathsf{C}^*$, let the* host *be $\mathsf{H} \subseteq ((\mathsf{C} \times \mathsf{D}) \times \mathsf{E})^*$, let the* security policy *be $\mathsf{S} \subseteq (\mathsf{C} \times \mathsf{E})^*$, and let the* functionality policy *be $\mathsf{F} \subseteq (\mathsf{C} \times \mathsf{E})^*$. Let $\mathsf{Q} \subseteq (\mathsf{C} \times \mathsf{D})^*$ be a language of plays, and let $\mathsf{Ins}(\mathsf{P}, \mathsf{Q}) \subseteq \mathsf{C}^* \times (\mathsf{C} \times \mathsf{D})^*$ be a binary relation from each execution from P to the execution instrumented according to Q:*

$$\mathsf{Ins}(\mathsf{P}, \mathsf{Q}) = \{(\sigma, \tau)|\sigma = [c_0, c_1, \ldots, c_n] \in \mathsf{P} \wedge \tau = [(c_0, d_0), (c_1, d_1), \ldots, (c_n, d_n)] \in \mathsf{Q}\}.$$

Let the alphabet of security events *be $\mathsf{C} \times \mathsf{E}$, and let the language of system traces be $(\mathsf{C} \times \mathsf{E})^*$. Let $\mathsf{SysTraces}(\mathsf{P}, \mathsf{Q}, \mathsf{H}) \subseteq (\mathsf{C} \times \mathsf{E})^*$ be the language of system traces induced by running P instrumented by Q on H:*

$$\mathsf{SysTraces}(\mathsf{P}, \mathsf{Q}, \mathsf{H}) = \mathsf{Zip}[\mathsf{Ins}(\mathsf{P}, \mathsf{Q}) \circ \mathsf{Unzip}[\mathsf{H}]]$$

The policy-weaving problem $\mathrm{POLWEAVE}(\mathsf{P}, \mathsf{H}, \mathsf{S}, \mathsf{F})$ *is to find a solution $\mathsf{Q} \subseteq (\mathsf{C} \times \mathsf{D})^*$ that satisfies the following conditions:*

1. Online*: Q is prefix-closed, and for each trace of Q, and each program command, there is a trace of Q that extends the original trace to respond to the program command. Formally, for each $\tau \in \mathsf{Q}$ and $c \in \mathsf{C}$ there is some host command $d \in \mathsf{D}$ such that $\tau \,.\, (c, d) \in \mathsf{Q}$, where "." denotes concatenation.*
2. Secure*: each system trace induced by P instrumented with Q executing on H is allowed by the security policy S:*
$$\mathsf{SysTraces}(\mathsf{P}, \mathsf{Q}, \mathsf{H}) \subseteq \mathsf{S}$$
3. Functional*: if a trace $\sigma \in \mathsf{P}$ is the sequence of program commands of some system trace $\tau \in \mathsf{F}$, then τ must be a system trace induced by P instrumented with Q executing on H:*
$$\mathsf{Zip}[\mathsf{Unzip}[\mathsf{F}]|_{\mathsf{P}}] \subseteq \mathsf{SysTraces}(\mathsf{P}, \mathsf{Q}, \mathsf{H})$$

3.2 Policy Weaving as a Parity Game

In this section, we solve the POLWEAVE problem by reducing it to a parity game. To do so, we first show that any POLWEAVE problem is equivalent to a POLWEAVE problem with a trivial functionality policy (though potentially with a different host), called a *(purely) upper-bounded weaving problem* UBWEAVE. We then reduce UBWEAVE to a parity game; i.e, the reduction is POLWEAVE $\Rightarrow$ UBWEAVE $\Rightarrow$ parity game.

The first step of the reduction is to restate each policy-weaving problem, which is explicitly defined by what privileges a program may and must have as it executes, to a policy-weaving problem that is explicitly defined by what privileges it must and must not have. The reduction proceeds as follows. For alphabet Σ, define the *negative alphabet* $\overline{\Sigma}$ by constructing, for each symbol $a \in \Sigma$, a negative symbol $\overline{a} \in \overline{\Sigma}$ (for negative alphabet $\overline{\Sigma}$, let Σ be the corresponding *positive alphabet*). For the alphabet of privilege events E, alphabet $\overline{\mathsf{E}}$ is an alphabet of negative privilege events, which explicitly denote when a program does not execute with a particular privilege.

For a policy-weaving problem $\mathrm{POLWEAVE}(\mathsf{P}, \mathsf{H}, \mathsf{S}, \mathsf{F})$, the languages H, S, and F are defined solely over alphabets partially constructed from the alphabet of privilege events. To reduce the problem to a problem in UBWEAVE, each language is lifted to a language defined over negative privilege events using the *positive* and *negative closure* of the each language. The positive closure of each language is, intuitively, the largest language that does not allow more positive privileges than the original language.

Definition 3. *For alphabets Σ_1 and Σ_2 and $L \subseteq (\Sigma_1 \times \Sigma_2)^*$, the* positive closure *of L, denoted by $\mathcal{C}_L^+ \subseteq (\Sigma_1 \times (\Sigma_2 \cup \overline{\Sigma_2}))^*$, is the largest language such that if $\mathcal{C}_L^+([(a_0, b_0), (a_1, b_1), \ldots, (a_n, b_n)])$ with $a_i \in \Sigma_1$, $b_i \in \Sigma_2 \cup \overline{\Sigma_2}$, then there is some trace $[(a_0, b_0'), (a_1, b_1'), \ldots, (a_n, b_n)] \in L$ such that for each $b_j \in \Sigma_2$, $b_j' = b_j$. This is a correction of the definition in the submission, which requires only that if $b_n \in \Sigma_2$, then $b_n' = b_n$.*

Lemma 1. *$\mathcal{C}_L^+$ is well-defined, and is equal to the following language:*

$$\mathcal{C}_L^+ = \mathsf{Zip}[\mathsf{Unzip}[L] \circ \mathsf{Repl}[\{\{b_i\} \cup \overline{\Sigma_2}/b_i\}_{b_i \in \Sigma_2}]]$$

Proof. Let $M = \mathsf{Zip}[\mathsf{Unzip}[L] \circ \mathsf{Repl}[\{\{b_i\} \cup \overline{\Sigma_2}/b_i\}_{b_i \in \Sigma_2}]]$. We first show that $M \subseteq \mathcal{C}_L^+$. Let $\sigma = (a_0, b_0), (a_1, b_1), \ldots, (a_n, b_n) \in M$. Then by the definition of Repl, there is some $\tau = (a_0, b_0')$, $(a_1, b_1'), \ldots, (a_n, b_n') \in \tau$, where $b_i' = b_i$ for each $b_i \in \Sigma_2$. Then by the definition of positive closure, $\sigma \in \mathcal{C}_L^+$.

We now show that $\mathcal{C}_L^+ \subseteq M$. Suppose, for a proof by contradiction, that $\sigma = (a_0, b_0), (a_1, b_1), \ldots, (a_n, b_n) \in \mathcal{C}_L^+ \setminus M$. By the definition of $\mathcal{C}_L^+$, there is some $\tau = (a_0, b_0'), (a_1, b_1'), \ldots, (a_n, b_n')$, with $b_i' = b_i$ for each $b_i \in \Sigma_2$. But by the defintion of Repl, $\sigma \in \mathsf{Zip}[\mathsf{Repl}[\{\{b_i\} \cup \overline{\Sigma_2}/b_i\}_{b_i \in \Sigma_2}](\tau)]$. □

For each security policy $\mathsf{S} \subseteq (\mathsf{C} \times \mathsf{E})^*$, the security policy $\mathcal{C}_{\mathsf{S}}^+ \subseteq (\mathsf{C} \times (\mathsf{E} \cup \overline{\mathsf{E}}))^*$ is the largest language that allows no system trace constructed from a program execution with more positive privilege events than some trace in S constructed from the same execution. For host $\mathsf{H} \subseteq ((\mathsf{C} \times \mathsf{D}) \times \mathsf{E})^*$, the host $\mathcal{C}_{\mathsf{H}}^+ \subseteq ((\mathsf{C} \times \mathsf{D}) \times (\mathsf{E} \cup \overline{\mathsf{E}}))^*$ is the largest language that, in response to any given play, allows no trace with more positive privilege events than some trace in H that responds to the same play. It follows that each program and solution that only induce system traces in S on H only induce system traces in $\mathcal{C}_{\mathsf{S}}^+$ on $\mathcal{C}_{\mathsf{H}}^+$, and vice-versa.

The negative closure of a language is, intuitively, the largest language that does not allow a negative privilege when the original language allows the corresponding positive privilege.

Definition 4. *For alphabets Σ_1 and Σ_2 and $L \subseteq (\Sigma_1 \times \Sigma_2)^*$, the* negative closure *of L, denoted by $\mathcal{C}_L^- \subseteq (\Sigma_1 \times (\Sigma_2 \cup \overline{\Sigma_2}))^*$, is the largest language that satisfies the following condition. For trace $\sigma = [(a_0, b_0), (a_1, b_1), \ldots, (a_n, b_n)] \in (\Sigma_1 \times \Sigma_2)^*$ and $\tau = [(a_0, b_0'), (a_1, b_1'), \ldots, (a_n, b_n')] \in (\Sigma_1 \times (\Sigma_2 \cup \overline{\Sigma_2}))^*$, say that σ and τ are* inconsistent *if there is some i for which $b_i' = \overline{b_i}$. Then $\mathcal{C}_L^-$ is the largest language that does not contain any trace that is inconsistent with some trace in L.*

Lemma 2. *$\mathcal{C}_L^-$ is well-defined, and is equal to the following language. Let $\mathsf{Flip} \subseteq \mathsf{E}^* \times (\mathsf{E} \cup \overline{\mathsf{E}})^*$ be the transducer such that for $\sigma = a_0, a_1, \ldots, a_n \in \mathsf{E}^*$ and $\tau = a_0', a_1', \ldots, a_n' \in (\mathsf{E} \cup \overline{\mathsf{E}})^*$, $\mathsf{Flip}(\sigma, \tau)$ if some $a_j' = \overline{a_j}$. Then*

$$\mathcal{C}_L^- = \overline{\mathsf{Zip}[\mathsf{Unzip}[L] \circ \mathsf{Flip}]}$$

Proof. Let $M = \overline{\mathsf{Zip}[\mathsf{Unzip}[L] \circ \mathsf{Flip}]}$. We first show that $M \subseteq \mathcal{C}_L^-$. Suppose, for a proof by contradiction, that there is some trace $\sigma \in M \setminus \mathcal{C}_L^-$. By the definition of $\mathcal{C}_L^-$, σ is inconsistent with some trace $\tau \in L$. It must be the case that $\sigma \in \mathsf{Zip}[\mathsf{Unzip}[L] \circ \mathsf{Flip}]$, by the definition of Flip, but then $\sigma \notin M$, by the definition of M. Thus $M \subseteq \mathcal{C}_L^-$.

We now show that $\mathcal{C}_L^- \subseteq M$. Suppose, for a proof by contradiction, that there is some trace $\sigma \in \mathcal{C}_L^- \setminus M$. Then σ is consistent with every trace in L. Thus $\sigma \notin \mathsf{Zip}[\mathsf{Unzip}[L] \circ \mathsf{Flip}]$ by the definition of Flip, and thus $\sigma \in \overline{\mathsf{Zip}[\mathsf{Unzip}[L] \circ \mathsf{Flip}]} = M$, which is a contradiction. Thus $\mathcal{C}_L^- \subseteq M$. □

For functionality policy $\mathsf{F} \subseteq (\mathsf{C} \times \mathsf{E})^*$, the functionality policy $\mathcal{C}_{\mathsf{F}}^- \subseteq (\mathsf{C} \times (\mathsf{E} \cup \overline{\mathsf{E}}))^*$ is the largest language that forbids traces that are inconsistent with some trace protected by the functionality policy F. For host $\mathsf{H} \subseteq ((\mathsf{C} \times \mathsf{D}) \times \mathsf{E})^*$, the host $\mathcal{C}_{\mathsf{H}}^- \subseteq ((\mathsf{C} \times \mathsf{D}) \times (\mathsf{E} \cup \overline{\mathsf{E}}))^*$ is the largest language that allows no system trace that is inconsistent with a system trace of H. It follows that each program and solution that enforces F on H only induces system traces in $\mathcal{C}_{\mathsf{F}}^-$ on $\mathcal{C}_{\mathsf{H}}^-$, and vice-versa.

Using the closures of a host and policies, we may reduce every policy-weaving problem with non-trivial security and functionality policies to a policy-weaving problem with only a non-trivial security policy.

Lemma 3. $\mathrm{POLWEAVE}(\mathsf{P}, \mathsf{H}, \mathsf{S}, \mathsf{F})$ *has exactly the same solutions as* $\mathrm{POLWEAVE}(\mathsf{P}, \mathcal{C}_{\mathsf{H}}^+ \cap \mathcal{C}_{\mathsf{H}}^-, \mathcal{C}_{\mathsf{S}}^+ \cap \mathcal{C}_{\mathsf{F}}^-, \emptyset)$.

Proof. We first prove that if Q is a solution to $\mathcal{P} = \mathrm{POLWEAVE}(\mathsf{P}, \mathsf{H}, \mathsf{S}, \mathsf{F})$ then Q is a solution of $\mathcal{P}' = \mathrm{POLWEAVE}(\mathsf{P}, \mathcal{C}_{\mathsf{H}}^+ \cap \mathcal{C}_{\mathsf{H}}^-, \mathcal{C}_{\mathsf{S}}^+ \cap \mathcal{C}_{\mathsf{F}}^-, \emptyset)$ by showing that Q satisfies each of the three conditions for a solution to $\mathcal{P}'$ given in Defn. 2. Q is online for $\mathcal{P}'$ because it is online for $\mathcal{P}$, and Q is functional for $\mathcal{P}'$ because the functionality policy for $\mathcal{P}'$ is $\emptyset$. Q satisfies security for $\mathcal{P}'$ if and only if $\mathsf{SysTraces}(\mathsf{P}, \mathsf{Q}, \mathsf{H}) \subseteq \mathcal{C}_{\mathsf{S}}^+ \cap \mathcal{C}_{\mathsf{F}}^-$. We first show that $\mathsf{SysTraces}(\mathsf{P}, \mathsf{Q}, \mathcal{C}_{\mathsf{H}}^+ \cap \mathcal{C}_{\mathsf{H}}^-) \subseteq \mathcal{C}_{\mathsf{S}}^+$. It suffices to show that $\mathsf{SysTraces}(\mathsf{P}, \mathsf{Q}, \mathcal{C}_{\mathsf{H}}^+) \subseteq \mathcal{C}_{\mathsf{S}}^+$, because $\mathsf{SysTraces}(\mathsf{P}, \mathsf{Q}, \mathcal{C}_{\mathsf{H}}^+ \cap \mathcal{C}_{\mathsf{H}}^-) \subseteq \mathsf{SysTraces}(\mathsf{P}, \mathsf{Q}, \mathcal{C}_{\mathsf{H}}^+)$. Note that $\mathsf{SysTraces}(\mathsf{P}, \mathsf{Q}, \mathsf{H}) \subseteq \mathsf{S}$ because Q is a solution to $\mathcal{P}$, and $\mathsf{S} \subseteq \mathcal{C}_{\mathsf{S}}^+$ by definition of positive closure. Thus

$$\mathsf{SysTraces}(\mathsf{P}, \mathsf{Q}, \mathcal{C}_{\mathsf{H}}^+) \cap \mathsf{SysTraces}(\mathsf{P}, \mathsf{Q}, \mathsf{H}) \subseteq \mathsf{S} \subseteq \mathcal{C}_{\mathsf{S}}^+$$

To show that Q satisfies security for $\mathcal{P}'$ it only remains to show that $\mathsf{SysTraces}(\mathsf{P}, \mathsf{Q}, \mathcal{C}_{\mathsf{H}}^+) \setminus \mathsf{SysTraces}(\mathsf{P}, \mathsf{Q}, \mathsf{H}) \subseteq \mathcal{C}_{\mathsf{S}}^+$. First, for traces $\sigma, \tau \in (\mathsf{C} \times \mathsf{E} \cup \overline{\mathsf{E}})^*$, say that τ *positively matches* σ when for each $(c_i, e_i) \in \sigma$, if (c_i', e_i') is the ith element of τ, then $c_i = c_i'$, and if $e_i \in \mathsf{E}$, then $e_i = e_i'$. For $\sigma \in \mathsf{SysTraces}(\mathsf{P}, \mathsf{Q}, \mathcal{C}_{\mathsf{H}}^+) \setminus \mathsf{SysTraces}(\mathsf{P}, \mathsf{Q}, \mathsf{H})$, there is some $\tau \in \mathsf{SysTraces}(\mathsf{P}, \mathsf{Q}, \mathsf{H})$ that positively matches σ, by the definition of $\mathcal{C}_{\mathsf{H}}^+$. Therefore, $\tau \in \mathsf{S}$, because Q satisfies security for $\mathcal{P}$. Therefore, $\sigma \in \mathcal{C}_{\mathsf{S}}^+$, by the definition of $\mathcal{C}_{\mathsf{S}}^+$. Thus $\mathsf{SysTraces}(\mathsf{P}, \mathsf{Q}, \mathcal{C}_{\mathsf{H}}^+) \subseteq \mathcal{C}_{\mathsf{S}}^+$.

We now show that Q satisfies the functionality condition of $\mathcal{P}'$, which holds if and only if $\mathsf{SysTraces}(\mathsf{P}, \mathsf{Q}, \mathcal{C}_{\mathsf{H}}^+ \cap \mathcal{C}_{\mathsf{H}}^-) \subseteq \mathcal{C}_{\mathsf{F}}^+$. First, $\mathsf{Zip}[\mathsf{Unzip}[\mathsf{F}]|_{\mathsf{P}}] \subseteq \mathsf{SysTraces}(\mathsf{P}, \mathsf{Q}, \mathsf{H})$, because Q satisfies the functionality condition of $\mathcal{P}$. It thus holds that $\mathcal{C}^-_{\mathsf{SysTraces}(\mathsf{P},\mathsf{Q},\mathsf{H})} \subseteq \mathcal{C}^-_{\mathsf{Zip}[\mathsf{Unzip}[\mathsf{F}]|_{\mathsf{P}}]}$, by the definition of $\mathcal{C}^-$. Next,

$$\begin{aligned}
\mathcal{C}^-_{\mathsf{SysTraces}(\mathsf{P},\mathsf{Q},\mathsf{H})} &= \overline{\mathsf{Zip}[\mathsf{Unzip}[\mathsf{SysTracesP}, \mathsf{Q}, \mathsf{H}] \circ \mathsf{Flip}]} && (1)\\
&= \overline{\mathsf{Zip}[\mathsf{Ins}(\mathsf{P}, \mathsf{Q}) \circ \mathsf{Unzip}[\mathsf{H}] \circ \mathsf{Flip}]} && (2)\\
&= \mathsf{Zip}[\overline{\mathsf{Ins}(\mathsf{P}, \mathsf{Q}) \circ \mathsf{Unzip}[\mathsf{H}] \circ \mathsf{Flip}}] && (3)\\
&= \mathsf{Zip}[\mathsf{Ins}(\mathsf{P}, \mathsf{Q}) \circ \overline{\mathsf{Unzip}[\mathsf{H}] \circ \mathsf{Flip}}] && (4)\\
&= \mathsf{Zip}[\mathsf{Ins}(\mathsf{P}, \mathsf{Q}) \circ \mathsf{Unzip}[\mathsf{Zip}[\overline{\mathsf{Unzip}[\mathsf{H}] \circ \mathsf{Flip}}]]] && (5)\\
&= \mathsf{SysTraces}(\mathsf{P}, \mathsf{Q}, \mathcal{C}_{\mathsf{H}}^-) && (6)
\end{aligned}$$

Eqn. (1) follows from Lem. 2, Eqn. (2) follows from the definition of $\mathsf{SysTraces}$ (Defn. 2), and Eqn. (3) follows from the definition of Zip (Defn. 1). Eqn. (4) follows from the fact that $\mathsf{Ins}(\mathsf{P}, \mathsf{Q})$ is a total function, Eqn. (5) follows from the definitions of Zip and Unzip (Defn. 1), and Eqn. (6) follows from the definition of $\mathsf{SysTraces}$.

Thus $\mathsf{SysTraces}(\mathsf{P}, \mathsf{Q}, \mathcal{C}_{\mathsf{H}}^-) \subseteq \mathcal{C}^-_{\mathsf{SysTraces}(\mathsf{P},\mathsf{Q},\mathsf{H})} \subseteq \mathcal{C}_{\mathsf{F}}^-$. Furthermore, $\mathsf{SysTraces}(\mathsf{P}, \mathsf{Q}, \mathcal{C}_{\mathsf{H}}^+ \cap \mathcal{C}_{\mathsf{H}}^-) \subseteq \mathsf{SysTraces}(\mathsf{P}, \mathsf{Q}, \mathcal{C}_{\mathsf{H}}^-)$, so $\mathsf{SysTraces}(\mathsf{P}, \mathsf{Q}, \mathcal{C}_{\mathsf{H}}^+ \cap \mathcal{C}_{\mathsf{H}}^-) \subseteq \mathcal{C}_{\mathsf{F}}^-$.

