Secure Programming via Visibly Pushdown Safety Games

William R. Harris[1], Somesh Jha[1], and Thomas Reps[1,2]

[1] University of Wisconsin-Madison, Madison, WI, USA
{wrharris, jha, reps}@cs.wisc.edu
[2] GrammaTech, Inc., Ithaca NY, USA

Abstract. Several recent operating systems provide system calls that allow an application to explicitly manage the privileges of modules with which the application interacts. Such *privilege-aware operating systems* allow a programmer to a write a program that satisfies a strong security policy, even when it interacts with untrusted modules. However, it is often non-trivial to rewrite a program to correctly use the system calls to satisfy a high-level security policy. This paper concerns the *policy-weaving problem*, which is to take as input a program, a desired high-level policy for the program, and a description of how system calls affect privilege, and automatically rewrite the program to invoke the system calls so that it satisfies the policy. We present an algorithm that solves the policy-weaving problem by reducing it to finding a winning modular strategy to a visibly pushdown safety game, and applies a novel game-solving algorithm to the resulting game. Our experiments demonstrate that our algorithm can efficiently rewrite practical programs for a practical privilege-aware system.

1 Introduction

Developing practical but secure programs remains a difficult, important, and open problem. Web servers and VPN clients execute unsafe code, and yet are directly exposed to potentially malicious inputs from a network connection [23]. System utilities such as Norton Antivirus scanner [19], `tcpdump`, the DHCP client `dhclient` [22], and file utilities such as `bzip`, `gzip`, and `tar` [16, 20, 21] have contained unsafe code with well-known vulnerabilities that allow them to be compromised if an attacker can control their inputs. Once an attacker compromises any of the above programs, they can typically perform any action allowed for the user that invoked the program, because the program does not restrict the privileges with which its code executes.

Traditional operating systems provide to programs only weak primitives for managing their privileges [10, 17, 22, 23]. As a result, if a programmer is to verify that his program is secure, he typically must first verify that the program satisfies very strong properties, such as memory safety. However, recent work [10, 17, 22, 23] has produced new operating systems that allow programmers to develop programs that execute unsafe code but still satisfy strong properties, and to construct such programs with significantly less effort than fully verifying the program. Such systems map each program to a set of privileges, and extend the set of system calls provided by a traditional operating system with security-

specific calls (which henceforth we will call *security primitives*) that the program invokes to manage its privileges. We call such systems *privilege-aware systems*.

This paper concerns the *policy-weaving problem*, which is to take a program and a security policy that defines what privileges the program must have, and to automatically rewrite the program to correctly invoke the primitives of a privilege-aware system so that the program satisfies the policy when run on the system. The paper addresses two key challenges that arise in solving the policy-weaving problem. First, a privilege-aware system cannot allow a program to modify its privileges arbitrarily, or an untrusted module of the program could simply give itself the privileges that it requires to carry out an attack. Instead, the system allows a program to modify its privileges subject to system-specific rules. In practice, these rules are subtle and difficult to master; the developers of the Capsicum capability system reported issues in rewriting the `tcpdump` network utility to use the Capsicum primitives to satisfy a security policy, while preserving the original functionality of `tcpdump` [22].

Second, the notions of privilege often differ between privilege-aware systems, and thus so too do the primitives provided by each system, along with the rules relating privileges to primitives. The Capsicum operating system defines privileges as capabilities [22], the Decentralized Information Flow Control (DIFC) operating systems Asbestos, HiStar, and Flume [10, 17, 23] define privileges as the right to send information, and each provide different primitives for manipulating information-flow labels [8]. Thus, a policy-weaving algorithm for a specific system must depend on the privileges and primitives of the system, yet it is undesirable to manually construct a new policy-weaving algorithm for each privilege-aware system that has been or will yet be developed.

We address the above challenges by reducing the policy-weaving problem to finding a winning Defender strategy to a two-player safety game. Each game is played by an Attacker, who plays program instructions, and a Defender, who plays system primitives. The game accepts all sequences of instructions and primitives that violate the given policy. A winning Defender strategy never allows the Attacker to generate a play accepted by the game, and thus corresponds to a correct instrumentation of the program, which invokes primitives so that the policy is never violated. If the rules describing how a system's primitives modify privileges can be encoded as an appropriate automaton, then the game-solving algorithm can be applied to rewrite programs for the system. We argue that stack-based automata, in particular *visibly pushdown automata* (VPAs) [5], are sufficient to model the rules of practical privilege-aware systems. Furthermore, *modular* winning strategies exactly correspond to correct instrumentations of programs for such systems.

Finding a modular winning strategy to a game defined by a VPA is NP-complete. However, games resulting from policy-weaving problems are constructed as products of input automata, and a game will often have a strategy whose structure closely matches one of the inputs. Inspired by this observation, we present a novel algorithm that, given a game, finds a modular strategy with structure similar to an additional, potentially smaller game called a *scaffold*. We show that our scaffolding algorithm generalizes two know algorithms for

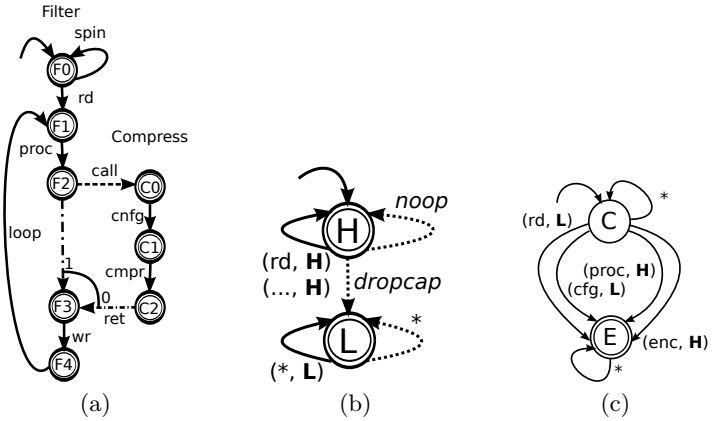

Fig. 1. Automata models of (a) the program `Filter`, (b) `Filter`'s MiniCap monitor, and (c) `Filter`'s policy for MiniCap. Notation is explained in §2.1.

finding modular strategies [6, 18] — in particular, those algorithms result from using two (different) "degenerate" scaffolds, and correspond to two ends of a spectrum of algorithms that can be implemented by our algorithm. We evaluated the scaffold-based algorithm on games corresponding to policy-weaving problems for six UNIX utilities with known vulnerabilities for the Capsicum capability system, and found that it could rewrite practical programs efficiently, and that the choice in scaffold often significantly affected the performance of the algorithm.

Organization §2 motivates by example the policy-weaving problem and our game-solving algorithm. §3 defines the policy-weaving problem, and reduces the problem to solving visibly-pushdown safety games. §4 presents a novel algorithm for solving visibly pushdown safety games. §5 presents an experimental evaluation of our algorithm. §6 discusses related work.

2 Overview

In this section, we motivate the policy-weaving problem. We sketch how the policy-weaving problem can be reduced to finding a winning strategy to a class of safety games, and how the structure of games constructed from policy-weaving problems makes them amenable to our novel game-solving algorithm.

2.1 An Example Policy-Weaving Problem: `Filter` on MiniCap

We illustrate the policy-weaving problem using an example program `Filter` that reads information from an input channel, processes and compresses the data, and then writes the data to an output channel. `Filter` is inspired by the UNIX utilities `tcpdump` and `gzip`, which have exhibited security vulnerabilities in the past, and have previously been rewritten manually for the Capsicum privilege-aware systems [22]. The executions of `Filter` are presented as the runs of the automaton F in Fig. 1(a), where each transition is labeled with a program action. Intraprocedural transitions are denoted with solid lines. Call transitions, which place their source node on a stack (see Defn. 3), are denoted with dashed lines.

Return transitions, defined by the top two states of the stack, are denoted with dash-dot lines, where the transition from the top state of the stack is labeled with a 0, and the transition from the next state down on the stack is labeled with a 1. In each figure, doubled circles denote accepting states.

`Filter` executes a no-op instruction (`spin`) until it reads data from its designated input channel (e.g., UNIX `stdin`) (`read`), processes a segment of its input data (`proc`), and calls a compression function `Compress` (`call`). `Compress` first opens and reads a configuration file (`cnfg`), compresses its input data (`cmpr`), and returns the result (`ret`). After `Compress` returns, `Filter` writes the data to its designated output channel (e.g., UNIX `stdout`) (`wr`), and loops to read another segment of data (`loop`).

Unfortunately, in practice, much of the code executed by a practical implementation of functions like `Filter` and `Compress` (e.g., `tcpdump` and `gzip` [22]) is not memory-safe, and thus allows an attacker to violate the security policy of a program. Suppose that the programmer wants to ensure that `Filter` only interacts with communication channels by opening and reading from its designated input at `read` and writing to its designated output at `wr`, and `Compress` only interacts with communication channels by reading from its configuration files at `cnfg`. However, suppose also that the data-processing action `proc` in `Filter` and the compression action `cmpr` in `Compress` perform memory-unsafe operations when passed particular inputs. Then an attacker who can control the inputs to `Filter` could craft a malicious input that injects code that opens a communication channel (e.g., a file) and violates the policy.

