Declarative, Temporal, and Practical Programming with Capabilities

William R. Harris*, Somesh Jha*, Thomas Reps*†, Jonathan Anderson‡, and Robert N. M. Watson‡
*{ *wrharris, jha, reps* }*@cs.wisc.edu; University of Wisconsin-Madison, Madison, WI, USA*
†*GrammaTech Inc., Ithaca, NY, USA*
‡{ *jonathan.anderson, robert.watson* }*@cl.cam.ac.uk; University of Cambridge, Cambridge, England, UK*

***Abstract*—New operating systems, such as the Capsicum capability system, allow a programmer to write an application that satisfies strong security properties by invoking security-specific system calls at a few key points in the program. However, rewriting an application to invoke such system calls correctly is an error-prone process: even the Capsicum developers have reported difficulties in rewriting programs to correctly invoke system calls.**

This paper describes `capweave`, a tool that takes as input (i) an LLVM program, and (ii) declarative specifications of the possibly-changing capabilities that a program must hold during its execution, and rewrites the program to use Capsicum system calls to enforce the policies. Our experiments demonstrate that `capweave` can be applied to rewrite security-critical UNIX utilities to satisfy practical security properties. `capweave` itself works quickly, and the amount of runtime overhead incurred in the programs that `capweave` produces is generally low for practical workloads.

I. INTRODUCTION

Developing practical but secure programs remains a difficult, important, and open problem. Network utilities such as `tcpdump` and `wget` process data read directly from a network connection, but execute vulnerable code [1], [2]. File utilities and language interpreters are often run by a trusted user to process untrusted data, but also execute vulnerable code [3]–[7]. Once an attacker compromises vulnerable code in any of the above programs, he can typically perform any action allowed for the user that invoked the program.

Traditional operating systems provide only weak primitives for applications to manage their privileges. As a result, if a programmer wants 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 [8]–[11] has produced new operating systems that allow programmers to develop programs that execute unmanaged code yet satisfy strong security requirements. Moreover, programmers can develop such programs with much less effort than fully verifying the program for a traditional operating system. 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. The developers of such systems have manually modified applications to invoke security primitives so that the application satisfies strong security policies, even when the application is composed of untrusted code.

One example of an operating system with strong security primitives is the capability operating system Capsicum [10], now an experimental feature in FreeBSD 9 [12]. Capsicum allows a programmer to compartmentalize his program into separate modules that each have a subset of the full set of privileges, following the principle of least privilege. 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 (i.e., open more files). Capsicum provides to each process a set of system calls that the process uses to limit its capabilities. Thus, trusted code in a program can first communicate with its environment unrestricted by Capsicum, and then invoke primitives to limit itself to have only the 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 [10]. However, in practice it is difficult for programmers to reason about the subtle, temporal effects of the primitives. When the Capsicum developers first evaluated Capsicum, they rewrote programs, such as `tcpdump`, in a way that they tentatively thought was correct, only to discover later that the program was incorrect and required a different rewriting [10]. Often, as in the case of `tcpdump`, the difficulty results from the conflicting demands of (i) using low-level primitives, (ii) ensuring that the program satisfies a strong, high-level security requirement, and (iii) preserving the core functionality of the original program.

This paper addresses the problem of writing programs for capability systems, like Capsicum, by presenting a system, called `capweave`, that takes from a programmer (1) a program that does not invoke Capsicum primitives, and (2) a declarative, temporal policy, stated in terms of the capabilities that the program should hold over the course of its execution, according to the principle of least privilege.

capweave automatically compartmentalizes the program and instruments it to invoke Capsicum primitives so that it satisfies the policy when executed on Capsicum. We call the problem of finding such an instrumentation the Capsicum *policy-weaving problem*.

Our capweave *policy weaver* addresses two key 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. Because the Capsicum developers did not have such a language when first developing Capsicum, it was impossible for them to formally 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 the program. However, the rewritten program may limit its capabilities too strongly at one point of an execution, and as a result, may not have the capabilities required to carry out core program functionality later in the execution. The incorrect rewriting of tcpdump [10] is an example of this issue. To resolve conflicts between security and functionality, a programmer must typically carefully rewrite his program to maintain additional state about an execution, and consult the state to determine when to invoke Capsicum primitives and execute a program function in a process with distinct capabilities.

An additional challenge in designing capweave was to structure it so that it uses a simple, declarative model of Capsicum. Capsicum system and application developers have developed and continue to develop libraries of functions that an application can invoke to more easily manage its capabilities [13]. For the remainder of this paper, we refer to both the system calls and library functions that a program invokes to manage its capabilities as *security primitives*. When a Capsicum architect implements a new primitive, he should be able to easily extend capweave so that it can instrument programs to invoke the new primitive, but he should not need to understand the details of the instrumentation algorithm used by capweave.

To address the programmer's first challenge, capweave provides a policy language with which a programmer can write an explicit, declarative, general policy that restricts the privileges of the program in terms of capabilities. Each policy is a regular expression over an alphabet of program points paired with capabilities. The policy allows all program executions that execute with the specified restricted privileges.

To address the programmer's second challenge, capweave takes an uninstrumented program and its policy, and automatically instruments the program to satisfy the policy. To do so, capweave constructs from the program, policy, and the semantics of Capsicum an *automata-theoretic safety game* [14] between an "Attacker," who "plays" program instructions, and a "Defender" who plays Capsicum primitives, by applying an *automata-theoretic policy weaver* [15]. The Attacker wins the game if the sequence of plays violates the policy, and the Defender wins otherwise. capweave searches for a winning Defender strategy, and from the strategy, instruments the program to (i) maintain instrumentation-state variables, and (ii) invoke Capsicum primitives based on the values of the variables so that the program satisfies its policy.

For a Capsicum architect to update capweave for an updated version of Capsicum, they only need to update declarative definitions of (i) the state maintained by Capsicum, (ii) the primitives available to a program, (iii) and each primitive's effect on the Capsicum state. In practice, the state and primitives are easy to define: together they account for only one tenth of the capweave source code.

We determined experimentally that capweave allows a programmer to rewrite practical programs to satisfy policies that rule out known critical exploits of the programs. We applied capweave to rewrite several UNIX utilities for Capsicum that have demonstrated security vulnerabilities. The rewritten programs included programs that were previously rewritten manually by the Capsicum team, programs suggested through discussion with the Capsicum development team, and the PHP CGI interpreter, whose policy was defined by independent security researchers at MIT Lincoln Laboratory. capweave allowed us to rewrite each utility using only a small handful of program annotations, no more than 11 lines, and a simple high-level policy of no more than 115 lines in our policy language. Each policy not only ruled out specific known exploits, but restricted the capabilities of significant segments of the program, potentially ruling out a large class of future vulnerabilities. Programs rewritten by capweave executed with equivalent behavior to programs instrumented manually by an expert, and incurred sufficiently low runtime overhead that they are still deployable: only 4% runtime overhead over unwoven programs on realistic workloads. We have provided a Capsicum virtual machine containing all programs and policies used in our experiments.[1]

Organization: §II uses the wget downloader to illustrate the Capsicum policy-weaving problem and capweave. §III discusses the design of capweave in detail. §IV presents an experimental evaluation of the correctness and performance of capweave and programs rewritten by applying capweave. §V discusses related work, and §VI concludes.

[1]The virtual machine used is available at https://www.dropbox.com/s/711q31mccz47rt4/capweave-exp-vm.tar.gz.

```
    void wget(char* uls[], int num_urls) {
      // For each URL input by the user:
      for (int i = 0; i < num_urls; i++) {
C0:     sync_fork();
L0:     char* url_nm = urls[i];
        // If the URL is an HTTP resource:
L1:     if (is_http(url_nm)) {
          // Open a socket to the server:
L2:       int svr_sock = open_http(url_nm);
          char* out_path = url_nm;
          bool redir_url = false;
          /* If server sends redirect
           * with status 3xx: */
          if (must_3xx_redirect(svr_sock)) {
            redir_url = true;
            /* Get the name of the output
             * file from the server: */
L3:         out_path = get_outnm(svr_sock);
          }
L4:       char* data = read_http(svr_sock);
C1:       redir_url ? cap_enter() : ;
L5:       write_data(out_path);
        } else { ... }
C2:     sync_join();
      }
    }
```

Figure 1. Pseudocode of the `wget` downloader instrumented to invoke Capsicum primitives. `wget` takes an array of URL's as input, and writes the data at each URL to the file system of its host. Particularly subtle segments of `wget`'s code are annotated with comments, and discussed in §II-A. Capsicum primitives are typeset in bold font.

II. Motivation

In this section, we motivate the Capsicum policy-weaving problem, and illustrate our solution by describing how `capweave` is used to secure the `wget` downloader.

A. `wget`: an Insecure Program and a Desired Policy

We now present a simplified version of the `wget` downloader and a desired security policy that past versions of `wget` do not satisfy. The `wget` downloader is a command-line utility that takes as input a list of URL's. For each URL, `wget` attempts to download the data addressed by the URL and write the data in the file system of `wget`'s host. `wget` is a mature, sophisticated tool that supports the HTTP, HTTPS, and FTP protocols, can be run non-interactively, and consists of 64,443 lines of C source code, including whitespace and comments [16].

Pseudocode for a simplified version of `wget` is given in Fig. 1. Important program points are annotated with C labels (e.g., `L0`). (Statements in Fig. 1 in bold font are invoked by a version of `wget` instrumented for Capsicum. Such statements are discussed in §II-B; for now, assume that `wget` does not execute such statements.) `wget` takes as input an array of URL's. For each input URL, `wget` fetches the data addressed by the URL and writes the data to the file system of the host system on which `wget` runs. In particular, for each URL, `wget` determines under what protocol the URL is addressed (Fig. 1, line `L1`). Once `wget` determines the protocol used, it runs protocol-specific functions to (i) open a socket to the server holding the URL (line `L2`), (ii) download the data addressed by the URL over the socket (lines `L3` and `L4`), and (iii) write the data to a file to the file system (line `L5`).

Unfortunately, versions of `wget` through v.1.12 demonstrate a vulnerability that allows an attacker who controls a server with which `wget` interacts to write data to any file on the host file system that can be written by the user who runs `wget`. The vulnerability is exposed when `wget` processes a particular HTTP response from the server. In particular, `wget` may receive from a server a redirect response, which directs `wget` to download data from a different network address. When `wget` receives such a response, it determines the path on its host file system to which it will write data directly from the information provided by the redirect server. A malicious server can exploit this behavior to craft a redirect response that causes `wget` to write data chosen by the attacker to a path in the file system chosen by the attacker. A server can exploit such a vulnerability to execute code on the host system by directing `wget` to write data to an appropriate startup or configuration file [2].