We now show that any solution to $\mathcal{P}'$ is a solution to $\mathcal{P}$. Let Q be a solution to $\mathcal{P}'$. We first show that Q satisfies the security

condition of $\mathcal{P}$. Observe that

$$\begin{aligned}
\mathcal{C}^+_{\mathsf{SysTraces(P,Q,H)}} &= \mathsf{Zip}[\mathsf{Unzip}[\mathsf{Zip}[\mathsf{Ins}(\mathsf{P},\mathsf{Q}) \circ \mathsf{Unzip}[\mathsf{H}]]] \quad (7)\\
&\quad \circ\mathsf{Repl}[\{\{b_i\} \cup \overline{\mathsf{E}}/b_i\}_{b_i \in \mathsf{E}}]]\\
&= \mathsf{Zip}[\mathsf{Ins}(P,Q) \circ \mathsf{Unzip}[\mathsf{H}] \quad (8)\\
&\quad \circ\mathsf{Repl}[\{\{b_i\} \cup \overline{\mathsf{E}}/b_i\}_{b_i \in \mathsf{E}}]]\\
&= \mathsf{Zip}[\mathsf{Ins}(P,Q) \circ \mathsf{Unzip}[\mathsf{Zip}[\mathsf{Unzip}[\mathsf{H}] \quad (9)\\
&\quad \circ\mathsf{Repl}[\{\{b_i\} \cup \overline{\mathsf{E}}/b_i\}_{b_i \in \mathsf{E}}]]]]\\
&= \mathsf{SysTraces}(\mathsf{P},\mathsf{Q},\mathcal{C}^+_{\mathsf{H}}) \quad (10)
\end{aligned}$$

Eqn. (7) follows from Lem. 1, and Eqn. (8) and follows from the definition of Zip and Unzip, as does Eqn. (9). Eqn. (10) follows from the definition of SysTraces (Defn. 2).

$\mathsf{SysTraces}(\mathsf{P},\mathsf{Q},\mathcal{C}^+_{\mathsf{H}} \cap \mathcal{C}^-_{\mathsf{H}}) \subseteq \mathcal{C}^+_{\mathsf{S}}$, because Q is a solution of $\mathcal{P}'$. In fact, $\mathsf{SysTraces}(\mathsf{P},\mathsf{Q},\mathcal{C}^+_{\mathsf{H}}) \subseteq \mathcal{C}^+_{\mathsf{S}}$. To see this, for any language L, let $f(L)$ be the union of all strings in L and all strings inconsistent with some string in L. $f(\mathsf{SysTraces}(\mathsf{P},\mathsf{Q},\mathcal{C}^+_{\mathsf{H}} \cap \mathcal{C}^-_{\mathsf{H}})) = \mathsf{SysTraces}(\mathsf{P},\mathsf{Q},\mathcal{C}^+_{\mathsf{H}})$ by the definition of $\mathcal{C}^-_{\mathsf{H}}$, and $f(\mathcal{C}^+_{\mathsf{S}}) = \mathcal{C}^+_{\mathsf{S}}$ by the definition of $\mathcal{C}^+_{\mathsf{S}}$. Furthermore, f is monotonic over language containment, so

$$\begin{aligned}
\mathsf{SysTraces}(\mathsf{P},\mathsf{Q},\mathcal{C}^+_{\mathsf{H}}) &= f(\mathsf{SysTraces}(\mathsf{P},\mathsf{Q},\mathcal{C}^+_{\mathsf{H}} \cap \mathcal{C}^-_{\mathsf{H}}))\\
&\subseteq f(\mathcal{C}^+_{\mathsf{S}})\\
&= \mathcal{C}^+_{\mathsf{S}}
\end{aligned}$$

Therefore $\mathcal{C}^+_{\mathsf{SysTraces(P,Q,H)}} \subseteq \mathcal{C}^+_{\mathsf{S}}$, by Eqn. (10).

We now how that for alphabets Σ_1, Σ_2 and languages $L, M \subseteq (A \times B)^*$, if $\mathcal{C}^+_L \subseteq \mathcal{C}^-_M$. Let $\sigma \in L$. Then $\sigma \in \mathcal{C}^+_M$, because $L \subseteq \mathcal{C}^+_L \subseteq \mathcal{C}^+_M$. But $\sigma \in (A \times B)$, so $\sigma \in M$, by the definition of $\mathcal{C}^+_M$ (Defn. 3). Thus, in particular, $\mathsf{SysTraces}(\mathsf{P},\mathsf{Q},\mathsf{H}) \subseteq \mathcal{C}^+_{\mathsf{S}}$, and thus Q satisfies the security condition of $\mathcal{P}$.

We now show Q satisfies the functionality condition of $\mathcal{P}$, which is equivalent to showing that $\mathsf{Zip}[\mathsf{Unzip}[\mathsf{F}]|_{\mathsf{P}}] \subseteq \mathsf{SysTraces}(\mathsf{P},\mathsf{Q},\mathsf{H})$. First, $\mathsf{SysTraces}(\mathsf{P},\mathsf{Q},\mathcal{C}^+_{\mathsf{H}} \cap \mathcal{C}^-_{\mathsf{H}}) \subseteq \mathcal{C}^-_{\mathsf{F}}$, because Q is a solution of $\mathcal{P}'$. Thus, $\mathsf{SysTraces}(\mathsf{P},\mathsf{Q},\mathcal{C}^+_{\mathsf{H}} \cap \mathcal{C}^-_{\mathsf{H}}) \subseteq \mathcal{C}^-_{\mathsf{Zip}[\mathsf{Unzip}[F]|_P]}$, by the definition of SysTraces. Thus, $\mathsf{SysTraces}(\mathsf{P},\mathsf{Q},\mathcal{C}^-_{\mathsf{H}}) \subseteq \mathcal{C}^-_{\mathsf{Zip}[\mathsf{Unzip}[F]|_P]}$. To see this, for any language L, let $g(L)$ be the set of all strings in L positively matched by any string in L. $g(\mathsf{SysTraces}(\mathsf{P},\mathsf{Q},\mathcal{C}^+_{\mathsf{H}} \cap \mathcal{C}^-_{\mathsf{H}})) = \mathsf{SysTraces}(\mathsf{P},\mathsf{Q},\mathcal{C}^-_{\mathsf{H}})$, by the definition of $\mathcal{C}^+_{\mathsf{H}}$, and $g(\mathcal{C}^-_{\mathsf{Zip}[\mathsf{Unzip}[F]|_P]}) = \mathcal{C}^-_{\mathsf{Zip}[\mathsf{Unzip}[F]|_P]}$, by the definition of $\mathcal{C}^-_{\mathsf{Zip}[\mathsf{Unzip}[F]|_P]}$. Furthermore, g is monotonic over language inclusion, so

$$\begin{aligned}
\mathsf{SysTraces}(\mathsf{P},\mathsf{Q},\mathcal{C}^-_{\mathsf{H}}) &= g(\mathsf{SysTraces}(\mathsf{P},\mathsf{Q},\mathcal{C}^+_{\mathsf{H}} \cap \mathcal{C}^-_{\mathsf{H}})\\
&\subseteq g(\mathcal{C}^-_{\mathsf{Zip}[\mathsf{Unzip}[F]|_P]})\\
&= \mathcal{C}^-_{\mathsf{Zip}[\mathsf{Unzip}[F]|_P]}
\end{aligned}$$

Therefore, $\mathcal{C}^-_{\mathsf{SysTraces(P,Q,H)}} \subseteq \mathcal{C}^-_{\mathsf{Zip}[\mathsf{Unzip}[F]|_P]}$, by (6).

We now show that for alphabets Σ_1, Σ_2 and languages $L, M \subseteq (\Sigma_1 \times \Sigma_2)^*$, if $\mathcal{C}^-_L \subseteq \mathcal{C}^-_M$, then $M \subseteq L$. Suppose, for a proof by contradiction, that there is a trace $\sigma \in M \setminus L$. Let τ be equal to σ, except that if (a_0, b_0) is the first element of σ, then $(a_0, \overline{b_0})$ is the first element of τ. Then $\tau \notin \mathcal{C}^-_M$, but the only string in $(\Sigma_1 \times \Sigma_2)^*$ that τ is inconsistent with is σ, and $\sigma \notin L$, so $\tau \in \mathcal{C}^-_L$. But this contradicts the assumption that $\mathcal{C}^-_L \subseteq \mathcal{C}^-_M$. In particular, because $\mathcal{C}^-_{\mathsf{SysTraces(P,Q,H)}} \subseteq \mathcal{C}^-_{\mathsf{Zip}[\mathsf{Unzip}[F]|_P]}$, it holds that $\mathsf{Zip}[\mathsf{Unzip}[F]|_P] \subseteq \mathsf{SysTraces}(\mathsf{P},\mathsf{Q},\mathsf{H})$. Thus Q satisfies the functionality condition of $\mathcal{P}$. □

Define the UBWEAVE problem as $\mathrm{UBWEAVE}(\mathsf{P},\mathsf{H},\mathsf{Pol}) = \mathrm{POLWEAVE}(\mathsf{P},\mathsf{H},\mathsf{Pol},\emptyset)$. Lem. 3 shows that to solve POLWEAVE, it suffices to solve UBWEAVE. Solving UBWEAVE reduces to finding a strategy to a two-player parity game, defined as follows.

A solution to $\mathrm{UBWEAVE}(\mathsf{P},\mathsf{H},\mathsf{Pol})$ must not allow any play that corresponds to an execution of P and induces a system trace not in Pol. The language of all plays constructed from executions of P is $\mathsf{ProgPlays}(\mathsf{P}) = \mathsf{Zip}[(\mathsf{C}^* \times \mathsf{D}^*)|_{\mathsf{P}}]$, and the language of all plays that induce a system trace in a language $L \subseteq (\mathsf{C} \times \mathsf{E})^*$ is

$$\mathsf{Plays}(\mathsf{H}, L) = \pi_{(\mathsf{C}\times\mathsf{D})\times\mathsf{E},1}(\mathsf{Zip}[\pi_{\mathsf{C}\times\mathsf{D},1} \circ \mathsf{Unzip}[L]] \cap \mathsf{H})$$

Therefore, a solution to $\mathrm{UBWEAVE}(\mathsf{P},\mathsf{H},\mathsf{Pol})$ may not allow any play in the language of violating plays:

$$\mathsf{Vio}(\mathsf{P},\mathsf{H},\mathsf{Pol}) = \mathsf{ProgPlays}(\mathsf{P}) \cap \mathsf{Plays}(\mathsf{H},\overline{\mathsf{Pol}})$$

We now use the representation of the language of violating plays to reduce POLWEAVE to a parity game. In particular, from this point on, we will restrict our attention to policy-weaving problems defined solely by regular languages. In §3.5.2, we relax this restriction to consider policy-weaving problems defined by input languages that support limited counting and recursion, such as context-free grammars and visibly-pushdown languages [2].

Each POLWEAVE problem defined by regular languages may be reduced to a parity game over a finite graph, defined here similarly to the literature [3].

Definition 5. *A* finite-graph parity game $\mathsf{G} = (\mathsf{S}_0, \mathsf{S}_1, s_0, A, \Sigma_0, \Sigma_1, \rho_0, \rho_1)$ *is defined by:*

1. *A finite set of Player-0 states* S_0
2. *A finite set of Player-1 states* S_1.
3. *An initial state* $s_0 \in \mathsf{S}_0$.
4. *A set of accepting states* $A \subseteq \mathsf{S}_0$.
5. *A Player-0 transition relation* $\rho_0 \subseteq \mathsf{S}_0 \times \Sigma_0 \times \mathsf{S}_1$, *and a Player-1 transition relation* $\rho_1 \subseteq \mathsf{S}_1 \times \Sigma_1 \times \mathsf{S}_0$.

A game play $\sigma = (s^0_0, l^0_0, s^0_1), (s^0_1, l^0_1, s^1_0), \ldots, (s^{n-1}_1, l^{n-1}_1, s^n_0)$, *is a finite run of the game automaton, where for each* $i < n$, $\rho_0(s^i_0, l^i_0, s^i_1)$, *and* $\rho_1(s^i_1, l^i_1, s^{i+1}_0)$. *Player 0* wins σ *if* $s^n_0 \in A$. *Otherwise, Player 1 wins* σ. *The* trace $\tau(\sigma)$ *of a play* σ *is the sequence of pairs of letters* $(l^0_0, l^1_0), (l^0_1, l^1_1), \ldots, (l^{n-1}_0, l^{n-1}_1)$.

Definition 6. *For game* $\mathsf{G} = (\mathsf{S}_0, \mathsf{S}_1, A, s_0, \Sigma_0, \Sigma_1, \rho_0, \rho_1)$, *a* Player-1 strategy $f_1 : \Sigma_0(\Sigma_1 \times \Sigma_0)^* \to \Sigma_1$ *is a function that maps each string of a Player-0 letter followed by Player-1 letters paired with Player-0 letters to a Player-1 letter.* f_1 *is a* winning Player-0 strategy *for* G *if the following hold. Let the set of runs of* f_1 *be all runs of* G *induced by choosing each* Σ_0 *letter in the run according to* f_1*:*

1. *For each l, and s such that* $\rho_0(s_0, l, s)$, (s_0, l, s) *is a run of* f_1.
2. *If* $r = (s^0_0, l^0_0, s^0_1), \ldots, (s^n_0, l^n_0, s^n_1)$ *is a run of* f_1 *with* $s^n_1 \in \mathsf{S}_1$, *and* $\rho_1(s^n_1, f_1(\tau(r)), s')$ *then* $r \,.\, (s^n_1, f_1(\tau(r)), s')$ *is a run of* f_1.
3. *If* $r = (s^0_0, l^0_0, s^0_1), \ldots, (s^{n-1}_1, l^{n-1}_1, s^n_0)$ *is a run of* f_1 *with* $s^n_0 \in \mathsf{S}_0$, *then for each* $(s^n_0, l, s') \in \rho_1$, *the run* $r \,.\, (s^n_0, l, s')$ *is a run of* f_1.

Let the rank *of a run be defined as follows. For any run that contains an accepting state, the rank for the run is* 0. *The rank of every other run* σ *is one more than the rank of all other runs of* f_1 *of the form* $\sigma \,.\, (s, l, s')$ *for some* $(s, l, s') \in \rho_0 \cup \rho_1$. *Strategy* f_1 *is a winning strategy if the rank of every run of* f_1 *is infinite; i.e. there is no run in which each Player-1 letter is defined by* f_1 *that reaches an accepting state.*

A Player 0-strategy f_0 *for* G*, and its runs, are defined symmetrically to a Player-1 strategy, with the Player-0 strategy mapping each trace to a Player-0 action. A Player-0 strategy is a winning strategy if no run of the strategy has finite rank.*

The game problem *for game* G *is to construct a winning Player-0 strategy or winning Player-1 strategy for* G.

Each parity game defined by Defn. 5 has either a winning Player-0 strategy or a winning Player-1 strategy. Furthermore, known algorithms can solve finite parity game problems in polynomial time. If a Player-1 strategy exists, then there is some winning strategy f_1 that may be represented as a finite-state acceptor $\mathcal{A}_1 \subseteq \Sigma_0(\Sigma_1 \times \Sigma_0)^*$, where $f_1(l_0^0(l_1^0, l_0^1), \ldots, (l_1^{n-1}, l_0^n)) = l_1^n$ if and only if for all $l_0^n \in \Sigma_0$, $\mathcal{A}_0([l_0^0(l_1^0, l_0^1), \ldots, (l_1^{n-1}, l_0^{n-1}), (l_1^n, l_0^n)])$. Each Player-0 strategy may similarly be represented as a finite-state acceptor.

Each policy-weaving problem $\text{POLWEAVE}(\mathsf{P}, \mathsf{H}, \mathsf{S}, \mathsf{F})$ with regular P, H, S, and F is reduced to finding a winning Player-1 strategy to a finite-graph parity game. By Lem. 3, the POLWEAVE problem reduces to $\text{UBWEAVE}(\mathsf{P}, \mathcal{C}_{\mathsf{H}}^+ \cap \mathcal{C}_{\mathsf{H}}^-, \mathcal{C}_{\mathsf{S}}^+ \cap \mathcal{C}_{\mathsf{F}}^-, \emptyset)$. It is easy to show that if P, H, S, and F are regular, then $\mathcal{C}_{\mathsf{H}}^+ \cap \mathcal{C}_{\mathsf{H}}^-$ and $\mathcal{C}_{\mathsf{S}}^+ \cap \mathcal{C}_{\mathsf{F}}^-$ are regular.

$\text{UBWEAVE}(\mathsf{P}, \mathsf{H}, \mathsf{Pol})$ defined by regular inputs is reducible to a finite-graph parity game as follows. If P, H, and Pol are regular, one can show that the language of violating plays $\mathsf{Vio}(\mathsf{P}, \mathsf{H}, \mathsf{Pol})$ is regular. Recall that for $\text{UBWEAVE}(\mathsf{P}, \mathsf{H}, \mathsf{Pol})$, the language $\mathsf{V} = \mathsf{Vio}(\mathsf{P}, \mathsf{H}, \mathsf{Pol})$ describes all plays that violate Pol. From $\mathcal{A}_{\mathsf{V}} = (\mathsf{S}, A, s_0, \mathsf{C} \times \mathsf{D}, \rho)$ a deterministic acceptor for V, we may construct a game $\mathsf{G} = \mathsf{Game}(\mathsf{V})$, such that a Player-1 strategy for $\mathsf{G} = (\widehat{\mathsf{S}_0}, \widehat{\mathsf{S}_1}, \widehat{A}, \widehat{s_0}, \widehat{\Sigma_0}, \widehat{\Sigma_1}, \widehat{\rho_0}, \widehat{\rho_1})$ is a solution to $\text{UBWEAVE}(\mathsf{P}, \mathsf{H}, \mathsf{Pol})$. Game G is defined as follows. $\widehat{\mathsf{S}_0} = \mathsf{S}$, $\widehat{\mathsf{S}_1} = \mathsf{S} \times \mathsf{C}$, $\widehat{s_0} = s_0$, $\widehat{A} = A$, $\Sigma_0 = \mathsf{C}$, and $\Sigma_1 = \mathsf{D}$. For each $s \in \mathsf{S}$ and each $c \in \mathsf{C}$, let $\widehat{\rho_0}(s, c, (s, c))$. For each $s, s' \in \mathsf{S}$, $c \in \mathsf{C}$, and $d \in \mathsf{D}$, let $\widehat{\rho_1}((s, c), d, s')$ if and only if $\rho(s, (c, d), s')$.

Lemma 4. *The solutions to* $\text{UBWEAVE}(\mathsf{P}, \mathsf{H}, \mathsf{Pol})$ *are exactly the Player-1 strategies to* $\mathsf{Game}(\mathsf{Vio}(\mathsf{P}, \mathsf{H}, \mathsf{Pol}))$.

Proof. Let Q be a solution to $\text{UBWEAVE}(\mathsf{P}, \mathsf{H}, \mathsf{Pol})$. Then Q defines a Player-1 strategy f_{Q}, using the construction of Defn. 6. Suppose, for a proof of contradiction, that there is a play $\sigma = c_0, d_0, c_1, d_1, \ldots, c_n, d_n$ of f_{Q} that is not a winning play of $\mathsf{G} = \mathsf{Game}(\mathsf{Vio}(\mathsf{P}, \mathsf{H}, \mathsf{Pol}))$, and thus is a play of G for Player 0. Then by the construction of G, the string $\tau = (c_0, d_0), (c_1, d_1), \ldots, (c_n, d_n)$ is a play in $\mathsf{ProgPlays}(\mathsf{P}) \cap \mathsf{Plays}(\mathsf{H}, \overline{\mathsf{Pol}})$. The play τ thus induces a system trace for which the program commands are an execution of P, and which violates Pol. Thus Q is not a solution of $\text{UBWEAVE}(\mathsf{P}, \mathsf{H}, \mathsf{Pol})$, which contradicts the initial assumption for Q. Thus Q is a Player-1 strategy to G.

Let f be a winning Player-1 strategy to G, represented as a language $\mathsf{Q}_f \subseteq (\mathsf{C} \times \mathsf{D})^*$. Suppose, for a proof by contradiction, that there is some play $\sigma = (c_0, d_1), (c_1, d_1), \ldots, (c_n, d_n)$ of Q_f such that $\sigma \in \mathsf{ProgPlays}(\mathsf{P}) \cap \mathsf{Plays}(\mathsf{H}, \overline{\mathsf{Pol}})$. Then by the construction of G, there is some run of G of the form $r(\tau) = (s_0^0, c_0, s_1^0)$, (s_1^0, d_0, s_0^1), (s_0^1, c_1, s_1^1), (s_1^1, d_1, s_0^2), $\ldots$, $(s_0^{n-1}, c_{n-1}, s_1^{n-1})$, $(s_1^{n-1}, d_{n-1}, s_0^n)$ with s_0^n an accepting state of G. But then f is not a winning strategy of G. □

Thus any policy-weaving problem defined by regular languages may be reduced to a finite-graph parity game.

Theorem 1. *The solutions to* $\text{POLWEAVE}(\mathsf{P}, \mathsf{H}, \mathsf{S}, \mathsf{F})$ *are exactly the Player-1 strategies to* $\mathsf{Game}(\mathsf{Vio}(\mathsf{P}, \mathcal{C}_{\mathsf{H}}^+ \cap \mathcal{C}_{\mathsf{H}}^-, \mathcal{C}_{\mathsf{S}}^+ \cap \mathcal{C}_{\mathsf{F}}^-))$.

Proof. By Lemmas 3 and 4. □

POLWEAVE restricted to regular languages is, in fact, equivalent to the problem of finding a strategy for parity games, in that one can reduce any parity game to a POLWEAVE problem constructed from regular languages. However, we omit the reduction in this direction.

