Programming for a Capability System via Safety Games

William R. Harris*, Benjamin Farley*, Somesh Jha*, Thomas Reps*†
*Computer Sciences Department; University of Wisconsin–Madison; Madison, WI, USA
{ wrharris, farleyb, jha, reps }@cs.wisc.edu
†GrammaTech, Inc.; Ithaca, NY, USA

Abstract—**New operating systems with security-specific system calls, such as the Capsicum capability system, allow programmers to write applications that satisfy strong security properties with significantly less effort than full verification. However, the amount of effort required is still high enough that even the Capsicum developers have reported difficulties in writing correct programs for their system.**

In this work, we present an algorithm that automatically rewrites a program for Capsicum so that it satisfies a given security policy by finding a winning strategy to an *automata-theoretic safety game*. We have implemented our algorithm as a tool, and we present experimental results that demonstrate that our algorithm can be applied to rewrite practical programs to satisfy practical security properties. Capsicum, combined with our algorithm, thus represents a sweet spot in the trade-off between the strength of policies that an operating system can enforce, and the ease of programming for such a system.

I. Introduction

Developing practical but secure programs remains a difficult, important, and open problem. Web servers and VPN clients execute untrusted code, and yet are directly exposed to potentially malicious inputs from a network connection [1]. System utilities such as Norton Antivirus scanner [2], `tcpdump` [3], the DHCP client `dhclient` [4], and file utilities such as `bzip2`, `gzip`, and `tar` [5]–[7] contain or have contained code with well-known vulnerabilities that allow the program to be compromised if they are exposed to an attack. Once an attacker compromises vulnerable code in 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 segments of its code execute.

Traditional operating systems provide to applications only weak primitives for managing their privileges. 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. Operating systems that support Mandatory Access Control (MAC) [8]–[10] allow a system administrator to specify a policy, and monitor the system calls of each program to ensure that the program does not violate the policy. Because MAC systems only monitor system calls, they cannot adjust the privileges with which a process executes based on events internal to the memory space of the process. However, many practical policies require the privileges of a process to change in this way [1], [4], [11], [12]. *Inline Reference Monitors* [13], [14] can enforce policies defined over internal events, but can only monitor managed code (i.e., code instrumented to be memory-safe).

However, recent work [1], [4], [11], [12] has produced new operating systems that allow programmers to develop programs that execute unmanaged code, but satisfy stronger properties than those that can be specified by a MAC system, and with significantly less effort than fully verifying the program. Such systems extend the set of system calls provided by a traditional operating system with security-specific calls (which henceforth we will call “security primitives”). Throughout a program’s execution, it interacts with the system by invoking security primitives to signal key events in its execution, which would not be observed by a MAC system. The developers of such systems have manually rewritten applications to invoke security primitives so that the application satisfies strong security policies, even when the application is composed of untrusted code. The application could not satisfy such policies if the operating system did not support such calls (see [1], [4], [11], [12] for detailed discussions of the advantages of such systems over traditional and MAC operating systems).

One example of an operating system with strong security primitives is the capability operating system Capsicum [4], now included in FreeBSD [15]. Capsicum tracks for each process (1) the set of *capabilities* available to the process, where a capability is a file descriptor and an access right for the descriptor, and (2) whether the process has the privilege to grant to itself more capabilities. Capsicum provides to each process a set of system calls that the process uses to limit its capabilities. Trusted code in a program can first communicate with its environment unconstrained, and then invoke primitives to limit itself to have only whatever capabilities that it needs for the rest of its execution. Untrusted code then executes with only the limited capabilities defined by the trusted code. Thus, even if the untrusted code is compromised, it will only be able to perform operations allowed by the limited capabilities.

The Capsicum primitives are sufficiently powerful that a programmer can rewrite a practical program to satisfy a strong security policy by inserting only a few calls to Capsicum primitives [4]. However, in practice it is difficult for programmers to reason about the subtle, temporal effects of the primitives. In fact, even Capsicum's own developers have rewritten programs, such as `tcpdump`, in a way that they tentatively thought was correct, only to discover later that the program was incorrect and required further rewriting [4]. Often, as in the case of `tcpdump`, the difficulty results from using the low-level primitives to ensure that the program simultaneously satisfies a strong, high-level security requirement, while preserving the core functionality of the original program.

This paper addresses the problem of writing programs for capability systems, like Capsicum, by presenting an algorithm that takes from a programmer (1) a program that does not invoke Capsicum primitives and (2) a declarative policy, stated in terms of the capabilities that the program should possess at various points. The algorithm automatically instruments the program to invoke Capsicum primitives, and partitions the program to execute in multiple processes if necessary, so that it satisfies the policy when executed on Capsicum. We call the problem of finding such an instrumentation and partitioning the *Capsicum policy-weaving problem.*

Our policy-weaving algorithm addresses three main challenges that a programmer faces when manually rewriting a program for Capsicum. The programmer's first challenge is to define what "secure behavior" means for his program. While Capsicum provides a powerful set of primitive operations, it does not provide an explicit language for describing policies per se. Because the Capsicum developers did not have such a language, it was impossible for them to define correctness for their rewritten programs.

The programmer's second challenge is to write his program to be both secure and functional. A programmer can typically secure a program on Capsicum by strongly limiting the capabilities of every process. However, the rewritten program may limit its capabilities too strongly at one point of execution, and as a result, may not have the capabilities required to carry out core program functionality later in the execution. The incorrect rewriting reported by the Capsicum developers [4] is an example of this issue.

The programmer's third challenge is to determine where he must partition the program, as well as instrument it to invoke Capsicum primitives. A programmer can potentially resolve the second challenge by partitioning a program into multiple processes, because Capsicum maintains different capabilities for each process. However, partitioning can itself lead to insecure behavior because each partitioned function that executes with many capabilities effectively serves as a library that malicious code can abuse.

We solve the above challenges by reducing the problem of rewriting a program to finding a winning strategy for an *automata-theoretic safety game* [16]. We represent a program as a language of traces of instructions, and we represent a policy as a language of traces of instructions paired with capabilities that the program must have when it executes the instructions. We model Capsicum as an automaton that relates executions of a program that has been instrumented with Capsicum primitives to the resulting instruction-capability trace allowed by Capsicum. From the program, policy, and Capsicum model, our algorithm constructs a game between an "Attacker," who "plays" program instructions, and a "Defender" who plays Capsicum primitives. The Attacker wins if the sequence of plays causes a policy violation, and the Defender wins otherwise. We show that the problem of simultaneously partitioning and instrumenting the program can be reduced to finding a *modular winning Defender strategy* to the game [16].

We find modular winning strategies for games by applying a symbolic algorithm that combines an algorithm for solving *reachability games* [17] with an algorithm for finding modular strategies [16]. Using the symbolic algorithm, we implemented a tool that automatically rewrites programs to correctly run on Capsicum. With our approach, when the policy weaver succeeds in finding a strategy, the rewritten program is correct by construction—modulo bugs in our implementation. We applied the tool to six system utilities that have demonstrated security vulnerabilities and to policies that counter the vulnerabilities. The utilities range in size from 8k to 108k lines of code. The tool was able to rewrite each of the utilities in minutes.

Organization: §II uses the `tcpdump` utility to illustrate the Capsicum policy-weaving problem and our reduction to safety games. §III formally defines the Capsicum policy-weaving problem and our reduction. §IV presents an experimental evaluation of our policy-weaving tool. §V discusses limitations of our approach. §VI discusses related work.

II. Overview

In this section, we discuss in more detail the problem of rewriting programs for Capsicum, and our algorithm for rewriting programs automatically. In particular, we illustrate the rewriting problem and our algorithm using the `tcpdump` system utility. We first describe the core functionality of `tcpdump`, give an informal but practical security policy for `tcpdump`, and summarize the Capsicum developers' experience rewriting `tcpdump` to satisfy the policy [4]. In §II-A, we then discuss how `tcpdump`, its policy, and Capsicum can be described formally as automata. In §II-B, we sketch how our algorithm uses the automata-based descriptions to determine how to rewrite `tcpdump` automatically to satisfy its policy when run on Capsicum.

