

Secure Programming via Visibly Pushdown Safety Games

William R. Harris¹, Somesh Jha¹, 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 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 [24]. System utilities such as Norton Antivirus scanner [20], `tcpdump`, the DHCP client `dhclient` [23], and file utilities such as `bzip`, `gzip`, and `tar` [16, 21, 22] 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 [9, 18, 23, 24]. 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 [9, 18, 23, 24] 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` [23].

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 [23], the Decentralized Information Flow Control (DIFC) operating systems Asbestos, HiStar, and Flume [9, 18, 24] define privileges as the right to send information, and each provide different primitives for manipulating information-flow labels [7]. 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 known algorithms for finding modular strategies [4, 19] — 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 [23]. 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`),

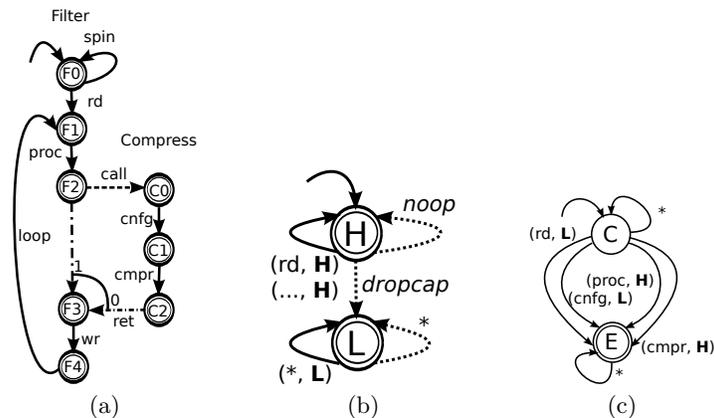


Fig. 1. Automata models of (a) the program **Filter** (F), (b) **Filter**'s MiniCap monitor (M), and (c) **Filter**'s policy for MiniCap (Pol). Executions of **Filter** are the runs of the automaton in (a). **Filter**'s MiniCap monitor allows sequences of privilege-instruction pairs and primitives accepted by the automaton in (b). **Filter**'s policy allows all sequences of privilege-instructions pairs accepted by the automaton in (c). Notation is explained in §2.1.

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** [23]) 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 [11, 23]. 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* [24].

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. In particular, `Filter` must (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

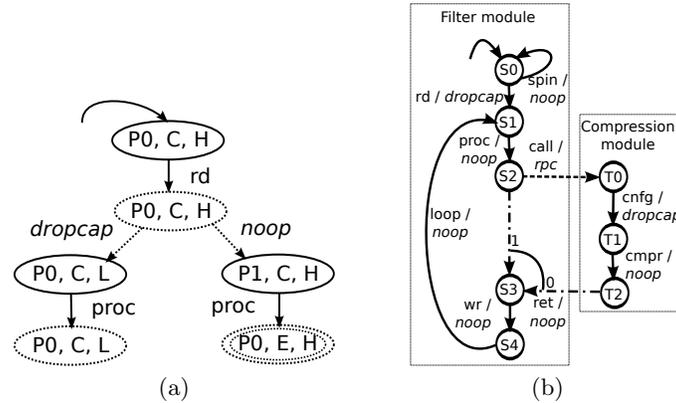


Fig. 2. (a) a selection of transitions of the game G_{ex} that is the product of F , Pol , and M from Fig. 1; (b) a strategy corresponding to a correct instrumentation of **Filter**.

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.

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*; $\tau_i : Q \times \Sigma_i \rightarrow Q$ is the *internal transition function*; $\tau_c : Q \times \Sigma_c \rightarrow Q$ is the *call transition function*; $\tau_r : Q \times \Sigma_r \times Q \rightarrow Q$ is the *return transition function*.