3.3 Symbolic Policy Weaving

Practical classes of POLWEAVE are challenging to solve scalably. As shown in §3.2, each policy-weaving problem $\text{POLWEAVE}(\mathsf{P}, \mathsf{H}, \mathsf{S}, \mathsf{F})$ defined by regular languages can be reduced to finding a winning Player-1 strategy to a finite-graph parity game. There are at least three issues that we must address. First, the size of the game is directly proportional to the size of the automata that represent P, H, S, and F, and in practice the size of the automata for P and H can be quite large. The size of P and H can be reduced by soundly over-approximating them with smaller automata (see §3.4), but natural over-approximations may still be large, and thus games defined by such automata may take prohibitively long to solve. Second, the size of a winning Player-1 strategy for such a game, and thus the size of the solution to the corresponding weaving problem, is in general proportional to the size of the game automaton. If a program is instrumented to query a large strategy at runtime to decide when to invoke host primitives, then the instrumentation could degrade the time and space performance of the program unacceptably. Third, the alphabet of host primitives used to define a weaving problem (i.e., the Player-1 alphabet) may be large, while the alphabet of program commands relevant to security (i.e., the Player-0 alphabet) may be relatively small. For example, Capsicum supports 63 rights, and allows a program to attempt to set each file descriptor to any subset of the rights, effectively providing 2^{63} host primitives for each resource. In the reduction of §3.2, each host primitive corresponds to a letter in the alphabet of Player 1 in the resulting game, but traditional algorithms for solving such parity games typically enumerate over the entire alphabet of Player 1, and the number of enumerations is proportional to the size of the game.

In this section, we address these scalability issues via a symbolic technique for solving POLWEAVE. We assume that the transition relation of the automaton for each input language may be represented as a formula in a theory $\mathcal{T}$ that allows an efficient decision procedure, such as a combination of SMT theories [30]. Modern decision procedures can often solve formulas in such theories very quickly in practice [12, 13].

Definition 7. *An acceptor automaton* $\mathcal{A} = (\mathcal{T}, \mathsf{S}, s_0, A, \Sigma, \rho)$ *is* represented symbolically *if*

1. *$\mathcal{T}$ is a logical theory that supports conjunction, disjunction, and negation.*
2. *The set of states S and alphabet of actions Σ are domains of values.*
3. *$s_0(s)$ can be expressed as a formula in $\mathcal{T}$, in which the variable s is free, satisfied only by the initial state of $\mathcal{A}$.*
4. *The set of accepting states $A(s)$ can be expressed as a formula in $\mathcal{T}$ in which the variable s is free, and which is only satisfied by the accepting states of $\mathcal{A}$.*
5. *The transition relation $\rho(s, l, s') \subseteq \mathsf{S} \times \Sigma \times \mathsf{S}$ can be expressed as a formula in $\mathcal{T}$ in which s, l, and s' are free, and which only hold for (s, l, s') in the transition relation of $\mathcal{A}$.*

A policy-weaving problem defined by languages given as symbolic acceptors can be efficiently reduced to solving a symbolically represented parity game. In particular, consider a policy-weaving problem $\text{POLWEAVE}(\mathsf{P}, \mathsf{H}, \mathsf{S}, \mathsf{F})$ accompanied by a bound $b > 0$ on the size of the state-space of a solution allowed for the problem, in which P, H, S, and F are given as symbolic acceptors. From the symbolic acceptors for the input languages, we can construct a symbolic representation of the game $\mathsf{G} = \mathsf{Game}(\mathsf{Vio}(\mathsf{P}, \mathcal{C}_{\mathsf{H}}^+ \cap \mathcal{C}_{\mathsf{H}}^-, \mathcal{C}_{\mathsf{S}}^+ \cap \mathcal{C}_{\mathsf{F}}^-))$, defined in §3.2.

Existing symbolic techniques [11, 24] for solving symbolic parity games are, in general, unsatisfactory for solving games constructed by reduction from POLWEAVE, for the following rea-

sons. First, most of the techniques given in [11, 24] take as input a symbolic representation of a game and produce either a decision as to which player can always win the game, or construct a symbolic strategy. However, instrumenting a program to query a symbolic solution at runtime does not resolve the second problem raised above, because such an instrumentation could degrade the performance of the application unacceptably. Second, the algorithms of [11, 24] explicitly enumerate over Player 1's alphabet. Thus, such algorithms do not address the third issue raised above.

We thus solve symbolic games produced by reduction from POLWEAVE by applying a novel symbolic construction for finding a Player-1 strategy to a symbolic parity game. The construction searches for a *sub-game* of G by searching for a "witness set" of b states in G, and transitions between them, such that no play of the sub-game is a winning play for Player 0 in G, and the sub-game is Player-1 simulated by G [4]. Any such sub-game corresponds to a winning Player-1 strategy for G. The construction is analogous to a known construction [24] for solving symbolic parity games; however, our solution finds strategies with a different set of winning conditions (i.e., the ones specified in Defn. 5).

Our construction defines a winning Player-1 strategy for a game $\mathsf{G} = (\mathsf{S}_0, \mathsf{S}_1, A, s_0, \Sigma_0, \Sigma_1, \rho_0, \rho_1)$ from an interpretation ι of a bounded set of logical constants as states in S_0 and S_1, and letters in Σ_1. By bounding the set of constants interpreted as states, we may bound the execution time of a decision procedure applied to solve the game, and simultaneously bound the size of any strategy found by our construction. In particular, to bound the search for a winning Player-1 strategy to strategies that may be represented by automata with b states, let there be b constants for Player-0 sub-game states. Moreover, for each Player-0 sub-game state s, and each Player-0 letter l, let there be one Player-1 sub-game state for which there is an l-transition from s. Thus, for each i, $0 \leq i < b + b|\Sigma_0|$, let there be a state constant q_i for each state in the collection of Player-0 and Player-1 sub-game states.

To bound the search through the large alphabet of Player-1 letters, we define a small, bounded set of constants that are interpreted as the different Player-1 letters chosen by a strategy. For each state constant q_i, let there be a Player-1 letter constant d_i to be interpreted as the letter chosen by Player-1 if the game reaches the state interpreted for q_i: for each i, $0 \leq i < b|\Sigma_0|$, let there be a Player-1-letter constant d_i.

We will search for an interpretation ι of the state constants q_i and letter constants d_j that interprets the constants as a sub-game of G from which we may construct a winning Player-1 strategy for G. Such an interpretation must be a model of the following constraints. First, no state of the sub-game is an accepting state of G, so for each state constant q_i we have the constraint

$$\neg A(\mathsf{q}_i) \tag{11}$$

Second, some state of the sub-game—we may arbitrarily choose the state interpreted for q_0—is the initial state of the game. Thus we have the constraint:

$$s_0(\mathsf{q}_0) \tag{12}$$

Third, each Player-0 state of the sub-game has a transition on each Player-0 letter to some other state of the sub-game. For each state constant q_i, we have the constraint

$$\mathsf{S}_0(\mathsf{q}_i) \implies \bigwedge_j \bigvee_k \rho_0(\mathsf{q}_i, c_j, \mathsf{q}_k) \tag{13}$$

Finally, each Player-1 state of the sub-game has a transition on some Player-1 letter to some other state of the sub-game. For each q_i, we have the constraint

$$\mathsf{S}_1(\mathsf{q}_i) \implies \bigvee_j \rho_1(\mathsf{q}_i, \mathsf{d}_i, \mathsf{q}_j) \tag{14}$$

The conjunction of these four kinds of constraints define the *bounded symbolic weaving formula* $\mathsf{B}(\mathsf{G}, b)$. The bounded symbolic weaving formula is a formula of $\mathcal{T}$, and thus a decision procedure for $\mathcal{T}$ can be applied to search for a satisfying interpretation for the constants $\{\mathsf{q}_i\}_i$ and $\{\mathsf{d}_j\}_j$. Consider the size of $\mathsf{B}(\mathsf{G}, b)$ in terms of the occurrences of formulas that define the symbolic representation of G (i.e., s_0, A, and ρ). For $n = b + b|\Sigma_0|$, the number of state and Player-1 alphabet letters defined, there are n occurrences of A, one occurrence s_0, n occurrences of S_0, $n^2|\Sigma_0|$ of ρ_0, n occurrences of S_1, and $nb|\Sigma_0|$ occurrences of ρ_1.

If the decision procedure for $\mathcal{T}$ finds a satisfying interpretation ι, then from ι we may construct a solution automaton $\mathsf{Q}(\iota) = (\widehat{\mathsf{S}}, \widehat{s_0}, \widehat{A}, \widehat{\Sigma}, \widehat{\rho})$ to represent a Player-1 solution for G (see Defn. 6). The state space of the solution $\widehat{\mathsf{S}}$ is all Player-1 states in the range of the interpretation: $\widehat{\mathsf{S}} = \mathsf{S}_0 \cap \mathsf{Rng}(\iota)$. The initial state of the solution $\widehat{s_0}$ is the initial state of the game: $\widehat{s_0} = s_0$. The accepting states of the solution $\widehat{A}$ are all states of the solution: $\widehat{A} = \widehat{\mathsf{S}}$ (except for the stuck state). The alphabet of the solution is $\widehat{\Sigma} = \Sigma_0 \times \Sigma_1$. The transition relation of the solution $\widehat{\rho}$ is defined as follows for each pair of game-state constants $s_i, s_j \in \mathsf{S}$, program command c_k, and host primitive d_l:

$$\begin{aligned} \widehat{\rho}(s_i, (c_k, d_l), s_j) \iff & \bigvee_m \rho_0(s_i, c_k, \iota(\mathsf{q}_m)) \\ & \wedge d_l = \iota(\mathsf{d}_m) \\ & \wedge \rho_1(\iota(\mathsf{q}_m), d_l, s_j) \end{aligned}$$

$\widehat{\rho}$ is defined by the set of state-constants $\{\mathsf{q}_i\}_i$, and the set of host-primitive constants $\{\mathsf{d}_i\}_i$, all of which are assumed to be small. Thus $\widehat{\rho}$ can be constructed directly from ι without further applying a decision procedure for $\mathcal{T}$.

For $\mathsf{Q}(\iota)$ described above, we have the following theorem.

Theorem 2. *For weaving problem* $\mathrm{POLWEAVE}(\mathsf{P}, \mathsf{H}, \mathsf{S}, \mathsf{F})$*, let* $\mathsf{G} = \mathsf{Game}(\mathsf{Vio}(\mathsf{P}, \mathcal{C}_{\mathsf{H}}^+ \cap \mathcal{C}_{\mathsf{H}}^-, \mathcal{C}_{\mathsf{S}}^+ \cap \mathcal{C}_{\mathsf{F}}^-))$*. For bound* b*, if* ι *is a model of* $\mathsf{B}(\mathsf{G}, b)$*, then* $\mathsf{Q}(\iota) \subseteq (\mathsf{C} \times \mathsf{D})^*$ *is a solution to* $\mathrm{POLWEAVE}(\mathsf{P}, \mathsf{H}, \mathsf{S}, \mathsf{F})$.

Proof. $\mathsf{Q}(\iota)$ is a solution to $\mathrm{POLWEAVE}(\mathsf{P}, \mathsf{H}, \mathsf{S}, \mathsf{F})$ if and only if it is a winning Player-1 strategy to the game $\mathsf{G} = \mathsf{Game}(\mathsf{Vio}(\mathsf{P}, \mathcal{C}_{\mathsf{H}}^+ \cap \mathcal{C}_{\mathsf{H}}^-, \mathcal{C}_{\mathsf{S}}^+ \cap \mathcal{C}_{\mathsf{F}}^-, \emptyset))$. Let $\mathsf{G} = (\mathsf{S}_0, \mathsf{S}_1, s_0, A, \Sigma_0, \Sigma_1, \rho_0, \rho_1)$. Let the *sub-game defined by* ι be $\mathsf{G}_\iota = (\widehat{\mathsf{S}_0}, \widehat{\mathsf{S}_1}, \widehat{s_0}, \widehat{A}, \widehat{\Sigma_0}, \widehat{\Sigma_1}, \widehat{\rho_0}, \widehat{\rho_1})$, with $\widehat{\mathsf{S}_0} = \mathsf{S}_0 \cap \mathsf{Rng}(\iota)$, $\widehat{\mathsf{S}_1} = \mathsf{S}_1 \cap \mathsf{Rng}(\iota)$, $\widehat{s_0} = s_0$, $\widehat{A} = A \cap \mathsf{Rng}(\iota)$, $\widehat{\Sigma_0} = \Sigma_0$, $\widehat{\Sigma_1} = \Sigma_1$, $\widehat{\rho_0} = \rho_0 \cap (\widehat{\mathsf{S}_0} \times \Sigma_0 \times \widehat{\mathsf{S}_1})$, and $\widehat{\rho_1} = \rho_1 \cap (\widehat{\mathsf{S}_1} \times (\Sigma_1 \cap \mathsf{Rng}(\iota)) \times \widehat{\mathsf{S}_0})$

Let f_ι be the strategy defined by $\mathsf{Q}(\iota)$ by Defn. 6. We will show that every play of f_ι is a winning play for Player 1. We will show that for each play σ of f_ι, $r(\sigma)$ (see proof of Lem. 4) ends in some state in $\widehat{\mathsf{S}_0} \cup \widehat{\mathsf{S}_1}$, by induction on the length of σ.

As the base case, consider a play of f_ι of length 1, which is a program command c. By constriant (12), there is some state $s_0 \in \mathsf{S}_0$ that is the initial state of G, and by constraint (13), for each $c \in \Sigma_0$, there is some state $(s_0, c) \in \widehat{\mathsf{S}_1}$ such $\rho(s_0, c, (s_0, c))$.

For the inductive case, let σ be a play of f_ι of length n. First, suppose $\sigma = \tau \, . \, c_n$. By the definition of the plays of f_ι (Defn. 6), τ is a play of f_ι. Furthermore, $|\tau| = n - 1 < n$, and thus by the inductive hypothesis, τ ends in a Player-0 state $s_0^n \in \widehat{\mathsf{S}_0}$. By constraint (13), if $s_1^n \in \mathsf{S}_1$ is such that $\rho_0(s_0^n, c, s_1^n)$, then s_1^n is a state of $\mathsf{G}(\iota)$. Thus σ ends in a state in $\widehat{\mathsf{S}_0}$.

Now, suppose $\sigma = \tau.d_n$. By the definition of the plays of f_ι, τ is a play of f_ι. Furthermore, $|\tau| = n - 1 < n$, and thus by the inductive hypothesis, τ ends in a Player-1 state $s_1^n \in \widehat{\mathsf{S}_1}$. By constraint (14), there is some state constant q and host-command

constant d such that $\rho_1(s_1^n, \iota(\mathsf{d}), \iota(\mathsf{q}))$. Thus σ ends in a state of $\widehat{s_1}$, and thus in a state of $\mathsf{G}(\iota)$.

Thus by induction, for each play σ of f_ι, σ ends in a state in $\widehat{\mathsf{S}_0} \cup \widehat{\mathsf{S}_1}$. No state in $\widehat{\mathsf{S}_0}$ is in A by constraint (11), and no state in $\widehat{\mathsf{S}_1}$ is in A, because $\widehat{\mathsf{S}_1} \subseteq \mathsf{S}_1$, $A \subseteq \mathsf{S}_1$, and S_0 and S_1 do not overlap. Thus no play of f_ι corresponds to a run in which G enters a state in A. Thus by Defn. 6, f_ι is a winning strategy of G. □

If for policy-weaving problem $\mathrm{POLWEAVE}(\mathsf{P}, \mathsf{H}, \mathsf{S}, \mathsf{F})$, the languages of P, H, S, and F are given by symbolic acceptors with finite state spaces, and the problem has a solution, then for the resulting parity game G, there is some b for which $\mathsf{B}(\mathsf{G}, b)$ is satisfiable. Furthermore, we can use the techniques of [24] to simultaneously search for a Player-0 strategy to the symbolic game, as a proof that the original weaving problem has no solution.

3.4 Policy Weaving over Abstractions

3.4.1 Sound Abstractions of Programs

§3.2 gives an algorithm for solving policy-weaving problems if all inputs languages, including the program, are represented as finite-state acceptors. Of course, real-world programs cannot be modeled precisely as finite-state acceptors, or even by simple stack machines; however, we now show that, as is the case for other well-known problems in software verification [5], a solution for a policy-weaving problem constructed from an over-approximation of a program is a solution for a policy-weaving problem constructed from the original program. Thus if we can solve a policy-weaving problem for an automaton that over-approximates the set of runs of the program, such as the automaton induced from the program's control-flow graph, then we can solve the weaving problem for the original program.

Theorem 3. *Let* Q *be a solution to* $\mathcal{P}^\# = \mathrm{POLWEAVE}(\mathsf{P}^\#, \mathsf{H}, \mathsf{S}, \mathsf{F})$, *with* $\mathsf{P} \subseteq \mathsf{P}^\#$. *Then* Q *is a solution to* $\mathcal{P} = \mathrm{POLWEAVE}(\mathsf{P}, \mathsf{H}, \mathsf{S}, \mathsf{F})$.

Proof. Q is a solution to $\mathcal{P}$ if and only if Q is a solution to $\mathcal{P}' = \mathrm{UBWEAVE}(\mathsf{P}, \mathcal{C}_\mathsf{H}^+ \cap \mathcal{C}_\mathsf{H}^-, \mathcal{C}_\mathsf{S}^+ \cap \mathcal{C}_\mathsf{F}^-) = \mathrm{POLWEAVE}(\mathsf{P}, \mathcal{C}_\mathsf{H}^+ \cap \mathcal{C}_\mathsf{H}^-, \mathcal{C}_\mathsf{S}^+ \cap \mathcal{C}_\mathsf{F}^-, \emptyset)$ (Lem. 3). We will show that Q is a solution to $\mathcal{P}'$ by showing that it satisfies every condition for a solution given in Defn. 2. First, Q is online, because it is a solution to $\mathcal{P}^\#$. Q is secure for $\mathcal{P}$, because $\mathsf{SysTraces}(\mathsf{P}, \mathsf{Q}, \mathsf{H}) \subseteq \mathsf{SysTraces}(\mathsf{P}^\#, \mathsf{Q}, \mathsf{H})$ by the definition of SysTraces (Defn. 2), and $\mathsf{SysTraces}(\mathsf{P}^\#, \mathsf{Q}, \mathsf{H}) \subseteq \mathcal{C}_\mathsf{S}^+ \cap \mathcal{C}_\mathsf{F}^-$, because Q is a solution to $\mathrm{POLWEAVE}(\mathsf{P}, \mathsf{H}, \mathsf{S}, \mathsf{F}) = \mathrm{UBWEAVE}(\mathsf{P}, \mathcal{C}_\mathsf{H}^+ \cap \mathcal{C}_\mathsf{H}^-, \mathcal{C}_\mathsf{S}^+ \cap \mathcal{C}_\mathsf{F}^-)$. Thus,

$$\mathsf{SysTraces}(\mathsf{P}, \mathsf{Q}, \mathsf{H}) \subseteq \mathsf{SysTraces}(\mathsf{P}^\#, \mathsf{Q}, \mathsf{H})$$

Thus Q satisfies the security condition of $\mathcal{P}$. Q satisfies the functionality condition of $\mathcal{P}$ because the functionality policy is $\emptyset$, and thus is satisfied trivially. □

3.4.2 Sound Abstractions of Hosts

Practical hosts can typically be formalized directly as state machines. However, we typically cannot solve weaving problems defined from such hosts, because, for example, their transition relation may be specified using formulas with alternating quantification (see §4.1). The following theorem allows us to soundly solve a policy-weaving problem over a "complex" host by substituting a "simpler" host, analogous to how Thm. 3 allows us to soundly solve a policy-weaving problem over a complex program by substituting a simpler program. As in Thm. 3, the substitution is incomplete, in the sense that the original weaving problem may have a solution, while the simpler weaving problem may not.

Theorem 4. *Let* Q *be a solution to* $\mathrm{UBWEAVE}(\mathsf{P}, \mathsf{H}^\#, \mathsf{Pol})$*, and let* $\mathsf{H} \subseteq \mathsf{H}^\#$. *Then* Q *is a solution to* $\mathrm{UBWEAVE}(\mathsf{P}, \mathsf{H}, \mathsf{Pol})$.

Proof. Let Q be a solution to $\mathcal{P}^\# = \mathrm{UBWEAVE}(\mathsf{P}, \mathsf{H}^\#, \mathsf{Pol})$. Q is a solution to $\mathrm{UBWEAVE}(\mathsf{P}, \mathsf{H}, \mathsf{Pol})$ if and only if it is the solution to $\mathcal{P} = \mathrm{POLWEAVE}(\mathsf{P}, \mathsf{H}, \mathsf{Pol}, \emptyset)$. We will show that Q is solution to $\mathcal{P}$ by showing that it satisfies every condition on a solution given in Defn. 2. First, Q is online, because it a solution to $\mathcal{P}^\#$. Q is secure for $\mathcal{P}$ because $\mathsf{SysTraces}(\mathsf{P}, \mathsf{Q}, \mathsf{H}) \subseteq \mathsf{SysTraces}(\mathsf{P}, \mathsf{H}^\#, \mathsf{Pol})$ by the definition of SysTraces, and $\mathsf{SysTraces}(\mathsf{P}, \mathsf{Q}, \mathsf{H}^\#) \subseteq \mathsf{Pol}$, because Q is a solution to $\mathcal{P}^\#$. Thus

$$\mathsf{SysTraces}(\mathsf{P}, \mathsf{Q}, \mathsf{H}) \subseteq \mathsf{SysTraces}(\mathsf{P}, \mathsf{H}^\#, \mathsf{Pol}) \subseteq \mathsf{Pol}$$

Q is functional for $\mathcal{P}$, as the functionality policy for $\mathcal{P}$ is $\emptyset$, and thus the functionality condition is satisfied trivially. □

We can apply Thm. 4 to solve an instance of UBWEAVE produced as a reduction from POLWEAVE. In doing so, we over-approximate, for host $\mathsf{H} \subseteq ((\mathsf{C} \times \mathsf{D}) \times \mathsf{E})^*$, its closure $\mathcal{C}_\mathsf{H}^+ \cap \mathcal{C}_\mathsf{H}^-$ with some abstract host $\mathsf{H}^\#$, such that $\mathcal{C}_\mathsf{H}^+ \cap \mathcal{C}_\mathsf{H}^- \subseteq \mathsf{H}^\#$. However, $\mathsf{H}^\#$ itself need not be the closure of any host. In fact, it turns out that for $\mathsf{H} \subseteq ((\mathsf{C} \times \mathsf{D}) \times \mathsf{E})^*$, if $\mathcal{C}_\mathsf{H}^+ \cap \mathcal{C}_\mathsf{H}^- \subset \mathsf{H}^\#$, there is no host $\mathsf{H}_1 \subseteq ((\mathsf{C} \times \mathsf{D}) \times \mathsf{E})^*$ for which $\mathcal{C}_{\mathsf{H}_1}^+ \cap \mathcal{C}_{\mathsf{H}_1}^- = \mathsf{H}^\#$. Thm. 4 shows that for the policy-weaving problem, this inconsistency may be tolerated as imprecision.