`tcpdump` is a popular system utility that allows a user to print incoming network packets that match a particular pattern. Pseudocode of `tcpdump` is provided in Fig. 1; for now, ignore the underlined code. `tcpdump` is given two inputs: a pattern `pat` over packets, and a network input device `netd`. `tcpdump` first compiles `pat` into a Berkeley Packet Filter (BPF) (line 1) `bpf` [18], and configures the network device `netd`. `tcpdump` then enters a loop, in

```
tcpdump(pat, netd) {
 1: bpf = compile_bpf(pat);
 2: config_input(netd);
 3: limit_fd(netd, { RD });
 4: limit_fd(stdout, { WR });
 5: enter_cm();
 6: while(*) {
 7:   rpc_resolve_dns();
 8:   pak = read_packet(netd);
 9:   if (match_pattern(pak, bpf))
10:     write_packet(pak, stdout);
    }
}
```

```
resolve_dns() {
D1: ...
D2: return;
}
```

Figure 1. Pseudocode of `tcpdump` instrumented to enforce a policy when run on the Capsicum capability system. `resolve_dns` performs a DNS resolution by opening various DNS tables stored as files on the system. The lines of code from the original `tcpdump` program are the ones not underlined. The underlined statements are invocations of Capsicum primitives. In line 7, the call to `resolve_dns` is changed to `rpc_resolve_dns`.

which it performs DNS resolution (line 7), reads a packet `pak` from `netd` (line 8), checks `pak` against the filter `bpf`, and if `pak` matches `bpf`, writes `pak` to standard output.

Historically, `tcpdump` has served as a target for various security attacks, because its packet-matching code is complex and brittle, and thus prone to compromise due to an input crafted by an attacker [4]. Once `tcpdump` is compromised on a traditional operating system, it can read packets, and write their contents to arbitrary locations, even over the network.

Remark 1. *When `tcpdump` executes pattern-matching code (line 9), it should only be able to access its environment (i.e., the file system and network) by reading data from `netd`, or writing data to `stdout`.*

To verify that `tcpdump` satisfies the security policy of Remark 1 when it runs on a traditional operating system, a programmer would have to verify a strong property of `tcpdump`, such as the memory safety of its complex packet-matching code.

However, the Capsicum developers rewrote `tcpdump` in a comparatively simple way to satisfy this policy when executed on Capsicum. Capsicum is a UNIX-based operating system that defines an extended set of 63 access rights, which describe how a program may access each descriptor that it opens. Capsicum provides to a program a standard set of UNIX system calls, and a set of security-specific system calls, which we refer to as security primitives. In particular, it provides a primitive `limit_fd(d,R)` that takes two arguments: a file descriptor `d`, and a set of rights `R`. When a process p calls `limit_fd(d,R)`, Capsicum limits the rights of p for `d` to `R`. Capsicum also provides a primitive `enter_cm()`; when a process calls `enter_cm`, it enters *capability mode*, at which point it can no longer open any new file descriptors.

The underlined code in lines 3–5 of Fig. 1 depicts the Capsicum primitives that the instrumented program invokes to satisfy the security policy of Remark 1. After `tcpdump` configures `netd`, but before it matches packets, it limits itself to be able to read only from `netd` (line 3) and to write only to `stdout` (line 4), and then enters capability mode (line 5) to ensure that it cannot open file descriptors to any other resource in its environment. Even if the instrumented `tcpdump` is compromised as it matches packets, whatever code that is injected by an attacker will only be able to read from `netd` or write to `stdout`. The instrumented `tcpdump` thus satisfies the informal security policy of Remark 1.

The Capsicum developers originally instrumented `tcpdump` as in Fig. 1 and tentatively declared the instrumentation to be correct. However, the developers later found, through testing, that the instrumented `tcpdump` did not behave as expected. In particular, `tcpdump` invoked the `libc` DNS resolver (line 7), and the resolver must access the file system and network to function correctly. However, the instrumented `tcpdump` calls `enter_cm` (line 5) before calling the resolver. Thus, the resolver was not able to open file descriptors.

The resolver can be configured to only open files on its local host when resolving DNS queries, and is trusted to have no vulnerability that allows an attacker to execute injected code. Thus the resolver cannot be manipulated to leak packets by malicious code injected into the process space of `tcpdump`. Thus, the Capsicum developers determined that it was acceptable to strengthen the policy of Remark 1 with the functionality requirement:

Remark 2. *The DNS resolver must be able to open files.*

The policies of Remark 1 and Remark 2 are consistent, in the sense that they do not simultaneously require and disallow `tcpdump` to have a particular capability at some point in its execution. However, the Capsicum developers could not rewrite `tcpdump` to satisfy the policies of Remark 1 and Remark 2 solely by instrumenting `tcpdump` to call Capsicum primitives, due to the semantics of `enter_cm`. Instead, they leveraged the fact that Capsicum allows each process to hold a distinct set of capabilities, and rewrote `tcpdump` so that the DNS resolver executes in a separate process space with the capability to open files. When `tcpdump` calls the resolver, it does so through a Remote Procedure Call (RPC). The resolver then executes on behalf of `tcpdump` with the capability to open files, without `tcpdump` itself holding the capability.

The Capsicum developers' experience rewriting `tcpdump` for Capsicum illustrates the general challenges in rewriting programs for Capsicum outlined in §I. First, Capsicum only provides system calls for configuring enforcement mechanisms, i.e., `limit_fd` and `enter_cm`. "Policies" that directly state what capabilities the program must have when it executes particular instructions, such as the ones in Remark 1 and Remark 2, are only implicit.

Second, it is fairly simple to rewrite `tcpdump` so that it behaves securely (i.e., it does not leak packets that it reads) by inserting a small set of calls to primitives. However, it is non-trivial to instrument `tcpdump` so that it behaves securely and yet still carries out its core functionality, e.g., successfully performs DNS resolution. In general, to rewrite practical programs for Capsicum, a programmer often must partition his program to execute in multiple process spaces, along with instrumenting some processes to call Capsicum primitives. Even to instrument the compression utility `gzip`, the Capsicum developers had to partition `gzip` to untrusted compression and decompression functions in processes distinct from the process in which most of `gzip` executes [4].

A. *`tcpdump`, Policies, and Capsicum as Automata*

The main contribution of our work is an algorithm that solves the Capsicum policy-weaving problem, which is to take (1) an unpartitioned, uninstrumented program and (2) a high-level policy that describes what capabilities the program must have as it executes, and rewrite the program to satisfy the policy when executed on Capsicum. Our algorithm is automata-theoretic, and operates over a program, policies, and a Capsicum model, all represented as automata.

A program, such as `tcpdump`, may be viewed as an automaton whose actions are program instructions [19], the set of which we call Instrs. While we cannot in general reason precisely about the language of sequences of instructions executed by a program, we can over-approximate the language by abstracting the program automaton as a finite-state or *visibly-pushdown automaton* (VPA) [20].

A policy can be defined naturally as an automaton that accepts traces of program instructions from Instrs paired with sets of capabilities, drawn from a set of capabilities Caps, that the program must have as it executes the instructions. For `tcpdump`, Caps contains (i) $(\mathtt{netd}, \mathsf{rd})$ and $(\mathtt{stdout}, \mathsf{wr})$ where rd and wr are the Capsicum access rights to read and write to a file, respectively, and (ii) the meta-capability Env to open file descriptors to resources in the environment.

The informal security requirements of Remark 1 can be represented as a policy automaton. Suppose, for clarity, that we weaken the policy of Remark 1 to require only that when `tcpdump` executes `match_pattern`, it must not be able to open file descriptors. This requirement can be formalized with a security policy defined by an automaton over the actions $\mathsf{Instrs} \times \mathcal{P}(\mathsf{Caps})$, where $\mathcal{P}(\mathsf{Caps})$ is the powerset of Caps. For clarity, we describe the policy as a *past-time linear temporal logic (PLTL)* [21] formula. Each example formula is defined over the following atomic predicates over actions: for each program instruction ins the predicate $\mathsf{InstrIs}(\mathsf{ins})$ denotes that the latest instruction executed is ins (each instruction is denoted by its line number in Fig. 1); $\mathsf{MustHaveCaps}(C)$ denotes that the set of capabilities of an action must include the set of capabilities $C \subseteq \mathsf{Caps}$; and $\mathsf{OnlyHasCaps}(C)$ denotes that C upper-bounds the set of capabilities of an action. Remark 1 is represented by the following formula:

$$\mathsf{Always}(\mathsf{InstrIs}(\mathtt{9}) \implies \mathsf{OnlyHasCaps}(\{(\mathtt{stdin}, \mathsf{rd}), (\mathtt{stdout}, \mathsf{wr})\})) \tag{1}$$

The informal policy of Remark 2 can be represented as a policy automaton as well. Suppose, for clarity, that we weaken the policy of Remark 2 to require that after `tcpdump` calls `resolve_dns` (line 7) but before it returns from `resolve_dns` (line 8), it must be able to open file descriptors. This requirement can be formalized with the following policy automaton, represented as a PLTL formula, where a trace t satisfies the PLTL formula $\mathsf{Btwn}(\varphi_0, \varphi_1)$ if some prefix u of t satisfies φ_0, and no extension of u satisfies φ_1, and instructions `D1` and `D2` are the beginning and end of `resolve_dns`:

$$\mathsf{Btwn}(\mathsf{InstrIs}(\mathtt{D1}), \mathsf{InstrIs}(\mathtt{D2})) \implies \mathsf{MustHaveCaps}(\{\mathsf{Env}\}) \tag{2}$$

We give examples of other Capsicum policies as PLTL formulas in §IV-A.

