Computer Sciences Department

Programming for a Capability System Via Safety Games

William R. Harris
Benjamin Farley
Somesh Jha
Thomas Reps

Technical Report #1705

November 2011

Programming for a Capability System via Safety Games

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

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

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

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

Abstract

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

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

We focus on an algorithm for rewriting programs for Capsicum. However, our algorithm can be naturally generalized to rewrite programs for systems different from Capsicum, such as decentralized information flow control and tagged-memory systems.

1. Introduction

Developing practical but secure programs remains a difficult, important, and open problem. Web servers and VPN clients execute untrusted code, and yet are directly exposed to potentially malicious inputs from a network connection [36]. System utilities such as Norton Antivirus scanner [28], `tcpdump`, the DHCP client `dhclient` [34], and file utilities such as `bzip2`, `gzip`, and `tar` [22, 31, 33] contain or have contained modules with well-known vulnerabilities that allow them to be compromised if they are exposed to an attacker. Once an attacker compromises a vulnerable module in any of the above programs, they can typically perform any action allowed for the user that invoked the program, because the program does not restrict the privileges with which its modules execute.

Traditional operating systems provide to applications only weak primitives for managing their privileges [17, 23, 34, 36]. As a result, if a programmer is to verify that his program is secure, he typically must first verify that the program satisfies very strong properties, such as memory safety. Operating systems that support Mandatory Access Control (MAC) [25, 29, 35] allow a system administrator to specify a policy, and monitor the system calls of each program to ensure that the program does not violate the policy. Because MAC systems only monitor system calls, they cannot adjust the privileges with which a process executes based on events internal to the memory space of the process. However, many practical policies require the privileges of a process to change in this way [18, 34]. *Inline Reference Monitors* [1, 18] can enforce policies defined over internal events, but can only monitor managed code (i.e., code instrumented to be memory-safe).

However, recent work [17, 23, 34, 36] has produced new operating systems that allow programmers to develop programs that execute unmanaged code, but satisfy stronger properties than those that can be specified to a MAC system, and with significantly less effort than fully verifying the program. Such systems extend the set of system calls provided by a traditional operating system with security-specific calls (which henceforth we will call "security primitives"). Throughout a program's execution, it interacts with the system by invoking security primitives to signal key events in its execution, which would not be observed by a MAC system. The developers of such systems have manually rewritten applications to invoke security primitives so that the application satisfies strong security policies, even when the application is composed partly of untrusted code. The application could not satisfy such policies if the operating system did not support such calls (see [17, 23, 34, 36] for detailed discussions of the advantages of such systems over traditional and MAC operating systems).

One example of an operating system with strong security primitives is the capability operating system Capsicum [34]. Capsicum tracks for each process (1) the set of *capabilities* available to the process, where a capability is a file descriptor and an access right for the descriptor, and (2) whether the process has the privilege to grant to itself more capabilities. Capsicum provides to each process a set of system calls that the process uses to limit its capabilities. Trusted code in a program can first communicate with its environment in an unrestrained fashion, and then invoke primitives to limit itself to have only whatever capabilities that it needs for the rest of its execution. Untrusted code then executes with only the limited capabilities defined by the trusted code. Thus, even if the untrusted code is compromised, it will only be able to perform operations allowed by the limited capabilities.

The Capsicum primitives are sufficiently powerful that practical programs can be rewritten to satisfy practical policies when run on Capsicum by making only moderate changes to the program [34]. However, while rewriting a program for Capsicum is not nearly as difficult as verifying the program for a traditional operating system, rewriting is still significantly more difficult than directly specifying a policy to a MAC system. In fact, even Capsicum's own developers have rewritten programs for Capsicum that they tentatively thought were correct, only to discover later that the program was insecure or non-functional, and required further rewriting [34].

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

The algorithm addresses three main challenges that a programmer faces when writing a program for Capsicum. The programmer's first challenge is to clearly define secure behavior of his program, independent of the rewritten program itself. 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, it was impossible for them to define correctness for their rewritten programs, much less determine if the rewritten programs were correct.

The programmer's second challenge is to write his program to be both secure and functional. A programmer can typically rewrite a program for Capsicum so that it is secure by strongly limiting the capabilities of every process. However, the rewritten program may limit its capabilities too strongly at one point of execution, and as a result, may not have the capabilities required to carry out core program functionality later in the execution. The incorrect rewriting reported by the Capsicum developers [34] is an example of this issue.

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

We solve the above challenges by reducing the problem of rewriting a program to finding a winning strategy for an *automata-theoretic safety game* [5]. We represent a program as a language of traces of instructions, and we represent a policy as a language of traces of instructions paired with capabilities that are needed by the program when it executes the instructions. We model Capsicum as an automaton that relates instrumented executions of the program to the resulting instruction-capability traces allowed by Capsicum. From the program, policy, and Capsicum model, our algorithm constructs a game between an "attacker," who "plays" program instructions, and a defender who plays Capsicum primitives. The attacker wins if the sequence of plays is an instrumented program execution that causes a policy violation, and the defender wins otherwise. The problem of simultaneously partitioning and instrumenting the program may be reduced to finding a *modular winning defender strategy* to the game [5].

We efficiently find modular winning strategies for games by a symbolic algorithm that combines a symbolic algorithm for solving *reachability games* [26] with an algorithm for finding modular strategies [5]. Using the symbolic algorithm, we implemented a tool that automatically rewrites programs to correctly run on Capsicum. We applied to the tool to six system utilities that have demonstrated security vulnerabilities, and their policies. The tool was able to rewrite each of the utilities in minutes.

```
tcpdump(pat, netd) {                    dns_resolve() {
 1: bpf = compile_bpf(pat);             D1: ...
 2: config_input(netd);                 D2: return;
 3: limit_fd(dev, { RD });              }
 4: limit_fd(stdout, { WR });
 5: enter_cap_mode();
 6: while(*) {
 7:   rpc_resolve_dns();
 8:   pak = read_packet(netd);
 9:   if match_pattern(pak, bpf)
10:     write_packet(pak, stdout);
   }
}
```

Figure 1. Pseudocode of `tcpdump` instrumented to enforce a policy when run on the Capsicum capability system. The lines of code from the original `tcpdump` program are the ones not underlined. The underlined lines are invocations of Capsicum primitives. In line 7, the call to `resolve_dns` is changed to `rpc_resolve_dns`.