3.5 Policy Weaving over Partially-Trusted Programs

A program that runs on a privilege-aware system either is often composed of or interacts with untrusted program modules. Recall from §1 and §2 that to secure `tcp_dump`, the programmer needed to reason about partially-trusted functions for matching packets that may be compromised, and then execute arbitrary commands. Weaving partially-trusted programs presents two main challenges. The first challenge is immediate: a partially-trusted program follows a weaker model, in which after certain points in an execution, it executes arbitrary program commands, and even invokes arbitrary host commands.

The second challenge arises from reasoning about programs that are partitioned to execute in separate process spaces. Programmers attempt to write programs that execute securely in the presence of untrusted modules by isolating different modules of the program in different processes. If some of the modules are compromised during execution, then they still may not cause the entire program to behave insecurely, if the insecure behavior requires collusion with modules that are not compromised. We currently assume that processes execute sequentially: a process executes until it transfers control to a module in another process by invoking the module via a remote procedure call (RPC), and the caller process blocks until the called process finishes executing. Given that our weaving algorithm is defined on state machines, it seems likely that we can extend the algorithm to weave systems constructed from concurrently executing processes; we leave such an extension for future work.

If we view a program as a collection of mutually untrusting processes, then this complicates the weaving problem, because we cannot instrument a program according to a weaving solution that monitors program commands and updates its internal state across multiple processes. Such an instrumentation is infeasible in practice because if a called process receives the current state of the strategy from a calling process, but the calling process may be compromised, then the callee cannot trust the validity of the strategy state.

We now formalize the problem of weaving over partially-trusted programs as the PARTWEAVE problem.

Definition 8. *Let a program be a set of n* modules. *For module i, let C_i be the alphabet of module program commands, Let D_i be the alphabet of host primitives that may be issued by module i, and let E_i be the alphabet of privilege events for module i. Let module 0 be the* root module*, and let C_0 contain a set of program commands $\{\mathsf{call}(j)\}_j$ for $0 < j < n$. The problem $\mathrm{PARTWEAVE}(\{\mathsf{P}_i\}_i, \{\mathsf{T}_i\}_i, \mathsf{H}, \mathsf{S}, \mathsf{F})$ is defined by:*

- *$\{\mathsf{P}_i\}_i$, with $\mathsf{P}_i \subseteq \mathsf{C}_i^*$, are the* well-behaved models *of each module.*
- *$\{\mathsf{T}_i\}_i$, with $\mathsf{T}_i \subseteq \mathsf{P}_i$, are the* trustworthy executions *of each module.*
- *$\mathsf{H} \subseteq ((\bigcup_i \mathsf{C}_i \times \bigcup_i \mathsf{D}_i) \times \bigcup_i \mathsf{E}_i)^*$ is the secure host.*
- *$\mathsf{S}, \mathsf{F} \subseteq \bigcup_i \mathsf{C}_i \times \bigcup_i \mathsf{E}_i$ are the security and functionality policies, respectively.*

The set of languages of trustworthy executions $\{\mathsf{T}_i\}_i$ define when a program may execute arbitrary program and host commands, as follows. Let the language of *untrustworthy executions* of each module U_i be defined from each module and set of trustworthy executions as $\mathsf{U}_i = \mathsf{P}_i \backslash \mathsf{T}_i$. Intuitively, if the ith module ever executes a sequence of commands in U_i, then the module may then execute an arbitrary sequence of program and host commands.

Defn. 8 may only be used to model programs in which a root module may RPC auxiliary modules in other processes, but the auxiliary modules may not RPC each other. In practice, it is simple to require each RPC module to only accept calls from the root module, and many practical programs fit this model. Moreover, if we allow processes to create arbitrarily many processes via RPC, with each process supported by the host, then intuitively we must model a host as an automaton with an unbounded stack, such as a pushdown system or nested-word automaton [2]. We leave this as future work.

A solution to a PARTWEAVE problem is a set of solutions $\{\mathsf{Q}_i\}_i$, with $\mathsf{Q}_i \subseteq (\mathsf{C}_i \times \mathsf{D}_i)^*$, subject to the following security and functionality conditions.

Security Condition The security condition describes all system traces that the instrumented program may induce, analogous to the security condition in Defn. 2. The key difference from Defn. 2 is that the security condition constrains the system traces of the program, even if modules of the program are compromised. For solution Q_i, let L_i be the language of module i instrumented with Q_i, including executions in which L_i is compromised after executing some untrustworthy execution:

$$\mathsf{L}_{i>0} = (\mathsf{Q}_i(\mathsf{P}_i) \cup (\mathsf{Q}_i(\mathsf{U}_i) \,.\, (\mathsf{C}_i \times \mathsf{D}_i)^*))$$
$$\mathsf{L}_0 = (\mathsf{Q}_0(\mathsf{P}_0) \cup (\mathsf{Q}_0(\mathsf{U}_0) \,.\, (\mathsf{C}_0 \times \mathsf{D}_0)^*))[\{\mathsf{L}_i/\mathsf{call}(i)\}_{j>0}]$$

Let $\mathsf{PartIns}(L)$, analogous to $\mathsf{Ins}(\mathsf{P}, \mathsf{Q})$ in Defn. 2, be

$$\mathsf{PartIns}(L) = \{(\sigma, \tau) | \tau = [(c_0, d_0), (c_1, d_1), \ldots, (c_n, d_n)] \wedge \sigma = \pi_{\mathsf{C}\times\mathsf{D},1}(\tau) \in L\}$$

Let $\mathsf{PSysTraces}(\mathsf{H}, L)$, analogous to $\mathsf{SysTraces}$ in Defn. 2, be

$$\mathsf{PSysTraces}(\mathsf{H}, L) = \mathsf{Zip}[\mathsf{PartIns}(L) \circ \mathsf{H}]$$

The security condition is:

$$\mathsf{PSysTraces}(\mathsf{H}, \mathsf{L}_0) \subseteq \mathsf{S} \tag{15}$$

Functionality Condition The functionality condition describes the system traces that the instrumented program must induce in response to sequences of program commands, analogous to the functionality condition in Defn. 2. The functionality condition requires that the instrumented program preserves the core functionality of the original program, but it makes little sense to try to preserve the functionality of the program when some of its modules are compromised. Thus we define functionality solely over the well-behaved models of program modules.

For solution Q_i, let W_i be the language of program module i instrumented with Q_i, restricted to executions where each module satisfies its well-behaved model:

$$\mathsf{W}_i = \mathsf{Q}_i(\mathsf{P}_i)$$
$$\mathsf{W}_0 = \mathsf{Q}_i(\mathsf{P}_0)[\{\mathsf{W}_i/((\mathsf{call}(j) \times \mathsf{D}_i)\}_{j>0}]$$

Let P be the set of all executions of the modules $\{\mathsf{P}_i\}_i$:

$$\mathsf{P} = \mathsf{P}_0[\{\mathsf{P}_j/\mathsf{call}(j)\}_{j>0}]$$

Then the functionality condition is:

$$\mathsf{Zip}[\mathsf{F}|_\mathsf{P}] \subseteq \mathsf{PSysTraces}(\mathsf{H}, \mathsf{W}_0) \tag{16}$$

3.5.1 Solving PARTWEAVE

In §3.2, we showed that POLWEAVE can be reduced to solving a single parity game. However, we cannot reduce PARTWEAVE to a single parity game, because the solution automata Q_i must be independent of each other, for the practical reasons discussed in §3.5. But while we cannot reduce PARTWEAVE to finding a strategy to a single parity game, we can reduce it to finding a set of games that satisfy a particular condition, and finding winning strategies for these games.

In §3.2, we reduced each POLWEAVE problem to a POLWEAVE problem constructed from a trivial functionality policy using a set of closure functions defined for hosts and policies. In other words, the reduction was from a weaving problem that placed upper and lower bounds on the set of system traces allowed to a weaving problem that only placed an upper bound on the set of traces allowed. Similarly, every PARTWEAVE problem may be reduced to a UBPRTWEAVE problem that only places an upper bound on the set of system traces. However, unlike UBWEAVE and POLWEAVE, UBPRTWEAVE is a variation of PARTWEAVE, not a restricted case of it.

Definition 9. *A* (purely) upper-bounded partial weaving problem $\mathrm{UBPRTWEAVE}(\{\mathsf{P}_i\}_i, \{\mathsf{T}_i\}_i, \mathsf{H}, \mathsf{S}, \mathsf{F})$ *and its solutions satisfy the same definition as Defn. 8, with inequality* (16) *replaced with:*

$$\mathsf{PSysTraces}(\mathsf{H}, \mathsf{W}_0) \subseteq \mathsf{F} \tag{17}$$

Lemma 5. *The solutions to $\mathrm{PARTWEAVE}(\{\mathsf{P}_i\}_i, \{\mathsf{T}_i\}_i, \mathsf{H}, \mathsf{S}, \mathsf{F})$ are exactly the solutions to $\mathrm{UBPRTWEAVE}(\{\mathsf{P}_i\}_i,\ \{\mathsf{T}_i\}_i,\ \mathcal{C}^+_\mathsf{H} \cap \mathcal{C}^-_\mathsf{H}, \mathcal{C}^+_\mathsf{S} \cap \mathcal{C}^-_\mathsf{F})$.*

$\mathrm{UBPRTWEAVE}(\{\mathsf{P}_i\}_i, \{\mathsf{T}_i\}_i, \mathsf{H}, \mathsf{Pol})$ may be reduced to the problem of finding a suitable set of two-player parity games and their strategies, as follows. First, restate the security condition Eqn. (15) as finding a set of pairs of languages $\{(\mathsf{L}^1_i, \mathsf{L}^2_i)\}_i$, with $\mathsf{L}^0_i, \mathsf{L}^1_i \subseteq (\mathsf{C}_i \times \mathsf{D}_i)^*$ such that

$$\mathsf{L}_{i>0} = (\mathsf{L}^1_i \cup (\mathsf{L}^2_i \,.\, (\mathsf{C}_i \times \mathsf{D}_i)^*))$$
$$\mathsf{L}_0 = (\mathsf{L}^1_0 \cup (\mathsf{L}^2_0 \,.\, (\mathsf{C}_0 \times \mathsf{D}_0)^*))[\{\mathsf{L}_j/\mathsf{call}(j)\}_j]$$
$$\mathsf{P}_i \subseteq \pi_1(\mathsf{L}^1_i)$$
$$\mathsf{U}_i \subseteq \pi_1(\mathsf{L}^2_i)$$
$$\mathsf{L}_0 \subseteq \mathsf{Plays}(\mathsf{H}, \mathsf{S})$$

and requiring each Q_i to be a winning Player-1 strategy for the games

$$\mathsf{G}^1_i = \mathsf{Game}(\mathsf{ProgPlays}(\mathsf{P}_i) \cap \overline{\mathsf{L}^1_i})$$
$$\mathsf{G}^2_i = \mathsf{Game}(\mathsf{ProgPlays}(\mathsf{U}_i) \cap \overline{\mathsf{L}^2_i})$$

for Game as defined in §3.2.

Second, restate the functionality condition of Eqn. (17) as finding a set of languages $\mathsf{W}_i \subseteq (\mathsf{C}_i \times \mathsf{D}_i)^*$ such that

$$\mathsf{P}_i \subseteq \pi_1(\mathsf{W}_i^1)$$
$$\mathsf{W} = \mathsf{W}_0[\{\mathsf{W}_j/\mathsf{call}j\}_j]$$
$$\mathsf{W} \subseteq \mathsf{Plays}(\mathsf{H}, \mathsf{F})$$

and require that each strategy Q_i is a winning Player-1 strategy for the game

$$\mathsf{G}_i^3 = \mathsf{Game}(\mathsf{ProgPlays}(\mathsf{P}_i) \cap \overline{\mathsf{W}_i})$$

Each module solution Q_i is thus constrained as a single Player 1 strategy that wins two games G_i^1 and G_i^2 defined by the security condition, and one game G_i^3 defined by the functionality condition. However, all three games may be combined to constrain Q_i as a winning strategy to a single game:

$$\begin{aligned}\mathsf{Game}((&\mathsf{ProgPlays}(\mathsf{P}_i) \cap \overline{\mathsf{L}_i^1})\\ &\cup (\mathsf{ProgPlays}(\mathsf{U}_i) \cap \overline{\mathsf{L}_i^2})\\ &\cup (\mathsf{ProgPlays}(\mathsf{P}_i) \cap \overline{\mathsf{W}_i}))\end{aligned}$$

Thus every instance of PARTWEAVE may be expressed as finding a set of triples of languages $\{(\mathsf{L}_i^1, \mathsf{L}_i^2, \mathsf{W}_i)\}_i$, and a winning strategy Q_i for a game defined by each triple. We are aware of no algorithm that can in general efficiently find languages that satisfy systems of inequalities of the above form, and we leave the problem for future work. However, we can symbolically search for solutions up to a bound by extending the algorithm in §3.3. The key idea is to simultaneously search for languages that may be represented with automata of size up to some bound as we search for strategies to the games defined by the languages, approximating language containment constraints with simulation.

Refining Program Abstractions Automatically Thm. 3 states that if we can find a suitable abstraction $\mathsf{P}^\#$ of a program P such that we can solve a weaving problem defined over $\mathsf{P}^\#$, then the solution holds for the analogous weaving problem defined over P. Given a program P, there are several natural abstractions that may serve for weaving. However, in general we cannot expect any fixed abstraction to be suitable for all weaving problems, and it would be much better if we could iteratively refine a given abstraction of a program until we either find a solution to the weaving problem over P, or we validate that no weaving exists. Such an approach is in the spirit of counterexample-guided abstraction refinement (CEGAR), which is a standard approach to checking properties of systems with large or infinite state spaces [5, 10, 19].

However, we cannot solve weaving problems by directly mimicking CEGAR-based techniques for checking programs against properties. To see this, note that in each step of a CEGAR-based model checker, the checker considers an execution of the original program that is a potential counterexample to the correctness of the program. The checker either determines that the counterexample is a true counterexample to correctness, or the checker refines the abstraction of the program so that the program does not allow the counterexample as an execution. However, for a given POLWEAVE problem with no solution, there may not always be a single trace of the program that serves as a counterexample.

While an unsolvable weaving problem will not always have a single program trace as a counterexample, the reduction from weaving problems to parity games in §3.2 shows what objects witness that a weaving problem is unsolvable. In particular, whenever a weaving problem is unsolvable, we obtain a Player-0 strategy $f_0 \subseteq (\mathsf{C} \times \mathsf{D})^*$ that foils any candidate solution to the weaving problem.

Thus, to iteratively refine a model of a program towards finding a weaving solution, we must refine the program model using a Player-0 strategy f_0. We can do so as follows. Observe that $\pi_2(f_0) \subseteq \mathsf{C}^*$ is the set of all sequences of program commands that the program may execute to foil a solution. If every sequence of program commands in $\pi_2(f_0)$ is a possible execution of P, formally that $\pi_2(f_0) \subseteq \mathsf{P}$, then f_0 is a true counterexample to the weaving problem, and we present it to the user. However, if there is some run of $\pi_2(f_0)$ that is not an execution of P, then it is possible that the true program P does allow a solution, but the abstraction $\mathsf{P}^\#$ is too coarse to determine this. In this case, we may use standard techniques from CEGAR-based model checking to obtain a new abstraction of the program that does not allow the spurious execution. Note that our model-checking problem $\pi_2(f_0) \subseteq \mathsf{P}$ is actually checking the finite-state specification $\pi_2(f_0)$ as an under-approximation of the program P. However, we may construct a program P' such that $\mathsf{P}' \subseteq \overline{\pi_2(f_0)}$ if and only if $\pi_2(f_0) \subseteq \mathsf{P}$, and from any counterexample trace to the claim $\mathsf{P}' \subseteq \overline{\pi_2(f_0)}$, we may construct a counterexample trace to the claim $\pi_2(f_0) \subseteq \mathsf{P}$.

3.5.2 Weaving over Stack-Based Models of Programs

The policy-weaving problem as defined in Defn. 2 is defined for programs, hosts, and policies as arbitrary languages. In §3.2, we discussed how to solve such problems when all languages are regular, and are represented as finite-state machines. However, the input languages may be irregular, and in particular, may be stack-based. There are multiple possible advantages to weaving over such languages:

1. Stack-based models of programs accurately capture the behavior of calls and returns in the program. This allows for more precise definitions of the plays that Player 0 is allowed in the resulting game, which may allow us to find winning Player-1 strategies that do not win for games constructed from any finite-state over-approximation of the same program.
2. Policies for some hosts, such as the Java Virtual Machine with Java Stack Inspection [16], are defined over the stack configurations of the program.

Enriching the languages of programs, host, and policies to stack-based languages has important consequences for our ability to solve the resulting policy-weaving problem. In particular, suppose that stack-based languages may be either context-free languages, represented as pushdown systems (PDS) [6], or visibly-pushdown languages, represented as nested-word automata (NWA) [2]. First, consider the reduction of the original POLWEAVE problem to a parity game, as discussed in §3.2. We can define a policy-weaving problem with P or F as a PDS, or if we construct $\overline{\mathsf{S}}$ as a PDS, we can define a POLWEAVE problem from it. However, we cannot in general represent H as a PDS, as the language of bad plays V produced by the reduction is defined from H and $\overline{\mathsf{H}}$. Furthermore, because V is defined by an intersection of languages defined by different input languages, we may define at most one input language as a PDS. Under these restrictions, V may be represented as a PDS, and from V we may construct a game as an alternating PDS, and solve the game using an algorithm given in [7]. Solving such games is EXPTIME-complete. However, with a careful reduction, we can ensure that the complexity of the policy-weaving problem is exponential in only the size of the goal states of Player 0 (i.e., the policy), and only polynomial in the size of the program.

We may also represent the program, host, and policies in a POLWEAVE problem all as NWAs, and represent V as an NWA, as NWAs are closed under language complement and intersection. From V, we may construct a game represented as an alternating NWA, and solve the game using an algorithm given in [23].

However, while our algorithm for solving POLWEAVE may be extended directly to handle stack-based languages, our algo-

rithm for solving symbolic weaving problems (§3.3) cannot, as the algorithm specifically searches for a game strategy represented as a finite-state machine of some bounded size. It seems non-trivial to extend this approach to symbolically search for bounded strategies to games defined by alternating PDSs or alternating NWAs. We leave this as future work.

When solving instances of the PARTWEAVE problem (§3.5) defined over stack-based languages, we encounter problems similar to those found in extending symbolic weaving to stack-based languages. Solving PARTWEAVE over stack-based languages is actually even more complicated than symbolic weaving over stack-based languages, as to solve a PARTWEAVE problem, we must find a set of languages (from §3.5, the L_i^j and W_i) and their complements. Thus we cannot, in general, find solutions where each language is a context-free grammar, although we could feasibly search for solutions in which every language is an NWA, provided that we can derive a technique to find NWAs with bounded representations that satisfy the language equations given in §3.5.

4. Practical Instances of Policy Weaving

In this section, we demonstrate that the policy-weaving problem formalizes the problem of instrumenting programs to run on real-world privilege-aware hosts. In particular, we formalize the Capsicum, HiStar, Asbestos, and Flume privilege-aware systems as languages from which one can define policy-weaving problems. We formally model each system as a symbolic acceptor. We then define a sound abstraction of the system as a symbolic acceptor whose transition relation may be described in quantifier-free SMT theories. Finally, we describe real-world programs that may be instrumented to execute securely on the system, along with their security and functionality policies.

These formalizations demonstrate that (i) our notion of a host introduced in §3.1 is expressive enough to describe the primitives and privilege events of a wide variety of real-world privilege aware systems, and (ii) our notion of security and functionality policies (§3.1) is expressive enough to describe the correctness requirements that real-world application programmers have for their programs. This exercise also allows us to evaluate qualitatively the complexity of programming for privilege-aware hosts. A programmer that manually instruments their program to invoke the primitives provided by a privilege-aware host must understand, at some level, the formalized semantics of the host in order to write a correct application for the host.

4.1 Concrete Model of Capsicum

§1 and §2 introduced the Capsicum host, giving an informal description of Capsicum's primitive operations, and demonstrating how a real-world application, `tcpdump`, can invoke the primitives to enforce desired security and functionality policies. §3.1 solved the weaving problem for hosts represented symbolically as automata whose runs are a language relating sequences of program commands and host primitives to the system traces that they induce. This section describes Capsicum as such an automaton.

By Defn. 2, a policy-weaving problem is defined for alphabets of program commands C, host primitives D, and privilege events E by a program $\mathsf{P} \subseteq \mathsf{C}^*$, policies $\mathsf{S}, \mathsf{F} \subseteq (\mathsf{C} \times \mathsf{E})^*$, and a host language $\mathsf{H} \subseteq ((\mathsf{C} \times \mathsf{D}) \times \mathsf{E})^*$. The language H thus describes how the host responds to commands and primitives and allows privilege events for a fixed program P.