For $\hat{\Sigma} = \Sigma_I \cup \Sigma_C \cup \Sigma_R$, each VPA accepts a set of *traces* of (i.e., sequences of actions in) $\hat{\Sigma}$. Let ϵ denote the empty sequence. Let “.” denote the concatenation of two sequences; for set X , $x \in X$, and $s \in X^*$, $x \cdot s = [x] \cdot s$ and $s \cdot x = s \cdot [x]$, where $[x] \in X^*$ is the sequence containing only x . For sets X_0 and X_1 , let $X_0 \cdot X_1$ be the set of all sequences $x_0 \cdot x_1$ for $x_0 \in X_0$ and $x_1 \in X_1$. Let $\tau : Q^* \times \hat{\Sigma} \rightarrow Q^*$

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 :

$$\begin{aligned} \tau(q \cdot s, a) &= \tau_I(q, a) \cdot s && \text{for } a \in \Sigma_I \\ \tau(q \cdot s, a) &= (\tau_C(q, a) \cdot q \cdot s) && \text{for } a \in \Sigma_C \\ \tau((q_0 \cdot q_1 \cdot s'), a) &= \tau_R(q_0, a, q_1) \cdot s' && \text{for } a \in \Sigma_R \end{aligned}$$

Let $\rho : \widehat{\Sigma}^* \rightarrow Q^*$ 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 $s \in \widehat{\Sigma}^*$, $\rho(s \cdot a) = \tau(\rho(s), a)$. A trace $t \in \widehat{\Sigma}^*$ is accepted by V if $\rho(t) = q \cdot s$ 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 . \square

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 $\text{Instrs} = (\Sigma_I, \Sigma_C, \Sigma_R)$, let $\widehat{\text{Instrs}} = \Sigma_I \cup \Sigma_C \cup \Sigma_R$, 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 $(\widehat{\text{Instrs}} \times \text{Privs})^*$. Let an $(\text{Instrs}, \text{Privs})$ -policy for P be a $(\Sigma_I \times \text{Privs}, \Sigma_C \times \text{Privs}, \Sigma_R, \times \text{Privs})$ -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 $(\text{Prims} \cdot \widehat{\text{Instrs}})^*$. 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 $(\text{Instrs}, \text{Privs}, \text{Prims})$ -privilege-aware monitor M be a $((\Sigma_I \times \text{Privs}) \cup \text{Prims}, \Sigma_C \times \text{Privs}, \Sigma_R \times \text{Privs})$ -VPA.

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

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

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

1. I instruments P to never violate Pol : $\text{PrivEx}_{\mathsf{M}}(I_{\text{tr}}(\mathcal{L}(\mathsf{P}))) \cap \mathcal{L}(\mathsf{Pol}) = \emptyset$.
2. I chooses primitives independently of the execution before the most recent call; i.e., I is *modular*. Let $p^0, p^1 \in \text{Img}(I_{\text{tr}})$, (where for a relation R , $\text{Img}(R)$ is the image of R), and $p^0 = p_0^0 \cdot c \cdot p_1^0 \cdot r_0 \cdot p_2^0$, $p^1 = p_0^1 \cdot c \cdot p_1^1 \cdot r_1 \cdot p_2^1$, where call action c is matched by r_0 in p^0 , and is matched by r_1 in p^1 . Let $p_1^0 = a_0, b_0^0, a_1, b_1^0, \dots, a_n, b_n^0$, and let $p_1^1 = a_0, b_0^1, a_1, b_1^1, \dots, a_n, b_n^1$. Then $b_i^0 = b_i^1$ for each i in each such p^0 and p^1 . \square

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 allow an instrumentation function to invoke only 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 [14] App. A, we describe two different privilege-aware systems as VPA.

3.2 From Policy Weaving to Safety Games

Each policy-weaving problem $\text{POLWEAVE}(\mathsf{P}, \mathsf{Pol}, \mathsf{M})$ can be reduced to a *single-entry VPA (SEVPA [3]) 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*.
- $\tau_{I,A} : Q_A \times \Sigma_{I,A} \rightarrow Q_D$ is the *Attacker internal transition function*.
- $\tau_D : Q_D \times \Sigma_D \rightarrow Q_A$ is the *Defender internal transition function*.
- $\tau_R : Q_A \times \Sigma_R \times (Q_A \times \Sigma_C) \rightarrow Q_D$ 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 \mathbf{G}* be $C = Q \times \Gamma^*$, let the *attacker configurations* be $C_A = C \cap (Q_A \times \Gamma^*)$, and let the *defender configurations* be $C_D = C \cap (Q_D \times \Gamma^*)$. Let the *Attacker actions* be $\Sigma_A = \Sigma_{I,A} \cup \Sigma_C \cup \Sigma_R$. $\tau_A : C_A \times \Sigma_A \rightarrow C_D$ maps each Attacker configuration and Attacker action to a Defender configuration:

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

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

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., $Q^{\mathbf{G}}$ are the states of SEVPA game \mathbf{G} .