Organization §2 uses the `tcpdump` system utility to describe informally the Capsicum policy-weaving problem and our algorithm to solve it. §3 formally defines the Capsicum policy-weaving problem and an algorithm for solving the problem. §4 presents an experimental evaluation of our tool. §5 discusses related work.

2. Overview

In this section, we discuss in more detail the problem of rewriting programs for Capsicum, and our algorithm for rewriting programs automatically. In particular, we illustrate the rewriting problem and our algorithm on the `tcpdump` system utility. We first describe the core functionality of `tcpdump`, give an informal but practical security policy for `tcpdump`, and recall the Capsicum developer's experience rewriting `tcpdump` to satisfy the policy. In §2.1, we then discuss how `tcpdump`, its policy, and Capsicum may be described formally as automata. In §2.2, we sketch how our algorithm uses the automata-based descriptions to rewrite `tcpdump` automatically to satisfy its policy when run on Capsicum.

`tcpdump` is a popular system utility that allows a user to print incoming network packets that match a particular pattern. Pseudocode of `tcpdump` is provided in Fig. 1; for now, ignore the underlined code. `tcpdump` is given two inputs: a pattern `pat` over packets, and a network input device `netd`. `tcpdump` first compiles `pat` into a Berkeley Packet Filter (BPF) (line 1) `bpf` [27], and configures the network device `netd`. `tcpdump` then enters a loop, in which it performs DNS resolution (line 7), reads a packet `pak` from `netd` (line 8), checks `pak` against the filter `bpf`, and if `pak` matches `bpf`, writes `pak` to standard output.

Historically, `tcpdump` has served as a target for various security attacks, because its packet-matching code is complex and brittle, and thus prone to compromise due to an input crafted by an attacker [34]. Once `tcpdump` is compromised on a traditional operating system, it can read packets, and write their contents to arbitrary locations, even over the network.

Remark 1. *When `tcpdump` executes pattern-matching code (line 9), it should only be able to access its environment (i.e., the file*

system and network) by reading data from `netd`*, or writing data to* `stdout`*.*

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

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

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

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

The resolver only opens files to perform DNS resolution, and cannot be manipulated to leak packets by malicious code injected into the process space of `tcpdump`. Thus, the Capsicum developers determined that it was acceptable to strengthen the policy of Remark 1 with the requirement:

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

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

The Capsicum developers' experience rewriting `tcpdump` for Capsicum illustrates the general challenges in rewriting programs for Capsicum outlined in §1. First, Capsicum only provides system calls for enforcing policies, `limit_fd` and `enter_cm`. Policies that directly state what capabilities the program *may* and *must* have when it executes particular instructions, such as the ones in Remark 1 and Remark 2, are only implicit. Second, it is fairly simple to rewrite `tcpdump` so that it behaves securely, i.e., it does not leak packets that it reads, by inserting calls to a small set of primitives. However, it is non-trivial to instrument `tcpdump` so that it behaves securely and yet still carries out its core functionality, e.g., performs DNS resolution. Finally, to rewrite practical programs for Capsicum, a programmer often must partition his program to execute in multiple process spaces, along with instrumenting some processes to call Capsicum primitives. Even to instrument the compression utility `gzip`, the Capsicum developers had to partition `gzip` to execute in multiple processes [34].

2.1 `tcpdump`, Policies, and Capsicum as Automata

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

A program, such as `tcpdump`, may be viewed as an automaton whose actions are program instructions [32], the set of which we call Instrs. While we cannot in general reason precisely about the language of sequences of instructions executed by a program, we can over-approximate the language by abstracting the program automaton as a finite-state or *visibly-pushdown automaton* (VPA) [2]. In §3.5, we show that a solution for a policy-weaving problem defined by an abstraction of a program is also a solution for a policy-weaving problem defined by the exact program.

A policy can be defined naturally as a pair of automata. One automaton, the *security policy*, describes what capabilities the program may have as it executes. An action for a security-policy automaton is a program instruction in Instrs paired with a capability, the set of which we call Caps. For the `tcpdump` security policy, Caps contains (i) (`netd`, rd) and (`stdout`, wr) where rd and wr are the Capsicum access rights to read and write to a file, respectively, (ii) the meta-capability Env to open file descriptors to resources in the environment, and (iii) a trivial capability Triv that the program always holds (motivated below when we introduce functionality policies).

A security policy describes what capabilities a program is allowed to have as it executes each instruction. A security policy is a language over the actions Instrs $\times$ Caps that only allows a program to execute instructions $i_0, i_1, \ldots, i_n$ with capabilities $c_0, c_1, \ldots, c_n$ at each corresponding instruction if (i_0, c_0), (i_1, c_1), $\ldots$, (i_n, c_n) is in the language of the policy. A security policy can specify that a program may have multiple capabilities as it executes. For example, to specify that when a program executes instructions $i_0, i_1, \ldots, i_n$ it may have capabilities $\{c_0^0, c_0^1\}, \{c_1^0, c_1^1\}, \ldots, \{c_n^0, c_n^1\}$, the security policy accepts the 2^n strings of the form $(i_0, c_0^{k_0}), (i_1, c_1^{k_1}), \ldots, (i_n, c_n^{k_n})$ with each $k_j \in \{0, 1\}$ (such a policy can be encoded with an automaton of size $O(n)$).