The Capsicum language for P is defined over an alphabet C of program commands of P, an alphabet D of host primitives that Capsicum provides to P, and an alphabet E of privilege events induced by P. Let the alphabet of program commands C be the following. Assume that P is given as a finite set of *process modules*. Each process module proc executes in its own process space, and thus during execution, Capsicum maintains separate rights for each module. proc defines an alphabet of program commands $\mathsf{C}_{\mathsf{proc}}$, with $\mathsf{C} = \bigcup_{\mathsf{proc}} \mathsf{C}_{\mathsf{proc}}$, and a set of descriptors $\mathsf{Descs}_{\mathsf{proc}}$, defined as follows. Let there be a fixed set of descriptors $\mathsf{U}_{\mathsf{proc}} \subseteq \mathsf{Descs}_{\mathsf{proc}}$ that correspond to STDIN, STDOUT, and STDERR on a standard UNIX system. Let there be a finite set of callsites to `open` in module proc, represented as $O_{\mathsf{proc}} \subseteq \mathsf{C}_{\mathsf{proc}}$. Then for each callsite $\mathsf{open}_j \in O_{\mathsf{proc}}$ and $k \geq 0$, $\mathsf{desc}(j,k) \in \mathsf{Descs}_{\mathsf{proc}}$ is the kth descriptor opened at callsite j. For descriptor $\mathsf{desc} \in \mathsf{Descs}_{\mathsf{proc}}$, let there be program commands $(\mathsf{proc}, \mathsf{open}(\mathsf{desc})), (\mathsf{proc}, \mathsf{close}(\mathsf{desc})) \in \mathsf{C}_{\mathsf{proc}}$. Capsicum treats every program command that does not open or close a descriptor the same, so let there be one program command $\mathsf{step} \in \mathsf{C}_{\mathsf{proc}}$ not of the form open or close.

Let the alphabet of host primitives D of Capsicum for P be as follows. For each process module proc, let there be a host primitive that places proc in capability mode, $(\mathsf{proc}, \mathsf{cm}) \in \mathsf{D}$, and a primitive that has no effect on the Capsicum state $(\mathsf{proc}, \mathsf{noop}) \in \mathsf{D}$. Let the set of all access rights be supported by Capsicum by R. For each descriptor $\mathsf{desc} \in \mathsf{Descs}$, and set of rights $S \subseteq \mathsf{R}$, let there be a primitive that limits proc to only have the rights of in S for desc: $(\mathsf{proc}, \mathsf{lim}(\mathsf{desc}, S)) \in \mathsf{D}$.

Let the alphabet of privilege events E of Capsicum for P be as follows. For each process module proc, let there be a privilege event denoting that proc may open descriptors into its environment $(\mathsf{proc}, \mathsf{allow}(\mathsf{ENV})) \in \mathsf{E}$, and for each descriptor $\mathsf{desc} \in \mathsf{Descs}$, and right $\mathsf{r} \in \mathsf{R}$, let there be a privilege event denoting that proc may access desc with right r: $(\mathsf{proc}, \mathsf{allow}(\mathsf{desc}, \mathsf{r})) \in \mathsf{E}$. Finally, let there be a privilege event that corresponds to no particular right $(\mathsf{proc}, \mathsf{null}) \in \mathsf{E}$.

For alphabets C, D, and E, the language of Capsicum for P may be defined as the traces of a symbolic acceptor (see Defn. 7) $\mathsf{H}_{\mathsf{Cap}} = (\mathcal{T}, \mathsf{S}, s_0, A, \Sigma, \rho)$. The theory $\mathcal{T}$ of Capsicum for P is first-order, with conjunction, disjunction, negation, and quantification. Formulas in $\mathcal{T}$ are interpreted over a domain that contains elements corresponding to the elements of alphabets C, D, and E, an identifier for each process module in P, each right in r, each set of rights, and each descriptor in Descs. The constants of $\mathcal{T}$ are the constant $\emptyset$ for the empty set of rights, and the constant R for the set of all rights. The functions of $\mathcal{T}$ are intersection over sets of rights, and constructors that construct elements in the alphabets. The predicates of $\mathcal{T}$ are an equality predicate and set membership. Functions and predicates not corresponding to sets of rights are interpreted as their corresponding elements in the domain of $\mathcal{T}$, and the set functions and predicates are axiomatized by a standard set theory. In all following logical formulas, both in the theory of Capsicum and in other theories, let all logical variables that are not explicitly quantified be existentially quantified, and for formulas φ_0, φ_1, and φ_2, let the formula $\mathsf{ite}(\varphi_0, \varphi_1, \varphi_2) \equiv (\varphi_0 \Rightarrow \varphi_1) \land (\neg\varphi_0 \Rightarrow \varphi_2)$. Let any variable using the symbol "proc" range over the set of process modules, and let any variable using the symbol "desc" range over the set of descriptors.

Each state in the state-space S is either the stuck state, or maps each process module proc to a Boolean flag denoting if proc is in capability mode, and a mapping from each descriptor to the rights that proc holds for the descriptor. Thus, S is the domain

$$\mathsf{S} = \mathsf{Procs} \to (\mathbb{B} \times (\mathsf{Descs}_{\mathsf{P}} \to \mathcal{P}(\mathsf{R})))$$

As shorthand, denote the capability mode of process proc in state s as $\mathsf{cm}(s, \mathsf{proc}) = \pi_1(s(\mathsf{proc}))$, and denote the rights of process proc in state s as $\mathsf{rights}(s, \mathsf{proc}) = \pi_2(s(\mathsf{proc}))$.

The initial state $s_0 \in \mathsf{S}$ is the unique state in which every process is not in capability mode, and for each process, each descriptor has no rights:

$$\forall \mathsf{proc} : \neg\mathsf{cm}(s_0, \mathsf{proc}) \land \forall \mathsf{desc} : \mathsf{rights}(s_0, \mathsf{proc})(\mathsf{desc}) = \emptyset$$

The accepting states A are all non-stuck states of $\mathsf{H_{Cap}}$.

The alphabet Σ of Capsicum for P is the alphabet of all program commands paired with host primitives, paired with all privilege events: $\Sigma = (\mathsf{C} \times \mathsf{D}) \times \mathsf{E}$.

The transition relation ρ is a ternary predicate $\rho(s, ((c, d), e), s')$ satisfied by pre-state s, program command c, host primitive d, privilege event e, and post-state s' that satisfy the following conjunction of constraints. The first constraints define what privilege events are allowed in each transition. A process proc is allowed to access descriptor desc with right r only if in s, proc has right r for desc:

$$e = (\mathsf{proc}, \mathsf{allow}(\mathsf{desc}, \mathsf{r})) \implies \mathsf{r} \in \mathsf{rights}(s, \mathsf{proc})(\mathsf{desc}) \quad (18)$$

proc is allowed to access the environment only if proc is not in capability mode:

$$e = (\mathsf{proc}, \mathsf{allow}(\mathsf{ENV})) \implies \neg\mathsf{cm}(s, \mathsf{proc}) \quad (19)$$

The null event is always allowed for every process:

$$e = (\mathsf{proc}, \mathsf{null}) \implies \mathsf{True}$$

The next constraints define how $\mathsf{H_{Cap}}$ responds to each program command c. Let $\mathsf{post} : \mathsf{Procs} \to \mathsf{Descs_P} \to \mathcal{P}(\mathsf{R})$ map each process proc to a map from each descriptor to the set of rights that proc has after executing program command $c \in \mathsf{C}$. First, if a process proc opens a descriptor desc, then $\mathsf{H_{Cap}}$ responds by giving proc all rights for desc.

$$\begin{aligned} c = &(\mathsf{proc}, \mathsf{open}(\mathsf{desc})) \implies \\ &\mathsf{ite}(\mathsf{cm}(s, \mathsf{proc}), \\ &\quad \mathsf{post}(\mathsf{proc})(\mathsf{desc}) = \mathsf{rights}(s, \mathsf{proc})(\mathsf{desc}), \\ &\quad \mathsf{post}(\mathsf{proc})(\mathsf{desc}) = \mathsf{R}) \end{aligned} \quad (20)$$

If proc closes descriptor desc, then $\mathsf{H_{Cap}}$ responds by clearing all of the rights that proc has for desc.

$$\begin{aligned} c = &(\mathsf{proc}, \mathsf{close}(\mathsf{desc})) \implies \\ &\mathsf{post}(\mathsf{proc}, \mathsf{desc}) = \emptyset \end{aligned} \quad (21)$$

If proc executes a command other than opening or closing a file, then the command does not change the rights of any process for any descriptor:

$$\begin{aligned} &c = (\mathsf{proc}, \mathsf{step}) \implies \quad (22) \\ &\forall \mathsf{proc} : \mathsf{post}(\mathsf{proc}) = \mathsf{rights}(s, \mathsf{proc}) \end{aligned}$$

The next constraints define how $\mathsf{H_{Cap}}$ responds to each host primitive in D. If proc requests to place itself in capability mode, $\mathsf{H_{Cap}}$ places proc in capability mode.

$$d = (\mathsf{proc}, cm) \implies \mathsf{cm}(s', \mathsf{proc})$$

If proc requests to limit its rights for descriptor desc to S, then $\mathsf{H_{Cap}}$ limits the rights of proc for desc to S intersected with whatever rights that proc had for desc before requesting to limit its rights.

$$\begin{aligned} d = &(\mathsf{proc}, \mathsf{lim}(\mathsf{desc}, S)) \implies \\ &\mathsf{rights}(s', \mathsf{proc})(\mathsf{desc}) = \mathsf{post}(\mathsf{proc})(\mathsf{desc}) \cap S \end{aligned} \quad (23)$$

Finally, if proc invokes the host primitive noop, then the state of $\mathsf{H_{Cap}}$ does not change:

$$d = \mathsf{noop} \implies \mathsf{rights}(s') = \mathsf{post} \wedge \mathsf{cm}(s') = \mathsf{cm}(s) \quad (24)$$

Unless a constraint explicitly requires that the rights or capability mode of a process proc change from s to s', then the rights and capability mode of proc do not change from s to s'. ρ is the conjunction of constraints, (18)–(24).

4.1.1 An Abstract Model of Capsicum

The concrete model of Capsicum $\mathsf{H_{Cap}}$ defined in §4.1 directly corresponds to the model of Capsicum described by the Capsicum developers [32]. However, if $\mathsf{H_{Cap}}$ is given as a component to a symbolic weaving problem, then existing SMT solvers will not be able to solve the weaving problem, because the transition relation of $\mathsf{H_{Cap}}$ is defined by formulas that quantify universally over the set of all descriptors created dynamically by a program, and this set is unbounded. To resolve this, we define an abstract model of Capsicum $\mathsf{H_{Cap}}^{\#}$ that soundly abstracts the concrete model $\mathsf{H_{Cap}}$. The key idea behind the definition of $\mathsf{H_{Cap}}^{\#}$ is that for each callsite o to `open` in P, $\mathsf{H_{Cap}}^{\#}$ keeps precise information about the capabilities of a process for only the most recent descriptor created at o, and approximate information about each process's capabilities for all other descriptors created at o. This approximate information may be represented without unbounded universal quantification.

We now define $\mathsf{H_{Cap}}^{\#} = (\mathcal{T}, \mathsf{S}, s_0, A, \Sigma, \rho)$ as a symbolic acceptor (Defn. 7). The theory $\mathcal{T}$ of $\mathsf{H_{Cap}}^{\#}$ is the same as the theory of the concrete model of Capsicum, but with only bounded universal quantification.

Each state in S is either a stuck state, or maps each process proc in P to a Boolean flag denoting if the process is in capability mode, and a map from each `open` callsite o to the index of the descriptor isolated for o, a lower bound on the set of rights that proc has for all summarized descriptors allocated at o, and an upper bound on the set of rights that proc has for all summarized descriptors allocated at o. S is thus the domain

$$\mathsf{Procs} \to (\mathbb{B} \times (O \to (\mathbb{N} \times \mathcal{P}(\mathsf{R})) \times \mathcal{P}(\mathsf{R})^2))$$

As shorthand, in addition to the function cm defined for $\mathsf{H_{Cap}}$, define the following functions for describing components of a state. For state s, process module proc, and `open` callsite o, let the index of the descriptor isolated for process proc for callsite o be $\mathsf{isoindx}(s, \mathsf{proc}, o) = \pi_1(\pi_1(\pi_2(s(\mathsf{proc}))(o)))$, let the capabilities of proc for the descriptor isolated for callsite o be $\mathsf{isocaps}(s, \mathsf{proc}, o) = \pi_2(\pi_1(\pi_2(s(\mathsf{proc}))(o)))$, let the lower-bound of capabilities of proc for all summarized descriptors allocated at o be $\mathsf{sumlb}(s, \mathsf{proc}, o) = \pi_1(\pi_2(\pi_2(s(\mathsf{proc}))(o)))$, and let the upper-bound of capabilities of proc for all summarized descriptors allocated at o be $\mathsf{sumub}(s, \mathsf{proc}, o) = \pi_2(\pi_2(\pi_2(s(\mathsf{proc}))(o)))$.

The initial state s_0 isolates no descriptor for any process, and maps each process to the highest possible lower bound and lowest possible upper bound. s_0 is the unique state that satisfies the formula over the free variable s:

$$\begin{aligned} \forall \mathsf{proc} : \neg\mathsf{cm}(s, \mathsf{proc}) \wedge \forall o : &\mathsf{isoindx}(s, \mathsf{proc}, o) = 0 \\ &\wedge \mathsf{isocaps}(s, \mathsf{proc}, o) = \emptyset \\ &\wedge \mathsf{sumlb}(s, \mathsf{proc}, o) = \mathsf{R} \\ &\wedge \mathsf{sumub}(s, \mathsf{proc}, o) = \emptyset \end{aligned}$$

The accepting states A are all non-stuck states.

The alphabet $\mathsf{H_{Cap}}^{\#}$ is $\Sigma = (\mathsf{C} \times \mathsf{D}) \times (\mathsf{E} \cup \overline{\mathsf{E}})$, for C, D, and E defined in §4.1. Note that Σ is different from the alphabet of $\mathsf{H_{Cap}}$, which is $(\mathsf{C} \times \mathsf{D}) \times \mathsf{E}$). $\mathsf{H_{Cap}}^{\#}$ actually abstracts the closures of $\mathsf{H_{Cap}}$, not $\mathsf{H_{Cap}}$ itself (see §3.4.2).

The transition relation $\rho(s, ((c, d), e), s')$ is a conjunction of the following constraints, over free variables s the pre-state of the transition, $c \in \mathsf{C}$ and $d \in \mathsf{D}$ the program command and host primitive executed by the instrumented program, $e \in \mathsf{E} \cup \overline{\mathsf{E}}$ a positive or negative privilege event that $\mathsf{H_{Cap}}^{\#}$ may allow, and s' a post-state. First, if the privilege event is positive, that is, if $e = (\mathsf{proc}, \mathsf{allow}(\mathsf{desc}(o, i), \mathsf{r}))$, then $\mathsf{H_{Cap}}^{\#}$ allows e if the descriptor $\mathsf{desc}(o, i)$ is isolated for proc and proc has the right r for $\mathsf{desc}(o, i)$, or if $\mathsf{desc}(o, i)$ is summarized, and proc may have r

for $\mathsf{desc}(o, i)$:

$$e = (\mathsf{proc}, \mathsf{allow}(\mathsf{desc}(o, i), \mathsf{r})) \implies \mathsf{ite}(\mathsf{isoindx}(s, \mathsf{proc}, o) = i, \mathsf{r} \in \mathsf{isocaps}(s, \mathsf{proc}, o), \mathsf{r} \in \mathsf{sumub}(s, \mathsf{proc}, o))$$

Next, if the privilege event is negative, that is, if $e = (\mathsf{proc}, \overline{\mathsf{allow}(\mathsf{desc}(o, i), \mathsf{r})})$, then $\mathsf{H_{Cap}}^{\#}$ allows e if $\mathsf{desc}(o, i)$ is isolated for proc and proc does not have the right r for $\mathsf{desc}(o, i)$, or if $\mathsf{desc}(o, i)$ is summarized, and proc may not have r for $\mathsf{desc}(o, i)$:

$$e = (\mathsf{proc}, \overline{\mathsf{allow}(\mathsf{desc}(o, i), \mathsf{r})}) \implies \mathsf{ite}(\mathsf{isoindx}(s, \mathsf{proc}, o) = i, \mathsf{r} \notin \mathsf{isocaps}(s, \mathsf{proc}, o), \mathsf{r} \notin \mathsf{sumlb}(s, \mathsf{proc}, o))$$

$\mathsf{H_{Cap}}^{\#}$ responds to program commands in C as follows. The function $\mathsf{post} : \mathsf{Procs} \to O \to (\mathbb{N} \times \mathcal{P}(\mathsf{R})) \times \mathcal{P}(\mathsf{R})^2$ maps each process to the rights for isolated and summarized descriptors after executing a program command. Suppose that a process proc opens a new descriptor at `open` callsite o by executing the program command $\mathsf{open}(o)$. Then $\mathsf{H_{Cap}}^{\#}$ isolates a new descriptor for proc for o, gives proc all rights for the new descriptor, and merges the rights for the previous descriptor isolated for o with the summary bounds for all other descriptors opened at o:

$$\begin{aligned} & c = (\mathsf{proc}, \mathsf{open}(o)) \implies \\ & \quad \mathsf{isoindx}(\mathsf{post}, \mathsf{proc}, o) = \mathsf{isoindx}(s, \mathsf{proc}, o) + 1 \\ & \wedge \mathsf{isocaps}(\mathsf{post}, \mathsf{proc}, o) = \mathsf{R} \\ & \wedge \mathsf{sumlb}(\mathsf{post}, \mathsf{proc}, o) = \mathsf{sumlb}(s, \mathsf{proc}, o) \cap \mathsf{isocaps}(s, \mathsf{proc}, o) \\ & \wedge \mathsf{sumub}(\mathsf{post}, \mathsf{proc}, o) = \mathsf{sumub}(s, \mathsf{proc}, o) \cup \mathsf{isocaps}(s, \mathsf{proc}, o) \end{aligned}$$

Suppose that a process proc closes a descriptor $\mathsf{close}(\mathsf{desc}(o, i))$ by executing the program command $\mathsf{close}(\mathsf{desc}(o, i))$. If $\mathsf{H_{Cap}}^{\#}$ currently isolates $\mathsf{desc}(o, i)$ for callsite o, then $\mathsf{H_{Cap}}^{\#}$ updates the set of rights for $\mathsf{desc}(o, i)$ to be empty. Otherwise, if $\mathsf{H_{Cap}}^{\#}$ does not isolate $\mathsf{desc}(o, i)$ for o, then $\mathsf{H_{Cap}}^{\#}$ notes that for all descriptors allocated at o, proc may have no rights:

$$c = (\mathsf{proc}, \mathsf{close}(\mathsf{desc}(o, i))) \implies \mathsf{ite}(\mathsf{isoindx}(s, \mathsf{proc}, o) = i), \mathsf{isocaps}(\mathsf{post}, \mathsf{proc}, o) = \emptyset, \mathsf{sumlb}(\mathsf{post}, \mathsf{proc}, o) = \emptyset)$$

All sets of rights not explicitly mentioned in the above constraints are unchanged between s and post.

$\mathsf{H_{Cap}}^{\#}$ responds to each host primitive that P may invoke as follows. If a process in P invokes cm, then $\mathsf{H_{Cap}}^{\#}$ responds similarly to how $\mathsf{H_{Cap}}$ responds to cm. If a process proc invokes lim on a descriptor $\mathsf{desc}(o, i)$, then $\mathsf{H_{Cap}}^{\#}$ updates the rights of proc for $\mathsf{desc}(o, i)$. If $\mathsf{desc}(o, i)$ is isolated, then $\mathsf{H_{Cap}}^{\#}$ precisely updates the rights for $\mathsf{desc}(o, i)$, and if $\mathsf{desc}(o, i)$ is summarized, then $\mathsf{H_{Cap}}^{\#}$ updates the rights for all descriptors summarized for o:

$$\begin{aligned} & d = (\mathsf{proc}, \mathsf{lim}((o, i), S)) \implies \\ & \mathsf{ite}(i = \mathsf{isoindx}(\mathsf{post}, \mathsf{proc}, o), \\ & \quad \mathsf{isocaps}(s', \mathsf{proc}, o) = \mathsf{isocaps}(\mathsf{post}, \mathsf{proc}, o) \cap S \\ & \quad \wedge \mathsf{sumlb}(s', \mathsf{proc}, o) = \mathsf{sumlb}(\mathsf{post}, \mathsf{proc}, o), \\ & \quad \mathsf{sumlb}(s', \mathsf{proc}, o) = \mathsf{sumlb}(\mathsf{post}, \mathsf{proc}, o) \cap S \\ & \quad \wedge \mathsf{isocaps}(s', \mathsf{proc}, o) = \mathsf{isocaps}(\mathsf{post}, \mathsf{proc}, o)) \end{aligned}$$

The rights of proc for all other callsites and the rights of all other processes are unchanged from post to s'.

4.1.2 Programs and Polices on Capsicum

We now describe problems of instrumenting real-world programs for Capsicum as policy-weaving problems. We describe each program as an automaton over program commands, and describe the security and functionality policies of the program as languages over program command and privilege events.

tcpdump In §2, we described the problem of instrumenting `tcpdump` for Capsicum.

gzip The problem of instrumenting the `gzip` compression tool for Capsicum can be expressed as a policy-weaving problem. At a high level, `gzip` executes as follows. `gzip` takes as input a set of input files and output file destinations, and a set of command-line arguments that configure details of its execution. `gzip` executes a small prologue of commands, and then enters a loop. In each iteration of the loop, `gzip` opens an output file for writing, opens an input file for reading, compresses or decompresses the input file, and outputs the result into the output file. `gzip` iterates through the loop until it processes all files.