However, if the programmer correctly rewrites `Filter` for a suitable *privilege-aware systems*, then the rewritten `Filter` will satisfy such a policy even if it executes code injected by an attacker. Consider a privilege-aware system MiniCap, which is a simplification of the Capsicum capability system now included in the "RELEASE" branch of FreeBSD [12, 22]. MiniCap maps each executing process to a two-valued flag denoting if the process has high or low privilege. If a process has high privilege H, then it can open communication channels, but if it has low privilege L, then it can only read and write to its designated input and output channels. A process on MiniCap begins executing with high privilege, but may invoke the MiniCap primitive `dropcap`, which directs MiniCap to give the process low privilege, and never give the process the high privilege again. A process thus might invoke `dropcap` after executing safe code that requires high privilege, but before executing unsafe code that requires only low privilege.

MiniCap also allows one process to communicate with another process via a *remote procedure call (RPC)*, in which case the called process begins execution with high privilege, independent of the privilege of the caller. The Capsicum capability system uses RPC in this way, while DIFC systems allow a process to call a process with different privileges via an analogous *gate call* [23].

MiniCap is partially depicted in Fig. 1(b) as an automaton M that accepts sequences of privilege-instruction pairs and primitives executed by `Filter`. We call M the MiniCap *monitor* of `Filter`. M accepts a trace of privilege-instruction pairs and primitives if when `Filter` executes the sequence of instructions and primitives, MiniCap grants `Filter` the privilege paired with each instruction.

The call and return transitions of M are omitted for simplicity; M transitions on an RPC to the high-privilege state H, and returns from an RPC to the calling state.

`Filter`'s policy can be expressed directly in terms of MiniCap's privileges by requiring that the instructions `read` and `cnfg` execute with high privilege, while the instructions `proc` and `cmpr` execute with low privilege. The policy is presented as an automaton Pol in Fig. 1(c), where each transition is labeled with a privilege-instruction pair (the label * denotes any label that does not appear explicitly on a transition from the same source state). The traces accepted by Pol are the sequences of instruction-privilege pairs that violate the policy.

For `Filter` to satisfy its policy when it is run on MiniCap, it must use the primitives of MiniCap in a way that is only indirectly related to, and significantly more complex than, its desired policy. (1) invoke `dropcap` after executing `read` but before executing `proc`, (2) call `Compress` via RPC so that `Compress` executes `cnfg` with high privilege, (3) invoke `dropcap` after executing `cnfg` but before executing `cmpr`. This rewritten `Filter` is "modular" across calls and returns, in the sense that the rewritten `Filter` and `Compress` invoke primitives independently of the actions of each other. On practical privilege-aware systems, a process that can be called via RPC cannot necessarily trust its caller, and thus cannot trust information passed by its caller. Thus a practical instrumentation must be modular.

The policy-weaving problem for `Filter` is to take F, its policy Pol, and MiniCap monitor M, and instrument `Filter` to use MiniCap's primitives modularly to satisfy Pol.

2.2 Policy-Weaving `Filter` via Safety Games

Each policy-weaving problem can be reduced to finding a winning strategy to a safety game. A safety game is played by two players, an Attacker and Defender, and is a transition system in which each state belongs to either the Attacker or the Defender. The goal of the Attacker is to drive the state of the game to an accepting state, while the goal of the Defender is to thwart the Attacker. The game is played in turns: when the game enters an Attacker state, the Attacker chooses the next transition, and when the game enters a Defender state, the Defender chooses the next transition. A strategy for the Defender takes as input a play of the game, and chooses the next transition for the Defender. A winning strategy chooses Defender transitions that never allow the Attacker to drive the state of the game to an accepting state.

From program, policy, and monitor automata, we can construct a game that accepts all policy-violating executions of a version of the program that is instrumented to invoke the primitives of the monitor. The game is constructed by (1) transforming the alphabets of the automata to a common alphabet defined by the instructions, privileges, and primitives, (2) constructing the product of the transformed automata, and (3) transforming the alphabet of the resulting product game so that all Attacker transitions are labeled with program instructions, and all Defender transitions are labeled with system primitives.

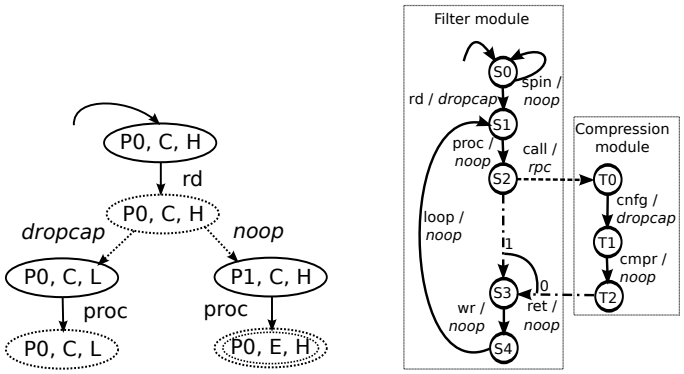

Fig. 2. (a) a selection of transitions of the game G_{ex} that is the product of F, Pol, and M; (b) a strategy corresponding to a correct instrumentation of `Filter`.

A subset of the transitions of the game G_{ex} constructed from F, Pol, and M are shown in Fig. 2(a). Each state of G_{ex} is either an Attacker or Defender state constructed from a triple of a state of F, state of Pol, and state of M, and each state in Fig. 2(a) is labeled with its triple. Each Attacker state and Attacker transition is denoted with a solid circle or line, while each Defender state is denoted with a dotted circle or line. The play "`read`, `noop`, `proc`" is accepted by G_{ex} (i.e., is a winning play for the Attacker) because it is an execution in which the instrumented `Filter` does not execute `dropcap` before executing `proc`, causing `proc` to execute with high-privilege, which violates the policy Pol. However, the play "`read`, `dropcap`, `proc`" is not accepted by G_{ex}, because it corresponds to an execution in which `Filter` invokes `dropcap`, causing `proc` to execute with low privilege, which satisfies the policy.

One winning Defender strategy to G_{ex}, which corresponds to the correct instrumentation of `Filter` given in §2.1, is presented in Fig. 2(b). The strategy is a transducer that, from its current state, reads an instruction executed by `Filter`, outputs the primitive paired with the instruction on the label of a transition t (Fig. 2(b) includes a primitive `noop` that denotes that no MiniCap primitive is invoked), transitions on t, and reads the next instruction. The strategy is partitioned into a `Filter` module that chooses what primitives are invoked during an execution of `Filter`, and a `Compress` module that chooses primitives are invoked during the execution of `Compress`. The modules are independent, in that the primitives chosen by the `Compress` module are independent of the instructions and primitives executed by `Filter` before the most recent call of `Compress`.

Solving games constructed from policy-weaving problems efficiently is a hard problem. The game G_{ex} is the product of F, Pol, and M, and thus has a state space whose size is proportional to the product of the sizes of F, Pol, and M (G_{ex} has 128 states). Furthermore, finding modular winning Defender strategies to games is NP-complete in the size of the game. However, in practice, games constructed from policy-weaving problems have a winning strategy whose structure closely

matches the structure of one of the input automata. For example, the winning strategy in Fig. 2(b) closely matches the structure of F. Each execution of `Filter` is in state F_n of F when the strategy is in state S_n, and in state C_n of F when the strategy is in state T_n (see Fig. 1(a) and Fig. 2(b)). To find winning modular strategies to games efficiently, we apply a novel algorithm that takes a game and an additional, potentially smaller, game called a *scaffold*, and searches for a winning strategy whose structure is similar to that of the scaffold. For G_{ex}, F serves as such a scaffold.

3 Policy Weaving as a Safety Game

3.1 Definition of the Policy-Weaving Problem

The policy-weaving problem is to take a program, a description of a privilege-aware system, and a policy that describes what privileges the program must have as it executes on the system, and to instrument the program so that it always has the privileges required by the policy. We model a program, policy, and privilege-aware system each as a Visibly Pushdown Automaton.

Definition 1. A *deterministic visibly-pushdown automaton (*VPA*)* for internal actions Σ_I, call actions Σ_C, and return actions Σ_R (alternatively, a $(\Sigma_I, \Sigma_C, \Sigma_R)$-VPA) is a tuple $V = (Q, \iota, Q_F, \tau_i, \tau_c, \tau_r)$, where: Q is the set of *states*; $\iota \in Q$ is the *initial state*; $Q_F \subseteq Q$ is the set of *accepting states*; $\tau_i : Q \times \Sigma_i \to Q$ is the *internal transition function*; $\tau_c : Q \times \Sigma_c \to Q$ is the *call transition function*; $\tau_r : Q \times \Sigma_r \times Q \to Q$ is the *return transition function*.