We have described Capsicum as an operating system that monitors a sequence of instructions and Capsicum primitives executed by a program, and decides what capabilities the program has as it executes each instruction. We can model how Capsicum monitors a program as a formal language, accepted by an automaton. Capsicum defines a set of primitive operations Prims, that represent primitive operations that can be called by an instrumented program. For `tcpdump`, Prims contains a primitive `limit_fd(d,R)` for every descriptor `d` opened by `tcpdump` and every subset `R` of access rights that Capsicum defines, and a primitive `enter_cm`, both of which were informally described above. Prims also contains a primitive `rpc`, described below, that causes a call to be treated as an RPC, and a primitive `noop`, which does not affect the capabilities of the program. The Capsicum model defines a language over the actions $(\mathsf{Instrs} \times \mathcal{P}(\mathsf{Caps})) \cup \mathsf{Prims}$, and a trace is in the language if and only if when a program executes the sequence of instructions and primitives, the program has the capabilities paired with each instruction. For example, the language of the Capsicum monitor for `tcpdump` includes the trace $(1, \{\mathsf{Env}\}), (2, \{\mathsf{Env}\}), (6, \{\mathsf{Env}\})$ but does not include the trace $(1, \{\mathsf{Env}\}), (2, \{\mathsf{Env}\}), \mathtt{enter\_cm}, (6, \{\mathsf{Env}\})$ because after `tcpdump` invokes `enter_cm`, it cannot hold the capability Env.

The language of the Capsicum monitor also formalizes how Capsicum responds to RPCs. For example, recall that if `tcpdump` invokes `enter_cm`, and then calls `resolve_dns` through a normal function call, then `resolve_dns` is not able to open file descriptors (i.e., it does not hold the capability Env). But if `tcpdump` calls `resolve_dns` through an RPC, then `resolve_dns` is able to open file descriptors. The language of the Capsicum monitor encodes this behavior. For example, the Capsicum monitor does not accept the trace

$$(1, \{\mathsf{Env}\}), (2, \{\mathsf{Env}\}), \texttt{enter\_cm}, (6, \emptyset), (7, \emptyset), (\texttt{D1}, \{\mathsf{Env}\})$$

because Capsicum models the action $(7, \emptyset)$ as a normal function call to `resolve_dns` (line 7 in Fig. 1), and thus rejects the last action $(\texttt{D1}, \mathsf{Env})$. However, the Capsicum monitor accepts the trace

$$(1, \{\mathsf{Env}\}), (2, \{\mathsf{Env}\}), \texttt{enter\_cm}, (6, \emptyset), \\ (7, \emptyset), \texttt{rpc}, (\texttt{D1}, \{\mathsf{Env}\})$$

because Capsicum models the sequence $(7, \emptyset), \texttt{rpc}$ as an RPC to `resolve_dns`, and thus accepts the action $(\texttt{D1}, \mathsf{Env})$.

The Capsicum language is accepted by a VPA, which uses a stack to model Capsicum's responses to RPC calls and returns (see §III-A).

B. Instrumenting `tcpdump` *via a Safety Game*

We have now illustrated by example how all components of the Capsicum policy-weaving problem can be represented as automata that define languages over various sets of actions: a program defines a language over Instrs, policies define languages over $\mathsf{Instrs} \times \mathcal{P}(\mathsf{Caps})$, and Capsicum defines a language over $(\mathsf{Instrs} \times \mathcal{P}(\mathsf{Caps})) \cup \mathsf{Prims}$. Our weaving algorithm takes such automata as inputs and instruments the program to satisfy the policy.

The algorithm proceeds in two steps. First, from the input automata, it constructs a *safety game automaton* G that accepts all instrumented executions in which the program violates one of its policies. G defines a game between the program, represented by the attacking player, and its potential instrumentations, represented by the defending player. Each state of G is either an Attacker state or a Defender state. If the game is in an Attacker state, then the Attacker chooses a program instruction on which the game transitions, and if the game is in a Defender state, then the Defender chooses a Capsicum primitive on which the game transitions. The Attacker wins if the game enters an accepting state of G, and otherwise the Defender wins. A *strategy* for the Defender is a function that reads the actions chosen so far by the Attacker and chooses the next action for the Defender to play. A *winning Defender strategy* is a strategy that the Defender can always follow to win. By the definition of G, a winning Defender strategy thus directly corresponds to an instrumentation that ensures that the program never violates its policy.

Fig. 2 gives the game constructed from an automaton model of `tcpdump` based on its control-flow graph, policy (i.e., conjunction of policies (1) and (2)), and our automaton

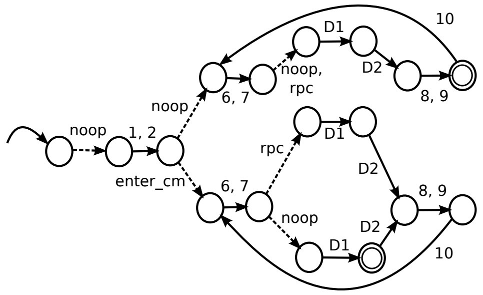

Figure 2. Safety game in which all winning plays for the Attacker are instrumentations of `tcpdump` that violate policy (1) or policy (2). Attacker (program) choices are represented as solid edges labeled with instruction line numbers from Fig. 1. Defender choices are represented as dashed edges labeled with Capsicum. Accepting states are represented as double circles.

model of Capsicum. To simplify the presentation, the game in Fig. 2 has been slightly simplified from the true game for `tcpdump` and policies (1) and (2). In particular, transitions for some instruction sequences have been collapsed into a single transition (e.g., the transition labeled with instructions "1, 2"), unnecessary Defender transitions have been removed (e.g., no Defender transition after "8, 9" is shown in Fig. 2), and only the primitives `enter_cm`, `rpc`, and `noop`are considered, because these are the only primitives relevant to policies (1) and (2).

In §III, we describe how our algorithm takes the program, policy, and Capsicum automata and constructs a game like the one shown in Fig. 2. For now, it is enough to observe that traces that cause the game to transition to an accepting state correspond to instrumented executions that cause the program to violate a policy. For example, the trace $[\texttt{noop}, 1, 2, \texttt{noop}, 6, 7, \texttt{noop}, D1, D2, 8, 9]$ corresponds to an execution in which `tcpdump` executes `9: match_pattern` without ever calling `enter_cm`, and thus violates security policy (1). As another example, the trace $[\texttt{noop}, 1, 2, \texttt{enter\_cm}, 6, 7, \texttt{noop}, D1]$ corresponds to an execution in which `tcpdump` calls `enter_cm` and then executes a non-RPC call to the DNS resolver, and thus executes the DNS resolver without the ability to open files, violating functionality policy (2).

In the algorithm's second step, it determines if there is a winning Defender strategy to the game. For the game in Fig. 2, one winning Defender strategy is one in which the Defender responds to instruction 1, 2 by calling `enter_cm`, and always responds to the call to `resolve_dns` by requiring the call to be an RPC (i.e., `rpc_resolve_dns`). Guided by this strategy, we can rewrite `tcpdump` to satisfy polices (1) and (2) by instrumenting `tcpdump` to call `enter_cm` after the program executes line 2, and rewriting `resolve_dns` to execute in a separate process space, invoked via RPC. This outcome corresponds to the instrumentation in lines 3 and 7 of Fig. 1. If we strengthen the security policy in (1) to exactly describe the informal security policy given in Remark 1, then our algorithm produces a game analogous to, but more complex than, the one shown in Fig. 1. One winning Defender strategy for such a game corresponds to the full instrumentation shown in Fig. 1.

III. Policy Weaving for Capsicum

In §II, we sketched how the problem of rewriting `tcpdump` to satisfy policies on Capsicum can be reduced to finding a winning strategy for a safety game. In this section, we describe our reduction in detail. In §III-A, we define the Capsicum policy-weaving problem in automata-theoretic terms. In §III-B, we discuss key properties of the problem formulation. In §III-C, we reduce the Capsicum policy-weaving problem to finding a strategy for a class of two-player safety games [16].

A. The Capsicum Policy-Weaving Problem

The Capsicum policy-weaving problem is defined by representing programs, policies, and Capsicum itself as visibly pushdown automata [20].