A *Defender strategy* of a two-player safety game \mathbf{G} is a function $\sigma : (\Sigma_A^{\mathbf{G}})^* \rightarrow \Sigma_D^{\mathbf{G}}$ 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 \mathbf{G} : formally, $\sigma_{\text{tr}}((\Sigma_A^{\mathbf{G}})^*) \cap \mathcal{L}(\mathbf{G}) = \emptyset$ (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 $\mathcal{P} = \text{POLWEAVE}(\mathbf{P}, \text{Pol}, \mathbf{M})$, there is a SEVPA safety game $\mathcal{G} = \text{PolWeaveGame}(\mathbf{P}, \text{Pol}, \mathbf{M})$ 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 $\mathcal{P} = \text{POLWEAVE}(\mathcal{P}, \text{Pol}, \mathcal{M})$ is given in §2.2. From \mathcal{P} , we construct a game $\mathbf{G}_{\mathcal{P}}$ that accepts all instrumented privileged executions of \mathcal{P} . From Pol , we constructed a game \mathbf{G}_{Pol} that accepts all instrumented privileged executions that violate Pol . We construct \mathcal{G} as the product of $\mathbf{G}_{\mathcal{P}}$, \mathbf{G}_{Pol} , and $\mathbf{G}_{\mathcal{M}}$. Proofs of all theorems stated in §3 and §4 are in [14] App. B.

4 Solving SEVPA Safety Games with Scaffolds

In this section, we present an algorithm `ScafAlgo` that finds a winning modular Defender strategy for a given SEVPA safety game. The algorithm uses an additional, potentially smaller game, which we call a *scaffold*. We present `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 [4] and a known symbolic algorithm for finding strategies of bounded size [19] as instances of `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 \mathbf{S} and \mathbf{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 . \mathbf{S} is a *scaffold* of \mathbf{G} under $\mathcal{R} \subseteq Q^{\mathbf{S}} \times Q^{\mathbf{G}}$ if and only if:

1. If $q_{\mathbf{S}} \in Q_F^{\mathbf{S}}$ and for $q_{\mathbf{G}} \in Q^{\mathbf{G}}$, $\mathcal{R}(q_{\mathbf{S}}, q_{\mathbf{G}})$, then $q_{\mathbf{G}} \in Q_F^{\mathbf{G}}$.
2. For $c \in \Sigma_C$, $\mathcal{R}(i_c^{\mathbf{S}}, i_c^{\mathbf{G}})$.
3. For $a \in \Sigma_{I,A}$, $q_{\mathbf{S}} \in Q_A^{\mathbf{S}}$, and $q_{\mathbf{G}} \in Q_A^{\mathbf{G}}$, if $\mathcal{R}(q_{\mathbf{S}}, q_{\mathbf{G}})$, then $\mathcal{R}(\tau_{I,A}(q_{\mathbf{S}}, a), \tau_{I,A}(q_{\mathbf{G}}, a))$.
4. For $a \in \Sigma_D$, $q_{\mathbf{S}} \in Q_D^{\mathbf{S}}$, and $q_{\mathbf{G}} \in Q_D^{\mathbf{G}}$, if $\mathcal{R}(q_{\mathbf{S}}, q_{\mathbf{G}})$, then $\mathcal{R}(\tau_D(q_{\mathbf{S}}, a), \tau_D(q_{\mathbf{G}}, a))$.
5. For $c \in \Sigma_C$, $q_c^{\mathbf{S}} \in Q_c^{\mathbf{S}}$, $q_c^{\mathbf{G}} \in Q_c^{\mathbf{G}}$, $q^{\mathbf{S}} \in Q^{\mathbf{S}}$, $q^{\mathbf{G}} \in Q^{\mathbf{G}}$, if $\mathcal{R}(q_c^{\mathbf{S}}, q_c^{\mathbf{G}})$ and $\mathcal{R}(q^{\mathbf{S}}, q^{\mathbf{G}})$, then $\mathcal{R}(\tau_R(q^{\mathbf{S}}, r, (q_c^{\mathbf{S}}, c)), \tau_R(q^{\mathbf{G}}, r, (q_c^{\mathbf{G}}, c)))$. \square

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

Each scaffold relation defines an Attacker bisimulation, with respect to actions, from configurations of the scaffold to configurations of the game. However, the bisimulation over configurations need not relate every accepting configuration of the game to an accepting configuration of the scaffold.