The informal security policy of Remark 1 can be represented as a security-policy automaton. Suppose, for clarity, that we weaken the policy of Remark 1 to require only that when `tcpdump` executes `match_pattern`, it must not be able to open file descriptors. This requirement can be formalized with the security policy defined by the following regular expression over the actions Instrs $\times$ Caps, where (Instrs $\times$ Caps)$\backslash(9, \text{Env})$ denotes any instruction paired with any capability, except for the instruction at line 9 of Fig. 1 with the capability Env (in all example languages, we refer to each

instruction in Instrs by its line number in Fig. 1):

$$((\mathsf{Instrs} \times \mathsf{Caps}) \backslash (9, \mathsf{Env}))^* \quad (1)$$

A functionality policy is a second policy automaton defined over the same actions as the security policy with which it is paired. However, unlike a security policy, which describes what capabilities a program *may* have as it executes, the functionality policy describes what capabilities the program *must* have as it executes. If a functionality policy accepts the string $(i_0, c_0), (i_1, c_1), \ldots, (i_n, c_n)$ and the program executes the instructions $i_0, i_1, \ldots, i_n$, then the program must have the capabilities $c_0, c_1, \ldots, c_n$ at each corresponding instruction. A functionality policy can specify that a program must have multiple capabilities as it executes. For example, to specify that when a program executes instructions $i_0, i_1, \ldots, i_n$ it must have capabilities $\{c_0^0, c_0^1\}, \{c_1^0, c_1^1\}, \ldots, \{c_n^0, c_n^1\}$, the functionality policy accepts the 2^n strings of the form $(i_0, c_0^{k_0}), (i_1, c_1^{k_1}), \ldots, (i_n, c_n^{k_n})$ with each $k_j \in \{0, 1\}$.

The informal functionality policy of Remark 2 may be represented as a functionality-policy automaton. Suppose, for clarity, that we weaken the policy of Remark 2 to require that after `tcpdump` calls `resolve_dns` (line 7) but before it returns from `resolve_dns` (line 8), it must be able to open file descriptors. This requirement can be formalized with the functionality policy defined by the following regular expression over the actions Instrs × Caps (where $((\mathsf{Instrs} \backslash \{8\}) \times \mathsf{Caps})$ denotes any instruction but 8, paired with any capability):

$$((\mathsf{Instrs} \times \mathsf{Triv})^* (7, \mathsf{Triv}) ((\mathsf{Instrs} \backslash \{8\}) \times \mathsf{Caps}))^* \quad (2)$$

By pairing some instructions with Triv in (2), we place no requirements on what privileges `tcpdump` has when it executes such instructions.

We have described Capsicum as an operating system that monitors a sequence of instructions and calls to Capsicum primitives executed by a program, and decides what capabilities the program has as it executes each instruction. We can model how Capsicum monitors a program as a formal language, accepted by an automaton. We may view Capsicum as defining a set of primitive operations Prims, that represent primitive operations that can be called by an instrumented program. For `tcpdump`, Prims contains a primitive `limit_fd(d, R)` for every descriptor `d` opened by `tcpdump` and every subset `R` of access rights that Capsicum defines, and a primitive `enter_cm`, both of which were informally described above. Prims also contains a primitive rpc, described below, that causes a call to be treated as an RPC, and a primitive `noop`, which does not affect the capabilities of the program. The Capsicum model defines a language over the actions $(\mathsf{Instrs} \times \mathsf{Caps}) \times \mathsf{Prims}$, where a string $((i_0, c_0), p_0), ((i_1, c_1), p_1), \ldots, ((i_n, c_n), p_n)$ is in the language if and only if Capsicum allows the program to execute each instruction i_j with capability c_j and then immediately invoke primitive p_j. For example, the language of the model of the Capsicum monitor for `tcpdump` accepts the string

$$((1, \mathsf{Env}), \texttt{noop}), ((2, \mathsf{Env}), \texttt{noop}), ((6, \mathsf{Env}), \texttt{noop})$$

but does not accept the string

$$((1, \mathsf{Env}), \texttt{noop}), ((2, \mathsf{Env}), \texttt{enter\_cm}), ((6, \mathsf{Env}), \texttt{noop});$$

after `tcpdump` calls `enter_cm`, it cannot hold the capability Env.

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

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

is not in the language of the Capsicum model, because Capsicum models the action $((7, \mathsf{Triv}), \texttt{noop})$ as a normal function call to `dns_resolve` (line 7 in Fig. 1), and thus does not accept the last action $((\texttt{D1}, \mathsf{Env}), \texttt{noop})$. However, the string

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

is in the language of Capsicum, because Capsicum models the action $((7, \mathsf{Triv}), \mathsf{rpc})$ as an RPC to `dns_resolve`, and thus accepts the action $((\texttt{D1}, \mathsf{Env}), \texttt{noop})$.

The Capsicum model is represented as a VPA, which uses a stack to model Capsicum's responses to RPC calls and returns. We give a detailed description of Capsicum as a VPA in §3.2.

2.2 Instrumenting `tcpdump` via a Safety Game

We have now given an intuitive explanation of how all components of the Capsicum policy-weaving problem can be represented as automata that define languages over various sets of actions: a program defines a language over Instrs, policies define languages over Instrs × Caps, and Capsicum defines a language over (Instrs × Caps) × Prims. Our weaving algorithm takes such automata as inputs and instruments and partitions the program to satisfy the policy.

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

Fig. 2 gives the game constructed from an automaton model of `tcpdump` based on its control-flow graph, security policy (1), functionality policy (2), and our automaton model of Capsicum. To simplify the presentation, the game in Fig. 2 has been slightly simplified from the true game for `tcpdump` and policies (1) and (2). In particular, transitions for some instruction sequences have been collapsed into a single transition (e.g., the transition labeled with instructions "1, 2"), unnecessary defender transitions have been removed (e.g., there is no defender transition after "8, 9"), and only the primitives `enter_cm` (abbreviated as `ecm`) and `noop` are considered, because these are the only primitives relevant to policies (1) and (2).