The high-level model of `gzip` described above may be represented as a regular language over an alphabet of program commands. The alphabet of program commands for this simplified view of `gzip` is

$$\mathsf{C} = \{\text{setup, openout, openin, operate, iter}\}$$

The alphabet of host primitives D and alphabet of privilege events E are defined by `gzip` according to §4.1, for descriptors infile and outfile.

The executions of `gzip` can be expressed by the following regular language over the alphabet C:

$$\mathsf{P} = \text{setup . (iter . openout . openin . operate)}^*$$

The security policy of `gzip` can be expressed as a regular language over the alphabet $\mathsf{C} \times \mathsf{E}$. The compression or decompression operations executed by `gzip` in each iteration of its loop are complex, and historically, have allowed vulnerabilities. Thus for `gzip` to be secure, it may only read from an input file and write to an output file when executing the program command operate. Thus the security policy for `gzip` is

$$\begin{aligned} \mathsf{S} = & (((\mathsf{C} \backslash \{\text{operate}\}) \times \mathsf{E}) \\ & \mid (\{\text{operate}\} \times \{\mathsf{allow}(\mathsf{infile}, \mathsf{rd}), \mathsf{allow}(\mathsf{outfile}, \mathsf{wr})\}))^* \end{aligned}$$

The functionality policy of `gzip` can be expressed as a regular language over the alphabet $\mathsf{C} \times \mathsf{E}$. When `gzip` opens each input file and output file, it must be able to open descriptors in its execution environment. When `gzip` executes a compression or decompression operation, it must be able to read from the input file, and write to the output file.

$$\begin{aligned} \mathsf{F} = & ((\mathsf{C} \times \mathsf{null}) \mid ((\text{openout} \times \mathsf{ENV}) \mid (\text{openin} \times \mathsf{ENV}) \\ & \mid (\{\text{operate}\} \times \{\mathsf{allow}(\mathsf{infile}, \mathsf{rd}), \mathsf{allow}(\mathsf{outfile}, \mathsf{wr})\}))^*) \end{aligned}$$

`gzip` cannot be instrumented to satisfy its S and F without being partitioned into multiple processes. To see this, note that `gzip` may execute the following sequence of program commands:

$$\text{setup iter openout openin operate iter openout}$$

For `gzip` to satisfy F, it must be able to access its environment each time that it executes openout. However, for `gzip` to be able to access its environment when it executes openout a second time, it must be able to access its environment when it executes operate, by the rules of Capsicum. But if `gzip` can access its environment when it executes operate, then it violates S. Thus, if we apply the policy-weaving algorithm to a policy-weaving problem constructed from P, S, and F, the algorithm will produce a winning strategy for Player 0. A `gzip` developer can use the strategy to partition `gzip` into multiple process modules, with the operate program command executing in a separate module from the rest of `gzip`. If a policy-weaving problem constructed from the repartitioned `gzip`

is given to the policy-weaving algorithm, then the algorithm finds a correct instrumentation of `gzip`.

dhclient `dhclient` configures network devices on a system using the Dynamic Host Configuration Protocol (DHCP). `dhclient` first reads configuration options in a configuration file. It then fetches a list of network devices on its host system. For each device dev in the list, `dhclient` opens a descriptor to dev, over which it configures dev.

The alphabet of program commands for this high-level model of `dhclient` is

$$\mathsf{C} = \{\text{config, listdev, opendev, cfgdev, iter}\}$$

The alphabets of host primitives D and privilege events E for `dhclient` are defined according to §4.1, with a descriptor dev.

The executions of the high-level model of `dhclient` correspond to strings in the following regular language over the alphabet C:

$$\mathsf{P} = \text{config . listdev . (iter . opendev . cfgdev)}^*$$

A security policy for `dhclient` can be expressed as regular language over the alphabet $\mathsf{C} \times \mathsf{E}$. When `dhclient` configures each network device, it must use complex code to parse input packets given in a complex format. Thus when `dhclient` executes cfgdev, it must only be able to read and write to the current device. Thus the security policy for `dhclient` is

$$\mathsf{S} = ((\mathsf{C} \backslash \{\text{cfgdev}\}) \times \mathsf{E}) | (\{\text{cfgdev}\} \times \{\mathsf{allow}(\mathsf{dev}, \mathsf{rd}), \mathsf{allow}(\mathsf{dev}, \mathsf{wr})\})^*$$

A functionality policy for `dhclient` can be expressed as a regular language over the alphabet $\mathsf{C} \times \mathsf{E}$. When `dhclient` configures itself, it must be able to access the execution environment to open various configuration files. When `dhclient` lists network devices, it must be able to access its execution environment. Finally, when `dhclient` configures each network device, it must be able to open the descriptor for each device and read from and write to the descriptor. Thus the functionality policy for `dhclient` is

$$\begin{aligned}\mathsf{F} =& ((\text{config} \times \mathsf{ENV}) \mid (\text{listdev} \times \mathsf{ENV}) \mid ((\text{iter} \times \mathsf{ENV}) \\ & \mid (\text{opendev} \times \mathsf{ENV}) \mid (\{\text{cfgdev}\} \times \{\mathsf{rd}(\mathsf{dev}), \mathsf{wr}(\mathsf{dev})\}))^*\end{aligned}$$

As with `tcpdump` and `gzip`, `dhclient` cannot satisfy its security and functionality policies without being partitioned to execute in multiple process modules.

wget `wget` downloads data from a server to a local host. `wget` first configures itself according to its command-line arguments. Next, `wget` iterates in a loop. In each iteration, `wget` opens a specified file for output, accesses a network device to retrieve the data for the file from the network, fetches the data from over the network device, and writes the data to the specified output file.

The alphabet of program commands for this high-level model of `wget` is:

$$\mathsf{C} = \{\text{setup, openout, connect, retr, iter}\}$$

The alphabet of host primitives D and privilege events E are as defined in §4.1 from the descriptors socket and outfile.

The executions of the high-level model of `wget` correspond to strings of the regular language

$$\mathsf{P} = \text{setup . (iter . openout . connect . retr)}^*$$

The security policy of `wget` may be described as a regular language over the alphabet $\mathsf{C} \times \mathsf{E}$. The code that `wget` executes to retrieve data from the network is complex, and vulnerable to compromise by a malicious input. Thus, when `wget` executes retr, it must only be able to read from and write to socket, and write to outfile. Thus the security policy is

$$\begin{aligned}\mathsf{S} =& (((\mathsf{C} \backslash \{\text{retr}\}) \times \mathsf{E}) \\ & \mid (\{\text{retr}\} \times \{\mathsf{allow}(\mathsf{socket}, \mathsf{rd}), \mathsf{allow}(\mathsf{socket}, \mathsf{wr}), \mathsf{allow}(\mathsf{outfile}, \mathsf{wr})\}))^*\end{aligned}$$

The functionality policy of `wget` may be described as a regular language over the alphabet $\mathsf{C} \times \mathsf{E}$. When `wget` opens files for output or connects to a resource over the network, it must be able to access its execution environment. Also, when `wget` retrieves data from each URL, it must be able to read from and write to the network socket, and write to the output file. Thus the functionality policy is

$$\begin{aligned}\mathsf{F} =& ((\mathsf{C} \times \{\mathsf{null}\}) \mid (\text{openout} \times \mathsf{ENV}) \mid (\text{connect} \times \mathsf{ENV}) \\ & \mid (\{\text{retr}\} \times \{\mathsf{allow}(\mathsf{socket}, \mathsf{rd}), \mathsf{allow}(\mathsf{socket}, \mathsf{wr}), \mathsf{allow}(\mathsf{outfile}, \mathsf{wr})\}))^*\end{aligned}$$

As with `tcpdump`, `gzip`, and `dhclient`, `wget` cannot be instrumented to satisfy its security and functionality policies without being partitioned to execute in multiple process modules.

4.2 HiStar

HiStar is a Decentralized Information Flow Control (DIFC) operating system [33]. HiStar, like most DIFC systems, assigns a label to each process executing on it. When one process attempts to communicate information to another process, HiStar interposes on the communication, and only allows the communication if the labels of the processes satisfy a particular relationship. The developers of HiStar have rewritten several real-world programs to securely run on HiStar, including the ClamAV [8] virus scanner and the OpenVPN [28] VPN client. In this section, we first formalize HiStar as a symbolic acceptor $\mathsf{H}_\star$, according to the description provided by its developers [33]. The transition relation of $\mathsf{H}_\star$ is defined by unbounded universal quantification, and thus $\mathsf{H}_\star$ is not be a suitable input for our symbolic weaving algorithm. Thus, we all also present a sound abstraction $\mathsf{H}_\star{}^\#$ of $\mathsf{H}_\star$ that is a suitable input to the symbolic weaving algorithm. Finally, we will describe the problem of instrumenting several real-world programs for HiStar as policy-weaving problems.

4.2.1 Concrete Model of HiStar

We may describe HiStar as a symbolic acceptor $\mathsf{H}_\star = (\mathcal{T}, \mathsf{S}, s_0, A, \Sigma, \rho)$. As with the Capsicum symbolic acceptor $\mathsf{H}_{\mathsf{Cap}}$, $\mathsf{H}_\star$ describes how HiStar allows privilege events in response to the program commands and host primitives executed by a fixed application P. We assume that each such P is represented by a finite set of process modules, where each process module proc is represented as an automaton over an alphabet of program commands.

The program commands of $\mathsf{H}_\star$, C, are the program commands of each process module in P. The model $\mathsf{H}_\star$ reacts to each program command identically, so we assume that there is one program command (proc, step) for each process module proc that denotes that proc takes a step of execution.

The privilege events of HiStar, E, are defined as follows. For every two process modules proc_1 and proc_2 in P, let there be a privilege event $\mathsf{allow}(\mathsf{proc}_1, \mathsf{proc}_2)$ that denotes that information is allowed to flow from proc_1 to proc_2.

To define the alphabet of host primitives D, we first define process labels. For every process, HiStar maintains a label, which HiStar uses to decide if each process may send information to another. HiStar provides to P a set of host primitives that allow each process in P to change its label. Each process's label maps each *category* allocated by P to a *level*. We define the set of categories by assuming a set of *bins* B, with each bin $b(i) \in B$ defined by some natural number. A category $\mathsf{cat}(b, i) \in \mathsf{Cats}$ is then defined by a bin b and a natural number i as index into the bin, that denotes in what order the category is allocated. The model of HiStar presented in [33] has no notion of bins: the processes in a program simply allocate a categories. Bins are instead a part of our framework for instrumenting programs to run in HiStar. However, clearly there is a direct correspondence between a system that constructs a category from a single natural number index, and a system that constructs a

category from a pair of natural number as bin number and natural number as index into the bin.

A process's label maps each category to a level, which denotes the highest confidentiality, or taint, of information that the process has read with respect to the given category. Let the set of levels L be the terms $\mathsf{level}(i)$ for $0 \leq i \leq 3$, which denote increasing levels of taint for a category, and the term $\star$, which denotes declassification privilege in a category.

Let the alphabet of host primitives D be as follows. For each process proc and bin b, let there be a host primitive $(\mathsf{proc}, \mathsf{make_cat}(b)) \in$ D that requests $\mathsf{H}_\star$ to allocate a new category in b. For each category cat and level l, let there be a host primitive $(\mathsf{proc}, \mathsf{set_taint}(l)) \in$ D that requests $\mathsf{H}_\star$ to set the level of cat for proc to l. Finally, let there be a host primitive $\mathsf{noop} \in \mathsf{D}$ that has no effect on any label of any process.

Let the theory of HiStar $\mathcal{T}$ be a first-order logic with conjunction, disjunction, negation and quantification. Each formula in $\mathcal{T}$ is interpreted over a domain that contains elements corresponding to elements of the alphabets C, D, and E, an identifier for each process module in P, each category that may be allocated, and each level. There is one constant in $\mathcal{T}$ for each level. The predicates of $\mathcal{T}$ are equality and $\sqsubseteq$, which is a binary predicate over labels, axiomatized below.

Each state in S, the state-space of $\mathsf{H}_\star$, is either the stuck state stuck, or a map from each process in P to a partial map from each category to a level, and a map from each bin to the number of categories allocated in the bin:

$$\mathsf{S} = (\mathsf{Procs} \to \mathsf{Cats} \to L) \times (B \to \mathbb{N})$$

As shorthand, define the following functions for describing components of a state. Let s be a non-stuck state. Let $\mathsf{label}(s, \mathsf{proc}) = \pi_1(s)(\mathsf{proc})$ be the map in s from each process to a label, and let $\mathsf{count}(s) = \pi_2(s)$ be the map in s from each bin to the count of categories allocated in the bin.

The initial state of $\mathsf{H}_\star$, s_0, is the function that maps each process to an empty label, and in which no categories have been allocated in any bin.

$$\forall \mathsf{proc} : \mathsf{label}(s_0, \mathsf{proc}) = \emptyset \wedge \forall b : \mathsf{count}(s_0)(b) = 0$$

The set of accepting states A is the set of a states that are not the stuck state.

The alphabet Σ is the product of the program commands, host commands, and privilege events: $\Sigma = (\mathsf{C} \times \mathsf{D}) \times \mathsf{E}$.

The transition relation of $\mathsf{H}_\star$, $\rho(s, ((c, d), e), s')$, is a conjunction of the following constraints. To define the constraints, we first define a flow relation $\sqsubseteq_{lev}$ over levels, and from it, a flow relation over labels. A category cat allows information to flow from a sending process proc_s to a receiving process proc_r if the level of cat for proc_s is at least as low as the level of cat for proc_r, or if the level of cat for either proc_s or proc_r is the declassification level $\star$. Define a binary flow relation $\sqsubseteq_{lev}$ over levels as:

$$l_1 \sqsubseteq_{lev} l_2 \iff (l_1 = \mathsf{level}(n) \wedge l_2 = \mathsf{level}(m) \wedge n \leq m) \vee l_1 = \star \vee l_2 = \star$$

Note that the relation $\sqsubseteq_{lev}$ is not an ordering (it is not anti-symmetric), even though the symbol "$\sqsubseteq$" is typically used to denote an order. We use the symbol to be consistent with [33].

In HiStar, a process proc_s may send information to a process proc_r if for each category cat that has been allocated by P, the level of cat for proc_s and the level of cat for proc_r satisfy the flow relation. Define a binary flow relation $\sqsubseteq$ over two labels l_1 and l_2 as:

$$l_1 \sqsubseteq l_2 \iff \forall \mathsf{cat} : l_1(\mathsf{cat}) \sqsubseteq_{lev} l_2(\mathsf{cat})$$

$\sqsubseteq$ defines when $\mathsf{H}_\star$ allows a privilege event. $\mathsf{H}_\star$ allows a process proc_s to send information to process proc_r only if proc_s and proc_r satisfy the flow relation:

$$e = \mathsf{allow}(\mathsf{proc}_s, \mathsf{proc}_r) \implies \mathsf{label}(s, \mathsf{proc}_s) \sqsubseteq \mathsf{label}(s, \mathsf{proc}_r)$$

$\mathsf{H}_\star$ responds to program commands executed by P as follows. Each program command is of the form $(\mathsf{proc}, \mathsf{step})$ for some proc in P. $\mathsf{H}_\star$ allows each such program command:

$$c = (\mathsf{proc}, \mathsf{step}) \implies \mathsf{True}$$

$\mathsf{H}_\star$ responds to host primitives invoked by P as follows. If a process proc_i in P allocates a new category in a bin, then for the new category, $\mathsf{H}_\star$ gives proc_i the declassification level, and gives every other process the default label of 1:

$$\begin{aligned} d =& (\mathsf{proc}_i, \mathsf{make_cat}(b)) \implies \\ & \mathsf{label}(s', \mathsf{proc}_i)(\mathsf{cat}(b, \mathsf{count}(b)) = \star \\ & \wedge \forall \mathsf{proc}_j \neq \mathsf{proc}_i : \mathsf{label}(s', \mathsf{proc}_j)(\mathsf{cat}(b, \mathsf{count}(b)) = 1 \\ & \wedge \wedge \mathsf{count}(s')(b) = \mathsf{count}(s)(b) + 1 \end{aligned}$$

If a process proc_i requests to set its level for a category cat to l_n, then HiStar sets the level of cat for proc_i to l_n, provided that the that old level l_o of proc_i for cat and l_n satisfy the "can-change relation" $\sqsubseteq_\delta$:

$$l_o \sqsubseteq_\delta l_n \iff l_o \sqsubseteq l_n \wedge (l_n \neq \star \vee l_o = \star)$$

Suppose that P requests to set the level of category cat to level l_n. Then $\mathsf{H}_\star$ sets the level of cat to l_n only if for l_o the current level of cat l_o can change to l_n:

$$\begin{aligned} d =& (\mathsf{proc}, \mathsf{set_taint}(\mathsf{cat}(b, i), l)) \implies \\ & \mathsf{ite}(\mathsf{label}(s, \mathsf{proc})(\mathsf{cat}) \sqsubseteq_\delta l, \\ & \quad \mathsf{label}(s', \mathsf{proc})(\mathsf{cat}) = l, \\ & \quad \mathsf{label}(s', \mathsf{proc})(\mathsf{cat}) = \mathsf{label}(s, \mathsf{proc})(\mathsf{cat})) \end{aligned}$$

Finally, if P invokes the host primitive, then the state of $\mathsf{H}_\star$ does not change:

$$d = (\mathsf{proc}, \mathsf{noop}) \implies s = s'$$

4.2.2 An Abstract Model of HiStar

The model of HiStar $\mathsf{H}_\star$ defined in §4.2.1 directly corresponds to the description of HiStar given by its developers. However, we cannot solve symbolic policy-weaving problems defined over $\mathsf{H}_\star$ by applying an SMT solver, as the transition relation ρ of $\mathsf{H}_\star$ is defined by formulas that are universally quantified over the set of all categories that may be allocated by a program P. In general, this set of categories may be unbounded. To resolve this issue, we define a host $\mathsf{H}_\star{}^\#$ from which one can define policy-weaving problems that may be solved using an SMT solver. There are sound abstractions of $\mathsf{H}_\star$ that are coarser or finer than $\mathsf{H}_\star{}^\#$. However, the definition of $\mathsf{H}_\star{}^\#$ is based on a how expert programmers write programs for HiStar. In particular, while HiStar allows a program to allocate arbitrarily many categories, expert HiStar programs typically write their real-world program to use only a small number of categories. Analogously, $\mathsf{H}_\star{}^\#$ maintains precise information for, or *isolates*, a small set of categories, and combines the precise information for the isolated categories with approximate but sound information about the non-isolated, or *summarized* categories, to ensure that a program is secure.

Let $\mathsf{H}_\star{}^\# = (\mathcal{T}, \mathsf{S}, s_0, A, \rho)$ be a symbolic acceptor that defines a language in $((\mathsf{C} \times \mathsf{D}) \times (\mathsf{E} \cup \bar{\mathsf{E}}))^*$. The theory $\mathcal{T}$ of $\mathsf{H}_\star{}^\#$ is a first-order theory with only bounded quantification. The formulas of $\mathcal{T}$ are interpreted in the semantics of *three-valued logic* [29]. We now give a brief background on three-valued logic sufficient to describe its use in defining $\mathsf{H}_\star{}^\#$. In three-valued logic, an interpretation

maps every predicate, and by extension, every formula, to one of three truth values in $\mathbb{B}_3$, intuitively corresponding to true (1), false (0), and unknown ($\frac{1}{2}$). We do not give the full semantics of three-valued logic, but define the following set of operators and predicates over truth values. For truth values x and y, $x \leq y$ yields a definite (two-valued) truth value that is true iff the numeric value for x is less than or equal to the numeric value for y. $x + y$ and $x - y$ produce the truth values from saturation arithmetic over the range $[0, 1]$. The *truth value least upper bound* $\sqcap^v$ is a binary function over truth values such that $x \sqcap^v y = \min(x, y)$. $\sqcap^v$ may be extended in the natural way to a function over arbitrary sets of truth values. The domain to which the constants and functions of $\mathcal{T}$ are interpreted is the same as the domain of the theory $\mathsf{H}_\star$.

Each state in the state-space S of $\mathsf{H}_\star{}^\#$ maps each bin to a counter of the category isolated for the bin and map from each process to its level for the isolated category, and a map from each pair of processes to a three-valued Boolean that denotes whether information can flow from the first process to the second process. Thus, S is the domain

$$\mathsf{S} = B \to ((\mathbb{N} \times (\mathsf{Procs} \to L)) \times (\mathsf{Procs}^2 \to \mathbb{B}_3))$$

As shorthand, let $\mathsf{isoindx}(s, b) = \pi_1(\pi_1(s(b)))$, let $\mathsf{isolabel}(s, b) = \pi_2(\pi_1(s(b)))$, and let $\mathsf{sum}(s, b) = \pi_2(s(b))$.

In the initial state s_0 of $\mathsf{H}_\star{}^\#$, no bin has an isolated category, and the summary flow relation allows information to flow from each process to each process:

$$\forall b : \mathsf{isoindx}(s, b) = 0 \land \mathsf{isolabel}(s, b) = \emptyset \\ \land \forall \mathsf{proc}_1, \mathsf{proc}_2 : \mathsf{sum}(s_0, b)(\mathsf{proc}_1, \mathsf{proc}_2) = 1$$

The accepting states of $\mathsf{H}_\star$ are all non-stuck states in S.