For $\widehat{\Sigma} = \Sigma_I \cup \Sigma_C \cup \Sigma_R$, each VPA accepts a set of *traces* of (i.e., sequences of actions in) $\widehat{\Sigma}$. Let ϵ denote the empty sequence. Let "." denote the concatenation of two sequences; for set X, $x \in X$, and $s \in X^*$, $x \mathbin{.} s = [x] \mathbin{.} s$ and $s \mathbin{.} x = s \mathbin{.} [x]$, where $[x] \in X^*$ is the sequence containing only x. For sets X_0 and X_1, let $X_0 \mathbin{.} X_1$ be the set of all sequences $x_0 \mathbin{.} x_1$ for $x_0 \in X_0$ and $x_1 \in X_1$. Let $\tau : Q^* \times \widehat{\Sigma} \to Q^*$ map a sequence of states $s \in Q$ (i.e., a *stack*) and action $a \in \widehat{\Sigma}$ to the stack to which V transitions from s on a:

$$\begin{aligned} \tau(q \mathbin{.} s, a) &= \tau_I(q, a) \mathbin{.} s && \text{for } a \in \Sigma_I \\ \tau(q \mathbin{.} s, a) &= (\tau_C(q, a) \mathbin{.} q \mathbin{.} s) && \text{for } a \in \Sigma_C \\ \tau((q \mathbin{.} s_0 \mathbin{.} s'), a) &= (\tau_R(q, a, s_0), s') && \text{for } a \in \Sigma_R \end{aligned}$$

Let $\rho : \widehat{\Sigma}^* \to Q^*$ map each trace to the stack that V is in after reading the trace. Formally, $\rho(\epsilon) = \iota$, and for $a \in \widehat{\Sigma}$ and $s \in \widehat{\Sigma}^*$, $\rho(s \mathbin{.} a) = \tau(\rho(s), a)$. A trace $t \in \widehat{\Sigma}^*$ is accepted by V if $\rho(t) = (q, s)$ with $q \in Q_F$. In a trace t, an instance c of a call action is *matched* by an instance r of a return action if c is before r in t, and each instance c' of a call action in t between c and r is matched by a return action r' between c and r. A trace is *matched* if all call and return actions in the string are matched. Let $\mathcal{L}(V)$ be the set of all traces accepted by V. □

A program is a language of traces of intraprocedural instructions, calls, and returns of the program (e.g., for `Filter` in §2, `spin`, `read`, etc.). Let $\mathsf{Instrs} = (\Sigma_I, \Sigma_C, \Sigma_R)$, let $\widehat{\mathsf{Instrs}} = \Sigma_I \cup \Sigma_C \cup \Sigma_R$, and let a program P be an Instrs-VPA.

A program policy is a language of traces of program instructions paired with privileges. A program's privilege is a system-specific ability (e.g., for MiniCap in §2, a program may have either the high privilege H or the low privilege L). Let Privs be a set of privileges, and let the set of *privileged executions of* P be $(\widehat{\mathsf{Instrs}} \times \mathsf{Privs})^*$. Let an $(\mathsf{Instrs}, \mathsf{Privs})$-*policy for* P be a $(\Sigma_I \times \mathsf{Privs}, \Sigma_C \times \mathsf{Privs}, \Sigma_R, \times\mathsf{Privs})$-VPA (e.g., Fig. 1(c)) that accepts all privileged executions that constitute violations.

A privilege-aware monitor is a language of privileged executions interleaved with primitives. The primitives of a privilege-aware system are the set of security-specific system calls that the application can invoke to manage its privileges (e.g., for MiniCap, the system call `dropcap`). Let Prims be a set of primitives and let the *instrumented executions* of P be $(\mathsf{Prims} \ . \ \mathsf{Instrs})^*$. A privilege-aware monitor of P reads an instrumented execution of P, and decides what privilege P has as it executes each instruction. Let an $(\mathsf{Instrs}, \mathsf{Privs}, \mathsf{Prims})$-privilege-aware monitor M be a $((\Sigma_I \times \mathsf{Privs}) \cup \mathsf{Prims}, \Sigma_C \times \mathsf{Privs}, \Sigma_R \times \mathsf{Privs})$-VPA.

Definition 2. (Policy-Weaving Problem) Let P be a program with internal, call, and return alphabets $\mathsf{Instrs} = (\Sigma_I, \Sigma_C, \Sigma_R)$. For privileges Privs, let Pol be an $(\mathsf{Instrs}, \mathsf{Privs})$-policy of P. For primitives Prims, let M be an $(\mathsf{Instrs}, \mathsf{Privs}, \mathsf{Prims})$-privilege-aware monitor.

Let an *instrumentation function* be a function $I : \mathsf{Instrs}^* \to \mathsf{Prims}$, and let $I_{\mathrm{tr}} : \mathsf{Instrs}^* \to (\mathsf{Prims}\ \mathsf{Instrs})^*$ map each sequence of instructions to the instrumentation of the sequence defined by I: $I_{\mathrm{tr}}(\epsilon) = I(\epsilon)$, and for $s \in \mathsf{Instrs}^*$ and $a \in \mathsf{Instrs}$, $I_{\mathrm{tr}}(s \ . \ a) = I_{\mathrm{tr}}(s) \ . \ I(s \ . \ a) \ . \ a$. Let $\mathsf{PrivEx}_{\mathsf{M}} : (\mathsf{Prims}\ \mathsf{Instrs})^* \to (\mathsf{Instrs} \times \mathsf{Privs})^*$ map each instrumented execution to the privileged execution that it induces on M: for primitives p_j, instructions i_j, and privileges r_j, $\mathsf{PrivEx}_{\mathsf{M}}([p_0, i_0, \ldots, p_n, i_n]) = [(p_0, r_0), \ldots, (p_n, r_n)]$ if and only if $[p_0, (i_0, r_0), \ldots, p_n, (i_n, r_n)] \in \mathcal{L}(\mathsf{M})$.

The *policy-weaving problem* PolWeave$(\mathsf{P}, \mathsf{Pol}, \mathsf{M})$ is to find an instrumentation function I such that:

1. I instruments P to never violate the Pol: $\mathsf{PrivEx}_{\mathsf{M}}(I_{\mathrm{tr}}(\mathcal{L}(\mathsf{P}))) \cap \mathcal{L}(\mathsf{Pol}) = \emptyset$.
2. I chooses primitives independently of the execution before the most recent call; i.e., I is *modular*. Let $p^0, p^1 \in \mathsf{Img}(I_{\mathrm{tr}})$, (where for a relation R, $\mathsf{Img}(R)$ is the image of R), and $p^0 = p^0_0 \ . \ c \ . \ p^0_1 \ . \ r_0 \ . \ p^0_2$, $p^1 = p^1_0 \ . \ c \ . \ p^1_1 \ . \ r_1 \ . \ p^1_2$, call action c is matched by r_0 in p^0, and is matched by r_1 in p^1. Let $p^0_1 = a_0, b^0_0, a_1, b^0_1, \ldots, a_n, b^0_n$, and let $p^1_1 = a_0, b^1_0, a_1, b^1_1, \ldots, a_n, b^1_n$. Then $b^0_i = b^1_i$ for each i in each such p^0 and p^1. □

Defn. 2 formalizes the informal policy-weaving problem illustrated in §2. As discussed in §4.1, if a policy-weaving problem has a solution I, then it has a solution I^* that may be represented as a VPA transducer T (i.e., a VPA where each action is labeled with input and output symbols). The problem of rewriting program P to satisfy the policy thus amounts to applying T to P, using a standard product construction from automata theory.

Privilege-aware systems are typically applied to monitor programs that could run injected code, yet an instrumentation function is defined in Defn. 2 to choose

a primitive after each instruction executed by the program. However, this is not a fundamental limitation, as if a programmer or static analysis tool determines that injected code might be run at a particular point in the program, then we can define the monitor so that no primitive other than a `noop` can be invoked by the instrumentation. Conversely, it is not too restrictive to only allow an instrumentation function to invoke a single primitive after each instruction, as we can rewrite the program to execute a sequence of security-irrelevant instructions between which the instrumentation can invoke a sequence of primitives. In App. A, we present two different privilege-aware systems as VPA.

3.2 From Policy Weaving to Safety Games

Each policy-weaving problem POLWEAVE($\mathsf{P}, \mathsf{Pol}, \mathsf{M}$) can be reduced to a *single-entry* VPA *(*SEVPA *[4]) safety game* that accepts plays corresponding to instrumented executions of P that violate Pol when run on M. A SEVPA safety game is a VPA structured as a set of modules with unique entry points whose transitions are decided in turn by an Attacker and a Defender. The states of the game are partitioned into modules, where the system transitions to a unique module on each call transition.

Definition 3. A SEVPA *safety game* for Attacker internal actions $\Sigma_{I,A}$, Defender internal actions Σ_D, call actions Σ_C, and return actions Σ_R is a tuple $\mathsf{G} = (Q_A, Q_D, Q_0, \iota_0, \{(Q_c, \iota_c)\}_{c\in\Sigma_C}, Q_F, \tau_{I,A}, \tau_D, \tau_R)$, where

- $Q_A \subseteq Q$ is a finite set of *Attacker states*.
- $Q_D \subseteq Q$ is a finite set of *Defender states*. Q_A and Q_D partition the states of the game Q.
- Q_0 is the *initial module*.
- $\iota_0 \in Q_0 \cap Q_D$ is the *initial state*.
- For $c \in \Sigma_C$, Q_c is the *module of* c. The sets $\{Q_c\}_{c\in\Sigma_C}$ and Q_0 are pairwise disjoint, and partition Q.
- For $c \in \Sigma_C$, $\iota_c \in Q_c \cap Q_D$ is the *initial state of* c.
- $Q_F \subseteq Q_0 \cap Q_D$ is the set of *accepting states*.
- $\tau_{I,A} : Q_A \times \Sigma_{I,A} \to Q_D$ is the *Attacker internal transition function*.
- $\tau_D : Q_D \times \Sigma_D \to Q_A$ is the *Defender internal transition function*.
- $\tau_R : Q_A \times \Sigma_R \times (Q_A \times \Sigma_C) \to Q_D$ is the *return transition function*.