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 $V = (\Sigma_I, \Sigma_C, \Sigma_R, Q, \iota, Q_F, \tau_i, \tau_c, \tau_r)$, 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 $\widehat{\Sigma} = \Sigma_I \cup \Sigma_C \cup \Sigma_R$, a $(\Sigma_I, \Sigma_C, \Sigma_R)$-VPA accepts a set of *traces* of (i.e., sequences of actions in) $\widehat{\Sigma}$. Let ϵ denote the empty sequence, and for set X, let $[x] \in X^*$ denote the sequence containing only x. Let "." denote the concatenation of two sequences. For $x \in X$ and $s \in X^*$, let $x \ . \ s = [x] \ . \ s$ and $s \ . \ x = s \ . \ [x]$. For sets X_0 and X_1, let $X_0 \ . \ X_1$ be the set of all sequences $x_0 \ . \ x_1$ 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 action a:

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

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 $s \in \widehat{\Sigma}^*$, $\rho(s \ . \ a) = \tau(\rho(s), a)$. 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. □

We model a program using a language of executions and a set of file descriptors. Let Σ_I be a set of intraprocedural program instructions, let Σ_C be an alphabet of calls, let Σ_R be an alphabet of returns, and let Descs be a set of resource descriptors. Then a $(\Sigma_I, \Sigma_C, \Sigma_R, \mathsf{Descs})-$*program* is $P = (V, \mathsf{Descs})$, where V is a $(\Sigma_I, \Sigma_C, \Sigma_R)$-VPA.

A policy for a program is a language of allowed traces of program instructions paired with sets of capabilities. Let $P = (V, \mathsf{Descs})$ be an $(\Sigma_I, \Sigma_C, \Sigma_R, \mathsf{Descs})$-program, let Rights be a fixed set of access rights provided by Capsicum, and let Env be the privilege to open files. Let the *capabilities of* P be $\mathsf{Caps}_P = (\mathsf{Descs} \times \mathsf{Rights}) \cup \{\mathsf{Env}\}$, and for $\mathsf{Instrs} = \Sigma_I \cup \Sigma_C \cup \Sigma_R$, let $(\mathsf{Instrs} \times \mathcal{P}(\mathsf{Caps}))^*$ be the set of *capability-traces*. Let a P-policy be a language of capability-traces accepted by an $(\Sigma_I \times \mathcal{P}(\mathsf{Caps}_P), \Sigma_C^P \times \mathcal{P}(\mathsf{Caps}_P), \Sigma_R^P \times \mathcal{P}(\mathsf{Caps}_P))$-VPA.

The Capsicum monitor of P is a VPA C_P whose language describes what capabilities Capsicum allows P to have as it executes. Let the space of primitives that may be invoked by a $(\Sigma_I, \Sigma_C, \Sigma_R, \mathsf{Descs})$-program $P = (\text{VPA}, \mathsf{Descs})$ be $\mathsf{Prims}_P = \{\texttt{enter\_cm}, \texttt{noop}, \texttt{rpc}\} \cup \{\texttt{limit\_fd}(d, R) | \ \mathsf{desc} \in \mathsf{Descs}, R \subseteq \mathsf{Rights}\}$. See §II for illustrations of the primitives.

The Capsicum monitor of P is the VPA $\mathsf{C}_P = ((\Sigma_I \times \mathcal{P}(\mathsf{Caps}_P)) \cup \mathsf{Prims}_P, \Sigma_C \times \mathcal{P}(\mathsf{Caps}_P), \Sigma_R \times \mathcal{P}(\mathsf{Caps}_P), Q, \iota, Q_F, \tau_I, \tau_C, \tau_R)$ with

- Q: each state is a map from each descriptor in Descs to the set of rights held by the program, paired with a Boolean flag that denotes whether the program is in capability mode (see §II) and a Boolean flag that denotes if the next call should be treated as an RPC, or the stuck state: $Q = ((\mathsf{Descs} \to \mathsf{Rights}) \times \mathbb{B} \times \mathbb{B}) \cup \{\mathsf{CapStuck}\}$.
- ι: in the initial state, the program has all rights for every descriptor, is not in capability mode, and the next call is not treated as an RPC.
- Q_F: all states but the stuck state are accepting states: $Q_F = Q \backslash \{\mathsf{CapStuck}\}$.
- $\tau_I(q, a) = q'$: suppose that the internal action a is a program instruction paired with a set of capabilities: $a = (\mathsf{ins}, C)$. Let $\mathsf{ExactCaps}(q, C)$ denote the condition (1) if $\mathsf{Env} \in C$, then the capability-mode flag of q is not set, and (2) for each capability $(\mathsf{desc}, r) \in C$, r is in the set of rights associated with desc at q. If $\mathsf{ExactCaps}(q, C)$, then $q' = q$; otherwise, $q' = \mathsf{CapStuck}$.

  Suppose that the internal action a is a Capsicum primitive p. If $p = \texttt{enter\_cm}$, then q' has the same rights and RPC flag as q, with the capability-mode flag set. If $p = \texttt{limit\_fd}(\mathsf{desc}, R)$, q' has the same capability-mode and RPC flag, but the rights of desc in q' are desc's rights in q restricted to R. If $p = \texttt{rpc}$, then q' has the rights and capability-mode flag of q, but its RPC-flag is set. Otherwise, $p = \texttt{noop}$, and $q' = \mathsf{CapStuck}$.

- $\tau_C(q, (c, C)) = q'$: if $\mathsf{ExactCaps}(q, C)$ (see τ_I) does not hold, then $q' = \mathsf{CapStuck}$. Otherwise, if the RPC flag is set in q, then $q' = \iota_c$, and if the RPC flag is not set, then $q' = q$. In all cases, q is pushed on the stack.
- $\tau_R(q_0, a, q_1) = q'$: if the RPC flag in the stack state q_1 is not set, then the post-state is the pre-state q_0: $q' = q_0$. Otherwise, the post-state is the stack state $q' = q_1$ with the RPC flag unset.

Let $(\mathsf{Prims}_P \ . \ \mathsf{Instrs})^*$, the set of sequences of alternating primitives and program instructions, be the set of *instrumented executions*; C_P defines a function $\mathsf{CapTraces}_P$ from each instrumented execution to the capability trace that it induces. Let this function be $\mathsf{CapTraces}_P : (\mathsf{Prims}_P \ . \ \mathsf{Instrs})^* \to (\mathsf{Instrs} \times \mathcal{P}(\mathsf{Caps}))^*$, where $\mathsf{CapTraces}_P([p_0, \mathsf{ins}_0, \ldots, p_n, \mathsf{ins}_n]) = ((\mathsf{ins}_0, C_0), \ldots, (\mathsf{ins}_n, C_n))$ if and only if $[p_0, (\mathsf{ins}_0, C_0), \ldots, p_n, (\mathsf{ins}_n, C_n)] \in \mathcal{L}(\mathsf{C}_P)$. By the definition of C_P, $\mathsf{CapTraces}_P$ is well-defined.

A program, its policy, and its Capsicum monitor define a *Capsicum policy-weaving problem.*

Definition 2. Let P be a $(\Sigma_I, \Sigma_C, \Sigma_R, \mathsf{Descs})$-program, and let Pol be a P-policy. Let an *instrumentation function* be a function $I : \mathsf{Instrs}^* \to \mathsf{Prims}_P^*$, and the *trace extension* of I, $I_{tr} : \mathsf{Instrs}^* \to (\mathsf{Prims}_P \ . \ \mathsf{Instrs})^*$ be defined as $I_{tr}(\epsilon) = \epsilon$, and $I_{tr}(s \ . \ a) = I_{tr}(s) \ . \ I(s \ . \ a) \ . \ a$.

The Capsicum policy-weaving problem $\mathsf{Weave}(P, \mathsf{Pol})$ is to find an instrumentation function I that is *safe* and *modular*. I is *safe* if P instrumented by I has the capabilities allowed by Pol in each execution. In other words, $\mathsf{CapTraces}_P(I_{tr}(\mathcal{L}(P))) \subseteq \mathcal{L}(\mathsf{Pol})$, where in a minor abuse of notation, we lift I_{tr} and $\mathsf{CapTraces}_P$ to sets of traces.

A function partitioned to be invoked via RPC can be invoked by arbitrary, injected code. Such a function cannot trust information passed by its caller about the execution of the program. Thus, a correct instrumentation function must be *modular*: it must choose primitives independently of the instructions and primitives executed before the most recent function call [16]. An instrumentation function I is modular if the following holds. Let $s^0, s^1 \in \mathrm{Img}(I_{tr})$, (where for a relation R, $\mathrm{Img}(R)$ is the image of R), with $s^0 = s^0_0 \ . \ c \ . \ s^0_1 \ . \ r_0 \ . \ s^0_2$, $s^1 = s^1_0 \ . \ c \ . \ s^1_1 \ . \ r_1 \ . \ s^1_2$, call action c is matched by r_0 in s^0, and is matched by r_1 in p^1. Let $s^0_1 = p^0_0, i_0, p^0_1, i_1, \ldots, i_n, p^0_{n+1}$, and let $s^1_1 = p^1_0, i_0, p^1_1, i_1, \ldots, i_n, p^1_{n+1}$. Then $p^0_i = p^1_i$ for each i in each such s^0 and s^1. □

B. Properties of the Weaving Problem

From an Instrumentation Functions to an Instrumented Program: The policy-weaving problem is defined as finding an instrumentation function (Defn. 2) that reads the history of an execution and outputs a Capsicum primitive. It remains to show that we can always use such a function to instrument a program for Capsicum. As we discuss in §III-C, every instrumentation function can be represented as a winning strategy for a *safety game*, which can be represented as a VPA transducer that reads program instructions as input, and outputs Capsicum primitives. The product of the program VPA and the strategy VPA transducer models an instrumented program that executes a Capsicum primitive chosen by the strategy, followed by the next program instruction executed by the program. We can thus use such a product to instrument the original program to invoke Capsicum primitives.