Scaffold relations and modular strategies are connected by the following key property, which provides the foundation for our algorithm. First, we define an $(\mathbf{S}, \mathcal{R}, k)$ -strategy of a game \mathbf{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 $Q' \subseteq Q^G$ such that $\{\iota_c\}_{c \in \Sigma_C^G} \subseteq Q' \subseteq Q^G$, let the *subgame* $G|_{Q'}$ be the game constructed by restricting the states and transition functions of G to the states in Q' . Each subgame G' of G defines a strategy $\sigma_{G'}$ as a VPA transducer. To compute $\sigma_{G'}(a_0, a_1, \dots, a_n)$, $\sigma_{G'}$ uses a_0, a_1, \dots, a_n as the Attacker actions for a play of G' . If G' is in an attacker state p , then σ_G transitions on the next unread a_i to $\tau_{I,A}(p, a_i)$. If G' is in a Defender state q , then $\sigma_{G'}$ picks the least Defender action d , under a fixed total ordering of Σ_D , such that $\tau_D(q, d) \in Q'$, outputs d , and transitions to $\tau_D(q, d)$. $\sigma_{G'}$ outputs the Defender action chosen by G' after it reads all of a_0, a_1, \dots, a_n .

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 $\mathcal{R} \subseteq Q^S \times Q^G$, and let $k \in \mathbb{N}$. An (S, \mathcal{R}, k) -*Defender strategy* σ' of G is a Defender strategy such that for some $\mathcal{R}' \subseteq \mathcal{R}$, $\mathcal{R}' \cap (Q_A^S \times Q_A^G)$ is *k-image-bounded*, $G' = G|_{\text{Img}(\mathcal{R}')}$, and $\sigma' = \sigma_{G'}$. \square

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 $\mathcal{R} \subseteq Q^S \times Q^G$. Then for some k , there is a winning modular (S, \mathcal{R}, k) -Defender 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 (S, \mathcal{R}, k) -strategies. The algorithm `ScafAlgo`, given in Alg. 1, takes a game G , scaffold S , relation $\mathcal{R} \subseteq Q^S \times Q^G$, and parameter k , and searches for an (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 $\mathcal{R}_A \subseteq Q_A^S \times Q_A^G$ by calling a function `nd-bounded-subrel`: $(Q_A^S \times Q_A^G) \times \mathbb{N} \rightarrow (Q_A^S \times Q_A^G)$, where `nd-bounded-subrel` $(\mathcal{R} \cap (Q_A^S \times Q_A^G), k)$ is a *k-image-bounded* subrelation of $Q_A^S \times Q_A^G$.

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

Input: G : a VPA safety game.
 S : a scaffold of G
 $\mathcal{R} \subseteq Q^S \times Q^G$: 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 \perp .

```

/* Choose  $\mathcal{R}_A$ : a  $k$ -image-bounded subrelation of  $\mathcal{R}$  that defines
Attacker states of a candidate strategy. */
1  $\mathcal{R}_A := \text{nd-bounded-subrel}(\mathcal{R} \cap (Q_A^S \times Q_A^G), k)$ ;
/* Construct  $\mathcal{R}_D$ : a relation to Defender states of the candidate
strategy defined by  $\mathcal{R}_A$ . */
2  $\mathcal{R}_{D,l} := \{(l_c^S, l_c^G) \mid c \in \Sigma_C^G\}$ ;
3  $\mathcal{R}_{D,i} := \{(\tau_{I,A}^S(p_A, a), \tau_{I,A}^G(q_A, a)) \mid (p_A, q_A) \in \mathcal{R}_A, a \in \Sigma_{I,A}^G\}$ ;
4  $\mathcal{R}_{D,r} := \{(\tau_R^S(p_A, a, s_A), \tau_R^G(q_A, a, t_A)) \mid (p_A, q_A), (s_A, t_A) \in \mathcal{R}_A, a \in \Sigma_R^G\}$ ;
5  $\mathcal{R}_D := \mathcal{R}_{D,l} \cup \mathcal{R}_{D,i} \cup \mathcal{R}_{D,r}$ ;
/* Check if the candidate strategy defined by  $\mathcal{R}_A$  and  $\mathcal{R}_D$  is a
winning strategy. */
6  $\text{StrWins} := \forall (p_D, q_D) \in \mathcal{R}_D : q_D \notin Q_F \wedge \exists a \in \Sigma_D : (\tau_D^S(p_D, a), \tau_D^G(q_D, a)) \in \mathcal{R}_A$ ;
7 if StrWins then return  $\sigma_{G|_{\text{Img}(\mathcal{R}_A \cup \mathcal{R}_D)}}$  else return  $\perp$ 