In §3, we describe how our algorithm takes the program, policy, and Capsicum automata and constructs the game in Fig. 2. For now, it is enough to observe that strings that cause the game to transition to an accepting state correspond to instrumented executions that cause the program to violate a policy. For example, the string $1, 2, 6, 7, \texttt{noop}, D1, D2, 8, 9$ corresponds to an execution in which `tcpdump` executes `9: match_pattern` without ever calling `enter_cm`, and thus violates security policy (1). As another example, the string $1, 2, \texttt{ecm}, 6, 7, \texttt{noop}, D1$ corresponds to an execution

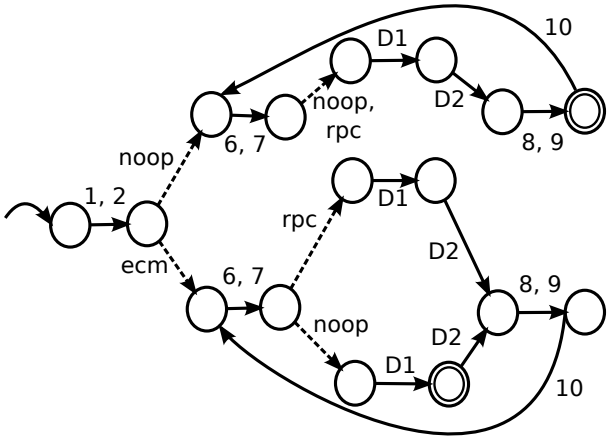

Figure 2. Safety game in which all winning plays for the attacker are instrumentations of `tcpdump` that violate policy (1) or (2). Attacker (program) choices are represented as solid edges labeled with instruction line numbers from Fig. 1; defender choices are represented as dashed edges labeled with Capsicum primitives; and accepting states are represented as double circles. The primitive `enter_cm` is abbreviated as `ecm`.

in which `tcpdump` calls `enter_cm` and then executes a non-RPC call to the DNS resolver, and thus executes the DNS resolver without the ability to open files, violating the functionality policy (2).

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

3. Policy Weaving for Capsicum

In §2, we sketched how the problem of rewriting `tcpdump` to satisfy simple policies on Capsicum may be reduced to finding a winning strategy for a safety game. In this section, we describe our reduction in detail. In §3.1 we review visibly pushdown automata, and safety games defined by them. Such games are the target of our reduction. In §3.2, we define the Capsicum policy-weaving problem in automata-theoretic terms. In §3.3, we present a symbolic algorithm for solving the automata-theoretic weaving problem. In §3.4, we discuss practical aspects of the algorithm, and its general implications.

3.1 Preliminaries

Visibly pushdown automata (VPA) [2] are a class of a stack-based machines that can use their stack to store unbounded information, but are restricted in how they can use their stack to transition.

Definition 1. *A deterministic* visibly pushdown automaton $V = (S, \iota, A, \Gamma, \Sigma_i, \Sigma_c, \Sigma_r, \tau_i, \tau_c, \tau_r)$ *is a tuple of:*

- *A finite* set of states S.
- *An* initial state $\iota \in S$.
- *A set of* accepting states $A \subseteq S$.
- *A finite* stack alphabet Γ.
- *A finite set of* internal actions Σ_i.
- *A finite set of* call actions Σ_c.
- *A finite set of* return actions Σ_r.
- *A* internal transition function $\tau_i : S \times \Sigma_i \to S$.
- *A* call transition function $\tau_c : S \times \Sigma_c \to S \times \Gamma$.
- *A* return transition function $\tau_r : S \times \Gamma \times \Sigma_r \to S$.

A VPA defines a language of strings over the actions $\widetilde{\Sigma} = \Sigma_i \cup \Sigma_c \cup \Sigma_r$, similar to how a pushdown automaton defines a language over its actions. A *configuration* of a VPA is a state in S paired with a stack of symbols in Γ. A VPA reads a string in $\widetilde{\Sigma}^*$ by starting in an initial configuration of ι paired with the empty stack. When the VPA reads an internal action $a \in \Sigma_i$ from state q, it updates its state to the $\tau_i(q, a)$, and leaves its stack unchanged. When the VPA reads a call action $c \in \Sigma_c$ from state q, and $\tau_c(q, c) = (q', \gamma)$, it updates its state to q', and pushes γ on the top of its stack. When the VPA reads a return action $r \in \Sigma_r$ from state q with stack symbol γ on the stop of its stack, it pops γ from its stack and updates its state to $\tau_r(q, \gamma, r)$. The VPA accepts a string if and only if after reading the string, it has transitioned to a state in A.

If when a reading a string s, a VPA begins reading a substring s' from a configuration with a stack T, finishes reading s' in another configuration with stack T, and the stack of the VPA has been an extension of T throughout, then we say that s' is a *matched substring* of s.

VPA can be extended to define turn-based safety games between two players: an attacker, and a defender.

Definition 2. *A* VPA turn-based safety game *is a tuple* $(S^a, S^d, \iota, A, \Gamma, \Sigma_i^a, \Sigma_i^d, \Sigma_c^a, \Sigma_c^d, \Sigma_r^a, \Sigma_r^d, \tau_i^a, \tau_i^d, \tau_c^a, \tau_c^d, \tau_r^a, \tau_r^d)$*, where*

- *The set of attacker states S^a and defender states S^d are non-overlapping.*
- *The initial state is $\iota \in S^a$.*
- *The set of accepting states are $A \subseteq S^a$.*
- *The attacker internal transition function is $\tau^a : S^a \times \Sigma_i^a \to S_d$.*
- *The attacker call transition function is $\tau_c^a : S^a \times \Sigma_i^a \to S_d \times \Gamma$.*
- *The attacker return transition function is $\tau_r^a : S^a \times \Gamma \times \Sigma_r^a \to S_d$.*

The defender transition functions τ_i^d, τ_c^d, and τ_r^d are defined analogously to the attacker transition functions, with each mapping a defender pre-state and defender action to an attacker post-state.

The attacker and defender play the game in turn, with each choosing on their turn an action on which the game transitions. The attacker wins if the string of all choices is a string accepted by the game automaton, and the defender wins otherwise.

Definition 3. *A* defender strategy $\sigma : (\widetilde{\Sigma}^a)^* \to \widetilde{\Sigma}^d$ *takes a string of attacker actions, and chooses a defender action. A* winning defender strategy *is a strategy such that if the defender always chooses actions according to the strategy, the resulting string will never be a winning string for the attacker. Formally, σ is a winning defender strategy for the game* G *if and only if for each finite string $a_0, a_1, \ldots, a_n \in \widetilde{\Sigma}^{a*}$, the string $a_0, \sigma([a_0]), a_1, \sigma([a_0, a_1]), \ldots, a_n, \sigma([a_0, a_1, \ldots, a_n])$ is not accepted by* G.