Soundness of Program Abstractions: The weaving problem is defined for a program whose executions can be represented as a VPA, but the executions of most practical programs cannot be represented exactly by a VPA. However, the executions of a program can be over-approximated by a VPA. For a given program, several natural such VPA's can be constructed automatically, which model the control flow of the program, along with a finite set of facts about its data.

To verify that a program satisfies a given safety property, it suffices to verify that an over-approximation of the program satisfies the property. A similar result holds in policy weaving. In particular, a correct instrumentation for a policy-weaving problem defined by an over-approximation of a program P is a correct instrumentation for the policy-weaving problem defined over P and the same policies. Intuitively, a satisfying instrumentation I for an over-approximation of a program is correct for the program because if I instruments each execution of the over-approximation to satisfy a policy, then it trivially instruments each execution of the approximated program to satisfy the policy. In §V, we consider automatically refining over-approximations of a program.

Primitives per instruction: The reduction of §III-C only searches for an instrumentation that invokes exactly one primitive after each instruction. This restriction does not fundamentally limit the number of primitives that the instrumentation can invoke, because we can either (1) redefine the space of primitives Prims to be all sequences of k Capsicum primitives for some sufficiently large fixed integer k, or (2) inject a block of "noop" program instructions that have no effect on the state of the policy, and allow a primitive to be invoked after each such instruction.

C. From a Capsicum Weaving Problem to a Safety Game

Each Capsicum policy-weaving problem can be reduced to finding a modular strategy for a two-player safety game played between an Attacker and a Defender. Such games are defined as follows.

Definition 3. A *single-entry* VPA *(*SEVPA*) [22] 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 $\mathcal{G} = (Q_A, Q_D, Q_0, \iota_0, \{(Q_c, \iota_c)\}_{c \in \Sigma_C}, Q_F, \tau_{I,A}, \tau_D, \tau_R)$, 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.
- $Q_0 \subseteq Q$ 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.
- $Q_F \subseteq Q_D$ 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 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 safety game are defined analogously to the traces of a VPA. Let the *configurations of* $\mathcal{G}$ 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 $\Sigma_A = \Sigma_{I,A} \cup \Sigma_C \cup \Sigma_R$. $\tau_A : C_A \times \Sigma_A \to C_D$ maps each Attacker configuration and Attacker action to a successor Defender configuration:

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

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 Defender 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 $\mathcal{G}$ if $\rho(p) = q \ . \ s$ with $q \in Q_F$ and $s \in Q^*$. Let $\mathcal{L}(\mathcal{G})$ be the set of all plays accepted by $\mathcal{G}$.

A *Defender strategy* of a two-player safety game $\mathcal{G}$ is a function $\sigma : (\Sigma_A^{\mathcal{G}})^* \to \Sigma_D^{\mathcal{G}}$ 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 $\mathcal{G}$: formally, $\sigma_{tr}(\Sigma_A^{\mathcal{G}}) \cap \mathcal{L}(\mathcal{G}) = \emptyset$ (for σ_{tr} defined analogously to I_{tr} in Defn. 2). Let σ be modular if it satisfies the condition analogous to a modular instrumentation function (Defn. 2). □

If a SEVPA safety game has a strategy σ, then it has a solution σ^* that may be represented as a VPA transducer (i.e., a VPA where each action is labeled with input and output symbols) [16].

Capsicum policy-weaving problems are strongly connected to SEVPA safety games by the following theorem.

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

Thm. 1 is a consequence of previous work in rewriting programs for general systems with primitives [23]. From a policy-weaving problem $\mathcal{P} = \mathsf{Weave}(P, \mathsf{Pol})$, we construct $\mathcal{G}_{\mathcal{P}} = \mathsf{CapGame}(P, \mathsf{Pol})$ as the product of two SEVPA safety games. The first game, $\mathcal{G}_P$, accepts all plays that correspond to instrumented executions of P. The second game, $\mathcal{G}_{\mathsf{Pol}}$ accepts all plays that correspond to instrumented executions that violate Pol. $\mathcal{G}_{\mathcal{P}}$ accepts all plays accepted by both $\mathcal{G}_P$ and $\mathcal{G}_{\mathsf{Pol}}$: $\mathcal{L}(\mathcal{G}_{\mathcal{P}}) = \mathcal{L}(\mathcal{G}_P) \cap \mathcal{L}(\mathcal{G}_{\mathsf{Pol}})$.

1) Finding a Winning Defender Strategy Efficiently: By Thm. 1, if $\sigma : \mathsf{Instrs}^* \to \mathsf{Prims}$ is a winning modular Defender strategy for G, then from σ we can construct a correct instrumentation function. However, finding a winning modular Defender strategy σ is NP-complete [16]. Algorithms for finding such strategies non-deterministically guess a set of call and return transitions in the game that a strategy can allow, and then apply efficient algorithms for finding winning strategies over finite automata to check if there is a set of strategies for each module that allow only the guessed call and return transitions. In previous work [23], we presented a symbolic algorithm that applies a Satisfiability Modulo Theories (SMT) [24], [25] solver to efficiently guess and check sets of call and return transitions. Our algorithm finds winning modular Defender strategies for games constructed from practical policy-weaving problems in minutes (see §IV).

Capsicum weaving problems exactly match finding modular winning Defender strategies safety games, as both problems are NP-complete. SEVPA safety games are NP-complete as an immediate consequence of the fact that finding modular winning strategies to games defined over recursive graphs is NP-complete [16]. Capsisum weaving problems are NP-complete by a proof sketched in App. A.

IV. EXPERIMENTS

We evaluated the policy-weaving algorithm presented in §III, using a set of experiments that were designed to answer the following questions: (1) Can policies capture security requirements that can only be enforced by comparatively complex use of the Capsicum primitives? and (2) Can the weaving algorithm efficiently instrument programs to satisfy their policies?

To answer these questions, we collected a set of real-world system utilities with known past security vulnerabilities, and informal policies for each program. Some of the programs were found through interaction with the Capsicum developers [26], while others were chosen as popular system utilities with well-known vulnerabilities [5], [6]. We specified a policy for each program as a policy automaton. We implemented the algorithm described in §III as a tool, and applied the tool to each program and its policies to rewrite the program.

The experiments indicate that our policy-weaving algorithm is practical. We were able to express desired policies for each of the programs as automata, and the tool found an instrumentation for each program and policies in minutes. In most cases, the instrumentations were subtle compared to the original policies, and on manual inspection, do not appear to be needlessly complex.

We now discuss the programs and policies used, and the instrumentations found by our tool. In some cases, the instrumentations are arguably "as small as" the policies, in that each Capsicum primitive inserted in an instrumented program corresponds directly to a subformula in the policy. However, it is typically only clear through subtle reasoning that the instrumented program satisfies the policy; in particular, five of the six programs are instrumented to use RPC's as well as Capsicum primitives. We inspected each of the these five programs by hand, and determined that in each case, RPC's were necessary (although in some cases, the tool inserted superfluous RPC's, as we discuss in §IV-B).

A. Programs and Polices

tcpdump: We described the structure of `tcpdump` and its policies by example in §II.

gzip, bzip2, tar: The `gzip` compression tool has exhibited vulnerabilities in the past, due to its complicated compression and decompression code [7]. `gzip` was previously rewritten manually by the Capsicum developers to execute securely on Capsicum [4].

`gzip` executes in a loop. In each iteration of the loop, `gzip` processes command-line arguments, opens files, and invokes compression and decompression routines `compress` and `decompress` to read input from and write output to the opened files [4]. While the code that processes arguments and configures files is simple and trusted, the compression and decompression routines are complex, and have exhibited vulnerabilties.