The modules are closed under internal transitions: for $x \in \{0\} \cup \Sigma_C$, $q \in Q_x$, and $a \in \Sigma_{I,A}$, $\tau_{I,A}(q, a) \in Q_x$, and for $a \in \Sigma_D$, $\tau_D(q, a) \in Q_x$. A SEVPA safety game is not defined by using an explicit call transition function, because each call on an action c pushes on the stack the calling Attacker state and calling action (we thus call $\Gamma = Q_A \times \Sigma_C$ the *stack symbols of the game*), and transitions to ι_c. The modules of a SEVPA safety game are closed under matching calls and returns: for $x \in \{0\} \cup \Sigma_C$, $c \in \Sigma_C$, $q_x \in Q_x$, $q_c \in Q_c$, and $r \in \Sigma_R$, $\tau_R(q_c, r, (q_x, c)) \in Q_x$.

The *plays* of a SEVPA are defined analogously to the traces of a VPA. Let the *configurations of* G be $C = Q \times \Gamma^*$, let the *attacker configurations* be $C_D = C \cap (Q_A \times \Gamma^*)$, and let the *defender configurations* be $C_D = C \cap (Q_D \times \Gamma^*)$. Let the *Attacker actions* be $\Sigma_A = \Sigma_{I,A} \cup \Sigma_C \cup \Sigma_R$. $\tau_A : C_A \times \Sigma_A \to C_D$ maps each

Attacker configuration and Attacker action to a Defender configuration:

$$\begin{aligned}
\tau_A((q,s),a) &= (\tau_{I,A}(q),s) && \text{for } a \in \Sigma_{I,A} \\
\tau_A((q,s),a) &= (\iota_c,(q,a)\ .\ s) && \text{for } a \in \Sigma_C \\
\tau_A((q,s_0\ .\ s'),a) &= (\tau_R(q,a,s_0),s') && \text{for } a \in \Sigma_R
\end{aligned}$$

Because each transition on a Defender action is to an Attacker state and each transition on an Attacker action is to a Defender state, all plays that transition to a defined configuration are in $(\Sigma_D\ .\ \Sigma_A)^*$. Let $\rho : (\Sigma_D\ .\ \Sigma_A)^* \to C_D$ map each play of alternating Defender and Attacker actions to the configuration that the game transitions to from reading the play: $\rho(\epsilon) = (\iota_0, \epsilon)$, and $\rho(p\ .\ a\ .\ b) = \tau_A(\tau_D(\rho(p), a), b)$. A play $p \in (\Sigma_D\ .\ \Sigma_A)^*$ is accepted by G if $\rho(p) = (q, \epsilon)$ with $q \in Q_F$. Let $\mathcal{L}(\mathsf{G})$ be the set of all plays accepted by G. □

Because all accepting states of a game are in the initial module, a game can only accept matched plays. Superscripts denote the VPA or SEVPA game to which various components belong; e.g., Q^{G} are the states of SEVPA game G.

A *Defender strategy* of a two-player safety game G is a function $\sigma : (\Sigma_A^{\mathsf{G}})^* \to \Sigma_D^{\mathsf{G}}$ that takes as input a sequence of Attacker actions, and outputs a Defender action. σ is a *winning strategy* if as long as the Defender uses it to choose his next transition of the game, the resulting play is not accepted by G: formally, $\sigma_{\text{tr}}(\Sigma_A^{\mathsf{G}}) \cap \mathcal{L}(\mathsf{G}) = \emptyset$ (for σ_{tr} as defined in Defn. 2). Let σ be modular if it satisfies the condition analogous to a modular instrumentation function (Defn. 2).

Theorem 1. *For each policy-weaving problem* $\mathcal{P} = \textsc{PolWeave}(\mathsf{P}, \mathsf{Pol}, \mathsf{M})$*, there is a* SEVPA *safety game* $\mathcal{G} = \mathsf{PolWeaveGame}(\mathsf{P}, \mathsf{Pol}, \mathsf{M})$ *such that each instrumentation function that satisfies* $\mathcal{P}$ *defines a winning modular Defender strategy of* $\mathcal{G}$*, and each winning modular Defender strategy of* $\mathcal{G}$ *defines a satisfying instrumentation function of* $\mathcal{P}$*.*

The intuition behind the construction of $\mathcal{G}$ from $\mathcal{P} = \textsc{PolWeave}(\mathsf{P}, \mathsf{Pol}, \mathsf{M})$ is given in §2.2. From P, we construct a game $\mathsf{G_P}$ that accepts all instrumented privileged instrumented executions of P. From Pol, we constructed a game $\mathsf{G_{Pol}}$ that accepts all instrumented privileged executions that violate Pol. We construct $\mathcal{G}$ as the product of $\mathsf{G_P}$, $\mathsf{G_{Pol}}$, and $\mathsf{G_M}$. Proofs of all theorems stated in §3 and §4 are in App. B.

4 Solving SEVPA Safety Games with Scaffolds

In this section, we present an algorithm $\mathsf{ScafAlgo}$ that finds a winning modular Defender strategy to a given SEVPA safety game. The algorithm uses an additional, potentially smaller game, which we call a *scaffold*. We present $\mathsf{ScafAlgo}$ as a non-deterministic algorithm, and demonstrate that a symbolic implementation builds a formula whose size is decided entirely by the size of the scaffold and an additional, tunable independent parameter. We describe a known algorithm for finding modular strategies [6] and a known symbolic algorithm for finding strategies of bounded size [18] as instances of $\mathsf{ScafAlgo}$.

4.1 Definition and Key Properties of Scaffolds

The key characteristic of our algorithm is that it finds a winning Defender strategy to a given game using an additional game, called a scaffold, and a specified relation between the states of the scaffold and the states of the game.

Definition 4. (Scaffolds) Let S and G be two SEVPA safety games defined for Attacker actions $\Sigma_{I,A}$, Defender actions Σ_D, call actions Σ_C, and return actions Σ_R. S is a *scaffold* of G under $\mathcal{R} \subseteq Q^{\mathsf{S}} \times Q^{\mathsf{G}}$ if and only if:

1. If $q_{\mathsf{S}} \in Q^{\mathsf{S}}_F$ and for $q_{\mathsf{G}} \in Q^{\mathsf{G}}$, $\mathcal{R}(q_{\mathsf{S}}, q_{\mathsf{G}})$, then $q_{\mathsf{G}} \in Q^{\mathsf{G}}_F$.
2. For $c \in \Sigma_C$, $\mathcal{R}(\iota^{\mathsf{S}}_c, \iota^{\mathsf{G}}_c)$.
3. For $a \in \Sigma_{I,A}$, $q_{\mathsf{S}} \in Q^{\mathsf{S}}_A$, and $q_{\mathsf{G}} \in Q^{\mathsf{G}}_A$, if $\mathcal{R}(q_{\mathsf{S}}, q_{\mathsf{G}})$, then $\mathcal{R}(\tau_{I,A}(q_{\mathsf{S}}, a), \tau_{I,A}(q_{\mathsf{G}}, a))$.
4. For $a \in \Sigma_D$, $q_{\mathsf{S}} \in Q^{\mathsf{S}}_D$, and $q_{\mathsf{G}} \in Q^{\mathsf{G}}_D$, if $\mathcal{R}(q_{\mathsf{S}}, q_{\mathsf{G}})$, then $\mathcal{R}(\tau_D(q_{\mathsf{S}}, a), \tau_D(q_{\mathsf{G}}, a))$.
5. For $c \in \Sigma_C$, $q^{\mathsf{S}}_c \in Q^{\mathsf{S}}_c$, $q^{\mathsf{G}}_c \in Q^{\mathsf{G}}_c$, $q^{\mathsf{S}} \in Q^{\mathsf{S}}$, $q^{\mathsf{G}} \in Q^{\mathsf{G}}$, if $\mathcal{R}(q^{\mathsf{S}}_c, q^{\mathsf{G}}_c)$ and $\mathcal{R}(q^{\mathsf{S}}, q^{\mathsf{G}})$, then $\mathcal{R}(\tau_R(q^{\mathsf{S}}, r, (q^{\mathsf{S}}_c, c)), \tau_R(q^{\mathsf{G}}, r, (q^{\mathsf{G}}_c, c)))$. □

If so, then $\mathcal{R}$ is a *scaffold relation* from S to G.