3.2 The Capsicum Policy-Weaving Problem

In this section, we define the problem of weaving programs for Capsicum in automata-theoretic terms.

3.2.1 Capsicum as a Visibly Pushdown Automaton

We can model the Capsicum monitor of a given program as a VPA. Let a program $P = (V, \mathsf{Descs})$ be:

- A VPA $V = (S, \iota, A, \Gamma, \mathsf{Intras}, \mathsf{Calls}, \mathsf{Rets}, \tau_i, \tau_c, \tau_r)$ whose internal actions Intras are programs intra-procedural instructions, call actions Calls are function calls Calls, and return actions Rets are function returns. Let the set of all program instructions by $\mathsf{Instrs} = \mathsf{Intras} \cup \mathsf{Calls} \cup \mathsf{Rets}$.
- A finite set Descs of resource descriptors.

We model the Capsicum monitor of P as a VPA C_P whose language describes what capabilities Capsicum allows P to have as it executes (all components of C_P are defined with respect to P, and thus we omit an explicit P subscript for the rest of this section). Here, we model a monitor only for a program that executes sequentially, only communicating with another process via a blocking RPC. However, the monitor model can naturally be generalized for programs that execute over multiple processes in parallel.

The actions of C are constructed from the following domains. Let Rights be a fixed set of access rights defined by Capsicum, and let Env be the meta-capability to open files. Then the space of all capabilities of P is $\mathsf{Caps} = (\mathsf{Descs} \times \mathsf{Rights}) \cup \{\mathsf{Env}\}$. Let the space of primitives that may be invoked by P be $\mathsf{Prims} = \{\texttt{enter\_cm}, \texttt{noop}, \mathsf{rpc}\} \cup \{\texttt{limit\_fd}(d, R) | \ \mathsf{desc} \in \mathsf{Descs}, R \subseteq \mathsf{Rights}\}$ (see §2 for intuitive definitions of the primitives).

The language of the Capsicum model, $L(\mathsf{C}) \subseteq ((\mathsf{Instrs} \times \mathsf{Caps}) \times \mathsf{Prims})$ is such that a string $((a_0, c_0), p_0), \ldots, ((a_n, c_n), p_n) \in L(\mathsf{C})$ if and only if when P executes each a_i followed immediately by each p_i, then it holds capability c_i when it executes a_i. $\mathsf{C}_P = (\widehat{S}, \widehat{\iota}, \widehat{A}, \widehat{\Gamma}, \widehat{\Sigma_i}, \widehat{\Sigma_c}, \widehat{\Sigma_r}, \widehat{\tau_i}, \widehat{\tau_c}, \widehat{\tau_r})$, with

- $\widehat{S}$: each state is a map from each descriptor in Descs to the set of rights held by the program, paired with a Boolean flag that denotes whether the program is in capability mode (see §2), or the stuck state: $S = ((\mathsf{Descs} \to \mathsf{Rights}) \times \mathbb{B}) \cup \{\mathsf{CapStuck}\}$.
- $\widehat{\iota}$: in the initial state, the program has all rights for every descriptor, and is not in capability mode.
- $\widehat{A}$: all states but the stuck state are accepting states: $\widehat{A} = S \backslash \{\mathsf{CapStuck}\}$.
- $\widehat{\Gamma} = S \cup \{\mathsf{intracall}\}$. See $\widehat{\tau_c}$ and $\widehat{\tau_r}$.
- $\widehat{\Sigma_i}$: each internal action of C is a program intra-procedural instruction paired with a capability, paired with one of the primitives: $\widehat{\Sigma_i} = (\mathsf{Intras} \times \mathsf{Caps}) \times \mathsf{Prims}$.
- $\widehat{\Sigma_c}$: each call action of C is a call action of P paired with a capability, paired with the $\texttt{noop}$ or rpc primitive: $\widehat{\Sigma_c} = (\mathsf{Calls} \times \mathsf{Caps}) \times \{\texttt{noop}, \mathsf{rpc}\}$.
- $\widehat{\Sigma_r}$: each return action of C is a return action of P paired with a capability, paired with the $\texttt{noop}$ primitive: $\widehat{\Sigma_r} = (\mathsf{Rets} \times \mathsf{Caps}) \times \{\texttt{noop}\}$ (we limit instrumentations to respond to returns only with $\texttt{noop}$ only for simplicity).
- $\widehat{\tau_i}(q, ((\mathsf{ins}, \mathsf{cap}), p))$: if $\mathsf{cap} = \mathsf{Env}$ and P is not in capability mode, then Capsicum transitions to $\mathsf{CapStuck}$. Otherwise, if $p = \texttt{enter\_cm}$, then C transitions to a state with the same rights, but which is in capability mode. If $p = \texttt{limit\_fd}(\mathsf{desc}, R)$, then C transitions to a state in which the rights of desc are limited to R. Otherwise, $p = \texttt{noop}$, and C transitions to q.
- $\widehat{\tau_c}(q, ((\mathsf{call}, \mathsf{cap}), p))$: if the call is not an RPC call, then the Capsicum-state does not change; otherwise, C initializes the called process with all capabilities, and stores the state of the calling process on its stack. In other words, $\widehat{\tau_c}(q, ((\mathsf{call}, \mathsf{cap}), \texttt{noop})) = (q, \mathsf{intracall})$, and $\widehat{\tau_c}(q, (\mathsf{call}, \mathsf{cap}), \mathsf{rpc}) = (\widehat{\iota}, q)$.
- $\widehat{\tau_r}$: if the program returns from an intra-process call, then the Capsicum-state does not change, but if the return is from an RPC, then Capsicum reverts to its state before the matching RPC was made. In other words, if $\gamma = \mathsf{intracall}$, then $\widehat{\tau_r}(q, \gamma, \mathsf{ret}) = q$, otherwise $\widehat{\tau_r}(q, \gamma, \mathsf{ret}) = \gamma$.