Ideally, a `wget` developer would formally specify that `wget` must not demonstrate a vulnerability along the lines of the one described above, and would rewrite `wget` so that it satisfies such a specification. However, rewriting `wget` to do so requires detailed knowledge of both the structure of `wget` and of the HTTP protocol. Thus, it would be useful if a developer could define an acceptable, if perhaps weaker, specification for `wget` in terms of commonly-used, well-understood operating-system objects, such as file descriptors, and automatically rewrite `wget` to satisfy such a policy. In particular, one useful policy for `wget` defined in terms of file descriptors would be:

Policy 1. *When `wget` executes `read_http`, it should always be able to open arbitrary files and sockets. But `wget` should execute `write_data` with the ability to open files if and only if it has not received an HTTP-redirect response.*

B. Securing `wget` on Capsicum

The Capsicum operating system [10] provides a set of powerful security-oriented system calls (i.e., *primitives*) that an application can invoke to ensure that it only demonstrates secure behavior, even if an attacker triggers a serious vulnerability in the application. Capsicum extends the notion of a file descriptor provided by UNIX to that of a *capability* by mapping each file descriptor opened by a process to a set of access rights that the process holds for the file descriptor. Each right corresponds roughly to a UNIX system call that operates over file descriptors (e.g., the access right `CAP_READ` corresponds to the `read` system call). When a

process running on Capsicum invokes a system call `c` on file descriptor `f`, Capsicum only carries out `c` if the process holds the right `CAP_C` for `f`. Capsicum also maps each executing process to an *ambient-capability flag*, which is a Boolean value that controls whether the process can open new file descriptors.

Capsicum's capabilities were designed so that a program executing on Capsicum begins by executing a small, trusted code segment that manages capabilities, and then executes complex, untrusted code that can interact with its environment only through the capabilities set by the trusted code. When a process opens a file descriptor, it holds all access rights for the descriptor. Throughout its execution, a process can invoke a Capsicum primitive `limitfd(d, R)` on descriptor `d` and set of rights `R` to decrease its rights for `d` to only those in `R`. A process begins executing with the capabilities of its parent, and can invoke the Capsicum primitive `cap_enter` to relinquish the ambient authority.

A programmer can instrument `wget` to invoke the Capsicum primitives so that it satisfies the informal security policy introduced in §II-A. The instrumented version of the example `wget` is the code shown in Fig. 1, including the Capsicum primitives shown in bold font. Essentially, `wget` is instrumented so that if it handles an HTTP redirection, then it invokes `cap_enter` before attempting to write data to its host's file system (line `L3`).

However, for a programmer to instrument his program to invoke Capsicum primitives correctly, he must address two challenges, illustrated by the instrumented version of `wget`.

First, once a programmer formulates a policy, he must modify his to invoke the Capsicum primitives to enforce the policy. However, the Capsicum primitives can have subtle consequences. In the example `wget`, once the programmer determines that under some conditions, `wget` should execute program point `L3` without ambient authority, then the programmer can immediately deduce that `wget` must sometimes invoke the `cap_enter` primitive before executing `L3`. However, once the programmer also determines that if `wget` does not receive a redirect response, then `wget` should execute `L3` with ambient authority, it is fairly difficult for him to decide how to instrument `wget`. The difficulty stems from the fact that once a process invokes `cap_enter`, then the process can never obtain the ambient authority for the rest of its execution. Thus, if a `wget` process attempts to download from URL `u`, receives a redirection response, and then invokes `cap_enter`, then the process must execute without ambient authority when downloading from all input URL's following `u`.

`wget` can be instrumented to satisfy the full informal policy of Policy 1 by compartmentalizing it to use multiple communicating processes, as follows. A "main" `wget` process executes the loop that iterates over the list of input URL's. To download data from each input URL, the main process synchronously forks a worker process to download the data and write it to the file system (line `C0`). Each worker process begins executing with ambient authority. If the worker receives an HTTP-redirect response while downloading from its URL, then it invokes `cap_enter`, but when the worker process terminates (line `C1`), the main `wget` process continues to execute with ambient authority, with which it forks the next worker process. (Capsicum only requires that a child process begin executing with the capabilities of its parent, but places no restrictions on the capabilities of the parent based on the capabilities of its children.)

```
let redir_exploit =
    any_instr* . [ L0 ] . any_instr*
  . [ L3 ] . [not L0]*
  . [ L5 with AMB ] in
let noredir_fails =
    any_instr* . [ L0 ]
  . [ not { L0, L3 } ]*
  . [ L5 with (no AMB) ] in
let http_fails =
  any_instr* . [ L4 with (no AMB) ] in
  redir_exploit | noredir_fails | http_fails
```

Figure 2. A `capweave` policy for the example `wget` given in Fig. 1. The policy is a regular expression that matches all executions of wget that constitute undesired executions of `wget`, and is described in §II-C.

Second, the instrumented program sometimes must update and consult additional instrumentation state to determine when to invoke Capsicum primitives. In Fig. 1, the instrumented `wget` maintains a Boolean variable `redir_url` that reflects whether or not `wget` received a redirection response when downloading from the current URL. The instrumented `wget` invokes `cap_enter` (line `C1`) if and only if `redir_url` is true.

Thus, an application can be rewritten to satisfy strong security requirements while preserving the functionality of the original program by correctly manipulating capabilities across multiple communicating processes and maintaining additional instrumentation state. However, it is non-trivial to determine how to rewrite an application to do so. In particular, the control locations at which an application must invoke primitives to satisfy a policy might not be near each other in the application's code. For example, in Fig. 1, `wget` invokes `fork` and `cap_enter` at distant program points.

C. Securing `wget` on Capsicum with `capweave`

§II-A and §II-B illustrate the general challenges that a programmer faces in rewriting a program to execute correctly on Capsicum. While a programmer can typically define the desired behavior of his rewritten program purely in terms of capabilities (e.g., Policy 1), Capsicum does not allow the programmer to state such a policy explicitly. Instead, the programmer must instrument his program manually to invoke primitives that manipulate both capabilities and

processes so that the resulting program executes with the desired capabilities. To help a programmer address this challenge, we have developed a Capsicum policy weaver, called `capweave`. `capweave` takes as input from the programmer (1) a program that invokes no Capsicum primitives (for the example `wget`, the code in Fig. 1 without the instrumentation statements), and (2) a policy describing correct executions of the program purely in terms of the capabilities that the instrumented program should have as it executes. For `wget`, a formalization of Policy 1 is given in Fig. 2, which is discussed below.

A policy that `capweave` takes as input is a regular language of *capability traces* that each constitute a policy *violation*, where a capability trace is a sequence of program points paired with the capabilities that the program has when it executes the program point. A policy regular expression that represents the Policy 1 is given in Fig. 2. The language of violations in Fig. 2 is defined as the union of three sublanguages: `redir_exploit`, `noredir_fails`, and `http_fails`. `redir_exploit` formally expresses the set of all `wget` executions in which an attacker exploits `wget`'s vulnerability in processing HTTP redirection responses. `redir_exploit` is defined as any sequence of instructions, followed by the program point at which the next URL in the array of inputs is selected (`L0`), followed by any sequence of instructions, followed by the program point at which `wget` processes an HTTP redirect response (`L3`), followed by any sequence of instructions before the selection of the next input URL (`not L0`), followed by `wget` writing downloaded data to the file system (`L5`) with ambient authority.

`noredir_fails` formally expresses the set of all `wget` executions in which `wget` does not receive an HTTP redirection response, but attempts to write downloaded data to the file system with insufficient capability. `noredir_fails` is defined as any sequence of instructions, followed by `L0`, followed by any sequence of instructions other than `L0` or `L3`, followed by executing `L5` with ambient authority.

`http_fails` formally expresses the set of all `wget` executions in which `wget` attempts to finish an HTTP session without ambient capability. `http_fails` is defined as any sequence of instructions followed by attempting to complete the HTTP protocol (`L4`) without ambient capability.

For the simplified version of `wget` given in Fig. 1, the accompanying `capweave` policy given in Fig. 2 is almost as large as the program itself. However, in practice, policies for real-world programs tend to grow very slowly in the size of the program. For example, the real `wget` program contains 64,443 lines of source code, but its entire policy can be expressed in only 35 lines of our policy language.

`capweave` outputs a version of the input program instrumented to invoke Capsicum primitives so that it satisfies the input policy. From the uninstrumented version of the example `wget` (i.e., Fig. 1 without the instrumentation statements) and the example policy given in Fig. 2, `capweave` outputs the correctly instrumented version of `wget` (Fig. 1 with the instrumentation statements).

D. `capweave` Parametrized on the Capsicum Semantics

The implemented version of `capweave` is actually structured slightly differently than described above: the implemented tool supports a more general model in which `capweave` is *generated* from an explicit description of the semantics of Capsicum [15]. The advantage of this approach is that it provides an easy way to adapt `capweave` either when Capsicum is extended or when the "packaging" of sequences of invocations of Capsicum primitives as a library API is changed.

The Capsicum semantics defines (i) the state maintained by Capsicum as a program executes, (ii) the set of primitives that an instrumented program can invoke, and (iii) the effects of each primitive on the Capsicum state. For instance, the state maintained by Capsicum is a stack of *process states*, where a process state is (a) a map from each descriptor to its current set of access rights, and (b) a Boolean value indicating whether the process has ambient authority. If a process state p_0 is below a process state p_1 on the stack, then the process whose state is p_0 spawned the process whose state is p_1 via a synchronous `fork`. The semantics also defines the effect of each primitive on the Capsicum state. For instance, `cap_enter` sets the Boolean value to `False` in the process state of the currently executing process (i.e., the top process on the stack); `fork` pushes a copy of the top process state onto the stack; `join` pops the top process state from the stack; etc.

It is significantly easier for a Capsicum architect to define a model of Capsicum using this mechanism than it would be for him to implement the entire policy weaver. The entire `capweave` implementation consists of 35k lines of OCaml that employs many subtle optimizations, whereas the Capsicum model is specified in only 3k lines, which essentially define a Capsicum interpreter. (The Capsicum state and interpretation functions are discussed in more detail in §III-A3.)