The transition relation $\rho(s, ((c, d), e), s')$ of $\mathsf{H}_\star{}^\#$ holds if and only if pre-state s, program command c, host primitive d, privilege event e, and post-state s' satisfy a conjunction of constraints. The constraints are defined using three-valued logic predicates and operators, but ρ itself is interpreted as a two-valued logic value. The constraints are defined in terms of a three-valued flow-relation $\sqsubseteq^\#$ that decides for each pair of process proc_1 and proc_2, either proc_1 definitely may send information to proc_2 (in which case, $\mathsf{proc}_1 \sqsubseteq^\# \mathsf{proc}_2 = 1$), or proc_1 definitely may not send information to proc_2 (in which case, $\mathsf{proc}_1 \sqsubseteq^\# \mathsf{proc}_2 = 0$), or $\mathsf{H}^\#$ cannot decide precisely if proc_1 can send information to proc_2 (in which case, $\mathsf{proc}_1 \sqsubseteq^\# \mathsf{proc}_2 = \frac{1}{2}$). $\mathsf{proc}_1 \sqsubseteq^\# \mathsf{proc}_2$ if and only if, for each bin b, the category isolated for b allows proc_1 to send information to proc_2, and the categories summarized for b allow proc_1 to send information to proc_2:

$$\mathsf{proc}_1 \sqsubseteq^\# \mathsf{proc}_2 = \prod_{b \in B}^{v} \mathsf{isolabel}(s, b)(\mathsf{proc}_1) \sqsubseteq \mathsf{isolabel}(s, b)(\mathsf{proc}_2) \\ \sqcap^v \mathsf{sum}(s, b)(\mathsf{proc}_1, \mathsf{proc}_2)$$

The $\sqsubseteq^\#$ relation defines when each privilege event $e \in \mathsf{E} \cup \overline{\mathsf{E}}$ is allowed:

$$e = \mathsf{allow}(\mathsf{proc}_1, \mathsf{proc}_2) \implies \mathsf{proc}_i \sqsubseteq^\# \mathsf{proc}_j \geq \frac{1}{2}$$
$$e = \overline{\mathsf{allow}(\mathsf{proc}_1, \mathsf{proc}_2)} \implies \mathsf{proc}_i \sqsubseteq^\# \mathsf{proc}_j \leq \frac{1}{2}$$

The only program commands in C are $(\mathsf{proc}, \mathsf{step})$ for each process module proc. $\mathsf{H}_\star{}^\#$ allows each such program command:

$$c = (\mathsf{proc}_1, \mathsf{step}) \implies \mathsf{True}$$

$\mathsf{H}_\star{}^\#$ responds to host primitives in D as follows. First, suppose that a process proc requests to create a new category in bin b. Then $\mathsf{H}_\star{}^\#$ merges information about the category isolated for b with the summary relation for b, and isolates a new category for b:

$$d = (\mathsf{proc}, \mathsf{make_cat}(b)) \implies \\ \mathsf{isoindx}(s', \mathsf{proc}) = \mathsf{isoindx}(s, \mathsf{proc}) + 1 \\ \land \mathsf{isolabel}(s, b)(\mathsf{proc}) = \star \\ \land \forall \mathsf{proc}_1 \neq \mathsf{proc} : \mathsf{isolabel}(s, b)(\mathsf{proc}_1) = 1 \\ \land \forall \mathsf{proc}_1, \mathsf{proc}_2 : \mathsf{sum}(s', b)(\mathsf{proc}_1, \mathsf{proc}_2) = \\ (\mathsf{sum}(s, b)(\mathsf{proc}_1, \mathsf{proc}_2) \\ \sqcap^v \mathsf{isolabel}(s, b)(\mathsf{proc}_1) \sqsubseteq_{lev} \mathsf{isolabel}(s, b)(\mathsf{proc}_2))$$

Now suppose that a process proc requests to set the level of some category cat in its label. If cat is isolated in its bin b, then $\mathsf{H}_\star{}^\#$ updates the precise level to which cat is mapped. Otherwise, $\mathsf{H}_\star{}^\#$ updates the summary relation for b, making the sound assumption that the label of proc is raised to an arbitrarily high level.

$$d = (\mathsf{proc}, \mathsf{set_taint}(\mathsf{cat}(b, i), l)) \implies \\ \mathsf{ite}(\mathsf{isoindx}(s, b) = i, \\ \mathsf{ite}(\mathsf{isolabel}(s, b)(\mathsf{proc}) \sqsubseteq_\delta l, \\ \mathsf{isolabel}(s', b)(\mathsf{proc}) = l, \\ \mathsf{isolabel}(s', b)(\mathsf{proc}) = \mathsf{isolabel}(s, b)(\mathsf{proc})), \\ \forall\ \mathsf{proc}_2 : \\ x = \mathsf{sum}(s, b)(\mathsf{proc}, \mathsf{proc}_2) \\ \land \mathsf{sum}(s', b)(\mathsf{proc}, \mathsf{proc}_2) = \left(x + \frac{1}{2}\right) - \frac{1}{2} \\ \land y = \mathsf{sum}(s, b)(\mathsf{proc}_2, \mathsf{proc}) \\ \land \mathsf{sum}(s', b)(\mathsf{proc}_2, \mathsf{proc}) = \left(y - \frac{1}{2}\right) + \frac{1}{2})$$

Finally, if a process proc invokes the noop host primitive, then $\mathsf{H}_\star{}^\#$ does not change its state:

$$d = (\mathsf{proc}, \mathsf{noop}) \implies s' = s$$

4.2.3 Real-World Programs and Policies on HiStar

Anti-Virus Scanner The developers of HiStar ported the ClamAV anti-virus scanner [8] to the HiStar operating system [33]. The version of ClamAV rewritten for HiStar can enforce strong security guarantees while making weak assumptions about the integrity of ClamAV. In particular, ClamAV executes over a Scanner process module. The Scanner module reads the private data of a user, reads entries in an updated virus database VirusDB, and checks the private data against information in the database. If the private data indicates that the host system has been infected with a virus, then the Scanner reports the virus to the user over a TTY terminal. However, the Scanner module may contain vulnerabilities, and if it is compromised after reading the user's private data, then it may leak the data directly over the network. Furthermore, a daemon process Update updates the virus database; Update should be able to read information from the network and write information to the virus database, but it should not be trusted to write the user's private data.

We may formalize the problem of instrumenting ClamAV to execute securely and functionally as a policy-weaving problem. In particular, let the ClamAV program be defined by the following set of process modules:

- Network: a network device.
- TTY: an output terminal to which the Scanner reports information to the user.
- UserData: a user's sensitive data that the Scanner should be able to read, but should not be able to leak directly over the network.

- $\mathsf{VirusDB}$: a virus database that stores information used by the $\mathsf{Scanner}$ to detect the signature of a virus in a user's files.
- Update: a process that fetches information about new viruses from the $\mathsf{Network}$ and updates the $\mathsf{VirusDB}$.
- $\mathsf{Scanner}$: the main module of ClamAV, which reads $\mathsf{UserData}$ and $\mathsf{VirusDB}$, and reports information about viruses on a system to TTY through wrap.
- wrap: a module that launches ClamAV by spawning the $\mathsf{Scanner}$ module. wrap also filters information that flows from the $\mathsf{Scanner}$ module to an output TTY device.

Let the alphabet of program commands for ClamAV C contain a command step that denotes any step of execution by any process module. Let the alphabet of privilege events E contain one event $\mathsf{allow}(\mathsf{proc}_1, \mathsf{proc}_2)$ for each pair of process modules proc_1 and proc_2.

Let the security policy of ClamAV be a regular language $\mathsf{S} \subseteq (\mathsf{C} \times \mathsf{E})^*$. Let $\mathsf{TransFlow}(\mathsf{proc}, \mathsf{proc}', \mathsf{Procs}) \subseteq (\mathsf{C} \times \mathsf{E})^*$ be the language of all sequences that contain a subsequence $(\mathsf{step}, \mathsf{allow}(\mathsf{proc}, \mathsf{proc}_1)), (\mathsf{step}, \mathsf{allow}(\mathsf{proc}_1, \mathsf{proc}_2)), \ldots, (\mathsf{step}, \mathsf{allow}(\mathsf{proc}_{n-1}, \mathsf{proc}'))$ for each $\mathsf{proc}_i \in \mathsf{Procs}$. Then the language of system traces that violate security is

$$\begin{aligned} V = &\ \mathsf{TransFlow}(\mathsf{Scanner}, \mathsf{Network}, \overline{\{\mathsf{wrap}\}}) \\ &\cup \mathsf{TransFlow}(\mathsf{Scanner}, \mathsf{TTY}, \overline{\{\mathsf{wrap}\}}) \\ &\cup .^*(\mathsf{step}, \mathsf{allow}(\mathsf{Update}, \mathsf{UserData})) \end{aligned}$$

Then the security policy is $\mathsf{S} = \overline{V}$.

Let the functionality policy of ClamAV be a regular language $\mathsf{F} \subseteq (\mathsf{C} \times \mathsf{E})^*$ that describes each required communication between processes described informally above. Let the set of protected privilege events be

$$\begin{aligned} R = \{&\mathsf{allow}(\mathsf{Scanner}, \mathsf{wrap}), \mathsf{allow}(\mathsf{wrap}, \mathsf{Scanner}), \\ &\mathsf{allow}(\mathsf{Network}, \mathsf{Update}), \mathsf{allow}(\mathsf{Update}, \mathsf{Network}), \\ &\mathsf{allow}(\mathsf{Update}, \mathsf{VirusDB}), \mathsf{allow}(\mathsf{VirusDB}, \mathsf{Update}), \\ &\mathsf{allow}(\mathsf{wrap}, \mathsf{TTY}), \mathsf{allow}(\mathsf{step}, \mathsf{UserData})\} \end{aligned}$$

Then the functionality policy is $\mathsf{F} = (\{\mathsf{step}\} \times R)^*$.

OpenVPN The developers of HiStar ported a version of the OpenVPN virtual private network (VPN) client [28] to HiStar, and described how to instrument programs so that they enforce strong guarantees using the VPN. A VPN allows a system not connected physically to a network to securely connect to the network remotely. If applications on such a system are compromised, then an attacker can use the compromised applications to leak information to and from the secure network. However, a system may use HiStar to ensure that even if applications are compromised, they cannot leak information to and from the secure network.

We may express the problem of instrumenting applications on a HiStar system with a VPN to a policy-weaving problem. Let an application consist of the following process modules:

- $\mathsf{VPNBrowser}$: a browser that must be able to connect to the VPN, but must not be able to send or receive information from the Internet, unless the information is sent through OpenVPN.
- $\mathsf{VPNStack}$: a process that mediates connections to the VPN.
- $\mathsf{InetBrowser}$: a browser that must be able to connect to the Internet, but must not be able to send or receive information from the VPN, unless the information is sent through OpenVPN.
- $\mathsf{InetStack}$: a process that mediates connections to the Internet.
- $\mathsf{OpenVPN}$: OpenVPN, a VPN client that is trusted to send and receive information from both the VPN and the Internet.

Let the alphabet of program commands for OpenVPN C contain a command step that denotes any other step of execution by any process module. Let the alphabet of privilege events E contain one event $\mathsf{allow}(\mathsf{proc}_1, \mathsf{proc}_2)$ for each pair of process modules proc_1 and proc_2.

Let the security policy for OpenVPN be $\mathsf{S} \subseteq (\mathsf{C} \times \mathsf{E})^*$, be defined as follows. $\mathsf{VPNBrowser}$ should not be able to leak information to $\mathsf{InetStack}$, and $\mathsf{InetBrowser}$ should not be able to leak information to $\mathsf{VPNStack}$. Thus, for

$$\begin{aligned} V = &\ \mathsf{TransFlow}(\mathsf{VPNBrowser}, \mathsf{InetStack}, \overline{\{\mathsf{OpenVPN}\}}) \\ &\cup \mathsf{TransFlow}(\mathsf{InetBrowser}, \mathsf{VPNStack}, \overline{\{\mathsf{OpenVPN}\}}) \end{aligned}$$

The security policy is $\mathsf{S} = \overline{V}$.

Let the functionality policy for OpenVPN be $\mathsf{F} \subseteq (\mathsf{C} \times \mathsf{E})^*$, defined as follows. $\mathsf{VPNBrowser}$ must always be able to send information to and receive information from $\mathsf{VPNStack}$, the $\mathsf{InetBrowser}$ should always be able to send information to and receive information from $\mathsf{InetStack}$, and $\mathsf{OpenVPN}$ should be able to send information to and receive information from $\mathsf{VPNStack}$ and $\mathsf{InetStack}$. Let the set of protected events be

$$\begin{aligned} R = \{&\mathsf{allow}(\mathsf{VPNBrowser}, \mathsf{VPNStack}), \mathsf{allow}(\mathsf{VPNStack}, \mathsf{VPNBrowser}), \\ &\mathsf{allow}(\mathsf{InetBrowser}, \mathsf{InetStack}), \mathsf{allow}(\mathsf{InetStack}, \mathsf{InetBrowser}), \\ &\mathsf{allow}(\mathsf{OpenVPN}, \mathsf{VPNStack}), \mathsf{allow}(\mathsf{VPNStack}, \mathsf{OpenVPN}), \\ &\mathsf{allow}(\mathsf{OpenVPN}, \mathsf{InetStack}), \mathsf{allow}(\mathsf{InetStack}, \mathsf{OpenVPN})\} \end{aligned}$$

Then $\mathsf{F} = (\{\mathsf{step}\} \times R)^*$.

4.3 Asbestos

The Asbestos operating system is a DIFC system similar to the HiStar operating system. Asbestos assigns a label to each process. When one process tries to communicate information to another process, Asbestos interposes on the communication, checks the label of the sending process against the label of the receiving process, and only allows the communication if the labels satisfy a particular relationship. However, while HiStar allows each process to explicitly manipulate its label, in Asbestos, labels are updated automatically when information is sent and received. Furthermore, Asbestos allows processes to raise the labels of other processes by providing a *contamination label* with each message, described below, which has no clear analogous object in HiStar. We thus present Asbestos alongside HiStar to illustrate how different privilege-aware systems provide different host primitives with different semantics, to allow programs to enforce policies defined over the same privilege events. A key feature of our policy-weaving problem and algorithm is that as long as the developers of two systems with different primitives provide models of their system, an application developer need only write a single policy for their application and may automatically obtain a version of their application instrumented for each system.

4.3.1 Concrete Model of Asbestos

We now give a model of Asbestos for a program P, represented as a symbolic acceptor $\mathsf{H_A} = (\mathcal{T}, \mathsf{S}, s_0, A, \Sigma, \rho)$. $\mathsf{H_A}$ is partly defined by various objects identical to objects used to define HiStar (§4.2), such as labels, categories, and levels; for clarity, we do not redefine these objects.

The theory of Asbestos $\mathcal{T}$ is identical to the theory of HiStar.

A state of Asbestos in S maps each process to a *send label* and a *receive label*. The state space of $\mathsf{H_A}$ is thus the space of all functions

$$\mathsf{Procs} \to (\mathsf{Cats} \to L)^2$$

along with a special stuck state. As shorthand, let $\mathsf{recvlab}(s, \mathsf{proc}) = \pi_1(s(\mathsf{proc}))$, and let $\mathsf{sndlab}(s, \mathsf{proc}) = \pi_2(s(\mathsf{proc}))$.

The initial state s_0 of $\mathsf{H_A}$ is the state that maps every process to an empty label:

$$\forall \mathsf{proc} : s_0(\mathsf{proc}) = \emptyset$$

The accepting states A of $\mathsf{H_A}$ are all non-stuck states.

The alphabet Σ of $\mathsf{H_A}$ is $(\mathsf{C} \times \mathsf{D}) \times \mathsf{E}$, where C and E are defined from P as they are for HiStar. The alphabet of host primitives D are defined as follows. Each process proc may invoke a host primitive to request to create a category ($(\mathsf{proc}, \mathsf{createcat}) \in \mathsf{D_A}$), may request to authorize another process to be able to declassify a category ($(\mathsf{proc}, \mathsf{authdecl}(\mathsf{proc}_1, c)) \in \mathsf{D_A}$), or may send a classification label to another process ($(\mathsf{proc}, \mathsf{sndclass}(L))$.

The transition relation ρ of Capsicum for P is a ternary predicate $\rho(s, ((c, d), e), s')$ satisfied by pre-state s, program command c, host primitive d, privilege event e and post-state s' that satisfy the conjunction of the following constraints. The first constraints define what privilege events are allowed. The privilege event $\mathsf{allow}(\mathsf{proc}_s, \mathsf{proc}_r)$ is allowed if and only if the sending label of proc_s and the receiving label of proc_r satisfy the flow relation defined in §4.2.1:

$$e = (\mathsf{proc}, \mathsf{allow}(\mathsf{proc}_s, \mathsf{proc}_r)) \implies \mathsf{sndlab}(\mathsf{proc}_s) \sqsubseteq \mathsf{recvlab}(\mathsf{proc}_r)$$

$\mathsf{H_A}$ allows each program command (proc, c) for each process module proc. $\mathsf{H_A}$ responds to each host command in D as follows. If a process requests to create a category on $\mathsf{H_A}$, then $\mathsf{H_A}$ responds analogously to how $\mathsf{H}_\star$ responds to the same command. If a process proc_1 requests to authorize another process proc_2 to be able to declassify a category cat, then $\mathsf{H_A}$ checks if proc_1 has the declassification label for cat, and if so, updates the label of proc_2 to be the declassification label:

$$d = (\mathsf{proc}_1, \mathsf{authdecl}(\mathsf{proc}_2, \mathsf{cat})) \implies \mathsf{sndlab}(s', \mathsf{proc}_1)(\mathsf{cat}) = \star \iff \mathsf{sndlab}(s', \mathsf{proc}_2)(\mathsf{cat}) = \star$$

Finally, if a process proc_s requests to raise the label of a receiving process proc_r by sending a classification label L to proc_r, then $\mathsf{H_A}$ responds to raising the sending label of proc_r accordingly:

$$\begin{aligned} d = (\mathsf{proc}_s, \mathsf{sndclass}(\mathsf{proc}_r, L)) \implies \\ \mathsf{ite}(\mathsf{sndlab}(s, \mathsf{proc}_r) \sqsubseteq L, \\ \mathsf{sndlab}(s', \mathsf{proc}_r) = L, \\ \mathsf{sndlab}(s', \mathsf{proc}_r) = \mathsf{sndlab}(s, \mathsf{proc}_r)) \end{aligned}$$

4.3.2 An Abstract Model of Asbestos

The concrete model of Asbestos $\mathsf{H_A}$ cannot be used to construct symbolic weaving problems that may be solved by applying an SMT solver, because the transition relation of $\mathsf{H_A}$ is defined by the $\sqsubseteq$ relation over labels, which is defined by universal quantification over the unbounded set of categories that a program may allocate at runtime. However, we may construct a sound abstraction $\mathsf{H_A}^\#$ of $\mathsf{H_A}$ that keeps an approximation of the flow relation that may be finitely represented. $\mathsf{H_A}^\#$ is constructed from $\mathsf{H_A}$ similarly to how $\mathsf{H}_\star^\#$ is constructed from $\mathsf{H}_\star$ (§4.2.2): $\mathsf{H_A}^\#$ partitions the unbounded set of categories that may be constructed by a program are partitioned into a finite set of bins, and $\mathsf{H_A}^\#$ maintains an over-approximation of the flow-relation over labels using the bins. Given that the constructions are nearly identical, we do not give an explicit definition of $\mathsf{H_A}^\#$.

4.3.3 Real-World Programs and Policies on Asbestos

OKWS The developers of Asbestos ported the OKWS webserver [20] to Asbestos, allowing it enforce stronger security properties than it could when running on UNIX [15]. OKWS is a webserver that receives requests over the network, and services each request by launching an untrusted Worker process to service the request. The full implementation of OKWS is complex, and in this report, we only discuss a high-level model of the system, along with its security and functionality policies. Let OKWS be structured as executing over the following process modules and files:

- netd: implements the network stack, and mediates all information sent into and out of the network.
- $\mathsf{okdemux}$: receives requests for services from netd.
- Worker_i: for each request u that $\mathsf{okdemux}$ receives, it forks a process Worker_i to service the request. In general, OKWS may fork an unbounded set of Worker processes. Our policy-weaving problem cannot reason about applications composed from unbounded sets of processes, so in this discussion, we suppose that OKWS only receives a bounded number (two) of requests.
- port_i: for each request i that $\mathsf{okdemux}$ receives, it creates a port to send and receive all information regarding the request.

The alphabet of program commands C contains, for each process module proc, a program command $(\mathsf{proc}, \mathsf{step})$ that denotes that proc performs a step of execution. The alphabet of privilege events E contains, for each pair of processes proc_1 and proc_2, a privilege events $\mathsf{allow}(\mathsf{proc}_1, \mathsf{proc}_2)$.

Let the security policy $\mathsf{S} \subseteq (\mathsf{C} \times \mathsf{E})^*$ for OKWS be the following. Worker_i for request i must not be able to send information to the Worker process for the other request, or to the port for the other request. Let the set of violating privilege events $V \subseteq \mathsf{E}$ be

$$\begin{aligned} V = \{&\mathsf{allow}(\mathsf{Worker}_0, \mathsf{Worker}_1), \mathsf{allow}(\mathsf{Worker}_0, \mathsf{port}_1), \\ &\mathsf{allow}(\mathsf{Worker}_1, \mathsf{Worker}_0), \mathsf{allow}(\mathsf{Worker}_1, \mathsf{port}_0)\} \end{aligned}$$

The security policy is $\mathsf{S} = (\{\mathsf{step}\} \times \overline{V})^*$.