Guided by the policy given informally by the Capsicum developers [4], we constructed a policy automaton for `gzip`. The policy allows `gzip` to access its environment when opening configuration files (instruction `setup`) and when opening files for input (instruction call `openin` in function `treat_file`) and output (`treat_file`: call `openout`) when iterating in the main loop. However, the policy allows `gzip` only to read from an input file `in` and write to an output file `out` when executing the compression and decompression functions (`compress` and `decompress`), which we assumed could inject arbitrary code. The complete policy automaton is represented by the PLTL formula:

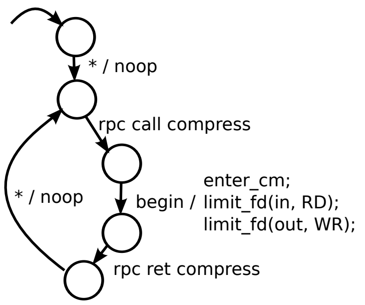

Figure 3. A fragment of the winning strategy for instrumenting the `compress` function, found by our tool when applied to `gzip` and its policy given in §IV-A.

$$\begin{aligned}
&\mathsf{Until}(\ \mathsf{MustHaveCaps}(\{\mathsf{Env}\}), \mathsf{InstrIs}(\texttt{end\_setup}))\\
\land\ &\mathsf{Btwn}(\mathsf{InstrIs}(\texttt{treat\_file:call openin}),\\
&\quad \mathsf{InstrIs}(\texttt{treat\_file:ret openin})) \implies\\
&\quad \mathsf{MustHaveCaps}(\{\mathsf{Env}\})\\
\land\ &\mathsf{Btwn}(\mathsf{InstrIs}(\texttt{treat\_file:call openout}),\\
&\quad \mathsf{InstrIs}(\texttt{treat\_file:ret openout})) \implies\\
&\quad \mathsf{MustHaveCaps}(\{\mathsf{Env}\})\\
\land\ &\mathsf{Btwn}(\texttt{call compress}, \texttt{ret compress}) \implies\\
&\quad \mathsf{OnlyHasCaps}(\{(\texttt{in}, \mathsf{rd}), (\texttt{stdout}, \mathsf{wr})\})\\
\land\ &\mathsf{Btwn}(\texttt{call decompress}, \texttt{ret decompress}) \implies\\
&\quad \mathsf{OnlyHasCaps}(\{(\texttt{in}, \mathsf{rd}), (\texttt{stdout}, \mathsf{wr})\})
\end{aligned}$$

The key challenge in instrumenting `gzip` to satisfy the above policy is that in some executions, `gzip` must open new files (i.e., have the Env capability) after executing the unsafe `compress` and `decompress` functions. Thus to rewrite `tcpdump` for Capsicum, a programmer must use both Capsicum primitives and RPC's.

We applied our tool to `gzip` and the above policy, and it found a correct instrumentation as a winning Defender strategy for the corresponding safety game. Under the strategy, the instrumented `gzip` iterates in its main loop with the Env capability and calls `compress` and `decompress` via RPC. `compress` and `decompress` invoke `limit_fd` to lower their capabilities to $\{(\texttt{in}, \mathsf{rd}), (\texttt{stdout}, \mathsf{wr})\}$ before each function executes its vulnerable code. Because both `compress` and `decompress` return from an RPC into iteration code that calls `openin` and `openout`, the iteration code has the Env capability when it executes both `openin`

	Program Stats			Policy Stats	Policy-Weaver Stats			
Name	LoC	*Procs*	*Call-sites*	*States*	*Non-RPC*	RPC		*Time*
bzip2-1.0.6	8,399	110	676	12	6	2	2	0m 03s
fetchmail-6.3.19	49,370	299	3,468	12	20	1	1	1m 05s
gzip-1.2.4	9,076	93	543	9	28	2	1	0m 23s
tar-1.25	108,723	940	5,135	12	18	0	0	2m 51s
tcpdump-4.1.1	87,593	723	10,075	12	11	2	1	0m 28s
wget-1.12	64,443	437	4,329	21	13	3	0	1m 00s

Table I
PROGRAM STATISTICS, POLICY STATISTICS, AND PERFORMANCE DATA FOR THE POLICY-WEAVING TOOL. *LoC* SHOWS THE NUMBER OF LINES OF C SOURCE CODE (INCLUDING BLANK LINES AND COMMENTS); *Procs* AND *Call-sites* SHOW THE NUMBER OF PROCEDURES AND CALL-SITES, RESPECTIVELY. *States* SHOWS THE NUMBER OF STATES IN THE POLICY. *Non-RPC* SHOWS THE NUMBER OF NON-RPC CAPSICUM PRIMITIVES THAT WERE INSERTED. THE TWO COLUMNS UNDER *RPC* SHOW THE NUMBER OF CALLS MARKED FOR TRANSFORMATION TO RPCS VS. THE NUMBER OF RPC'S THAT APPEAR TO BE NEEDED (OBTAINED VIA MANUAL INSPECTION). *Time* LISTS THE TIME TAKEN TO FIND AN INSTRUMENTATION, IN MINUTES AND SECONDS.

and openout. A fragment of the strategy dealing with the compress function is depicted as a transducer in Fig. 3.

Like gzip, the bzip2 compression utility and tar archiving utility have demonstrated security vulnerabilties [5], [6]. We defined policies for bzip2 and tar analogous to the policy described above for gzip, and our tool found suitable strategies.

fetchmail: fetchmail downloads mail from a list of servers. In a typical execution, fetchmail reads a configuration file (setup), reads a list of mail servers, and then iteratively opens a connection (connect) and downloads mail from each one (fetch_messages). fetchmail must be able to open its configuration file, and open a network connection when it executes connect. However, when fetchmail executes fetch_messages, it reads data over an untrusted network connection. Thus, as fetchmail executes fetch_messages it should only be able to read from a network socket sock. fetchmail also should always be able to write to an output file out and a log file log. fetchmail's policy is represented by the following PLTL formula:

$$\begin{aligned}
&\mathsf{Until}(\mathsf{MustHaveCaps}(\{\mathsf{Env}\}), \mathsf{InstrIs}(\texttt{setup}))\\
&\quad\wedge \mathsf{Btwn}(\texttt{fetch\_messages:begin},\\
&\qquad\quad \texttt{fetch\_messages:end}) \implies\\
&\quad\ \mathsf{OnlyHasCaps}(\{(\texttt{log}, \mathsf{wr}), (\texttt{out}, \mathsf{wr}), (\texttt{sock}, \mathsf{rd})\})\\
&\quad\wedge \mathsf{Btwn}(\texttt{fetch\_messages:begin},\\
&\qquad\quad \texttt{fetch\_messages:end}) \implies\\
&\quad\ \mathsf{MustHaveCaps}(\{(\texttt{sock}, \mathsf{rd})\})\\
&\quad\wedge \mathsf{Always}(\mathsf{MustHaveCaps}(\{(\texttt{log}, \mathsf{wr}), (\texttt{out}, \mathsf{wr})\}))
\end{aligned}$$

The key challenge in instrumenting fetchmail to satisfy the above policy is that in some executions, fetchmail must open new connections (i.e., have the Env capability) after executing the unsafe fetch_messages function. Thus, to rewrite fetchmail for Capsicum, a programmer must use both Capsicum primitives and RPC's. We applied our tool to fetchmail and the above policy, and the tool found a correct instrumentation as a winning Defender strategy to a corresponding safety game. Under the strategy, the instrumented fetchmail iterates over and opens connections to URL's while holding the Env capability. fetchmail then calls fetch_messages via RPC. fetch_messages then invokes the enter_cm primitive, and invokes limit_fd so that fetch_messages executes only with the capability to read and write to a fixed network socket, or write to the log file. Because fetch_messages returns from an RPC into iteration code that opens a connection to each URL, fetch_messages executes with a small set of capabilities, while the iterating code still executes with the Env capability.

wget: In a typical execution, wget first opens and configures output and logging files. It then iterates through a given list of URL's. For each URL, it opens a network connection to the URL, and downloads data from the URL. As wget executes, it thus handles data read from a network connection that may be untrusted. Consequently, we defined a policy for wget analogous to that of fetchmail's.

bzip2, fetchmail, gzip, tcpdump, and wget each must be instrumented to use RPC's and Capsicum primitives so that the capabilities of the program are raised or lowered only temporarily. However, the instrumented programs combine RPCs with Capsicum primitives in significantly different ways. In the instrumented tcpdump, untrusted code (i.e., code that may run injected code) calls the trusted DNS resolver via RPC, because the resolver must execute with the capability to open files. However, the resolver uses the capability in only a limited way, so that the resolver cannot be abused to leak information. Instrumented versions of other programs, such as gzip, call untrusted code via RPC so that it executes with a lower set of capabilities. If the instrumented version of such a program allowed, e.g., a function that opens files to execute with high capability and be called via RPC, then injected code could abuse the RPC function.

When manually rewriting a program for Capsicum, a programmer must carefully choose a combination of RPC and Capsicum primitives, because code injected by an attacker can feasibly call and abuse any function that executes in a separate process with high capabilities. In contrast, a programmer can simply provide to our policy-weaving tool a description of the capabilities that functions must have, and our tool automatically finds a safe combination of RPC's and Capsicum primitives by searching for a modular winning Defender strategy, or it reports that the search failed (for further discussion, see §V).