In general, the Capsicum semantics would be specified by a Capsicum architect, rather than an application implementer, and would be changed rarely—either when new Capsicum primitives are introduced or when there are changes in the API of a library that packages Capsicum calls into routines that are more convenient to use than "raw" Capsicum. Application programmers can then regenerate an updated `capweave` tool and weave policies into as many applications as they wish.

III. Design of the Policy Weaver

In this section, we formally define the Capsicum policy-weaving problem, and describe our algorithm to solve it.

$$
\begin{aligned}
\mathsf{prog} &:= (\mathsf{block}_0, \{\mathsf{block}_1, \ldots, \mathsf{block}_n\}) \\
\mathsf{block} &:= \texttt{LABEL} : \mathsf{stmt}; \mathsf{termin} \\
\mathsf{stmt} &:= \mathtt{v}_0 := \mathsf{op}(\mathtt{v}_1, \ldots, \mathtt{v}_n) && \mathtt{v}_i \in \mathsf{Vars} \\
&\quad \mid \mathsf{dscinst} \\
&\quad \mid \mathsf{wvinstrs} \\
\mathsf{termin} &:= \texttt{halt} \mid \texttt{br v ? LABEL}_t \texttt{ : LABEL}_f && \mathtt{v} \in \mathsf{Vars} \\
\mathsf{dscinst} &:= \mathsf{os} : \mathtt{v}_0 := \mathtt{open}(\mathtt{v}_1), \\
&\quad \mathtt{v}_0, \mathtt{v}_1 \in \mathsf{Vars}, \mathsf{os} \in \mathsf{Opens}
\end{aligned}
$$

Figure 3. Syntax of the IMP⟨wvinstrs⟩ language: an imperative language parametrized on a set of woven instructions wvinstrs.

$$
\begin{aligned}
\mathsf{capinstr} &:= \mathtt{v}_0 := \mathsf{op}(\mathtt{v}_1, \ldots, \mathtt{v}_n) && \mathtt{v}_i \in \mathsf{WVars} \\
&\quad \mid \mathtt{v} \texttt{ ? } \mathsf{capprim} && \mathtt{v} \in \mathsf{WVars} \\
\mathsf{capprim} &:= \texttt{cap\_enter} \\
&\quad \mid \texttt{limitfd}(\mathsf{os}, \mathsf{rs}) && \mathsf{os} \in \mathsf{Opens}, \mathsf{rs} \subseteq \mathsf{Rights} \\
&\quad \mid \texttt{fork} \\
&\quad \mid \texttt{join}
\end{aligned}
$$

Figure 4. Syntax of the set of Capsicum woven instructions capinstr.

We also describe how components of the problem, such as unwoven and woven programs and Capsicum, are modeled in practice.

A. The Policy-Weaving Problem

1) Language Syntax: The syntax of languages of both unwoven and woven programs will be defined as instances of a language of simple imperative programs, IMP. IMP is a small "core" language that supports only updates to program state with the result of language operations, operations on descriptors, invocations of woven instructions, and conditional branches of control-flow. However, the implementation of our weaving algorithm instruments programs in the LLVM intermediate language [17], and thus can weave programs compiled from widely-used, practical languages, such as C and C++.

Syntax of Unwoven Programs: The syntax of language IMP⟨wvinstrs⟩ (Fig. 3) is defined for a fixed set of program variables Vars, a fixed set of control labels Labels, and a set of *open sites* Opens that label program instructions at which descriptors are opened. The syntax is parametrized on a set of woven instructions wvinstrs. An IMP program prog is a set of instruction blocks, including an initial instruction block block_0. Each instruction block is a unique label, a statement, and a block-terminator instruction. A statement stores the result of a language operation, opens a descriptor, or executes a weaving instruction. A block terminator halts the program or branches. The language of unwoven programs UNWOVEN is the language of imperative programs with no woven instructions: UNWOVEN = IMP$\langle\emptyset\rangle$.

Syntax of Woven Programs: The language of woven programs is the language of IMP programs that may execute Capsicum woven instructions (Fig. 4), defined over a set of weaving variables WVars. A Capsicum woven instruction may store the result of a language operation in a variable in WVars, or may execute a guarded invocation of a *Capsicum primitive*. A Capsicum primitive is either cap_enter, fork, join, or limitfd(os, rs), for os ∈ Opens and rs ⊆ Rights. A woven program is an IMP program instrumented to execute Capsicum instructions: WOVEN = IMP⟨capinstr⟩.

2) Language Semantics: In this section, we define a semantics of WOVEN programs by mapping each WOVEN program to the executions that it may perform. In particular, we define a semantic function τ that maps every WOVEN program P and initial program state p to the trace of capabilities that P holds throughout its execution from p. τ is defined using an operational-semantic function σ_s that describes how each program statement updates the state of the program. σ_s is defined in terms of the operational semantic functions σ_w and σ_c, which define how each weaving instruction updates the program state. σ_w and σ_c are defined in terms of the *Capsicum interpretation* of Capsicum primitives, which defines how each primitive updates the state maintained by Capsicum (§III-A3).

The semantics of WOVEN, given in Fig. 5, is defined by a function τ (Fig. 5, Eqns. (1) and (2)) that maps each program in WOVEN and initial program state p to the sequence of capabilities that the program holds during an execution that starts from p. Let a *program state* be an assignment from each program variable to an integer value: progstates = Vars → $\mathbb{Z}$, where $\mathbb{Z}$ denotes the set of integers. A *capability state* is the state maintained by the Capsicum operating system. The set of capability states capstates is defined by the Capsicum architect (see §III-A3). Let a *capability trace* be a sequence of program labels paired with the capabilities that the program has as it executes the block with the given label: captraces = (Labels × capstates)*. τ maps each program $P \in$ WOVEN and program state $p \in$ progstates to the capability trace that P generates in an execution that starts from p (Fig. 5, Eqn. (1)). The trace generated by P from a program state p is the trace that the initial block of P generates in an execution that starts from p, along with a fixed initial woven state and capability state (Fig. 5, Eqn. (2)).

The semantic function τ_b, given in Fig. 5, Eqns. (3) and (4), defines the capability trace that a block generates in an execution from a given state. Let a *woven state* be a map from each weaving variable to an integer value: wvstates = WVars → $\mathbb{Z}$. Let a *full state* be a program-state, woven-state, and capability-state triple: fullstates = progstates × wvstates × capstates. Then τ_b defines the capability trace generated by executing a given block from a given full-state (Fig. 5, Eqn. (3)). The trace generated by executing a block LABEL : $s; t$ from a full state f is LABEL

$$\tau \;:\; \textsc{Woven} \to \mathsf{progstates} \to \mathsf{captraces} \tag{1}$$

$$\tau[\![(\mathsf{block}_0, \{\mathsf{block}_1, \ldots, \mathsf{block}_n\})]\!](p) = \tau_b[\![P, \mathsf{block}_0]\!](p, \emptyset, c_i) \tag{2}$$

$$\tau_b \;:\; (\textsc{Woven} \times \mathsf{block}) \to \mathsf{fullstates} \to \mathsf{captraces} \tag{3}$$

$$\tau_b[\![P, \texttt{LABEL} : \mathsf{stmt}; \mathsf{termin})]\!](p, i, c) = (\texttt{LABEL}, c) :: \tau_t[\![P, \mathsf{termin}]\!](\sigma_s[\![\mathsf{stmt}]\!](p, i, c)) \tag{4}$$

$$\tau_t \;:\; (\textsc{Woven} \times \mathsf{termin}) \to \mathsf{fullstates} \to \mathsf{captraces} \tag{5}$$

$$\tau_t[\![P, \texttt{halt}]\!](p, i, c) = \epsilon \tag{6}$$

$$\tau_t[\![P, \texttt{br v ? LABEL}_t \;:\; \texttt{LABEL}_f]\!](p, i, c) = \mathsf{let\ dest} = \mathsf{if}\ p(\mathtt{v}) \neq 0\ \mathsf{then}\ \texttt{LABEL}_t\ \mathsf{else}\ \texttt{LABEL}_f\ \mathsf{in}\ \tau_b[\![\mathsf{labelblk}_P(\mathsf{dest})]\!](p, i, c) \tag{7}$$

$$\sigma_s \;:\; \mathsf{stmt} \to \mathsf{fullstates} \to \mathsf{fullstates} \tag{8}$$

$$\sigma_s[\![\mathtt{v}_0 := \mathsf{op}(\mathtt{v}_1, \ldots, \mathtt{v}_n)]\!](p, i, c) = (\iota_{\mathsf{o}}[\![\mathtt{v}_0 := \mathsf{op}(\mathtt{v}_1, \ldots, \mathtt{v}_n)]\!](p), i, c) \tag{9}$$

$$\sigma_s[\![\mathsf{dscinst}]\!](p, i, c) = \mathsf{let}\ (p', c') = \iota_{\mathsf{d}}[\![\mathsf{dscinst}]\!](p, c)\ \mathsf{in}\ (p', i, c') \tag{10}$$

$$\sigma_s[\![\mathsf{capinstr}]\!](p, i, c) = \sigma_c[\![\mathsf{capinstr}]\!](i, c) \tag{11}$$

$$\sigma_c \;:\; \mathsf{capinstr} \to (\mathsf{wvstates} \times \mathsf{capstates}) \to (\mathsf{wvstates} \times \mathsf{capstates}) \tag{12}$$

$$\sigma_c[\![\mathtt{v}_0 := \mathsf{op}(\mathtt{v}_1, \ldots, \mathtt{v}_n)]\!](i, c) = (\iota_{\mathsf{o}}[\![\mathtt{v}_0 := \mathsf{op}(\mathtt{v}_1, \ldots, \mathtt{v}_n)]\!](i), c) \tag{13}$$

$$\sigma_c[\![\mathtt{v}\ ?\ \mathsf{capprim}]\!](i, c) = (i, \mathsf{if}\ i(\mathtt{v}) \neq 0\ \mathsf{then}\ \iota_{\mathsf{p}}[\![\mathsf{capprim}]\!](c)\ \mathsf{else}\ c) \tag{14}$$

Figure 5. Semantics of WOVEN. τ, τ_b, and τ_t define the capability trace that a WOVEN program generates by executing a given program, block, or terminator instruction, respectively, from a given state. In the definition of τ_t, $\mathsf{labelblk}_P$ maps each label to the instruction block that it labels. σ_s and σ_c define how a program statement and woven instruction, respectively, update the state of a program. ι_{o} and ι_{p} denote the interpretation of program operations and Capsicum primitives (Fig. 6), respectively). progstates, wvstates, capstates, and fullstates denote the spaces of program states, woven states, capability states, and "full" program states, respectively (see §III-A2).

paired with the capability state in f, followed by the trace generated by executing the terminator instruction t starting in the full-state obtained by updating f with the statement s (Fig. 5, Eqn. (4)).