Let the functionality policy $\mathsf{F} \subseteq (\mathsf{C} \times \mathsf{E})^*$ for OKWS be the following. Each Worker_i must be able to send information to and receive information from port_i. $\mathsf{okdemux}$ must be able to send information to and receive information from netd and each Worker_i. Let the set of protected events be

$$\begin{aligned} R = \{&\mathsf{allow}(\mathsf{Worker}_0, \mathsf{port}_0), \mathsf{allow}(\mathsf{port}_0, \mathsf{Worker}_0), \\ &\mathsf{allow}(\mathsf{Worker}_1, \mathsf{port}_1), \mathsf{allow}(\mathsf{port}_1, \mathsf{Worker}_1), \\ &\mathsf{allow}(\mathsf{okdemux}, \mathsf{netd}), \mathsf{allow}(\mathsf{netd}, \mathsf{okdemux}), \\ &\mathsf{allow}(\mathsf{okdemux}, \mathsf{Worker}_0), \mathsf{allow}(\mathsf{Worker}_0, \mathsf{okdemux}), \\ &\mathsf{allow}(\mathsf{okdemux}, \mathsf{Worker}_1), \mathsf{allow}(\mathsf{Worker}_1, \mathsf{okdemux})\} \end{aligned}$$

Then the functionality policy is $\mathsf{F} = (\{\mathsf{step}\} \times R)^*$.

4.4 Flume

The Flume [21] DIFC operating system provides a set of host primitives for allowing applications to enforce information-flow security. The Flume primitives are distinct from those of HiStar or Asbestos, because in the HiStar and Asbestos systems, the label of each process completely determines both when it can send or receive information, and how the label itself my change. However, in Flume, each process has, along with its label, a distinct *positive capability* that describes all categories that it may add to its label, and a *negative capability* that describes all categories that it may remove from its label.

4.4.1 Concrete Model of Flume

The concrete model of Flume $\mathsf{H_F} = (\mathcal{T}, \mathsf{S}, s_0, A, \Sigma, \rho)$ for a given program P is a symbolic acceptor defined as follows.

$\mathsf{H_F}$ is defined over an alphabet of program commands C, alphabet of host primitives D, and alphabet of privilege events E. The program commands in C are the following. Each process module

proc in P executes only a single program command that denotes a step of execution, $(\mathsf{proc}, \mathsf{step}) \in \mathsf{C}$.

The host primitives in D are the following. Each process module proc in P can request to create a new category by invoking $(\mathsf{proc}, \mathsf{createcat}) \in \mathsf{D}$. proc can request to add a category to its label by invoking $(\mathsf{proc}, \mathsf{addcat}) \in \mathsf{D}$, or can request to remove a category from its label by invoking $(\mathsf{proc}, \mathsf{addcat}) \in \mathsf{D}$. Finally, proc can request to remove a positive capability by invoking $(\mathsf{proc}, \mathsf{rmpos}) \in \mathsf{D}$, or it can request to remove a negative capability by invoking $(\mathsf{proc}, \mathsf{rmneg}) \in \mathsf{D}$.

The alphabet the privilege events E for P in Flume is the same alphabet of privilege events for P in HiStar.

The theory $\mathcal{T}$ of Flume for P is first-order, with conjunction, disjunction, negation, and quantification. Formulas in $\mathcal{T}$ are interpreted over a domain that contains elements corresponding to the elements of alphabets C, D, and E, an identifier for each process module in P, and element and set of elements in a countable set of categories in Cats. The constants of $\mathcal{T}$ are the constant $\emptyset$ for the empty set of categories, and the constant Cats for the set of all categories. The functions of $\mathcal{T}$ are intersection over sets of categories, and constructors that construct elements in the alphabets. The predicates of $\mathcal{T}$ are an equality predicate and set membership. The subset predicate is defined from the membership predicate. Functions and predicates not corresponding to sets of rights are interpreted as their corresponding elements in the domain of $\mathcal{T}$, and the set functions and predicates are axiomatized by a standard set theory.

Each state in the state-space S of Flume stores the index of the next category to be created, and maps each process to (1) its label, which is the set of categories that define what information it may send and receive, (2) its positive capability, which is the set of categories that it may add to its label, and (3) its negative capability, which is the set of categories that it may remove from its label. For Procs the set of process modules in a program and Cats the set of categories that may be allocated dynamically by a program, S is the domain

$$\mathsf{S} = \mathbb{N} \times (\mathsf{Procs} \to (\mathcal{P}(\mathsf{Cats}))^3)$$

along with a "stuck" state. As shorthand, for each non-stuck state s and process proc, let $\mathsf{catindx}(s) = \pi_1(s)$, let $\mathsf{lab}(s, \mathsf{proc}) = \pi_1(\pi_2(s)(\mathsf{proc}))$, let $\mathsf{pos}(s, \mathsf{proc}) = \pi_2(\pi_2(s)(\mathsf{proc}))$, and let $\mathsf{neg}(s, \mathsf{proc}) = \pi_3(\pi_2(s)(\mathsf{proc}))$.

The initial state s_0 of Flume for P maps every process in P to an empty label, positive capability, and negative capability:

$$\forall \mathsf{proc} : \mathsf{lab}(s_0, \mathsf{proc}) = \mathsf{pos}(s_0, \mathsf{proc}) = \mathsf{neg}(s_0, \mathsf{proc}) = \emptyset$$

The accepting states A of Flume for P are all non-stuck states.

The alphabet of Flume is $((\mathsf{C} \times \mathsf{D}) \times \mathsf{E})$.

The transition relation ρ of Flume for P is a ternary predicate $\rho(s, ((c, d), e), s')$ over a pre-state s, program command c, host primitive d, privilege event e, and post-state s'. ρ is a conjunction of constraints. The first constraint requires that a process proc_s may only send information to a process proc_r if the label of proc_s is a subset of the label of proc_r:

$$e = \mathsf{allow}(\mathsf{proc}_s, \mathsf{proc}_r) \implies \mathsf{lab}(s, \mathsf{proc}_s) \subseteq \mathsf{lab}(s, \mathsf{proc}_r)$$

where subset over sets of categories has its standard definition:

$$L_1 \subseteq L_2 \iff \forall c : c \in L_1 \implies c \in L_2$$

For each process module proc in P, there is only one program command $(\mathsf{P}, \mathsf{step})$, and it is allowed as a component of each transition. $\mathsf{H_F}$ responds to host primitives in D as follows. If a process proc requests to create a category by invoking the host primitive $(\mathsf{P}, \mathsf{createcat})$, then $\mathsf{H_F}$ creates a category, gives proc positive and negative capabilities for the category, and updates the index of the next category to be created:

$$\begin{aligned} d = (\mathsf{proc}, \mathsf{createcat}) \implies & \\ \mathsf{pos}(s', \mathsf{proc}) &= \mathsf{pos}(s, \mathsf{proc}) \cup \{\mathsf{catindx}(s)\} \\ \land\ \mathsf{neg}(s', \mathsf{proc}) &= \mathsf{neg}(s, \mathsf{proc}) \cup \{\mathsf{catindx}(s)\} \end{aligned}$$

If a process proc requests to add a category cat to its label by invoking the host primitive $(\mathsf{proc}, \mathsf{addcat}(\mathsf{cat}))$, then $\mathsf{H_F}$ only adds cat to the label of proc if proc has cat category in its positive capability:

$$\begin{aligned} & d = (\mathsf{proc}, \mathsf{addcat}(c)) \implies \\ & \mathsf{ite}(c \in \mathsf{pos}(s, \mathsf{proc}), \\ & \quad \mathsf{lab}(s', \mathsf{proc}) = \mathsf{lab}(s, \mathsf{proc}) \cup \{c\}, \\ & \quad \mathsf{lab}(s', \mathsf{proc}) = \mathsf{lab}(s, \mathsf{proc})) \end{aligned}$$

If a process proc requests to remove a category cat from its label by invoking the host primitive $(\mathsf{proc}, \mathsf{addcat}(\mathsf{cat}))$, then $\mathsf{H_F}$ removes cat from the label of proc only if cat is in the negative capability of proc.

$$\begin{aligned} d = (\mathsf{proc}, \mathsf{addcat}(c)) \implies & \mathsf{ite}(c \in \mathsf{neg}(s, \mathsf{proc}), \\ & \mathsf{lab}(s', \mathsf{proc}) = \mathsf{lab}(s, \mathsf{proc}) \backslash \{c\}, \\ & \mathsf{lab}(s', \mathsf{proc}) = \mathsf{lab}(s, \mathsf{proc})) \end{aligned}$$

If a process proc requests to remove a category from its positive capability, then $\mathsf{H_F}$ removes the category:

$$d = (\mathsf{proc}, \mathsf{rmpos}(c)) \implies \mathsf{pos}(s', \mathsf{proc}) = \mathsf{pos}(s, \mathsf{proc}) \backslash \{c\}$$

If a process proc requests to remove a category from its negative capability, then $\mathsf{H_F}$ removes the negative capability:

$$d = (\mathsf{proc}, \mathsf{rmneg}(c)) \implies \mathsf{neg}(s', \mathsf{proc}) = \mathsf{neg}(s, \mathsf{proc}) \backslash \{c\}$$

4.4.2 An Abstract Model of Flume

The concrete model of Flume $\mathsf{H_F}$ cannot be used to construct weaving problems that may be solved by applying an SMT solver, because the transition relation of $\mathsf{H_F}$ is defined by the subset relation over sets of categories created by a program, which is defined by universal quantification over the set of all categories, which is unbounded. However, we may construct a sound abstraction $\mathsf{H_F}^\#$ of $\mathsf{H_F}$ that keeps an approximation of the containment relation that may be finitely represented. $\mathsf{H_F}^\#$ is constructed from $\mathsf{H_F}$ similarly to how $\mathsf{H_\star}^\#$ is constructed from $\mathsf{H_\star}$ (§4.2.2): $\mathsf{H_F}^\#$ partitions the unbounded set of categories that may be constructed by a program into a finite set of bins, and $\mathsf{H_F}^\#$ maintains an approximation of the containment relation defined by the bins. Given that the construction of $\mathsf{H_F}^\#$ from $\mathsf{H_F}^\#$ is nearly identical to the construction of $\mathsf{H_\star}^\#$ from $\mathsf{H_\star}$, we do not give an explicit definition of $\mathsf{H_F}^\#$.

4.4.3 Real-World Programs and Policies on Flume

The developers of Flume ported the MoinMoin wiki [26] to Flume, allowing it to enforce stronger security properties than it could when running on a traditional operating system, like UNIX [21]. MoinMoin is a wiki server that allows users to send remote requests over the network to read or edit content on the wiki. The Flume developers repartitioned MoinMoin into FlumeWiki, a wiki server more appropriately structured to use the Flume primitives to enforce security. The developers then instrumented FlumeWiki to use the Flume primitives to enforce security. The full implementation of FlumeWiki is a complex server, and in this report, we only describe the server at a high-level, and discuss a component over its security and functionality policies. Let FlumeWiki execute over the following process modules and files:

- port80: a special network port from which requests come in for the wiki server.

- httpd_i: an instance of the `http` daemon that communicates information about request i to and from $\mathsf{port80}$. In practice, FlumeWiki may service an unbounded set of requests, and thus may create an unbounded set of instances of httpd_i. However, for this report, we assume that FlumeWiki services at most two requests.
- $\mathsf{wikilaunch}_i$: a small, trusted process that receives a request from the httpd process and creates an untrusted Worker process to service the request.
- Worker_i: a complex, untrusted process that implements most of the logic in servicing a request.

Let the alphabet of program commands C contain a single program command step that denotes one step of execution of FlumeWiki. Let the alphabet of privilege events E contain for each two processes proc_1 and proc_2, a privilege event $\mathsf{allow}(\mathsf{proc}_1, \mathsf{proc}_2)$ that denotes that information is allowed to flow between proc_1 and proc_2.

Let the security policy $\mathsf{S} \subseteq (\mathsf{C} \times \mathsf{E})^*$ be defined as follows. No Worker_i process should be able to directly leak information to the network over $\mathsf{port80}$, or to another Worker_i process. Let the set of violating privilege events be

$$V = \{\mathsf{allow}(\mathsf{Worker}_0, \mathsf{port}80), \mathsf{allow}(\mathsf{Worker}_1, \mathsf{port}80), \\ \mathsf{allow}(\mathsf{Worker}_0, \mathsf{Worker}_1), \mathsf{allow}(\mathsf{Worker}_1, \mathsf{Worker}_0)\}$$

Then the security policy is $\mathsf{S} = (\{\mathsf{step}\} \times \overline{V})^*$.

Let the functionality policy $\mathsf{F} \subseteq (\mathsf{C} \times \mathsf{E})^*$ be defined as follows. Each httpd_i process must always be able to send information to and receive information from port, each $\mathsf{wikilaunch}_i$ process must be able to send information to and receive information from the corresponding httpd_i process, and each Worker_i process must be able to send information to and receive information from the corresponding $\mathsf{wikilaunch}_i$ process. Let the set of protected privilege events be

$$R = \{\mathsf{allow}(\mathsf{httpd}_0, \mathsf{port}), \mathsf{allow}(\mathsf{port}, \mathsf{httpd}_0), \\ \mathsf{allow}(\mathsf{httpd}_1, \mathsf{port}), \mathsf{allow}(\mathsf{httpd}_1, \mathsf{port}), \\ \mathsf{allow}(\mathsf{wikilaunch}_0, \mathsf{httpd}_0), \mathsf{allow}(\mathsf{httpd}_0, \mathsf{wikilaunch}_0), \\ \mathsf{allow}(\mathsf{wikilaunch}_1, \mathsf{httpd}_1), \mathsf{allow}(\mathsf{httpd}_1, \mathsf{wikilaunch}_1), \\ \mathsf{allow}(\mathsf{Worker}_0, \mathsf{wikilaunch}_0), \mathsf{allow}(\mathsf{wikilaunch}_0, \mathsf{Worker}_0), \\ \mathsf{allow}(\mathsf{Worker}_1, \mathsf{wikilaunch}_1), \mathsf{allow}(\mathsf{wikilaunch}_1, \mathsf{Worker}_1)\}$$

The functionality policy is $\mathsf{F} = (\{\mathsf{step}\} \times R)^*$.

5. Related Work

Formal games: Formal games have been studied [3, 4, 11, 24] as a framework for synthesizing reactive programs and control mechanisms. Previous work describes algorithms that take a parity game represented symbolically, determine which player may always win the game, and sometimes synthesize a winning strategy for the player [11, 24]. One contribution of our work is connecting these problems to an apparently different problem of instrumenting programs for privilege-aware systems. Accordingly, we present an algorithm that addresses problems that arise in games constructed from instrumenting programs for such systems. In particular, our algorithm searches for strategies with size up to a given bound, and considers the case where the alphabet of the defending player is much larger than the alphabet of the attacking player. The approach is similar to a known algorithm that searches for a witness set of states of bounded size [24]. Our approach differs from that algorithm in that we search for a strategy with a different condition for winning.

Reference monitors: Inline reference monitors describe a policy for a program to satisfy, and monitor the program to ensure that it satisfies the policy [1, 16]. Reference monitors traditionally monitor a stream of security-sensitive events, and respond with one of two possible actions: either the monitor may allow the program to continue executing, or it may abort the program. In contrast, our policy-weaving algorithm instruments a program to enforce a policy by instrumenting the program to (i) track the security-sensitive events of a program, and (ii) respond by invoking primitives drawn from a rich set provided by a privilege-aware system. Furthermore, an inline reference monitor guarantees the correctness of the program that it monitors only if the integrity of the control-flow of the program is preserved. In contrast, our policy-weaver can potentially instrument programs to enforce security policies even when the control-flow integrity of an application is subverted, provided that the integrity of the host operating system is not subverted.

Edit automata: *Edit automata* form a hierarchy of runtime enforcement mechanisms that generalize IRMs [22]. The hierarchy consists of *supression automata*, which can remove events from the event stream of a monitored program while allowing the program to continue to execute; *insertion automata* which can insert events into the event stream of the program; and *edit automata*, which can both suppress and insert. As in the case of IRMs, edit automata both monitor and alter a stream of events, whereas a solution to a weaving problem monitors events and responds by calling primitive operations, subject to rules defined by a privilege-aware system. The work in [22] introduces a notion of *transparency* to restrict how an edit automaton may alter the events of a program. This concept is similar in spirit to our notion of a functionality policy (Defn. 2), and the two notions intuitively seem equivalent, although we have not proven any formal equivalence.

Privilege-aware OS: Privilege-aware operating systems [15, 21, 32, 33] provide powerful primitive operations that applications call to enforce a high-level notions of security. DIFC operating systems [15, 21, 33] allow applications to enforce information-flow policies using labels, while Capsicum [32] allows applications to enforce access policies using capabilities and lists of rights. Prior work automatically verifies that a program instrumented to use the Flume OS [21] primitives enforces a high-level policy [17]; automatically instruments programs to use the primitives of the HiStar OS [33] to satisfy a policy [14], and automatically instruments programs [17] to use the primitives of the Flume OS [21]. However, the languages of policies used in the approaches presented in [14, 18] are not temporal and cannot clearly be applied to other privilege-aware systems, and the proofs of the correctness of the instrumentation algorithms are ad hoc. The work presented in this paper generalizes the work presented in [14, 18] by reducing the instrumentation problem to a parity game. As a result, the technique in this paper can be instantiated to instrumentation algorithms for a variety of privilege-aware systems, including HiStar and Flume, that support rich languages of policies, and admit direct proofs of correctness.

6. Conclusion

Privilege-aware operating systems make feasible the problem of constructing large, secure applications from untrusted components. However, such systems do so by greatly complicating the task of application programmers. In this paper, we have formalized the problem of writing programs for such systems with a game semantics. As a result, we have obtained an algorithm that takes a formal description of the semantics of a privilege-aware system, a program that makes no use of the primitive operations of the system, a set of executions allowed for security, and a set of executions required for functionality, and produces a secure but functional program. We believe that these results will allow for a set of tools that

will significantly increase the efficacy, and encourage the adoption of, privilege-aware systems.

References

[1] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity. In *CCS*, 2005.

[2] R. Alur and P. Madhusudan. Visibly pushdown languages. In *STOC*, 2004.

[3] R. Alur, T. A. Henzinger, and O. Kupferman. Alternating-time temporal logic. In *FOCS*, 1997.

[4] R. Alur, T. A. Henzinger, O. Kupferman, and M. Y. Vardi. Alternating refinement relations. In *CONCUR*, 1998.

[5] T. Ball and S. K. Rajamani. Automatically Validating Temporal Safety Properties of Interfaces. In *SPIN*, 2001.

[6] O. Burkart, D. Caucal, F. Moller, and B. Steffen. Verification on infinite structures, 2000.

[7] T. Cachat. Symbolic strategy synthesis for games on pushdown graphs. In *ICALP*, 2002.

[8] ClamAV. Clamav, 2011. URL `http://www.clamav.net`.

[9] ClamAV. Wu-ftpd development group, 2011. URL `http://wu-ftpd.therockgarden.ca/`.

[10] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 2003.

[11] L. de Alfaro, T. A. Henzinger, and R. Majumdar. Symbolic algorithms for infinite-state games. In *CONCUR*, 2001.

[12] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, 2008.

[13] B. Dutertre and L. de Moura. The yices smt solver. Tool paper at `http://yices.csl.sri.com/tool-paper.pdf`, August 2006.

[14] P. Efstathopoulos and E. Kohler. Manageable fine-grained information flow. In *EuroSys*, pages 301–313, 2008.

[15] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazières, F. Kaashoek, and R. Morris. Labels and event processes in the Asbestos operating system. In *SOSP*, 2005.

[16] Ú. Erlingsson and F. B. Schneider. IRM enforcement of Java stack inspection. In *IEEE SP*, 2000.

[17] W. R. Harris, N. A. Kidd, S. Chaki, S. Jha, and T. Reps. Verifying information flow control over unbounded processes. In *FM*, 2009.

[18] W. R. Harris, S. Jha, and T. Reps. DIFC programs by automatic instrumentation. In *CCS*, 2010.

[19] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy Abstraction. In *POPL*, 2002.

[20] M. Krohn. Building secure high-performance web services with okws. In *ATEC '04: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 15–15, Berkeley, CA, USA, 2004. USENIX Association.

[21] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard OS abstractions. In *SOSP*, 2007.

[22] J. Ligatti, L. Bauer, and D. Walker. Edit automata: enforcement mechanisms for run-time security policies. *Int. J. Inf. Sec.*, 4(1-2), 2005.

[23] C. Löding, P. Madhusudan, and O. Serre. Visibly pushdown games. In *FSTTCS*, 2004.

[24] P. Madhusudan, W. Nam, and R. Alur. Symbolic computational techniques for solving games. *Electr. Notes Theor. Comput. Sci.*, 89(4), 2003.

[25] S. McCanne and V. Jacobson. The bsd packet filter: a new architecture for user-level packet capture. In *USENIX Winter Conference*, 1993.

[26] MoinMoin. The moinmoin wiki engine, Dec. 2006. URL `http://moinmoin.wikiwikiweb.de`.

[27] A. One. Smashing the stack for fun and profit. *Phrack Magazine*, 49 (14), 1998.

[28] OpenVPN. Openvpn, 2011. URL `http://www.openvpn.net`.

[29] H. Putnam. Three-valued logic. *Philosophical Studies*, 8:73–80, 1957. ISSN 0031-8116. URL `http://dx.doi.org/10.1007/BF02304905`. 10.1007/BF02304905.

[30] R. E. Shostak. Deciding combinations of theories. *J. ACM*, 31, January 1984.

[31] TCPDUMP. Tcpdump/libcap public repository, 2011. URL `http://www.tcpdump.org/`.

[32] R. N. M. Watson, J. Anderson, B. Laurie, and K. Kennaway. Capsicum: Practical capabilities for UNIX. In *USENIX Security*, 2010.

[33] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *OSDI*, 2006.