Each scaffolding relation $\mathcal{R}$ defines an Attacker simulation [3], but there are SEVPA related by an Attacker simulation that are not related by any scaffold relation. However, scaffold relations and modular strategies are connected by the following key property, which provides the foundation for our algorithm. First, we define an $(\mathsf{S}, \mathcal{R}, k)$-strategy of a game G, which intuitively is a strategy whose structure tightly corresponds to a scaffold S, according to a relation $\mathcal{R}$ from the states of S to those of G. For a game G and $Q' \subseteq Q^{\mathsf{G}}$ such that $\{\iota_c\}_{c \in \Sigma^{\mathsf{G}}_C} \subseteq Q' \subseteq Q^{\mathsf{G}}$, let the *subgame* $\mathsf{G}|_{Q'}$ be the game constructed from restricting the states and transition functions of G to the states in Q'. Each subgame G' of G defines a strategy $\sigma_{\mathsf{G}'}$ as a VPA transducer. To compute $\sigma_{\mathsf{G}'}(a_0, a_1, \ldots, a_n)$, $\sigma_{\mathsf{G}'}$ uses $a_0, a_1, \ldots, a_n$ as the Attacker actions for a play of G'. If G' is in an attacker state, then σ_{G} transitions it on the next unread a_i, and if G' is in a Defender state, then $\sigma_{\mathsf{G}'}$ picks a fixed Defender transition of G' on which to transition. $\sigma_{\mathsf{G}'}$ outputs the Defender action chosen by G' after it reads all of $a_0, a_1, \ldots, a_n$. This construction is standard, and we leave a full definition for App. B.

Definition 5. For sets A and B, let a relation $\mathcal{R} \subseteq A \times B$ be *k-image-bounded* if for each $a \in A$, $|\{b \mid b \in B, \mathcal{R}(a, b)\}| \leq k$. Let G be a game, let S be a scaffold of G under $\mathcal{R} \subseteq Q^{\mathsf{S}} \times Q^{\mathsf{G}}$, and let $k \in \mathbb{Z}$. An *$(\mathsf{S}, \mathcal{R}, k)$-Defender strategy* σ' of G is a Defender strategy such that for some $\mathcal{R}' \subseteq \mathcal{R}$, $\mathcal{R}' \cap (Q^{\mathsf{S}}_A \times Q^{\mathsf{G}}_A)$ is k-image-bounded, $\mathsf{G}' = \mathsf{G}|_{\mathsf{Img}(\mathcal{R}')}$, and $\sigma' = \sigma_{\mathsf{G}'}$. □

Let game G have a winning Defender strategy, and let S be a scaffold of G under a scaffold relation $\mathcal{R}$. Then S is a scaffold of some subgame of G' that defines a winning strategy of G, under a finer scaffold relation than $\mathcal{R}$.

Theorem 2. *Let G have a winning modular Defender strategy, and let S be a scaffold of G under $\mathcal{R} \subseteq Q^{\mathsf{S}} \times Q^{\mathsf{G}}$. Then for some k, there is a winning modular $(\mathsf{S}, \mathcal{R}, k)$-Defender strategy of G.*

Input: G: a VPA safety game.
S: a scaffold of G
$\mathcal{R} \subseteq Q^{\mathsf{S}} \times Q^{\mathsf{G}}$: a scaffold relation.
Output: If G has a winning $(S, \mathcal{R}, k)$-strategy, then it returns a winning $(S, \mathcal{R}, k)$-strategy. Otherwise, it returns $\bot$.

`/* Choose` $\mathcal{R}_A$`: a` k`-image-bounded subrelation of` $\mathcal{R}$ `that defines Attacker states of a candidate strategy. */`

1 $\mathcal{R}_A := \mathsf{nd\text{-}bounded\text{-}subrel}(\mathcal{R} \cap (Q^{\mathsf{S}}_A \times Q^{\mathsf{G}}_A), k)$;

`/* Construct` $\mathcal{R}_D$`: a relation to Defender states of the candidate strategy defined by` $\mathcal{R}_A$`. */`

2 $\mathcal{R}_{D,\iota} := \{(\iota^{\mathsf{S}}_c, \iota^{\mathsf{G}}_c) \mid c \in \Sigma^{\mathsf{G}}_C\}$;

3 $\mathcal{R}_{D,i} := \{(\tau^{\mathsf{S}}_{I,A}(p_A, a), \tau^{\mathsf{G}}_{I,A}(q_A, a)) \mid (p_A, q_A) \in \mathcal{R}_A, a \in \Sigma^{\mathsf{G}}_{I,A}\}$;

4 $\mathcal{R}_{D,r} := \{(\tau^{\mathsf{S}}_R(p_A, a, s_A), \tau^{\mathsf{G}}_R(q_A, a, t_A)) \mid (p_A, q_A), (s_A, t_A) \in \mathcal{R}_A, a \in \Sigma^{\mathsf{G}}_R\}$;

5 $\mathcal{R}_D := \mathcal{R}_{D,\iota} \cup \mathcal{R}_{D,i} \cup \mathcal{R}_{D,r}$;

`/* Check if the candidate strategy defined by` $\mathcal{R}_A$ `and` $\mathcal{R}_D$ `is a winning strategy. */`

6 $\mathsf{StrWins} := \forall (p_D, q_D) \in \mathcal{R}_D : q_D \notin Q_F \wedge \exists a \in \Sigma_D : (\tau^{\mathsf{S}}_D(p_D, a), \tau^{\mathsf{G}}_D(q_D, a)) \in \mathcal{R}_A$;

7 **if** $\mathsf{StrWins}$ **then return** $\sigma_{\mathsf{G}|_{\mathsf{Img}(\mathcal{R}_A \cup \mathcal{R}_D)}}$ **else return** $\bot$

Algorithm 1. ScafAlgo: non-deterministic algorithm that takes a game G, scaffold S, and relation $\mathcal{R} \subseteq Q^{\mathsf{S}} \times Q^{\mathsf{G}}$, and finds a winning modular Defender $(\mathsf{S}, \mathcal{R}, k)$-strategy of G.

4.2 An Algorithm Parametrized on Scaffolds

To find modular winning Defender strategies to games, we can apply Thm. 2 to search for $(\mathsf{S}, \mathcal{R}, k)$-strategies. The algorithm ScafAlgo, given in Alg. 1, takes a game G, scaffold S, relation $\mathcal{R} \subseteq Q^{\mathsf{S}} \times Q^{\mathsf{G}}$, and parameter k, and searches for an $(\mathsf{S}, \mathcal{R}, k)$-strategy by searching for an $\mathcal{R}' \subseteq \mathcal{R}$ that satisfies the condition given in Defn. 5.

ScafAlgo searches for such an $\mathcal{R}'$ in three main steps. In the first step, ScafAlgo non-deterministically chooses a k-image-bounded subrelation of $\mathcal{R}$ from the Attacker states of S to the Attacker states of G. Specifically, on line [1], ScafAlgo defines such a relation $\mathcal{R}_A \subseteq Q^{\mathsf{S}}_A \times Q^{\mathsf{G}}_A$ by calling a function $\mathsf{nd\text{-}bounded\text{-}subrel} : (Q^{\mathsf{S}}_A \times Q^{\mathsf{G}}_A) \times \mathbb{Z} \to (Q^{\mathsf{S}}_A \times Q^{\mathsf{G}}_A)$, where $\mathsf{nd\text{-}bounded\text{-}subrel}(\mathcal{R} \cap (Q^{\mathsf{S}}_A \times Q^{\mathsf{G}}_A), k)$ is a k-image-bounded subrelation of $Q^{\mathsf{S}}_A \times Q^{\mathsf{G}}_A$.

In the second step (lines [2]–[5]), ScafAlgo constructs a relation $\mathcal{R}_D \subseteq Q^{\mathsf{S}}_D \times Q^{\mathsf{G}}_D$ such that if there is any $\mathcal{R}^* \subseteq Q^{\mathsf{S}}_D \times Q^{\mathsf{G}}_D$ such that $\mathsf{G}|_{\mathsf{Img}(\mathcal{R}_A \cup \mathcal{R}^*)}$ defines a winning strategy of G, then the *candidate strategy* defined by $\mathsf{G}|_{\mathsf{Img}(\mathcal{R}_A \cup \mathcal{R}_D)}$ is a winning strategy of G. On line [2], ScafAlgo defines $\mathcal{R}_{D,\iota} \subseteq Q^{\mathsf{S}}_D \times Q^{\mathsf{G}}_D$ that relates each module-initial state of S to its corresponding module-initial state in G. On line [3], ScafAlgo defines $\mathcal{R}_{D,i} \subseteq Q^{\mathsf{S}}_D \times Q^{\mathsf{G}}_D$ that, for each $(p_A, q_A) \in \mathcal{R}_A$ and internal Attacker action $a \in \Sigma^{\mathsf{G}}_{I,A}$, relates the a-successor of p_A to the a-successor of q_A. On line [4], ScafAlgo defines $\mathcal{R}_{D,r} \subseteq Q^{\mathsf{S}}_D \times Q^{\mathsf{G}}_D$ that, for each $(p_A, q_A), (s_A, t_A) \in \mathcal{R}_A$ and return action $a \in \Sigma^{\mathsf{G}}_R$, relates the r-successor of (p_A, s_A) to the r-successor of (q_A, t_A). On line [5], ScafAlgo defines $\mathcal{R}_D \subseteq Q^{\mathsf{S}}_D \times Q^{\mathsf{G}}_D$ as the union of $\mathcal{R}_\iota$, $\mathcal{R}_{D,i}$, and $\mathcal{R}_{D,r}$.