The terminator semantic function τ_t, given in Fig. 5, Eqns. (5)–(7), defines the trace generated by executing a block terminator from a given full-state $f \in \mathsf{fullstates}$. The terminator `halt` generates the empty trace (Fig. 5, Eqn. (6)). The terminator `br v ? LABEL`$_t$ `: LABEL`$_f$ generates the trace of executing either the block labeled `LABEL`$_t$ or the block labeled `LABEL`$_f$ from full-state f, depending on whether `v`'s value is non-zero or zero, respectively (Fig. 5, Eqn. (7)).

The statement semantic function σ_s, given in Fig. 5, Eqns. (8)–(11), defines how a statement $s \in \mathsf{stmt}$ updates a full-state $f \in \mathsf{fullstates}$. If s is a language operation, then it updates the program state in f according to the language semantics of the operation (Fig. 5, Eqn. (9)). In Eqn. (9), the language semantics is denoted by the function $\iota_{\mathsf{o}} : \mathsf{stmt} \to \mathsf{progstates} \to \mathsf{progstates}$, and omitted for brevity. If s is a descriptor instruction, then it updates the program and capability state in f according to the Capsicum interpretation of descriptor instructions ι_{d} (Fig. 5, Eqn. (10); for a discussion of ι_{d}, see §III-A3). If s is a weaving instruction, then it updates the weaving and capability state in f as defined by the weaving-instruction semantic function σ_w (Fig. 5, Eqn. (11)).

The woven-instruction semantic function σ_w, given in Fig. 5, Eqns. (12)–(14), defines how a woven instruction $v \in \mathsf{wvinstrs}$ updates a woven state $w \in \mathsf{wvstates}$ and a capability state $c \in \mathsf{capstates}$. If v stores the value of a language operation in a woven-state variable, then the woven state is updated according to the language semantics (Fig. 5, Eqn. (13)). If the woven instruction is a guarded Capsicum primitive `v ? p`, then if `v` is zero in w, `v ? p` does not update the woven state, and otherwise, `v ? p` updates c according to the Capsicum interpretation of primitive `p` (Fig. 5, Eqn. (14); for a discussion of the interpretation of primitives ι_{p}, see §III-A3).

3) Capsicum Interpretation Functions: The semantics of WOVEN (§III-A2) is defined from (1) the space of capability states maintained by Capsicum, (2) the initial capability state with which a program executes, and (3) the Capsicum interpretations, which define how program instructions and Capsicum primitives update capability state. If the semantics of Capsicum were to be extended or revised in some way, these are the only pieces of information that a Capsicum architect would have to modify to obtain an updated version of `capweave`.

A capability state defines what capabilities are held by a program (Fig. 6, Eqns. (15) and (16)). Let a *process capability state* be a Boolean flag, denoting whether a process is in capability mode, together with a map from each descriptor to a set of rights (Fig. 6, Eqn. (15)). Then a capability state is a stack of process capability states (Fig. 6, Eqn. (16)).

The initial capability state c_i is a singleton stack containing a process state denoting that the process has ambient capability, and that the rights of no descriptors are defined: $c_i = [(\mathsf{True}, \emptyset)]$.

The Capsicum interpretation functions are given in Fig. 6.

$$\mathsf{proccap} = \mathbb{B} \times (\mathsf{Opens} \to \mathcal{P}(\mathsf{Rights})) \tag{15}$$

$$\mathsf{capstates} = \mathsf{proccap}^* \tag{16}$$

$$\iota_{\mathsf{d}} \;:\; \mathsf{dscinst} \to (\mathsf{progstates} \times \mathsf{capstates}) \to (\mathsf{progstates} \times \mathsf{capstates}) \tag{17}$$

$$\iota_{\mathsf{d}}[\![d : x := \texttt{open}(y)]\!](p, r :: rs) = \mathsf{let}\ (fd, R') = \mathsf{if}\ \mathsf{ambcap}(r)\ \mathsf{then}\ (\mathsf{fresh}(r), \mathsf{Rights})\ \mathsf{else}\ (-1, \bot)\ \mathsf{in}\ (p[x \mapsto fd], (\mathsf{ambcap}(r), \mathsf{rights}(r)[d \mapsto R']) :: rs) \tag{18}$$

$$\iota_{\mathsf{p}} \;:\; \mathsf{capprim} \to \mathsf{capstates} \to \mathsf{capstates} \tag{19}$$

$$\iota_{\mathsf{p}}[\![\texttt{cap\_enter}]\!](r :: rs) = (\mathsf{False}, \mathsf{rights}(r)) :: rs \tag{20}$$

$$\iota_{\mathsf{p}}[\![\texttt{limitfd}(d, R)]\!](r :: rs) = (\mathsf{ambcap}(r), \mathsf{rs}(r)[d \mapsto \mathsf{rs}(r)(d) \cap R]) :: rs \tag{21}$$

$$\iota_{\mathsf{p}}[\![\texttt{fork}]\!](r :: rs) = r :: r :: rs \tag{22}$$

$$\iota_{\mathsf{p}}[\![\texttt{join}]\!](r :: r' :: rs) = (\mathsf{ambcap}(r'), \mathsf{descrights}(r)) :: rs \tag{23}$$

Figure 6. Definition of the space of capability states and interpretation functions. proccap denotes the space of process states. In Eqn. (16), $\mathcal{P}$(Rights) denotes the power set of Capsicum access rights. capstates denotes the space of capability states. ι_{d} and ι_{p} denote the interpretations of descriptor instructions and Capsicum primitives, respectively. In Eqns. (18), (21), and (23), ambcap(r) and rs(r) denote the ambient-authority flag and map from descriptors to access rights, respectively, in process state r. In Eqn. (18), fresh(r) denotes a new descriptor value that is not bound in process state r.

The first interpretation function ι_{d} defines how each descriptor instruction $i \equiv$ `d: x := open(y)` (Fig. 6, Eqns. (17) and (18)) updates a program state $p \in$ progstates and capability state $c \in$ capstates (Fig. 6, Eqn. (17)). If the program holds ambient authority in c, then i updates p so that `x` holds a fresh descriptor, and updates c so that the fresh descriptor has all access rights. Otherwise, i updates p so that `x` holds the value -1, and the latest descriptor opened at `d` is not mapped to any set of access rights (Fig. 6, Eqn. (18)).

The second interpretation function ι_{p} (Fig. 6, Eqns. (19) and (20)) specifies how a Capsicum primitive $p \in$ Capprims updates a capability state $c \in$ capstates (Fig. 6, Eqn. (19)). If a program executes the Capsicum primitive `cap_enter`, then the program relinquishes ambient authority (Fig. 6, Eqn. (20)). If a program invokes the Capsicum primitive `limitfd(d,R)`, then the program's rights for the last descriptor opened at d are updated to the intersection of the program's rights in c and the set of rights R (Fig. 6, Eqn. (21)). If a program invokes the Capsicum primitive `fork`, then the program pushes a copy of the current process-capability state onto the stack of process capability states (Fig. 6, Eqn. (22)). If a program invokes the Capsicum primitive `join`, then the program pops its top process capability state pc, and updates the new top process state in its capability state to have the descriptor rights in pc (Fig. 6, Eqn. (23)).

Policy Semantics of WOVEN*:* A policy is a set of executions of a program annotated with the capabilities that the program must have as it executes. Although the capability state of a program completely defines the capabilities held by a program as it executes, writing policies defined by the complete capability-state may be complicated or infeasible. In particular, the Capsicum interpretation functions in Fig. 6 are defined over capability states that are stacks of process capability states, but practical policies typically are defined over only the currently executing process (i.e., the top process on the stack of process capability states).

To bridge the gap between the capability state maintained by Capsicum and the state used to define policies, the Capsicum architect defines a space of *policy states* polstates and a policy state abstraction α : capstates $\to$ polstates that maps each capability state to the policy state that represents it. Our implementation of `capweave` allows policies to be defined using the capabilities of the currently executing process: polstates = proccap, and $\alpha(\mathsf{cur} :: \mathsf{procs}) = \mathsf{cur}$.

A policy state abstraction α defines a *policy semantics function* that maps each woven program and initial program state to the trace of program labels paired with policy states that the program generates in an execution from the initial program state. For poltraces = (Labels $\times$ polstates)*, the policy semantics function τ_α : WOVEN $\to$ (progstates $\to$ poltraces) is

$$\tau_\alpha[\![\mathsf{prog}]\!](p) = m_\alpha(\tau[\![\mathsf{prog}]\!](p))$$

where

$$m_\alpha(\epsilon) = \epsilon$$
$$m_\alpha((\texttt{LABEL}, c) :: t) = (\texttt{LABEL}, \alpha(c)) :: m_\alpha(t)$$

4) Problem Definition: The policy-weaving problem is to take an unwoven program and a policy, and construct a weaving of the unwoven program that satisfies the policy. We formally define the weaving problem using the policy semantics of a program and the definition of a weaving. To simplify the definition of the policy-weaving problem, we fix the definition of the Capsicum interpretation functions, initial state, and policy-state abstraction to be as defined in Fig. 6 and §III-A2. The definition of the program and policy semantics of WOVEN programs is thus fixed as well.

For an unwoven program $P \in$ UNWOVEN and a woven program $P' \in$ WOVEN, P' is a weaving of P if P' is constructed by only adding woven instructions to P.

Defn. 1. *For* IMP *statements s and s', s'* is a weaving of *s if one of the following holds*

- *s is not a sequence of statements, and (1) $s' = s$ or (2) $s' = s'_0; s'_1$ and either s'_0 is a weaving of s and s'_1 is a sequence of woven instructions, or s'_0 is a sequence of woven instructions and s'_1 is a weaving of s.*
- *s is a sequence of statements $s_0; s_1$ and s' is a sequence of statements $s'_0; s'_1$ where s'_0 is a weaving of s_0 and s'_1 is a weaving of s_1.*

A program $P' = (\mathsf{block}'_0, \{\mathsf{block}_1, \ldots, \mathsf{block}'_n\})$ is a weaving *of a program $P = (\mathsf{block}_0, \{\mathsf{block}_1, \ldots, \mathsf{block}_n\})$ if for each $0 \leq i \leq n$, $\mathsf{block}_i = \texttt{LABEL}_i : s_i; t_i$ and $\mathsf{block}'_i = \texttt{LABEL}_i : s'_i; t_i$, where s'_i is a weaving of s_i.*

The policy-weaving problem is to take an unwoven program and a policy defining the allowed executions of the program, and instrument the program so that it satisfies the policy.