Let $(\mathsf{Instrs} \times \mathsf{Prims})^*$, the set of program actions paired with primitives, be the set of *instrumented executions*, and let $(\mathsf{Instrs} \times \mathsf{Caps})^*$, the set of program actions paired with capabilities, be the set of *capability-traces*. C defines a relation from each instrumented execution of P to the set of capability traces that it induces. Let this relation be $\mathsf{CapTraces}_P \subseteq (\mathsf{Instrs} \times \mathsf{Prims})^* \times (\mathsf{Instrs} \times \mathsf{Caps})^*$, where $((\mathsf{ins}_0, p_0), \ldots, (\mathsf{ins}_n, p_n)), ((\mathsf{ins}_0, \mathsf{cap}_0), \ldots, (\mathsf{ins}_n, \mathsf{cap}_n)) \in \mathsf{CapTraces}$ if and only if $((\mathsf{ins}_0, \mathsf{cap}_0), p_0), \ldots, ((\mathsf{ins}_n, \mathsf{cap}_n), p_n) \in \mathsf{C}$.

3.2.2 The Automata-Theoretic Weaving Problem

Using the VPA model of Capsicum defined in §3.2.1, we define the *Capsicum policy-weaving problem* as follows. Let program P be a VPA defined over actions Instrs, and let S and F be a security and functionality policy, both represented as VPAs over the actions $\mathsf{Instrs} \times \mathsf{Caps}$. Let an instrumentation function be some function $I : \mathsf{Instrs}^* \to (\mathsf{Instrs} \times \mathsf{Prims})^*$ such that for each sequence of program instructions $\mathsf{ins}_0, \ldots, \mathsf{ins}_n$, $I([\mathsf{ins}_0, \ldots, \mathsf{ins}_n]) = [(\mathsf{ins}_0, p_0), \ldots, (\mathsf{ins}_1, p_n)]$ for some Capsicum primitives $p_0, \ldots, p_n$. The Capsicum policy-weaving problem $\mathsf{Prob}(P, S, F)$ is to find an instrumentation function I that satisfies the following conditions.

Secure: P instrumented by I has only the capabilities allowed by S in each execution. For a policy $\mathsf{Pol} \subseteq (\mathsf{Instrs} \times \mathsf{Caps})^*$, let $\mathsf{Pol}|_P$ be all strings $(\mathsf{ins}_0, \mathsf{cap}_0), \ldots, (\mathsf{ins}_n, \mathsf{cap}_n)$ in Pol such that $\mathsf{ins}_0, \ldots, \mathsf{ins}_n \in P$. Then

$$\mathsf{CapTraces}(I(P)) \subseteq S|_P$$

Functional: P instrumented by I has all of the capabilities required by F for executions of P:

$$F|_P \subseteq \mathsf{CapTraces}(I(P))$$

RPC-modular: A function invoked via an RPC may be invoked by arbitrary, injected code. As a result, each RPC function cannot trust any information supposedly passed by the instrmentation of the caller. Thus, I must be *modular*, in the sense that it chooses primitives after an RPC independently of instructions and primitives chosen before the RPC [5].

Definition 4. *For $L \subseteq (\mathsf{Instrs} \times \mathsf{Prims})^*$, L is (a, p)-*modular *if and only if the following holds. Let $s \in L$ contain as a subsequence a matched string $(a, p), (a_0, p_0^0), (a_1, p_1^0), \ldots, (r, p_n^0)$, and let $s' \in L$ contain a matched a string whose actions are constructed from identical Instrs-components: $(a, p), (a_0, p_0^1), (a_1, p_1^1), \ldots, (r, p_n^1)$. Then the corresponding Prims-component of each action must be identical: for each i, $p_i^0 = p_i^1$.*

For each $\mathsf{call} \in \mathsf{Calls}$, the range of I must be $(\mathsf{call}, \mathsf{rpc})$-modular.

3.3 From a Weaving Problem to a Safety Game

We now give an algorithm that solves a Capsicum policy weaving problem $\mathcal{P}$. From $\mathcal{P} = \mathsf{Prob}(P, S, F)$, the algorithm first constructs a VPA $\mathsf{Vio}_{\mathcal{P}}$ that accepts all instrumented executions of P that violate S or F (§3.3.1). From $\mathsf{Vio}_{\mathcal{P}}$, the algorithm constructs a VPA safety game $\mathsf{G}_{\mathcal{P}}$ for which a winning defender strategy corresponds to a solution to $\mathcal{P}$ (§3.3.2). The algorithm finds a winning

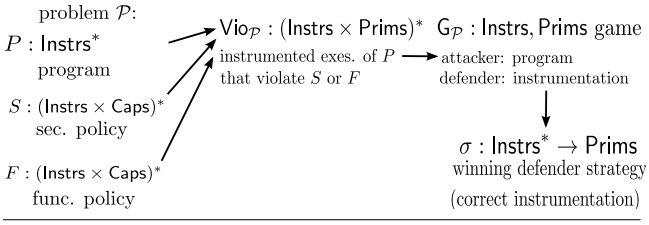

Figure 3. Outline of reduction presented in §3.3.1.

defender strategy to $\mathsf{G}_\mathcal{P}$ by reducing the search for a strategy to finding a model of a Satisfiability Modulo Theories (SMT) formula $\varphi_\mathcal{P}$, and applying an SMT solver (§3.3.3). In other words, the algorithm performs a reduction $\mathcal{P} \Rightarrow \mathsf{Vio}_\mathcal{P} \Rightarrow \mathsf{G}_\mathcal{P} \Rightarrow \varphi_\mathcal{P}$. Fig. 3 gives a visual summary of the reduction.

3.3.1 From Weaving Problem to Violating Executions

For $\mathcal{P} = \mathsf{Prob}(P, S, F)$ the algorithm constructs $\mathsf{Vio}_\mathcal{P}$ as the intersection of two languages. The first is the language of all possible instrumented executions of the P. Let this language be $P' \subseteq (\mathsf{Instrs} \times \mathsf{Prims})^*$, the set of all strings $(\mathsf{ins}_0, p_0), \ldots, (\mathsf{ins}_n, p_n)$ such that $\mathsf{ins}_0, \ldots, \mathsf{ins}_n \in P$.