In the third step (lines [6] and [7]), ScafAlgo checks if the candidate strategy defined by $\mathsf{G}|_{\mathsf{Img}(\mathcal{R}_A \cup \mathcal{R}_D)}$ is a winning strategy of G. On line [6], ScafAlgo defines

$\mathsf{StrWins} : \mathbb{B}$, which is true if and only if for each Defender-state of the candidate strategy, the state is not an accepting state of the game, and there is some action that the Defender can take to reach some Attacker-state of the candidate strategy. On line [7], ScafAlgo returns the strategy defined by $\mathsf{G}|_{\mathsf{Img}(\mathcal{R}_A \cup \mathcal{R}_D)}$ if and only if $\mathsf{G}|_{\mathsf{Img}(\mathcal{R}_A \cup \mathcal{R}_D)}$ is a winning strategy. Otherwise, ScafAlgo returns failure.

Theorem 3. *Let* G *be a game, let* S *be a scaffold of* G *under* $\mathcal{R} \subseteq Q^{\mathsf{S}} \times Q^{\mathsf{G}}$*, and let* k *be a positive integer. If* $\sigma = \mathsf{ScafAlgo}(\mathsf{G}, \mathsf{S}, \mathcal{R}, k)$*, then* σ *is a winning Defender strategy for* G*. If* G *has a winning Defender strategy, then for each scaffold* S *and scaffolding relation* $\mathcal{R} \subseteq Q^{\mathsf{S}} \times Q^{\mathsf{G}}$*, there is some* k *such that* $\mathsf{ScafAlgo}(\mathsf{G}, \mathsf{S}, \mathcal{R}, k)$ *is a winning Defender strategy of* G*.*

A deterministic implementation of ScafAlgo runs in worst-case time exponential in the number of Attacker states. However, a symbolic implementation of ScafAlgo can represent its input problem with a formula whose size depends only on the scaffold, and the tunable parameter k. Assume that each component of an input game G is given as interpreted symbolic functions and predicates (i.e., states and actions are given as domains, and the transition functions are given as interpreted functions), and that the relation $\mathcal{R}$ is given as an interpreted relation. Then ScafAlgo can be implemented by reinterpreting its steps to build a symbolic formula StrWins (line [6]) whose models correspond to values of $\mathcal{R}_A$ and $\mathcal{R}_D$ for which $\mathsf{G}|_{\mathsf{Img}(\mathcal{R}_A \cup \mathcal{R}_D)}$ defines a winning strategy.

The size (i.e., the number of literals in) of StrWins is determined by S and k. The universal quantification on line [6] is bounded, and can thus be encoded as a finite conjunction; the nested existential quantification can then be Skolemized. To check the membership $(\tau_D^{\mathsf{S}}(p_D, a), \tau_D^{\mathsf{G}}(q_D, a)) \in \mathcal{R}_A$, we can apply the fact that $\mathcal{R}_A$ is a k-bounded-image relation to represent the membership check with k disjuncts. From these observations, the size of the StrWins formula can be bounded by $O(|Q_A^{\mathsf{S}}|^2 k^3)$.

Two known algorithms for finding modular strategies can be defined as ScafAlgo applied to degenerate scaffolds. A naive implementation of the original algorithm presented for finding modular strategies [6] can be defined as ScafAlgo applied to the game itself as a scaffold. A symbolic algorithm for finding strategies of bounded size [18], generalized to Vpa games, can be defined as ScafAlgo applied to a scaffold with a single Attacker and Defender state for each module. The known algorithms are thus ScafAlgo applied to scaffolds that have complete and no information about their games, respectively. However, any game defined as a product of "factor" games, such as games constructed from the policy-weaving reduction, is scaffolded by its factors under a relation that relates each factor state to the product states for which it is a component. See App. C for a further discussion of known algorithms as instances of ScafAlgo.

5 Experiments

In this section, we discuss experiments that evaluate the reduction from policy-weaving problems to safety games presented in §3, and the scaffold-based game-solving algorithm presented in §4. The experiments were designed to answer two questions. First, by reducing policy-weaving problems to solving games, can we

Name	LoC	Pol. States	Scaffolds					
			Triv.		Prog.-Pol.		Prog.-Pol.-Mon.	
			k	Time	k	Time	k	Time
bzip2-1.0.6	8,399	12	12	-	1	0:04	1	0:09
fetchmail-6.3.19	49,370	12	7	-	1	1:13	1	1:39
gzip-1.2.4	9,076	9	12	-	1	1:47	1	-
tar-1.25	108,723	12	3	3:47	1	1:20	1	-
tcpdump-4.1.1	87,593	12	15	-	1	0:30	1	0:45
wget-1.12	64,443	21	7	0:43	1	0:25	1	18:59

Table 1. Performance of the Capsicum policy weaver. Column "LoC" contains lines of C source code (not including blank lines and comments), "Pol. States" contains the number of states in the policy, k contains the simulation bound, and "Times" contains the times used to find a strategy using the complete, trivial, and factor scaffolds. "-" denotes a time-out of 20 minutes.

efficiently instrument practical programs for a real privilege-aware system so that they satisfy practical high-level policies? Second, which scaffolds allow our scaffolding game-solving algorithm to most efficiently solve games constructed by our policy-weaving algorithm?

To answer these questions, we instantiated our policy-weaving algorithm to a policy weaver for the Capsicum [22] capability operating system. We collected a set of six UNIX utilties, given in Tab. 1, that have exhibited critical security vulnerabilities [16, 20–22]. For each utility, we defined a policy that describes the capabilities that the program must have as it executes. The policies were defined by working with the Capsicum developers, or using general knowledge of the utility. Detailed descriptions of the policies for each utility are given in [14].

We applied our Capsicum policy-weaver to each utility and its policy, each scaffold defined as a product of some subset of the program, policy, and monitor. The data from all scaffolds is given in App. D; Tab. 1 presents data for several illustrative scaffolds: the trivial scaffold "Triv." defined in §4.2, the product of the program and policy "Prog.-Pol", and the product of all program, policy, and monitor "Prog.-Pol.-Mon." For each scaffold, we measured how long it took our weaver to find a strategy, and with what minimum simulation bound (i.e., value of k from §4) it either found a strategy or timed out. The results for each scaffold are in the subcolumns of "Scaffolds" in Tab. 1, with each simulation bound in subcolumn "k," and each time in subcolumn "Time."

The results indicate that while many scaffolds give similar results for some practical problems, an *intermediate* scaffold constructed as a product of some but not all of the inputs, e.g. Prog.-Pol., leads to the best performance. The difference in performance could be due to fact that a scaffold with little information about the structure of its game (e.g., "Triv.") generates a formula that allows many transitions between a small set of states in a candidate strategy, while a scaffold with total information (e.g., Prog.-Pol.-Mon.) generates a formula that allows few transitions between a large set of states in a candidate strategy. An intermediate scaffold strikes a balance between the two, generating a formula that allows a

moderate number of transitions between a moderate set of states. We further discuss the experimental results in App. D.

6 Related Work

Privilege-aware operating systems: Decentralized Information Flow Control (DIFC) operating systems such as Asbestos [10], HiStar [23], and Flume [17] manage privileges describing how information may flow through a system, and provide primitives that allow an application to direct flows by managing the labels of each object in the system. Tagged memory systems such as Wedge [7] enforce similar policies per byte of memory by providing primitives for managing memory tags. Capability operating systems such as Capsicum [22] track the capabilities of each process, where a capability is a file descriptor paired with an access right, and provide primitives that allow an application to manage its capabilities.

Our work complements privilege-aware operating systems by allowing a programmer to give an explicit, high-level policy, and automatically rewriting the program to satisfy the policy when run on the system. Prior work in aiding programming for systems with security primitives automatically verifies that a program instrumented to use the Flume primitives enforces a high-level policy [15], automatically instruments programs to use the primitives of the HiStar to satisfy a policy [9], and automatically instruments programs [13] to use the primitives of the Flume OS. However, the languages of policies used in the approaches presented in [9, 13] are not temporal and cannot clearly be applied to other systems with security primitives, and the proofs of the correctness of the instrumentation algorithms are ad hoc. The work in [14] describes the approach in this paper instantiated to a policy weaving for Capsicum. This paper describes how the work in [14] may be generalized to arbitrary privilege aware systems, and describes the novel game-solving algorithm applied in [14].

Inlined Reference Monitors: An *Inlined Reference Monitor (IRM)* [1, 11] is code that executes in the same memory space as a program, observes the security-sensitive events of the program, and halts the program immediately before it violates a policy. IRMs have shortcomings that prohibit them from monitoring many practical programs and policies. Because an IRM executes in the same process space as the program it monitors, it cannot enforce policies throughout the system. Furthermore, an IRM must be able to monitor security-sensitive events of a program throughout the program's execution, but there are known techniques to subvert an IRM [1]. Privilege-aware operating systems address the shortcomings of IRM by monitoring policies in the operating system, and providing a set of primitives that an application invokes to direct the operating system. The primitives are distinct from the security-sensitive events of interest.

Safety Games: Automata-theoretic games formalize problems in synthesizing reactive programs and control mechanisms [2]. Alur et. al. give an algorithm that takes a single-entry recursive state machine and searches for a strategy that is modular, as defined in §3, and show that this problem is NP-complete [6]. Recursive state machines are directly analogous to Sevpa [4]. Madusudan et. al. give a set of symbolic algorithms that find a winning strategy for a given game

whose transition relation is represented symbolically [18]. The practical contribution of our work is that we express the emerging, important, and practical problem of rewriting programs for privilege-aware operating systems in terms of such games. We also give an algorithm for finding modular strategies that can be instantiated to a symbolic implementation of the algorithm of [6], to the "bounded-witness" algorithm of [18].