Defn. 2. *Let $P \in$ UNWOVEN be an unwoven program, and let $Q \subseteq \mathsf{captraces}$ be a regular language of capability traces. For a woven program P', let the traces of P', denoted as $\mathcal{T}(P') \subseteq \mathsf{captraces}$, be the set of capability traces generated by some input to the program: $\mathcal{T}(P') = \{\tau_\alpha[\![P']\!](i) \mid i \in (\mathsf{Vars} \to \mathbb{Z})\}$. A solution to the policy-weaving problem $\text{WEAVE}(P, Q)$ is a woven program $P' \in$ WOVEN such that P' is a weaving of P (Defn. 1) and $\mathcal{T}(P') \subseteq Q$.*

WEAVE is undecidable in general; it can be shown that any algorithm that could solve WEAVE could decide if a program in a Turing-complete language satisfies an arbitrary safety property. `capweave` uses a sound but incomplete solver for WEAVE, described in §III-B.

B. Solving Policy Weaving with Automata Games

We have developed a sound but incomplete solver for WEAVE, called `capweave`, that reduces WEAVE to finding a winning strategy to a two-player safety game, played by an Attacker and a Defender. `capweave` uses an existing automata-theoretic weaver-generator algorithm [15] as its core engine. To make the paper self-contained, this section summarizes that algorithm, and describes how `capweave` applies the weaver generator to weave practical programs for Capsicum.

The weaver generator solves a version of the policy-weaving problem in which an input program, a policy, and the operating system are all modeled as automata. The weaver generator solves such a problem by reducing it to finding a *modular winning strategy to a two-player safety game*. Intuitively, a two-player safety game is an automaton in which the set of states is partitioned into a set of Attacker states and a set of Defender states. When the game is in an Attacker state, the Attacker can transition the state to any adjacent state, and analogously for the Defender. The goal of the Attacker is to eventually transition the game to an accepting state of the automaton, and the goal of the Defender is to prevent the Attacker from doing so. A

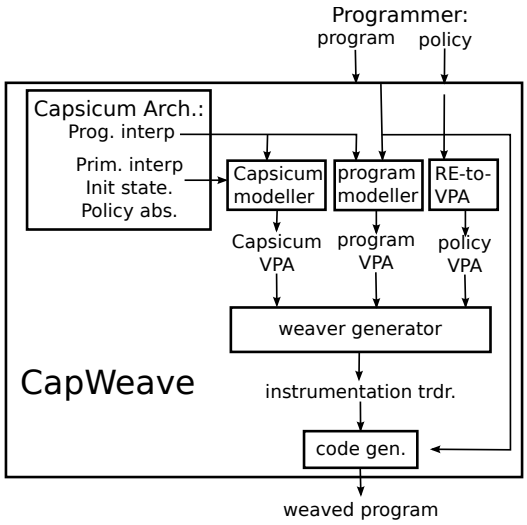

Figure 7. Architecture of `capweave`. Items in the box labeled "Capsicum Arch." are defined by the Capsicum architect. "Init. state" denotes the initial Capsicum state, "Desc. interp." denotes the interpretation of Capsicum program statements, "Prim. interp" denotes the interpretation of Capsicum primitives, and "Policy abs." denotes the policy abstraction.

winning Attacker (Defender) strategy is a function that reads the transitions chosen by both the Attacker and Defender and outputs a transition for the Attacker (Defender) such that if the Attacker (Defender) always chooses the transition output by the strategy, then the Attacker (Defender) always wins the game. For a game defined by a pushdown automaton, a *modular* Attacker (Defender) strategy is a strategy that outputs transitions independent of the transitions chosen before the most recent unmatched push transition.

If a game defined by a restricted classes of pushdown automata called *Visibly Pushdown Automata (VPA)* [18], has a winning modular Attacker or Defender strategy, then the strategy can be found efficiently [14].

Our policy weaver `capweave` soundly reduces a policy-weaving problem $\text{WEAVE}(P, Q)$ to the problem of finding a winning Defender strategy to a game defined by a VPA. Intuitively, `capweave` constructs a game in which the choices of an Attacker correspond to instructions that a program can execute, the choices of a Defender correspond to Capsicum primitives that can be invoked, and accepting states are reached when the program violates a policy. A winning Defender strategy for the game thus corresponds to a weaving that ensures that the woven program never violates the policy. The problem of finding a winning Defender strategy is NP-complete in general, but in practice `capweave` finds a winning strategy to a game efficiently by applying heuristics introduced in previous work([15], Sec. 4).

Fig. 7 illustrates how `capweave` applies the weaver generator. The weaver generator takes as input a program and policy represented as VPA's, and an operating system

(e.g., Capsicum) modeled as a visibly-pushdown transducer. Thus, to apply the weaver generator, `capweave` must soundly model its input program, policy, and Capsicum as VPA's. `capweave` models the program as a VPA constructed directly from the program's interprocedural control-flow graph, which is a standard technique in program analysis [18]. However, in principle, the program can be modeled by any VPA that overapproximates the possible executions of the program (e.g., models constructed via predicate abstraction [19]). Each policy that `capweave` takes as input is a regular language, so the policy can be represented as a finite-state automaton, and thus as a visibly-pushdown automaton [18]. `capweave` constructs the transducer model of Capsicum from the space of capability states and Capsicum interpretations (§III-A3). Details of this construction are given in App. VII-A.

The weaver generator produces an instrumentation strategy represented as a transducer that reads a sequence of program instructions and outputs the next Capsicum primitive that a woven program should execute. `capweave` compiles such an instrumentation transducer to a woven program by representing the transition function of the instrumentation transducer using a state variable and the woven instructions in `capinstr` (§III-A1). `capweave` weaves its input program to consult its state variable to determine which Capsicum primitive to invoke next as the program executes, and then update the state variable. This compilation scheme is described further in App. VII-B.

If a programmer provides a program P and policy Q for which `capweave` cannot find a solution to $\textsc{Weave}(P, Q)$, then `capweave` can, in principle, provide useful diagnostic information to the programmer. There are multiple reasons why `capweave` may not be able to find a solution to a weaving problem $\textsc{Weave}(P, Q)$: (1) $\textsc{Weave}(P, Q)$ may not have a solution. (2) $\textsc{Weave}(P, Q)$ may have a solution, but `capweave` may not find a solution because either (a) when `capweave` constructs a VPA model of P, the resulting model allows P to perform more executions than P can actually perform, or (b) the solution does not correspond to a modular winning Defender strategy [15]. In all cases except (b), the weaver generator produces a *winning Attacker strategy* that describes the executions that P can perform to violate Q, no matter what Capsicum primitives are invoked by any weaving. From such an Attacker strategy and the Capsicum semantics, `capweave` could construct an unweavable policy Q' that is no more restrictive than Q (i.e., $Q' \supseteq Q$). `capweave` could then either validate that Q' is truly unsatisfiable and provide Q' to the user as an explanation of `capweave`'s failure, or use Q' to refine its model of the input program. In case (b), `capweave` could still apply various heuristics to iteratively weaken the input policy, check if the weakened policy has a weaving, and provide this information as diagnostics to the programmer. We have not implemented support for handling failures in `capweave`, and do not evaluate `capweave`'s practicality in handling such failures. We leave such an evaluation for future work.

C. Weaving Practical Programs

In §III-A4, we defined the policy-weaving problem for a simple imperative language. However, the weaving problem and our policy weaver can be extended to handle programs written in practical programming languages, such as C, in which programs may have multiple recursive procedures, or manipulate compound datatypes. In particular, our actual policy weaver has been implemented to weave programs represented in the LLVM intermediate language [17].

A key strength of Capsicum is that a program that runs on Capsicum may run code injected by an attacker (e.g., via a stack-smashing attack [20]), and yet can still satisfy a non-trivial security policy. Thus, in practice it is critical that our policy weaver be able to correctly weave programs that can run injected code. We could implement such a weaver by extending the language semantics and policy-weaving problem described in §III-A2 and §III-A4 in a straightforward way. The only change we would need to make is that the policy weaver would not be able to choose what Capsicum primitives the woven program will execute after it executes specified program points at which the program might run injected code.

A programmer who uses `capweave` must understand his program sufficiently well to define a correct policy in terms of program actions paired with capabilities. Furthermore, in practice, the size of a policy may not differ significantly from the size of the code required to instrument the policy. However, the key utility of `capweave` is that it allows the programmer to reason purely in terms of capabilities that the program must hold over its execution. Because a programmer's ultimate goal is to write a program that holds desired capabilities, this reasoning is strictly easier than determining desired capabilities and then rewriting the program to use the intricate Capsicum primitives to induce the capabilities.

More sophisticated programming tools could further ease the burden of using `capweave` by, e.g., inferring a likely policy from the callsites of system calls that manipulate descriptors. Such a tool need only determine the capabilities that the program requires as it executes, and discharge to `capweave` the problem of instrumenting the program to hold the required capabilities. To evaluate fully the utility of `capweave` and related tools would require a comprehensive programmer study. We leave this as future work.

IV. Practical Experience and Experiments

We carried out a set of experiments to evaluate the practical utility of `capweave`. The experiments were designed to answer the following questions:

1) Does `capweave` allow a programmer to rewrite a program with less effort and with higher assurance than if he manually rewrote the program to invoke Capsicum's primitives?
2) Does `capweave` rewrite real-world programs to enforce practical policies efficiently?
3) Do programs produced by `capweave` behave comparably, both in terms of correctness and performance, to programs manually modified by an expert to satisfy the same policy?

To answer these questions, we applied `capweave` to a set of UNIX utilities—all of which had previously demonstrated security vulnerabilities—so that the instrumented program satisfied a policy that thwarted the vulnerabilities. The programs and policies were derived from either previous work done in developing Capsicum [10], discussions with Capsicum system and application developers on the Capsicum mailing list [13], or collaborative work with security researchers at MIT Lincoln Laboratory (MITLL).