```

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

each $(p_A, q_A), (s_A, t_A) \in \mathcal{R}_A$ and return action $a \in \Sigma_R^G$, relates the r -successor of (p_A, s_A) to the r -successor of (q_A, t_A) . On line [5], ScafAlgo defines $\mathcal{R}_D \subseteq Q_D^S \times Q_D^G$ as the union of $\mathcal{R}_l, \mathcal{R}_{D,i}$, and $\mathcal{R}_{D,r}$.

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

Theorem 3. *Let G be a game, let S be a scaffold of G under $\mathcal{R} \subseteq Q^S \times Q^G$, and let k be a positive integer. If $\sigma = \text{ScafAlgo}(G, S, \mathcal{R}, k)$, then σ is a winning Defender strategy for G . If G has a winning Defender strategy, then for each scaffold S and scaffolding relation $\mathcal{R} \subseteq Q^S \times Q^G$, there is some k such that $\text{ScafAlgo}(G, S, \mathcal{R}, k)$ 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 $G_{\text{Img}(\mathcal{R}_A \cup \mathcal{R}_D)}$ 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 $(\tau_D^S(p_D, a), \tau_D^G(q_D, a)) \in \mathcal{R}_A$, 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 $O(|Q_A^S|^2 k^3)$.

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 [4] can be defined as `ScafAlgo` applied to the game itself as a scaffold. A symbolic algorithm for finding strategies of bounded size [19], 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 information and no information about their games, respectively. However, any game defined as a product of “factor” games has as a scaffold the product of any subset of its factors. In particular, for each policy-weaving game, we can automatically construct scaffolds from products of any subset of the program, monitor, and policy automata.

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 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 [23] capability operating system. We collected a set of six UNIX utilities, given in Tab. 1, that have exhibited critical security vulnerabilities [16, 21–23]. 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 [13].

We applied our Capsicum policy-weaver to each utility and its policy, with each scaffold defined as a product of some subset of the program, policy, and monitor. The data from all scaffolds is given in [14] App. D; Tab. 1 presents data for several illustrative scaffolds: the trivial scaffold “Triv.” defined in §4.2, the

Name	LoC	Pol.	Scaffolds								
			Triv.			Prog.-Pol.			Prog.-Pol.-Mon.		
			k	Time	Prims.	k	Time	Prims.	k	Time	Prims.
bzip2-1.0.6	8,399	12	12	-	-	1	0:04	6	1	0:09	6
fetchmail-6.3.19	49,370	12	7	-	-	1	1:13	5	1	1:39	5
gzip-1.2.4	9,076	9	12	-	-	1	1:47	15	1	-	-
tar-1.25	108,723	12	3	3:47	15	1	1:20	15	1	-	-
tcpdump-4.1.1	87,593	12	15	-	-	1	0:30	6	1	0:45	6
wget-1.12	64,443	21	7	0:43	11	1	0:25	11	1	18:59	11

Table 1. Performance of the Capsicum policy weaver. Column “LoC” contains lines of C source code (not including blank lines and comments) and “Pol. States” contains the number of states in the policy. For the trivial scaffold “Triv.,” intermediate scaffold “Prog.-Pol.,” and complete scaffold “Prog.-Pol.-Mon.,” k contains the simulation bound, “Times” contains the times used to find a strategy, and “Prims.” contains the number of callsites to primitives inserted. “-” denotes a time-out of 20 minutes.

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 the 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. The time taken to find a strategy does not directly depend on the size of the original program, because we apply several optimizations when constructing a policy-weaving game that cause the size of the constructed game to depend only on the size of program modules relevant to a given policy.