The second language is the language of all instrumented executions that violate S or F. To define this language, we first define operators that allow us to translate languages between different actions. For a policy $\mathsf{Pol} \subseteq (\mathsf{Instrs} \times \mathsf{Caps})^*$, let $\mathsf{lift}(\mathsf{Pol}) \subseteq ((\mathsf{Instrs} \times \mathsf{Caps}) \times \mathsf{Prims})^*$ be the set of all strings $((\mathsf{ins}_0, \mathsf{cap}_0), p_0), \ldots, ((\mathsf{ins}_n, \mathsf{cap}_n), p_n)$ such that $(\mathsf{ins}_0, \mathsf{cap}_0)$, $\ldots$, $(\mathsf{ins}_n, \mathsf{cap}_n) \in \mathsf{Pol}$, for some primitives $p_0, \ldots, p_n$. For a language $L \subseteq ((\mathsf{Instrs} \times \mathsf{Caps}) \times \mathsf{Prims})^*$, let the projection $\pi(L) \subseteq (\mathsf{Instrs} \times \mathsf{Prims})^*$ be the language of all instrumented executions (ins_0, p_0), $\ldots$, (ins_n, p_n) such that $((\mathsf{ins}_0, \mathsf{cap}_0), p_0)$, $\ldots, ((\mathsf{ins}_n, \mathsf{cap}_n), p_n) \in L$ for some $\mathsf{cap}_0, \ldots, \mathsf{cap}_n$.

For C_P the Capsicum monitor of P defined in §3.2.1, $\pi(\mathsf{C}_P \cap \overline{\mathsf{lift}(S)})$ is the language of all instrumented executions that induce Capsicum to allow a capability trace not allowed by S. $\pi(\overline{\mathsf{C}_P} \cap \mathsf{lift}(F))$ is the language of all instrumented executions that induce Capsicum to disallow a capability trace accepted by F. Thus the language of all instrumented executions of P that violate S or F is:

$$L_\mathcal{P} = P' \cap (\pi(\mathsf{C}_P \cap \overline{\mathsf{lift}(S)}) \cup \pi(\overline{\mathsf{C}_P} \cap \mathsf{lift}(F))) \qquad (3)$$

Lemma 1. *Let $\mathcal{P} = \mathsf{Prob}(P, S, F)$ be a weaving problem, and let $L_\mathcal{P}$ be defined as in* (3)*. An instrumentation function $I : \mathsf{Instrs}^* \to (\mathsf{Instrs} \times \mathsf{Prims})^*$ is secure and functional if and only if for each execution $s \in P$, $I(s) \notin L_\mathcal{P}$.*

Proof. First, suppose that I is not secure. This holds if and only if there is some program execution $r = \mathsf{ins}_0, \ldots, \mathsf{ins}_n \in P$ for which $\mathsf{CapTraces}(I(r)) = (\mathsf{ins}_0, \mathsf{cap}_0), \ldots, (\mathsf{ins}_n, \mathsf{cap}_n) \notin S|_P$. Let $I(r) = (\mathsf{ins}_0, p_0), \ldots, (\mathsf{ins}_n, p_n)$. Let $s = ((\mathsf{ins}_0, \mathsf{cap}_0), p_0)$, $\ldots$, $((\mathsf{ins}_n, \mathsf{cap}_n), p_n)$. By the definition of P', $I(r) \in P'$. By the definition of $\mathsf{CapTraces}$ and C_P, $s \in \mathsf{C}_P$. By the definition of $\mathsf{CapTraces}$ and $\mathsf{lift}(S)$, $s \in \overline{\mathsf{lift}(S)}$. Thus by the definition of π, $I(r) = \pi(s) \in \pi(\mathsf{C}_P \cap \overline{(\mathsf{lift}S)}$. Thus I is not secure if and only if there is some program execution r for which $I(r) \in P' \cap \pi(\mathsf{C}_P \cap \overline{\mathsf{lift}(S)}$.

Now suppose that I is not functional. This holds if and only if there is some program execution $r = \mathsf{ins}_0, \ldots, \mathsf{ins}_n \in P$ and capability trace $s = (\mathsf{ins}_0, \mathsf{cap}_0), \ldots, (\mathsf{ins}_n, \mathsf{cap}_n) \in P$ for which $s \in F|_P$ and $s \notin \mathsf{CapTraces}(I(r))$. Let $I(r) = (\mathsf{ins}_0, p_0), \ldots, (\mathsf{ins}_n, p_n)$. $I(r) \in P'$, by the definition of P'. Let $t = ((\mathsf{ins}_0, \mathsf{cap}_0), p_0), \ldots, ((\mathsf{ins}_n, \mathsf{cap}_n), p_n)$. Then $t \notin \mathsf{C}_P$, for otherwise, $s \in \mathsf{CapTraces}(I(r))$ by the definition of $\mathsf{CapTraces}$. $t \in \mathsf{lift}(F)$ by the definition of $\mathsf{lift}(F)$. Thus $t \in (\overline{\mathsf{CapTraces}(I(r))} \cap \mathsf{lift}(F))$, which holds if and only if $I(s) = \pi(t) \in \pi(\overline{\mathsf{C}_P} \cap \mathsf{lift}(F))$. Thus I is not functional if and only if there is some program execution r for which $I(r) \in P' \cap \pi(\overline{\mathsf{C}_P} \cap \mathsf{lift}(F))$.

Thus an instrumentation is not secure or not functional if and only if it there is some program execution r such that

$$\begin{aligned} I(r) &\in (P' \cap \pi(\mathsf{C}_P \cap \overline{\mathsf{lift}(S)})) \cup (P' \cap \pi(\overline{\mathsf{C}_P} \cap \mathsf{lift}(F))) \\ &= P \cap (\pi(\mathsf{C}_P \cap \overline{\mathsf{lift}(S)}) \cup \pi(\overline{\mathsf{C}_P} \cap \mathsf{lift}(F))) \\ &= L_\mathcal{P} \end{aligned}$$

□

From the VPA representations of P, S, F, and C_P, we can apply (3) to construct a VPA $\mathsf{Vio}_\mathcal{P}$ that accepts $L_\mathcal{P}$, because VPAs are closed under intersection, complement, lift, and π.