The `capweave` implementation is 35k lines of OCaml, and uses the LLVM OCaml API, which corresponds closely to the LLVM API provided for C and C++. `capweave` takes as input programs in the LLVM intermediate representation (called *bitcode*), and outputs instrumented programs as bitcode as well. As a result, it can be inserted into any compiler toolchain that compiles a source program to LLVM bitcode in some intermediate phase.

`capweave` generates instrumentation code as a multi-dimensional array that defines what Capsicum primitive should be called as each program point is executed. While the generated code likely cannot be understood easily by a programmer, the instrumentation transducer (§III-B) can be recovered from the generated array. The transducer could perhaps be used by a programmer to more easily understand the instrumentation, or could be reused to generate code directly for a different version of the program.

The results of our experiments demonstrate that `capweave` is useful for rewriting programs for Capsicum. In particular:

1) For each of our subject programs, a programmer could apply `capweave` by annotating their program with only 4–11 lines of code, and writing a policy, expressed purely in terms of Capsicum capabilities, that could be represented with 35–114 lines of our policy language. Thus, `capweave` can be applied to rewrite programs to satisfy explicit, declarative policies with minimal effort.
2) `capweave` wove all programs in less than five minutes, except for the PHP CGI interpreter, which it wove in 46 minutes. Thus, `capweave` is efficient enough to be applied to programs in, say, a nightly build system, and in many cases could reasonably be integrated into a compiler toolchain used in an edit-compile-run cycle.
3) Programs rewritten by `capweave` to satisfy a given policy match programs manually written by an expert to satisfy the same policy, and run with overhead within 4% of unwoven programs on practical workloads. Thus, in practice, `capweave` produces programs that behave comparably to those written by an expert programmer.

A. Methodology

To answer the experimental questions presented in §IV, we applied `capweave` to weave a set of UNIX utilities as security and performance benchmarks. In this section, we describe each of the benchmarks applied, and then describe the experiments that we performed on each benchmark.

1) Benchmark Programs and Policies: We now describe each of the benchmark programs and policies used, including its role as a security-critical application, and the source of its policy. While each policy used was inspired by a known vulnerability in the benchmark, each policy restricts the capabilities of large portions of its program's execution. For example, the policies for `bzip2` and `gzip` strongly limit the capabilities of both programs for as long as they execute their compression and decompression functions. Each policy thus potentially mitigates a large class of vulnerabilities that may be unknown when the policy is written. The policy also explicitly describes the limitations of any program rewritten to satisfy the policy, and thus the limits of any attacker who compromises the rewritten program.

`bzip2` and `gzip`: The compression programs `bzip2` and `gzip` can be used by a trusted user to compress data from an untrusted source. On BSD systems, they are often used by `root` to decompress ports of applications. The compression and decompression functions of `bzip2` and `gzip` are heavily optimized and quite complex, and have demonstrated security vulnerabilities in the past [3], [4]. An attacker who can control the inputs to `bzip2` and `gzip` can craft an input that allows him to execute arbitrary code with the privileges of the user who invoked `bzip2` or `gzip`.

We defined a `capweave` policy that strictly limits the abilities of an attacker who compromises `bzip2` or `gzip`. The policy restricts `bzip2` and `gzip` to execute with only the capability to read from the source file that holds uncompressed data and write to the file opened to store the compressed output. An attacker who compromises a version of `bzip2` or `gzip` that satisfies such a policy can compromise the integrity of the output files of `bzip2` or `gzip`, but cannot carry out other malicious actions. For instance, the attacker cannot overwrite arbitrary files. Our `capweave` policies for `bzip2` and `gzip` were inspired by previous work on manually writing programs for Capsicum [10], [21].

`php-cgi`: Executing programs written in web scripting languages, such as PHP, raises multiple security issues. First, it is inherently difficult to analyze, monitor, and restrict the behavior of a program written in a scripting language.

Second, a maliciously-crafted web program can potentially compromise the interpreter that executes it, and then perform any action on its host system that is allowed for the user who launched the interpreter [5].

Using `capweave`'s policy language, we defined a policy for the PHP CGI interpreter `php-cgi` that allows the interpreter to only read from and write to files defined by a small set of simple, easily-audited checking functions. Our policy strictly limits the file I/O of `php-cgi` itself, and thus indirectly limits the I/O of any PHP script that the interpreter executes. We defined the policy by collaborating with a group of researchers from MITLL.

`tar`: The `tar` archiving utility archives sets of files into a single file. Unfortunately, past versions of `tar` have demonstrated vulnerabilities that allow an attacker who controls the inputs to `tar` to run injected code with the privileges of the user who invoked `tar` [6], [7].

We defined a `capweave` policy that strictly limits the abilities of an attacker who compromises `tar`. The policy restricts `tar` to execute vulnerable functions without ambient authority. An attacker who compromises a version of `tar` that satisfies such a policy can compromise the integrity of output files opened by `tar`, but cannot carry out other malicious actions.

`tcpdump`: `tcpdump` is a widely-used network-facing application that historically has been the target of many exploits. `tcpdump` takes as input a Berkeley Packet Filter (BPF), and a device from which to read packets. In a correct execution, it reads packets from the device, matches them against the input BPF, and if the packet matches, prints the packet to standard output. Unfortunately, the packet-matching code in `tcpdump` is complex; in previous versions of `tcpdump`, an attacker who controls the network input to `tcpdump` can craft a packet that allows him to take control of the process executing `tcpdump` [1].

We defined a policy for `tcpdump` in the `capweave` policy language that strictly limits the power of an attacker who is able to compromise `tcpdump`. In previous work on Capsicum [10], the Capsicum developers instrumented `tcpdump` so that it could only read from its input network device and write to standard output. The Capsicum developers later found through testing that this instrumentation did not allow `tcpdump` to resolve network addresses in a packet, and the developers reinstrumented `tcpdump` so that only a small, trusted DNS resolver could open files. The capweave policy for `tcpdump` describes the policy that our revised instrumentation satisfies.

`wget`: The `wget` downloader, its vulnerabilities, and its `capweave` policy were discussed in §II. Our policy for `wget` was inspired by discussion on the Capsicum-developer mailing list and the known vulnerabilities of `wget` [2], [13].

B. Experimental Procedure

For each of the benchmark programs and policies described in §IV-A1, we defined the benchmark's policy in the `capweave` policy language and applied `capweave` to the program and policy. We also obtained a version of each program that was manually modified to satisfy the policy. In previous work, we manually rewrote `bzip2`, `gzip`, and `tcpdump` to satisfy informal versions of the policies described in §IV-A1. We recompiled these versions with the LLVM compiler so that we could compare their runtime overhead with the runtime overhead of the programs woven by `capweave`. We manually instrumented the other three benchmarks to satisfy each of their policies ourselves. The woven and manually instrumented programs were compiled with the default optimization of each benchmark ("-O2" for each program). `capweave` was applied to optimized LLVM bitcode.

We ran each benchmark on a set of test workloads that exercised various behaviors of each benchmark. We ran `bzip2`, `gzip`, `php-cgi`, `tcpdump`, and `wget` on the test workloads included in the source distribution of each program. We ran `tar` to archive its own source directory. We ran each original benchmark, woven benchmark, and manually rewritten benchmark on the benchmark's test suite, and compared the executions. In particular, we counted the number of tests that each of the benchmark programs passed and measured the runtime performance overhead of the rewritten programs compared to the original benchmark. Because the total time taken by each benchmark on the test workloads supplied with the source was often less than a second, we also measured performance on a larger, more realistic workload.

To validate that the woven programs mitigated attacks according to their policy, we introduced into each program a "backdoor" into that attempted to carry out an attack disallowed by the program's policy. We then ran the woven program on an input that triggered the backdoor, and observed that the backdoor failed to carry out its attack.

C. Analysis of Results

The results of our experiments are given in Tab. I. (The performance numbers reported in Tab. I are from the test workloads included in the source distribution of each program. Performance on larger, more realistic workloads is discussed below.) For each benchmark, Tab. I contains three groups of measurements of our experience weaving the benchmark. The first group (cols. 3–5) measures the complexity of the policy for which the benchmark was woven, and contains the number of lines of code in which each policy is represented in our policy language, as well as the size of the policy DFA constructed by `capweave`. The number of lines of policy-language code ("Lines") indicates that the policy language supported by `capweave` can express practical policies relatively concisely.

Program Features		Policy Size			Weaver		Woven-Program Performance							
Prog. Name	Size (KLOC)	Lines	States	Alpha. Size	Weaving Time	Weaving Memory	Tests	OK	Weaved Points	Interproc. Funcs.	Base (s)	Weaved Overhd.	Hand Overhd.	capweave/ Hand (%)
bzip2-1.0.6	8	70	5	5,156	4m57s	0.3 GB	6	6	66	1	0.593	1.099	0.909	20.90
gzip-1.2.4	9	68	5	1,787	3m26s	0.2 GB	2	2	55	1	0.036	1.278	1.111	15.03
php-cgi-5.3.2	852	114	11	15,777	46m36s	25.3 GB	11	2	213	2	0.289	1.938	1.170	65.64
tar-1.25	108	49	8	143	0m08s	0.2 GB	1	1	62	2	0.156	21.917	13.301	64.78
tcpdump-4.1.1	87	52	6	223	0m09s	0.3 GB	29	27	88	1	1.328	1.224	0.981	24.77
wget-1.12	64	35	3	549	0m10s	0.2 GB	4	4	246	1	4.539	1.106	1.096	0.91

Table I
EXPERIMENTAL DATA FOR A SET OF BENCHMARKS PROGRAMS AND POLICIES. THE FIELDS OF THE TABLE ARE DISCUSSED IN §IV-C. IN THIS TABLE, THE PERFORMANCE NUMBERS REPORTED ARE THOSE FROM THE TEST WORKLOADS INCLUDED IN THE SOURCE DISTRIBUTION OF EACH PROGRAM. PERFORMANCE ON A LARGER, MORE REALISTIC WORKLOAD IS DISCUSSED IN §IV-C.

The second group of measurements in Tab. I (cols. 6–7) measures the performance of `capweave`, and contains the time and peak memory used by `capweave`. Each benchmark was woven on a server that has sixteen 2.4 GHz cores and 32 GB of memory, although the `capweave` implementation executes serially. The running time (cf. "Weaving Time") and peak memory (cf. "Weaving Mem.") indicate that `capweave` could be included in the edit-compile-run toolchain of many programs of small-to-medium size, and could be included in the nightly build system of a program of large size. The running time and peak memory also indicate that the performance of `capweave` is strongly determined by the size of the policy, in particular the size of the policy-automaton alphabet, more than the size of the input program (cf. "States", "Alpha. Size", "Weaving Time", and "Weaving Mem").