For each scaffold, the column “Prims.” contains the number of callsites to primitives dictated by the strategy. Our current game-solving algorithm does not minimize the number of such callsites, and as a result, the number of callsites may be larger than necessary. Moreover, in the current implementation, the number of callsites does not depend on the scaffold used to find a strategy for the game.

6 Related Work

Privilege-aware operating systems: Decentralized Information Flow Control (DIFC) operating systems such as Asbestos [9], HiStar [24], and Flume [18] 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 [6] enforce similar policies per byte of memory by providing primitives for managing memory tags. Capability operating systems such as Capsicum [23] 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 [8], and automatically instruments programs [12] to use the primitives of the Flume OS. However, the languages of policies used in the approaches presented in [8, 12] 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 [13] describes the approach in this paper instantiated to a policy weaver for Capsicum. This paper describes how the work in [13] may be generalized to arbitrary privilege aware systems, and describes the novel game-solving algorithm applied in [13].

Inlined Reference Monitors: An Inlined Reference Monitor (IRM) [1, 10] 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 [4]. Recursive state machines are directly analogous to SEVPA [3]. Madhusudan et. al. give a set of symbolic algorithms that find a winning strategy for a given game whose transition relation is represented symbolically [19]. The practical

contribution of our work is that we express the emerging 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 [4], to the “bounded-witness” algorithm of [19].

Jobstmann et al. [17] consider the problem of rewriting a program to satisfy a Linear Temporal Logic (LTL) specification, and reduce the problem to an LTL game. Their reduction constructs Defender actions (i.e., “system choices” [17]) from failure-inducing assignments of expressions in the program, whereas our work constructs Defender actions from a set of security system calls. Also, they reduce program repair to finite-state games, while our reduction relies crucially on modular strategies for SEVPA games. Thus, while the work of Jobstmann et al., like ours, is formalized in terms of automata games, the approaches differ in both the meaning of actions performed by players of the game, and the fact that we require a context-sensitive model of a target program.

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, V. Kumar, P. Madhusudan, and M. Viswanathan. Congruences for visibly pushdown languages. In *ICALP*, 2005.
4. R. Alur, S. La Torre, and P. Madhusudan. Modular strategies for recursive game graphs. In *TACAS*, 2003.
5. R. Alur and P. Madhusudan. Visibly pushdown languages. In *STOC*, 2004.
6. A. Bittau, P. Marchenko, M. Handley, and B. Karp. Wedge: Splitting applications into reduced-privilege compartments. In *NSDI*, 2008.
7. D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7):504–513, 1977.
8. P. Efstathopoulos and E. Kohler. Manageable fine-grained information flow. In *EuroSys*, 2008.
9. 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.
10. Ú. Erlingsson and F. B. Schneider. IRM enforcement of Java stack inspection. In *IEEE SP*, 2000.
11. FreeBSD 9.0-RELEASE announcement. <http://www.freebsd.org/releases/9.0R/announce.html>, Jan. 2012.
12. W. R. Harris, S. Jha, and T. Reps. DIFC programs by automatic instrumentation. In *CCS*, 2010.
13. W. R. Harris, S. Jha, and T. Reps. Programming for a capability system via safety games. Technical Report TR1705, University of Wisconsin-Madison, April 2012.
14. W. R. Harris, S. Jha, and T. W. Reps. Secure programming via visibly pushdown safety games. Technical Report TR1710, University of Wisconsin, Dept. of Computer Sciences, April 2012.

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. B. Jobstmann, A. Griesmayer, and R. Bloem. Program repair as a game. In *CAV*, 2005.
18. 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.
19. P. Madhusudan, W. Nam, and R. Alur. Symbolic computational techniques for solving games. *Electr. Notes Theor. Comput. Sci.*, 89(4), 2003.
20. R. Naraine. Symantec antivirus worm hole puts millions at risk. *eWeek.com*, 2006.
21. Ubuntu sec. notice USN-709-1. <http://www.ubuntu.com/usn/usn-709-1/>, 2009.
22. Vuln. note VU#381508. <http://www.kb.cert.org/vuls/id/381508>, July 2011.
23. R. N. M. Watson, J. Anderson, B. Laurie, and K. Kennaway. Capsicum: Practical capabilities for UNIX. In *USENIX Security*, 2010.
24. N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *OSDI*, 2006.