B. Tool Performance

Data concerning the performance of the tool is shown in Tab. I. *LoC* (lines-of-code) shows the number of lines of C source code of the program (library code is not included), *Procs* shows the number of procedures in the program, and *Call-sites* shows the number of procedure call-sites. Each time is the average of three runs on a machine with 16 2.4 GHz processors and 32 GB of memory, measured by the UNIX utility `time`. Each processor has four cores and a 12 MB cache. However, our tool does not explicitly exploit parallelism.

The data supports several claims about the effectiveness of the weaving algorithm. The policies are relatively small compared to the size of the programs because they are defined over only a small handful of important program instructions and capabilities. The performance times indicate that the tool can rewrite programs efficiently enough that a developer could feasibly integrate it into a system that periodically secures a program under development, or perhaps into a compiler toolchain.

Runtime overhead: Watson et al. [4] demonstrate that in practice, the only appreciable runtime overhead incurred in rewriting a program for Capsicum is typically due to RPCs; each invocation of the `enter_cm` and `limit_fd` primitives induces a small overhead on the order or microseconds or nanoseconds. The *RPC* column of Tab. I contains the number of RPCs added by our instrumentation vs. the number that we believe are required from a manual inspection. Although our algorithm is not guaranteed to minimize the number of RPCs, the number of RPCs added is extremely low compared to the number of all call sites in the program. This is partly because adding too many RPCs would cause most programs to violate their security policy, and partly because various standard optimizations applied to construct the game determine that the vast majority of program call sites are irrelevant to the policy. However, even though the number of RPCs is small, each can add considerable overhead, and so we consider the problem of minimizing them to be an important issue. The *Non-RPC* column shows the number of calls to non-RPC primitives (i.e., `enter_cm` and `limit_fd`) inserted by the tool. Many of the calls appear to have no effect on the state of the Capsicum monitor, and can likely be removed by a simple peephole optimizer. Extending our algorithm to find strategies that optimize a given performance or cost metric is left for future work.

V. LIMITATIONS

Injected code: An instrumentation function as defined in Defn. 2 invokes a Capsicum primitive in response to each program instruction. However, if a program is compromised, then it may execute arbitrary code that is not instrumented with Capsicum primitives. Defn. 2 can be extended to allow a programmer to incorporate knowledge of potentially vulnerable portions of code. In particular, the programmer models their program as executing a special instruction `inj` at program states corresponding to vulnerable portions of code. After the Capsicum monitor reads `inj`, the only primitive that it accepts is `noop`. After the program executes `inj`, it can execute any instruction, or RPC any function that the instrumented program calls via RPC. This models the conservative assumption that any function partitioned to execute in a different RPC can be called by trusted and injected code alike. The reduction to safety games described in §III-C can be extended in this way to handle injected code.

Policy-weaving policies as a foundation for richer policies: A policy-weaving problem is defined using a policy represented as a language of capability traces. In practice, such policies sometimes do not completely describe desired policies for a program. For example, a richer policy for `tcpdump` (§II) might require that an adversary who can observe all of `tcpdump`'s output channels, including any data that it sends over the network during DNS resolution, should only be able to determine information about certain address fields in an incoming packet sent over the network during a correct execution of `tcpdump`. A policy-weaving instrumentation of `tcpdump` can ensure that even if `tcpdump` is compromised, it cannot leak information about a packet directly to an adversary, e.g., by opening a file and writing the information over file. However, a correct policy-weaving policy for `tcpdump` must also ensure that the DNS resolver has the Env capability, in order to perform resolution. Such a policy allows executions in which a compromised `tcpdump` potentially leaks information about a packet through the DNS resolver.

However, while policy-weaving policies cannot always express all security properties desired of a given program, they can provide a foundation, expressed purely in terms of capabilities, on which a programmer can rewrite or reconfigure their program to satisfy a policy. The Capsicum developers rewrote `tcpdump` to invoke the Capsicum primitives so that it satisfied a policy similar to the one described in §II, ensuring that `tcpdump` could only leak packet information through the DNS resolver. The developers then rewrote Capsicum to be completely secure (informally) by configuring the DNS resolver to perform resolution by only reading tables stored on the local host, but not sending resolution requests over the network. We leave as future work the problem of formalizing such reasoning, and integrating it with policy-weaving policies.

Completeness: The reduction of §III-C is complete for finding solutions to Capsicum policy-weaving problems. We have found that in practice, if a SEVPA safety game constructed by a reduction from a Capsicum weaving problem has a winning Defender strategy, then it has a winning modular Defender strategy (see §IV). However, in principle, there may be weaving problems for which the corresponding safety game has no modular winning Defender strategy. Such problems fall into one of two cases.

First, some games have a winning Attacker strategy, which defeats any Defender strategy. We can search for a winning Attacker strategy in parallel to searching for a winning Defender strategy, and if one is found, provide it to the programmer. However, the problem of finding any global (i.e., not necessarily modular) Attacker strategy is EXPTIME-complete [27].

Second, some games have a winning global Defender strategy, but do not have a modular winning Defender strategy. We cannot use such a strategy to instrument a program, because the instrumentation function defined by such a strategy is not modular.

In both cases, while a given VPA abstraction of the program may not allow for a winning modular Defender strategy, there might be some more precise, yet sound, abstraction of the program that has a winning modular Defender strategy. It may be possible to automatically refine a VPA abstraction of a program from either a winning Attacker strategy or a proof of the absence of a modular Defender strategy, perhaps by extending CEGAR-style automatic refinement [28], [29]. We leave this as future work.

VI. Related Work

Security monitors: This paper describes an algorithm and a tool that automatically rewrite programs for Capsicum, an operating system that provides a set of capability-specific primitives [4]. Operating systems that provide security system calls as primitives allow one to define program-specific policies. In comparison, Mandatory Access Control (MAC) operating systems such as SELinux [8]–[10] only support system-wide policies described in terms of standard system calls. Such policies cannot refer to important events in the execution of a particular program, but many practical policies can only be defined in terms of such events [14]. UNIX can monitor programs to ensure that they satisfy policies if the program correctly uses the `setuid` system call, but in general this approach suffers the same shortcomings as MAC systems. In comparison, systems with security primitives allow an application to signal key events in its execution to the operating system.

An *Inline Reference Monitor* (IRM) rewriter takes a policy expressed as an automaton and instruments a target program with an IRM, which executes in the same memory space as the program, and halts the program if it attempts to perform some sequence of actions that would violate the policy [13], [14]. *Edit automata* [30] generalize IRMs by also supressing or adding security-sensitive events to ensure that the program satisfies a policy. Because an IRM (or edit automaton) executes in the same memory space as the program that it monitors, it can enforce policies defined over arbitrary events in the execution of the program. However, for the same reason, an IRM can only monitor the execution of managed code. In comparison, systems with security primitives can safely and efficiently monitor programs composed largely of unmanaged code [1], [4].

Writing programs for security monitors: Prior work in aiding programming for systems with security primitives automatically verifies that a program instrumented to use the Flume OS [12] primitives enforces a high-level policy [31], automatically instruments programs to use the primitives of the HiStar OS [1] to satisfy a policy [32], and automatically instruments programs [31] to use the primitives of the Flume OS [12]. However, the languages of policies used in the approaches presented in [32], [33] are not temporal and cannot clearly be applied to other systems with security primitives. The instrumentation algorithm presented in this paper is one instance of a general, automata-theoretic algorithm [23]. The main contribution of this paper is to describe how the general algorithm can be applied to rewrite programs for a practical capability-based system.

Previous work [34], [35] automatically partitions programs so that high and low confidentiality data are processed by separate processes, or on separate hosts. We automatically partition programs so that each process of the partitioned program can correctly invoke operating system primitves to satisfy a policy, when a single monolithic process cannot invoke primitives to satisfy the policy.

Skalka and Smith [36] present an algorithm that takes a Java program instrumented with capability security checks, and attempts to show statically that some checks are always satisfied. Hamlen et al. [37] verify that programs rewritten by an IRM rewriter are correct. Thus, the work in both of those papers concerns identifying superfluous capability checks in managed programs, whereas our work concerns correctly applying primitives to restrict the capabilities of unmanaged programs.

Safety games: Safety games have been studied as a framework for synthesizing reactive programs and control mechanisms [17], [38]–[40]. Previous work describes algorithms that take a safety game represented symbolically, determine which player may always win the game, and sometimes synthesize a winning strategy for the player [17], [40]. One contribution of our work is connecting these game-theoretic problems to the problem of rewriting a program for a capability system.