The third group of measurements in Tab. I (cols. 8–15) measures the performance of programs rewritten by `capweave`, and contains the number of reference tests that the unwoven (i.e., *baseline*) program passed ("Tests"), the number of tests passed by the woven version of each benchmark ("OK"), the number of program points at which `capweave` added instrumentation ("Weaved Points"), the number of functions that the benchmark executes in a synchronous `fork` ("Interproc. Funcs."), the runtime of the baseline program ("Base"), the runtime of the woven program expressed as a multiple of the runtime of the baseline ("Weaved Overhead"), the runtime of the hand-woven program as a multiple of the runtime of the baseline ("Hand Overhead"), and the percentage overhead of the runtime of the woven program over the runtime of the hand-woven program ("`capweave`/ Hand"). The geometric mean of all "`capweave`/ Hand" values is 1.298 (i.e., `capweave` overhead is 29.8%). Each benchmark was run on a host machine with eight 2.2 GHz processors and 6 GB of memory, in a Capsicum virtual machine with one processor and 2 GB of memory.

Each woven program behaved identically to its corresponding hand-woven program on each test, and behaved identically to its corresponding unwoven program on each test, except for some tests included with `tcpdump` and `php-cgi`. The woven `tcpdump` failed tests included with `tcpdump` that gave a filepath to a file containing a secret key for decrypting IPsec ESP packets, instead of giving the secret key directly on the command line. The `tcpdump` policy specified that the woven `tcpdump` should not be able to open any file except for the input network device, and hence could not open the file containing the secret key. The woven `php-cgi` failed tests included with `php-cgi` because no program can simultaneously satisfy the policy specified by the MITLL group and pass all of the tests. For example, the MITLL policy does not allow a PHP program to create a new file in its current directory. The woven `php-cgi` thus failed any test in which a program tried to create a file for output in its current directory. `tcpdump` and `php-cgi` thus illustrate one key aspect of `capweave`: `capweave` allows a programmer to specify the capabilities that a program should hold throughout an execution, and automatically obtain a program that holds the specified capabilities. However, a programmer still must determine manually whether the capabilities specified by a particular policy strike an acceptable balance between the security requirements of the program and its original functionality.

In future work, we plan to extend `capweave` to address another of its current limitations, namely, essentially no information is provided when policy weaving fails. We will provide diagnostic aids designed to help the programmer determine how a policy can be weakened so that policy weaving will succeed.

The number of program points at which `capweave` introduced instrumentation (i.e., "Weaved Points") was small relative to the size of each benchmark. Furthermore, the number of functions that each woven benchmark executed between a synchronous `fork` and `join` (i.e., "Interproc. Funcs") was small, and matched the number of functions that each hand-woven benchmark executed between a synchronous `fork` and `join` (and thus likely was minimal). However, woven versions of `bzip2`, `gzip`, `php-cgi`, and `wget` incurred noticeable overhead. We suspected that the

woven versions of these programs would introduce the most overhead on small workloads, because on such workloads, the fixed overhead of executing a synchronous `fork` and `join` dominates the overall runtime of the program.

To measure the performance of the woven programs on larger workloads, we generated a 1 GB file of source code from the Capsicum kernel source tree, and used it as a workload for `bzip2`, `gzip`, and `wget`. The unwoven `bzip2` compressed the large file in 25m31s, and the woven `bzip2` compressed the large data with 4% overhead over the baseline time. The unwoven `gzip` compressed the large file in 5m27s, and the woven `gzip` compressed the large data with 3% overhead over the baseline time. The unwoven `wget` downloaded the large file from a server on the same local network in 1m06s, and the woven `wget` downloaded the large data with -4% overhead over the baseline time, indicating that the overhead of the weaving is obscured by noise introduced by network traffic. Thus, the maximum overhead of the woven programs over unwoven programs is 4%, and geometric mean of all the overheads is 1%. The overhead for `php-cgi` depends entirely on how frequently an input PHP script opens files over the course of its execution.

The woven versions of `tar` and `tcpdump` introduced noticeable overhead on operations that execute frequently on all workloads, such as a subprocedure in `tar` that reads data into a buffer, or a subprocedure in `tcpdump` that resolves network addresses to names. The per-operation overhead induces an enormous overhead in `tar` in particular, and illustrates a third limitation of `capweave`: some policies induce `capweave` to instrument costly primitives, such as `fork`, at program points that induce considerable overhead, when `capweave` potentially may be able to instrument other program points that induce much less overhead. The overhead of the hand-woven program, while less than the overhead of the woven program, is still considerable: in our experience, weaving `tar` efficiently is a difficult problem, and one that could benefit significantly from further automatic-tool support. In future work, we hope to address this limitation by extending `capweave` to use a cost metric, and generalizing the game solver to find *optimal* strategies to *quantitative* games [22].

V. Related Work

Capability systems: Karger [23] introduced a capability system that mitigates the effects of an attack by a malicious program run on the system. The Capsicum operating system [10] provides security primitives to support isolating components of a program in sandboxes that run with different capabilities based on UNIX file descriptors. This paper describes the `capweave` tool, which greatly eases the burden of using Capsicum by allowing policies to be stated as separate policy specifications that `capweave` weaves into the program automatically.

Security monitors: Operating systems that provide security system calls, such as Capsicum, HiStar [11], Wedge [24] etc., allow an application developer to define program-specific policies (where the nature of the policy depends on the security primitives offered by the operating system). In contrast, Mandatory Access Control (MAC) operating systems, such as [25]–[27] only support system-wide policies described in terms of standard system events. 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 [28]. 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. Watson has described the challenges of developing an access-control system, and has surveyed recent implementations of such systems [29].

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 [28], [30]. *Edit automata* [31] 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 [10], [11].

Writing programs for security monitors: Prior work on programming aids for systems with security primitives automatically verifies that a program instrumented to use the Flume OS [9] primitives enforces a high-level policy [32], automatically instruments programs to use the primitives of the HiStar OS to satisfy a policy [33], and automatically instruments programs [32] to use the primitives of the Flume OS [9]. However, the languages of policies used in the approaches presented in [33], [34] are not temporal and cannot clearly be applied to other systems with security primitives. The weaving algorithm presented in this paper applies a known automata-theoretic weaving algorithm [15]. The main contribution of this paper is to describe how the automata-theoretic algorithm can be applied as an engine to rewrite programs for a practical capability system.

In the `privsep` project [35], OpenSSH was rewritten manually to execute using a trusted, privileged parent process and an unprivileged child process. Privman [36] is a library that a programmer can use to manually compartmentalize a UNIX daemon into high and low-privilege

processes. Previous work [37], [38] automatically partitions programs so that high and low confidentiality data are processed by separate processes, or on separate hosts. The SOAPP project [39] proposes a semi-automatic technique in which a programmer annotates a program with a hypothetical sandbox, and a program analysis validates that the sandbox does not introduce unexpected program behavior. The SOAAP approach is similar in spirit to [32], which uses model checking to verify that a programmer-proposed partitioning and set of calls to security primitives satisfies a given policy. In contrast, `capweave` automatically infers where to invoke library functions that cause the program to execute in different processes (if necessary), and rewrites the program accordingly.

Skalka and Smith [40] 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. [41] 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 how to infer the correct placement of 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 [42]–[45]. Previous work describes algorithms that take a safety game, determine which player can always win the game, and synthesize a winning strategy for the winning player [44], [45]. The key contribution of our work is to demonstrate that such game-theoretic problems can be applied in practice to rewrite secure programs.

VI. Conclusion

New operating systems, such as the Capsicum capability system, define powerful system-level primitives for secure programming, but such primitives are non-trivial to use. This paper presents a policy-weaver for Capsicum, `capweave`, that takes from a programmer an uninstrumented program and a high-level policy that describes correct behavior of the program. `capweave` automatically infers where to invoke security primitives and rewrites the program accordingly. In practice, `capweave` produces programs that match the behavior and performance of programs manually modified by an expert. `capweave` is designed so that a Capsicum Architect can easily add, remove, or update new programming libraries as they continue to be developed.

Acknowledgments: We would like to gratefully acknowledge the work of the Capsicum development team, in particular Pawel Dawidek, Khilan Gudka, and Ben Laurie, in developing the Capsicum and manually instrumenting programs for Capsicum. We thank our shepherd Niels Provos.

VII. Appendix

A. From Capsicum Semantics to a Transducer Model

`capweave` takes as input a program and policy, constructs VPA models of the program, policy, and Capsicum, and provides these models to the policy-weaver generator. In this section, we describe the construction of the models in more detail. We first define the automata that the weaver generator takes as input.

A deterministic visibly-pushdown automaton (VPA) is a stack machine that reads distinct sets of actions that cause it to modify that top state on its stack, push a new state onto its stack, and pop the top state from its stack. [18]

Defn. 3. *A deterministic $(\Sigma_I, \Sigma_C, \Sigma_R)$-visibly-pushdown automaton $V = (\Sigma_I, \Sigma_C, \Sigma_R, Q, q_0, Q_F, \tau_I, \tau_C, \tau_R)$ is a tuple, where*

- Σ_I *is the alphabet of* internal actions.
- Σ_C *is the alphabet of* call actions.
- Σ_R *is the alphabet of* return actions. Σ_I, Σ_C *and* Σ_R *are mutually disjoint.*
- Q *is the set of* states.
- $q_0 \in Q$ *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.
- $\tau_C : Q \times \Sigma_c \to (Q \times Q)$ *is the* call transition function.
- $\tau_R : Q \times Q \times \Sigma_R \to Q$ *is the* return transition function.

A visibly-pushdown transducer (VPT) is a VPA extended with an alphabet symbols, which outputs a symbol from on each transition.