References

1. M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity. In *CCS*, 2005.
2. R. Alur, T. A. Henzinger, and O. Kupferman. Alternating-time temporal logic. In *FOCS*, 1997.
3. R. Alur, T. A. Henzinger, O. Kupferman, and M. Y. Vardi. Alternating refinement relations. In *CONCUR*, 1998.
4. R. Alur, V. Kumar, P. Madhusudan, and M. Viswanathan. Congruences for visibly pushdown languages. In *ICALP*, 2005.
5. R. Alur and P. Madhusudan. Visibly pushdown languages. In *STOC*, 2004.
6. R. Alur, S. L. Torre, and P. Madhusudan. Modular strategies for recursive game graphs. In *TACAS*, 2003.
7. A. Bittau, P. Marchenko, M. Handley, and B. Karp. Wedge: Splitting applications into reduced-privilege compartments. In *NSDI*, 2008.
8. D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7):504–513, 1977.
9. P. Efstathopoulos and E. Kohler. Manageable fine-grained information flow. In *EuroSys*, 2008.
10. 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.
11. Ú. Erlingsson and F. B. Schneider. IRM enforcement of Java stack inspection. In *IEEE SP*, 2000.
12. FreeBSD 9.0-RELEASE announcement. http://www.freebsd.org/releases/9.0R/announce.html, Jan. 2012.
13. W. R. Harris, S. Jha, and T. Reps. DIFC programs by automatic instrumentation. In *CCS*, 2010.
14. W. R. Harris, S. Jha, and T. Reps. Programming for a capability system via safety games. Technical report, University of Wisconsin-Madison, November 2011.
15. W. R. Harris, N. A. Kidd, S. Chaki, S. Jha, and T. Reps. Verifying information flow control over unbounded processes. In *FM*, 2009.
16. M. Izdebski. bzip2 'BZ2_decompress' function integer overflow vuln., Oct. 2011.
17. 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.
18. P. Madhusudan, W. Nam, and R. Alur. Symbolic computational techniques for solving games. *Electr. Notes Theor. Comput. Sci.*, 89(4), 2003.
19. R. Naraine. Symantec antivirus worm hole puts millions at risk. *eWeek.com*, 2006.
20. Ubuntu sec. notice USN-709-1. http://www.ubuntu.com/usn/usn-709-1/, 2009.
21. Vuln. note VU#381508. http://www.kb.cert.org/vuls/id/381508, July 2011.
22. R. N. M. Watson, J. Anderson, B. Laurie, and K. Kennaway. Capsicum: Practical capabilities for UNIX. In *USENIX Security*, 2010.
23. N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *OSDI*, 2006.

A Practical Privilege-Aware Systems as Automata

Practical privilege-aware systems can often be modeled as privilege-aware monitors as presented in §3. In this section, we informally model two apparently different privilege-aware systems, the Capsiscum capability system [22] and the HiStar decentralized information-flow control (DIFC) system [23].

A.1 Capsicum as a VPA

The Capsicum capability system is the inspiration for the MiniCap system presented in §2.1, but is a robust, practical system included in the "RELEASE" version of FreeBSD 9.0 [12]. The full implementation of Capsicum extends the standard UNIX file descriptor to a *capability*, which is a descriptor paired with a set of *access rights*, which denote the ability of a process to read, write, seek, etc. a file. Capsicum defines 63 rights, significantly enriching the standard set of rights provided by UNIX. As each process executes, Capsicum maps the process to its set of capabilities, and only allows a process to access a file in a particular way (e.g., to read it) if the process has a sufficient set of capabilities. When a process begins to execute, it may open descriptors to any files available to the user, allowing the process to grant itself coarse sets of capabilities. As a process executes, it may invoke a primitive that restricts its capabilities for a file descriptor, and it may invoke another primitive to enter "capability mode", which causes Capsicum to disallow it from opening additional descriptors. The primitives were designed for a programmer to rewrite their program to execute trusted code with high privilege, then restrict its capabilities by invoking the primitives, and then execute untrusted code with capabilities that are sufficiently reduced such that even a malicious or compromised module invoked by the program cannot violate a particular security policy.

Capsicum may be modeled as a privilege-aware system monitor as defined in §3. The privileges of the monitor are sets of capabilities. The primitives of the monitor are the system calls that a program invokes to restrict its capabilities or enter capability mode. The state of the monitor is the set of capabilities held by the program, and a Boolean that decides whether the program is in capability mode. A full description of Capsicum as a VPA monitor is given in [14].

A.2 HiStar as a VPA

The HiStar DIFC system [23] maps every process to a *label*, interposes on each time that one process attempts to send data to a receiving process, and allows the data to be sent if and only if the labels of the sending and receiving process satisfy a particular condition. A label is a finite set of named *categories*, which each may have a value in the range $0 - 3$, or the special value $\star$. Intuitively, each category corresponds to a source of information, and a process with a higher numeric value for a category has read more sensitive information from the source. If a process has the $\star$ value for a category c, then it has the ability to declassify information corresponding c. The names of the categories in the label of each process are the same, and HiStar allows a process p to send information to process q if and only if for each category c, the value of c in the label of p is

a numeric value less than or equal to the numeric value of c in p, or the value of p or q for c is $\star$.

HiStar provides system calls that each process may invoke to manipulate labels, subject to particular rules. A process may invoke one primitive to introduce a new category c, giving itself $\star$ for c and assigning a default value of c for every other process. A process may also invoke a primitive to raise the numeric value of its label for any category.

HiStar may be modeled as a privilege-aware system monitor as defined in §3. The privileges of HiStar are the flows that it allows, and the primitives are the system calls that allow each process to manipulate its label. Each state of the HiStar VPA is a map from each process to its label.

B Proofs of Theorems

In this section, we sketch informal proofs of the theorems stated in §3 and §4.

Proof. (Thm. 1) First, assume that the languages accepted by the program automaton P, policy Pol, and monitor M only contain matched words. Then from each automaton, we construct a (non-game) SEVPA that accepts the language of the automaton [4]. We then construct a pair of games whose Attacker actions are program instructions, and whose Defender actions are monitor primitives. Let $\mathsf{G_P}$ be a SEVPA game that accepts all instrumented executions of P: $\mathcal{L}(\mathsf{G_P}) = \{[p_0, i_0, \ldots, p_n, i_n] \mid [i_0, \ldots, i_n] \in \mathcal{L}(\mathsf{P}), p_j \in \mathsf{Prims}\}$. Let $\mathsf{G_{Pol,M}}$ be a SEVPA game that accepts all instrumented executions for M that violate Pol: $\mathcal{L}(\mathsf{G}_{pol,\mathsf{M}}) = \{[p_0, i_0, \ldots, p_n, i_n] \mid \exists r_j \in \mathsf{Privs} : [p_0, (i_0, r_0), \ldots, p_n, (i_n, r_n)] \in \mathcal{L}(\mathsf{M}) \wedge [(i_0, r_0), \ldots, (i_n, r_n)] \in \mathcal{L}(\mathsf{Pol})\}$. Let G be such that $\mathcal{L}(\mathsf{G}) = \mathcal{L}(\mathsf{G_P}) \cap \mathcal{L}(\mathsf{G_{Pol,M}})$, and let $\mathsf{PolWeaveGame}(\mathsf{P}, \mathsf{Pol}, \mathsf{M}) = \mathsf{G}$. It is straightforward to show that the instrumentation functions of POLWEAVE$(\mathsf{P}, \mathsf{Pol}, \mathsf{M})$ are exactly the modular winning Defender strategies of $\mathsf{PolWeaveGame}(\mathsf{P}, \mathsf{Pol}, \mathsf{M})$.

The direct construction of $\mathsf{G_{Pol,M}}$ may allow $\mathsf{G_{Pol,M}}$ to be non-deterministic. To avoid this in practice, we require that the M in fact be a *transducer function* from instrumented executions to privileges. Using this assumption, one can directly construct a deterministic $\mathsf{G_{Pol,M}}$ from deterministic Pol and M without requiring an explicit determinization.

To prove Thm. 2, we first recall a well-known construction of strategies, stated informally in §4. For a SEVPA safety game G with subgame G', define the strategy $\sigma_{\mathsf{G}'}$ as follows. For each Defender state $q \in Q_D^{\mathsf{G}'}$, fix a Defender action $\delta(q)$ such that $\tau_D(q, \delta(q)) \in Q^{\mathsf{G}'}$. Let $\gamma_{\mathsf{G}'} : (\Sigma_A^{\mathsf{G}})^* \to C^{\mathsf{G}'}$ map each trace of Attacker actions to the configuration of G' that the actions interleaved with fixed Defender actions drive G': $\gamma_{\mathsf{G}'}(\epsilon) = (\iota_0, \epsilon)$, and $\gamma_{\mathsf{G}'}(s \,.\, a) = \tau_A(\tau_D(\gamma_{\mathsf{G}'}(s), \delta(\mathsf{St}(\gamma_{\mathsf{G}'}(s)))), a)$, where $\mathsf{St}(c)$ is the state of configuration c. Then $\sigma_{\mathsf{G}'}(s) = \delta(\mathsf{St}(\gamma_{\mathsf{G}'}(s)))$.