References

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

[2] R. Naraine, "Symantec antivirus worm hole puts millions at risk," *eWeek.com*, 2006. [Online]. Available: http://www.eweek.com/article2/0,1895,1967941,00.asp

[3] "Tcpdump/libcap public repository," 2011. [Online]. Available: http://www.tcpdump.org/

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

[5] M. Izdebski, "bzip2 'BZ2_decompress' function integer overflow vuln." Oct. 2011. [Online]. Available: http://www.securityfocus.com/bid/43331

[6] "Ubuntu sec. notice USN-709-1," http://www.ubuntu.com/usn/usn-709-1/, 2009.

[7] "Vuln. note VU#381508," http://www.kb.cert.org/vuls/id/381508, Jul. 2011.

[8] P. Loscocco and S. Smalley, "Integrating flexible support for security policies into the Linux operating system," in *USENIX Annual Technical Conference, FREENIX Track*, 2001.

[9] O. S. Saydjari, "Lock : An historical perspective," in *ACSAC*, 2002.

[10] C. Wright, C. Cowan, J. Morris, and S. S. G. Kroah-Hartman, "Linux security modules: General security support for the Linux kernel," in *Foundations of Intrusion Tolerant Systems*, 2003.

[11] 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.

[12] 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.

[13] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti, "Control-flow integrity," in *CCS*, 2005.

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

[15] "FreeBSD 9.0-RELEASE announcement," http://www.freebsd.org/releases/9.0R/announce.html, Jan. 2012.

[16] R. Alur, S. L. Torre, and P. Madhusudan, "Modular strategies for recursive game graphs," in *TACAS*, 2003.

[17] P. Madhusudan, W. Nam, and R. Alur, "Symbolic computational techniques for solving games," *Electr. Notes Theor. Comput. Sci.*, vol. 89, no. 4, 2003.

[18] S. McCanne and V. Jacobson, "The BSD packet filter: A new architecture for user-level packet capture," in *USENIX Winter Conference*, 1993.

[19] M. Y. Vardi and P. Wolper, "An automata-theoretic approach to automatic program verification (preliminary report)," in *LICS*, 1986.

[20] R. Alur and P. Madhusudan, "Visibly pushdown languages," in *STOC*, 2004.

[21] J. van Leeuwen, Ed., *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*. Elsevier and MIT Press, 1990.

[22] R. Alur, V. Kumar, P. Madhusudan, and M. Viswanathan, "Congruences for visibly pushdown languages," in *ICALP*, 2005.

[23] W. R. Harris, S. Jha, and T. Reps, "Secure programming via visibly pushdown safety games," University of Wisconsin, Dept. of Computer Sciences, Tech. Rep., January 2012. [Online]. Available: http://www.cs.wisc.edu/~wrharris/tech_reports/secure_programming_via_visibly.pdf

[24] L. de Moura and N. Bjørner, "Z3: An efficient SMT solver," in *TACAS*, 2008.

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

[26] P. J. Dawidek, Personal communication, Jan. 2011.

[27] T. Cachat, "Symbolic strategy synthesis for games on pushdown graphs," in *ICALP*, 2002.

[28] T. Ball and S. K. Rajamani, "The SLAM project: debugging system software via static analysis," in *POPL*, 2002.

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

[30] J. Ligatti, L. Bauer, and D. Walker, "Edit automata: Enforcement mechanisms for run-time security policies," *Int. J. Inf. Sec.*, vol. 4, no. 1-2, 2005.

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

[32] P. Efstathopoulos and E. Kohler, "Manageable fine-grained information flow," in *EuroSys*, 2008.

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

[34] D. Brumley and D. X. Song, "Privtrans: Automatically partitioning programs for privilege separation," in *USENIX Security Symposium*, 2004.

[35] S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng, "Secure web application via automatic partitioning," in *SOSP*, 2007.

[36] C. Skalka and S. F. Smith, "Static enforcement of security with types," in *ICFP*, 2000, pp. 34–45.

[37] K. W. Hamlen, G. Morrisett, and F. B. Schneider, “Certified in-lined reference monitoring on .NET,” in *PLAS*, 2006, pp. 7–16.

[38] R. Alur, T. A. Henzinger, and O. Kupferman, “Alternating-time temporal logic,” in *FOCS*, 1997.

[39] R. Alur, T. A. Henzinger, O. Kupferman, and M. Y. Vardi, “Alternating refinement relations,” in *CONCUR*, 1998.

[40] L. de Alfaro, T. A. Henzinger, and R. Majumdar, “Symbolic algorithms for infinite-state games,” in *CONCUR*, 2001.

APPENDIX

In this section, we sketch a proof that the Capsicum policy-weaving problem is NP-complete.

Theorem 2. *For program P and P-policy* Pol*, the Capsicum policy-weaving problem* $\mathsf{Weave}(P, \mathsf{Pol})$ *is* NP*-complete in the size of P and* Pol.

Proof: (sketch) Membership of $\mathsf{Weave}(P, \mathsf{Pol})$ in NP follows from Thm. 1, because the problem of finding modular strategies for SEVPA safety games is in NP.

We will show that the Capsicum-policy-weaving problem is NP-hard by reduction from 3SAT, similar to the hardness proof given in [16]. From an instance of 3SAT φ, we construct a weaving problem $\mathcal{P}_\varphi = (P, \mathsf{Pol})$ such that a satisfying instrumentation function for $\mathcal{P}_\varphi$ corresponds to a solution to φ. For every variable that occurs in φ, P has a function that executes three internal instruction, and then returns. The instrumentation function chooses whether or not to invoke `enter_cm` before a special internal instruction in each function. Pol stores the choices of the instrumentation function in each program function, and only accepts the resulting execution if the choices correspond to a satisfying assignment to φ. The choice regarding `enter_cm` for the internal instruction of the function for propositional variable x defines an assignment to x.

We now describe the reduction in more detail. Let φ be a 3SAT formula in CNF. Construct a program $P_\varphi = (V_\varphi, \emptyset)$ from φ as follows. For each variable x that appears in φ, introduce a program function f_x that executes in sequence the internal instructions ins_x^0, ins_x^1, and ins_x^2. For each clause $C = \langle l_0, l_1, l_2 \rangle$, introduce a program function f_C that calls in sequence f_{x_i}, where x_i is the variable of literal l_i. Let V_φ call in sequence f_C for each clause C.

Construct a policy VPA Pol_φ from φ as follows. Introduce policy states q_0 and q_1, with q_0 non-accepting, and q_1 the accepting initial state. For each variable x, if the instruction ins_x^0 executes without Env or executes ins_x^2 with Env, then let Pol_φ transition to a non-accepting stuck state s. Otherwise, if the instruction ins_x^1 executes with Env, then let Pol_φ transition to $q_{x,0}$, and transition to state $q_{x,1}$ otherwise. For each clause C, introduce a policy state $q_{C,0}$, to which the policy transitions on each call to f_C, and a state $q_{C,1}$. On a call to function f_x from f_C, let Pol_φ place its current state on the stack. If (1) on a return from f_x, $q_{x,1}$ is the top state of Pol_φ and x occurs as a positive literal in C, or (2) on a return from f_x, $q_{x,0}$ is the top state of Pol_φ and x occurs as a negative literal in C, or Pol_φ pushed the state $q_{C,1}$ on the stack when f_x was called, then Pol_φ transitions to $q_{C,1}$. Otherwise, Pol_φ transitions to $q_{C,0}$. When f_c returns, if Pol_φ pushed q_1 onto its stack when f_c was called and the top state of Pol_φ is $q_{C,1}$, then Pol_φ transitions to q_1. Otherwise, Pol_φ transitions to q_0. Pol_φ can be represented with two states per variable, two states per clause, the states q_0 and q_1, and the stuck state.

Each satisfying assignment of φ corresponds to an instrumentation function for $\mathsf{Weave}(V_\varphi, \mathsf{Pol}_\varphi)$. Let I be a satisfying instrumentation function for $\mathsf{Weave}(V_\varphi, \mathsf{Pol}_\varphi)$. Define an assignment σ_I for φ as $\sigma_I(x) = \mathsf{True}$ if and only if I causes V_φ to execute ins_x^1 with capability Env. Because each ins_x^0 must execute with Env and each ins_x^2 must execute without Env, a satisfying instrumentation function must call each function f_x with an RPC. Thus, whether instruction ins_x^1 executes with Env depends only on whether I inserts an invocation of `enter_cm` immediately before ins_x^1. I thus always requires ins_x^1 to executes with the same set of capabilities, either $\{\mathsf{Env}\}$ or $\emptyset$, because I is modular. As a result, we can use the above construction to define an assignment σ_I from I. Conversely, it follows that from each satisfying assignment σ for φ, we can define a satisfying instrumentation function for $\mathsf{Weave}(P_\varphi, \mathsf{Pol}_\varphi)$. ■