Defn. 4. *A deterministic $(\Sigma_I, \Sigma_C, \Sigma_R, \Sigma_O)$-*visibly-pushdown transducer *(VPT) $T = (\Sigma_I, \Sigma_C, \Sigma_R, \Sigma_O, Q, q_0, Q_F, \tau_I, \tau_C, \tau_R)$ is a tuple, where*

- Σ_I *is the alphabet of* internal actions.
- Σ_C *is the alphabet of* call actions.
- Σ_R *is the alphabet of* return actions.
- Σ_O *is the alphabet of* output actions. Σ_I, Σ_C, Σ_R *and* Σ_O *are mutually disjoint.*
- Q *is a set of* states.
- $q_0 \in Q$ *is the* initial state.
- $Q_F \subseteq Q$ *is the set of* accepting states.
- $\tau_I : Q \times \Sigma_I \to Q \times \Sigma_O$ *is the* internal transition function.
- $\tau_C : Q \times \Sigma_C \to Q \times Q \times \Sigma_O$ *is the call transition function.*
- $\tau_R : Q \times Q \to Q \times \Sigma_O$ *is the return transition function.*

The input of an automata-theoretic policy-weaving problem is a program VPA defined over an alphabet of program actions, a policy VPA defined over an alphabet of program actions paired with privileges, and a system transducer that maps a sequence of program actions and primitives to a sequence of privileges held by the program [15].

Defn. 5. *An* input of an automata-theoretic policy-weaving problem *is a tuple* $I = (\Sigma_I, \Sigma_C, \Sigma_R, R, \Sigma_{PI}, \Sigma_{PC}, \Sigma_{PR}, P, Q, S)$*, where*

- Σ_I *is the alphabet of* program internal actions.
- Σ_C *is the alphabet of* program call actions.
- Σ_R *is the alphabet of* program return actions.
- R *is the set of* privileges.
- Σ_{PI} *is the alphabet of* internal primitives.
- Σ_{PC} *is the alphabet of* primitive calls.
- Σ_{PR} *is the alphabet of* primitive returns.
- *The program model* V_P *is a* $(\Sigma_I, \Sigma_C, \Sigma_R$*-VPA.*
- *The policy model* V_Q *is a* $(\Sigma_I \times R, \Sigma_C \times R, \Sigma_R \times R)$*-VPA.*
- *The system model* S *is a* $(\Sigma_I \cup \Sigma_{PI}, \Sigma_C \cup \Sigma_{PC}, \Sigma_R \cup \Sigma_{PR}, R)$*-VPT.*

`capweave` takes a program P and policy Q, and constructs a policy-weaving problem $\mathcal{P} = (\Sigma_I, \Sigma_C, \Sigma_R, R, \Sigma_{PI}, \Sigma PI, \Sigma_{PC}, \Sigma PR, P, Q, S)$, defined as follows. Let Labels_P be the set of control labels that occur in P. Let $\Sigma_I = \mathsf{Labels}_P$ (§III-A2), $\Sigma_C = \emptyset$, $\Sigma_R = \emptyset$. Let Opens_P be the set of open-sites that occur in P. Then $\Sigma_{PI} = \{\texttt{cap\_enter}\} \cup \{\texttt{limitfd}(o, R) \mid o \in \mathsf{Opens}_P, R \subseteq \mathsf{Rights}\}$, $\Sigma_{PC} = \{\texttt{fork}\}$, and $\Sigma_{PR} = \{\texttt{join}\}$.

To construct the program-model VPA V_P, `capweave` constructs the control-flow graph of the P, and from the control-flow graph, constructs V_P using standard techniques P [18]. For simplicity, in this section, we have defined the alphabets of program calls Σ_C and program returns Σ_R to be empty. However, our actual implementation of `capweave` defines Σ_C and Σ_R from the calls and returns of P, in order to construct a more precise model of P.

The policy Q is a regular expression over the alphabet $\mathsf{Labels}_P \times \mathsf{proccap}$. To construct the policy VPA V_Q, `capweave` constructs the deterministic finite automaton (DFA) that accepts that language of Q, and from the DFA, directly constructs V_Q [18].

The Capsicum system transducer $S = (\Sigma_I, \Sigma_C, \Sigma_R, \Sigma_O, Q, q_0, Q_F, \tau_I, \tau_C, \tau_R)$ is constructed from the Capsicum semantics as follows:

- The internal alphabet is the set of program labels and internal primitives: $\Sigma_i = \mathsf{Labels}_P \cup \Sigma_{PI}$.
- The call alphabet is the set of call primitives: $\Sigma_C = \Sigma_{PC}$.
- The return alphabet is the set of return primitives: $\Sigma_R = \Sigma_{PR}$.
- The output alphabet is the set of process-capability states: $\Sigma_O = \mathsf{proccap}$.
- The set of states is the set of process-capability states: $Q = \mathsf{proccap}$.
- The initial state is the process state that defines the initial capability state c_i (§III-A3): $q_0 = (\mathsf{True}, \emptyset)$.
- The accepting states are all states: $Q_F = Q$.
- The internal transition function is defined by the interpretation of descriptor instructions and the interpretation of Capsicum primitives: for $q \in Q$, $i \in \Sigma_i$, if $i \equiv d$: `x := open(y)`, then $\tau_I(q, i) = q'$, where $\iota_{\mathtt{d}}[\![i]\!](\emptyset, [q]) = (p', [q'])$.
- The call transition function is defined by the interpretation of the `fork` primitives: for $q \in Q$, $\tau_C(q, \texttt{fork}) = (q, q)$.
- The return transition function is defined by the interpretation of the `join` primitive: for $q, r \in Q$, $\tau_R(q, r, \texttt{join}) = q'$, where $\iota_{\mathtt{p}}[\![\texttt{join}]\!]([q; r]) = [q']$.

`capweave` applies the policy-weaver generator to the policy-weaving problem $\mathcal{P}$ defined by the program model, policy, and Capsicum model. If the weaver generator solves $\mathcal{P}$, then it constructs an instrumentation transducer T_I. `capweave` uses T_I to instrument the program P, as described in App. VII-B.

B. From an Instrumentation Transducer to a Woven Program

From an input program P and policy Q, `capweave` constructs an automata-theoretic weaving problem $\mathcal{P}$ (App. VII-A), and gives $\mathcal{P}$ to the policy weaver generator. If the weaver generator solves $\mathcal{P}$, then it constructs an instrumentation transducer T_I, which directs when an instrumented program should invoke Capsicum primitives. In this section, we describe how `capweave` uses T_I to instrument an IMP program. We then describe how the practical implementation of `capweave` instruments LLVM programs.

`capweave` takes an IMP program P and instrumentation transducer T_I and instruments P by (1) updating a state variable `s`that maintains a state of T_I and (2) checking the value of `s`to determine what primitive to execute in each step of its execution. Let $T_I = (\Sigma_I, \Sigma_C, \Sigma_R, \Sigma_O, Q, q_0, Q_F, \tau_I, \tau_C, \tau_R)$. Then the instrumentation generated for program action a, $\mathcal{I}(a)$, is defined as follows:

$$\begin{aligned}
\mathcal{I}'(a, []) &= \texttt{x := noop} \\
\mathcal{I}'(a, q :: qs) &= \texttt{d := s} - i(q); \\
&\quad\ \texttt{nz := iszero(d)}; \\
&\quad\ \texttt{nz ? } \tau_{\mathtt{I}}(\mathtt{q}, \mathtt{a}); \\
&\quad\ \mathcal{I}(a, qs) \\
\mathcal{I}(a) &= \mathcal{I}'(a, Q)
\end{aligned}$$

REFERENCES

[1] "CVE-2007-3798," http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2007-3798, July 2007.

[2] "CVE-2004-1488," http://cve.mitre.org/cgi-bin/cvename.cgi?name=CAN-2004-1488, Feb 2005.

[3] "CVE-2010-0405," http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-0405, April 2010.

[4] "Vulnerability note VU#381508," http://www.kb.cert.org/vuls/id/381508, July 2011.

[5] "Vulnerability note VU#520827," http://www.kb.cert.org/vuls/id/520827, May 2012.

[6] "CVE-2007-4476," http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2007-4476, Aug 2007.

[7] "GNU Tar and GNU Cpio rmt_read__() function buffer overflow," http://xforce.iss.net/xforce/xfdb/56803, Mar 2010.

[8] 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.

[9] 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.

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

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

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

[13] "cl-capsicum-discuss – Capsicum project discussion list," https://lists.cam.ac.uk/mailman/listinfo/cl-capsicum-discuss, 2012.

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

[15] W. R. Harris, S. Jha, and T. W. Reps, "Secure programming via visibly pushdown safety games," in *CAV*, 2012.

[16] "GNU Wget 1.13.4 manual," http://www.gnu.org/software/wget/manual/wget.html, 2012.

[17] C. Lattner, http://llvm.org/, Nov. 2011.

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

[19] S. Graf and H. Saïdi, "Construction of abstract state graphs with PVS," in *CAV*, 1997.

[20] A. One, "Smashing the stack for fun and profit," *Phrack Magazine*, vol. 49, no. 14, 1998.

[21] "Using Capsicum for sandboxing," http://www.links.org/?p=1242, April 2012.

[22] A. Ehrenfeucht and J. Mycielski, "Positional strategies for mean payoff games," *International Journal of Game Theory*, vol. 8, no. 2, 1979.

[23] P. A. Karger, "Limiting the damage potential of discretionary trojan horses," in *IEEE S&P*, 1987.

[24] A. Bittau, P. Marchenko, M. Handley, and B. Karp, "Wedge: Splitting applications into reduced-privilege compartments," in *NSDI*, 2008.

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

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

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

[28] Ú. Erlingsson and F. B. Schneider, "IRM enforcement of Java stack inspection," in *IEEE S&P*, 2000.

[29] R. N. M. Watson, "A decade of os access-control extensibility," *Commun. ACM*, vol. 56, no. 2, Feb. 2013.

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

[31] 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.

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

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

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

[35] N. Provos, "Privilege separated OpenSSH," http://www.citi.umich.edu/u/provos/ssh/privsep.html, Aug 2003.

[36] D. Kilpatrick, "Privman: A library for partitioning applications," in *USENIX Annual Technical Conference*, 2003.

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

[38] 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.

[39] K. Gudka, R. N. M. Watson, S. Hand, B. Laurie, and A. Madhavapeddy, "Exploring compartmentalization hypothesis with SOAPP," in *AHANS 2012*, 2012.

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

[41] K. W. Hamlen, G. Morrisett, and F. B. Schneider, "Certified in-lined reference monitoring on .NET," in *PLAS*, 2006.

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

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

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

[45] P. Madhusudan, W. Nam, and R. Alur, "Symbolic computational techniques for solving games," *ENTCS 89, 4*, 2003.