If a SEVPA game has a modular winning Defender strategy, then it has a modular winning strategy $\sigma_{\mathsf{G}'}$ for some subgame G' of G. This follows immediately from results for finding modular strategies to recursive game graphs [6].

Proof. (Thm. 2) Let G have a modular winning Defender strategy σ_{G^*}, where G^* is a subgame of G. Let $R' = R \cap (Q^{\mathsf{S}} \times Q^{\mathsf{G}^*})$. For $\mathsf{G}' = \mathsf{G}|_{\mathsf{Img}(R')}$, $\sigma_{\mathsf{G}'}$ is a

modular winning Defender strategy, by a straightforward argument. Thus for $k = \max_{q \in Q_A^{\mathsf{S}}} |\{q' | R'(q, q')\}|$, G' is an (S, R, k) strategy, where R' satisfies the condition for R' given in Defn. 5.

Proof. (Thm. 3) *Soundness:* Let $\mathsf{G}' = \mathsf{G}|_{\mathsf{Img}(\mathcal{R}_A \cup \mathcal{R}_D)}$, for $\mathcal{R}_A$ and $\mathcal{R}_D$ as defined in Alg. 1, it is straightforward to show that $\sigma_{\mathsf{G}'}$ is a winning strategy using the defintion of StrWins. The modularity of $\sigma_{\mathsf{G}'}$ follows from the fact that it is a subgame of a SEVPA.

Completeness: If G has a winning strategy, then by Thm. 2, it has an $(\mathsf{S}, \mathcal{R}, k)$ winning modular strategy for some k. Let this $(\mathsf{S}, \mathcal{R}, k)$ be σ_{G^*} for a $\mathsf{G}^* = \mathsf{G}|_{\mathsf{Img}(\mathcal{R}')}$, where $\mathcal{R}' \subseteq \mathcal{R}$, and let $\mathcal{R}_A = \mathcal{R}' \cap (Q_A^{\mathsf{S}} \times Q_A^{\mathsf{G}})$. Let ScafAlgo non-deterministically guess $\mathcal{R}_A$ on Alg. 1 line [1], and then construct $\mathcal{R}_D$ on line [5]. It follows that $\mathsf{Img}(\mathcal{R}_D) \subseteq Q^{\mathsf{G}'}$, and thus that $\mathsf{Img}(\mathcal{R}_D) \cap Q_F^{\mathsf{G}} = \emptyset$ and each state in $\mathsf{Img}(\mathcal{R}_D)$ transitions on some Defender action to some state in $\mathsf{Img}(\mathcal{R}_A)$. It then follows that for $\mathsf{G}' = \mathsf{G}|_{\mathsf{Img}(\mathcal{R}_A) \cup \mathsf{Img}(\mathcal{R}_D)}$, $\sigma_{\mathsf{G}'}$ is a winning modular strategy of G.

C Known Game Algorithms as Scaffolds

Various algorithms have been presented for finding modular winning strategies to games. We now show that a naive version of a known algorithm for finding modular strategies for recursive game graphs [6] and a known symbolic algorithm for finding strategies of bounded size for symbolic games [18] are analogous to instances of ScafAlgo.

Symbolic Implementation of the Standard Algorithm Alur et. al. gave an explicit algorithm for finding modular Attacker strategies to recursive game graphs [6], and proved that the problem of finding modular strategies is NP-complete. Thus the algorithm may run in time exponential in the size of the game. The algorithm given in [6] can be adopted to find modular Defender strategies to SEVPA games, and we call the adopted algorithm Standard. Standard finds, for each state q, a set of states X_q such that some modular strategy can ensure that if the game reaches q, then it always stays within X_q, and X_q does not contain an accepting state. Standard then picks a set of Defender transitions that ensure that the game stays in X_q.

To compute X_q for each state s, Standard computes a set of sets of candidate X_q, and as a result, Standard may run in time exponential in the size of an input game G. ScafAlgo can analogously compute the sets X_q using the input game itself as a scaffold. Let $=_{Q^{\mathsf{G}}}$ be the equality relation over the states of G. If we apply $\mathsf{ScafAlgo}(\mathsf{G}, \mathsf{G}, =_{Q^{\mathsf{G}}}, 1)$, then ScafAlgo non-deterministically guesses whether each state in G is in a candidate winning strategy, and then checks if a winning strategy can be constructed from the guessed Attacker states. A symbolic implementation of Standard using ScafAlgo constructs formulas of size $O(|Q_A^{\mathsf{G}}|^2)$.

The algorithm presented in [6] is defined over recursive game graphs, in which the internal states of each game module are disjoint from entry and exit states. The algorithm executes in time exponential only in the number of *exit* states, not

Performance data of scaffolds

Name	LoC	Pol. States	Scaffolds							
			Triv.		Prog.		Mon.		Pol.	
			k	Time	k	Time	k	Time	k	Time
bzip2-1.0.6	8,399	12	12	-	4	1:26	15	-	7	17:22
fetchmail-6.3.19	49,370	12	7	-	1	0:16	2	-	2	1:54
gzip-1.2.4	9,076	9	12	-	1	-	1	-	8	-
tar-1.25	108,723	12	3	3:47	1	-	1	-	3	5:17
tcpdump-4.1.1	87,593	12	15	-	4	1:13	23	-	6	-
wget-1.12	64,443	21	7	0:43	1	0:15	1	-	2	0:42

Performance data (cont.)

Name	Scaffolds							
	Prog.-Mon.		Prog.-Pol.		Mon.-Pol.		Prog.-Mon.-Pol.	
	k	Time	k	Time	k	Time	k	Time
bzip2-1.0.6	15	-	1	0:04	8	-	1	0:09
fetchmail-6.3.19	1	1:13	1	1:13	1	-	1	1:39
gzip-1.2.4	1	-	1	1:47	1	-	1	-
tar-1.25	1	-	1	1:20	1	-	1	-
tcpdump-4.1.1	18	-	1	0:30	9	-	1	0:45
wget-1.12	1	-	1	0:25	1	-	1	18:59

Table 2. Performance of the Capsicum policy weaver. Column "LoC" contains lines of C source code (not including blank lines and comments), "Pol. States" contains the number of states in the policy, k contains the simulation bound, and "Times" contains the times used to find a strategy using the complete, trivial, and factor scaffolds. "-" denotes a time-out of 20 minutes.

the number of all states. In this work, we use SEVPA's to define games, which are analogous to recursive game graphs [4], but do not have disjoint sets of internal, entry, and exit states, and thus are not amenable to deriving such bounds.

Symbolic Search for Bounded Witness Sets The algorithm of [6] computes information for each state in a given game. However, many games have state spaces too large to enumerate explicitly. Madhusudan et. al. [18] describe how to construct, from a game G and bound k on the number of states that may be in a strategy, a constraint $\beta(\mathsf{G}, k)$ whose models correspond to winning strategies with at most k states. Each strategy is constructed from a bounded *witness set* of states of the game [18], and thus we call the algorithm BoundedWS. The states in a witness set are derived from a model of $\beta(\mathsf{G}, k)$.

We can construct $\beta(\mathsf{G}, k)$ by applying ScafAlgo. For a game G, define a "trivial" scaffold S_T with a module for each call in Σ_C^{G} that contains a single Defender and Attacker state. Let $\mathcal{R}$ relate the Defender state of each module Q_c^{S} for $c \in \Sigma_C^{\mathsf{G}}$ of S_T to each Defender state in module Q_c^{G}, and relate the Attacker state of module Q_c^{S} to each Attacker state in module Q_c^{G}. Then the StrWins formula constructed by a symbolic implementation of ScafAlgo for $\mathsf{ScafAlgo}(\mathsf{G}, \mathsf{S}, \mathcal{R}, k)$ has models that directly correspond to the models of $\beta(\mathsf{G}, k)$. The size of the formula constructed by $\mathsf{ScafAlgo}(\mathsf{G}, \mathsf{S}, \mathcal{R}, k)$ is $O(k^3)$.

D Complete Experimental Data

In §5, we described a set of experiments in which we implemented a policy weaver for the Capsicum operating system, and applied it to a set of UNIX utilities and the policies, using a variety of scaffolds to search for a winning strategy. In §5, we presented results for the two extreme scaffolds Triv. and Prog.-Pol.-Mon., and the intermediate scaffold Prog.-Pol. The tables in Table 2 present the results of applying all scaffolds constructed as subsets of products of the program, policy, and monitor automata. Each scaffold is labeled by the automata of which it is a product.

While many scaffolds provide performance comparable to the Prog.-Pol. scaffold on particular benchmarks, Prog.-Pol consistently finds strategies in minutes, whereas each other scaffold times out on at least two benchmarks. Prog.-Pol.'s consistency likely results from the fact that it always finds a winning strategy with a simulation bound of 1, whereas if ScafAlgo is applied to the other scaffolds, it often must use a higher simulation bound. Thus, apparently all of the practical policy-weaving problems that we found can be solved by assigning a single monitor state to each program-policy state pair, such that the strategy ensures that when an execution reaches the program and policy states, it will be in the assigned monitor state. In comparison, the scaffold Prog. sometimes requires a simulation bound of 4, indicating that some policy problems require finding up to 4 monitor-policy states for each program state.