3.3.2 From Violating Executions to a Safety Game

From $\mathsf{Vio}_\mathcal{P} = (S, \iota, A, \Gamma, \mathsf{Intras} \times \mathsf{Prims}, \mathsf{Calls} \times \mathsf{Prims}, \mathsf{Rets} \times \mathsf{Prims}, \tau_i, \tau_c, \tau_r)$, the algorithm constructs a VPA safety game $\mathsf{G}_\mathcal{P} = (\widehat{S^a}, \widehat{S^d}, \widehat{\iota}, \widehat{A}, \widehat{\Gamma}, \widehat{\Sigma_i^a}, \widehat{\Sigma_i^d}, \widehat{\Sigma_c^a}, \widehat{\Sigma_c^r}, \widehat{\Sigma_r^a}, \widehat{\Sigma_r^d}, \widehat{\tau_i^a}, \widehat{\tau_i^d}, \widehat{\tau_c^a}, \widehat{\tau_c^d}, \widehat{\tau_r^a}, \widehat{\tau_r^d})$ such that the winning plays of the attacker for $\mathsf{G}_\mathcal{P}$ directly correspond to the instrumented executions accepted by $\mathsf{Vio}_\mathcal{P}$. Intuitively, $\mathsf{G}_\mathcal{P}$ is constructed by serializing each internal transition $\tau_i(q, (\mathsf{ins}, p)) = q'$ of $\mathsf{Vio}_\mathcal{P}$ into an attacker transition $\tau_i^a(q, \mathsf{ins}) = q_0$ and subsequent defender transition $\tau_i^d(q_0, p) = q'$ in $\mathsf{G}_\mathcal{P}$ for some defender state q_0. Call and return transitions of $\mathsf{Vio}_\mathcal{P}$ are serialized analogously.

$\mathsf{G}_\mathcal{P}$ is constructed from $\mathsf{Vio}_\mathcal{P}$ as follows:

- The attacker states are the states of $\mathsf{Vio}_\mathcal{P}$: $\widehat{S^a} = S$.
- Each defender state stores the last attacker state, and the last internal, call, or return action chosen by the attacker: $\widehat{S^d} = S \times (\Sigma_i \cup \Sigma_c \cup (\Gamma \times \Sigma_r))$
- The initial state is the initial state of $\mathsf{Vio}_\mathcal{P}$: $\widehat{\iota} = \iota$
- The accepting states are the accepting states of $\mathsf{Vio}_\mathcal{P}$: $\widehat{A} = A$.
- The stack alphabet is the stack alphabet of $\mathsf{Vio}_\mathcal{P}$ along with "placeholder" state μ: $\widehat{\Gamma} = \Gamma \cup \{\mu\}$.
- The attacker internal actions are intraprocedural program instructions: $\widehat{\Sigma_i^a} = \mathsf{Intras}$.
- The defender internal actions are Capsicum primitives: $\widehat{\Sigma_i^d} = \mathsf{Prims}$.
- The attacker call actions are program calls: $\widehat{\Sigma_c^a} = \mathsf{Calls}$.
- The defender call actions are to select a call as RPC or leave it as an intraprocess call: $\widehat{\Sigma_c^d} = \{\mathtt{rpc}, \mathtt{noop}\}$.
- The attacker return actions are program returns: $\widehat{\Sigma_r^a} = \mathsf{Rets}$.
- The defender return actions are Capsicum primitives are `noop`: $\widehat{\Sigma_r^d} = \{\mathtt{noop}\}$.
- When the attacker chooses an intra-process instruction, the game transitions to a state that stores the choice: $\widehat{\tau_i^a}(q, a) = (q, a)$.
- When the defender chooses a Capsicum primitive p, the game transitions to a state of $\mathsf{Vio}_\mathcal{P}$ reached by p with the last intra-process instruction chosen by the attacker: $\widehat{\tau_i^d}((q, a), p) = \tau_i(q, (i, p))$
- The attacker and defender call and transition functions are defined analogously to the internal transition functions:

The strings accepted by $\mathsf{Vio}_{\mathcal{P}}$ correspond to the winning attacker plays of $\mathsf{G}_{\mathcal{P}}$.

Lemma 2. *$(\mathsf{ins}_0, p_0), \ldots, (\mathsf{ins}_n, p_n) \in \mathsf{Vio}_{\mathcal{P}}$ if and only if $\mathsf{ins}_0, p_0, \ldots, \mathsf{ins}_n, p_n$ is a winning attacker play of $\mathsf{G}_{\mathcal{P}}$.*

Proof. Follows directly from the definitions of $\widehat{\tau_i^a}$, $\widehat{\tau_c^a}$, $\widehat{\tau_r^a}$, $\widehat{\tau_i^d}$, $\widehat{\tau_c^d}$, and $\widehat{\tau_r^d}$. □

3.3.3 Finding a Winning Defender Strategy Symbolically

From Lem. 1 and Lem. 2, it holds that if $\sigma : \mathsf{Instrs}^* \to \mathsf{Prims}$ is a winning defender strategy to $\mathsf{G}_{\mathcal{P}}$, then from σ we can construct an instrumentation function that is secure and functional. In particular, from σ, we can construct the instrumentation function $I_\sigma : \mathsf{Instrs}^* \to (\mathsf{Instrs} \times \mathsf{Prims})^*$ where for the empty string ϵ and concatenation operator ".", $I_\sigma(\epsilon) = \epsilon$, and for each $s \in \mathsf{Instrs}^*$ and $a \in \mathsf{Instrs}$, $I(s.a) = I(s).(s, \sigma(s.a))$.

However, finding a winning defender strategy σ such that I_σ is modular (i.e., a modular strategy) is exactly as hard as finding a modular strategy for a *recursive game graph*, which is NP-complete [5]. To cope with this high worst-case complexity, we reduce the problem of finding a modular winning defender strategy for $\mathsf{G}_{\mathcal{P}}$ to finding a model of a Satisfiability Modulo Theories (SMT) formula. Modern SMT solvers can often efficiently find models for large formulas [14, 15]. Such solvers can find models in minutes for formulas derived from practical programs and policies (see §4).

From $\mathsf{G}_{\mathcal{P}}$, the algorithm constructs an SMT formula $\varphi_{\mathcal{P}}$ such that a model of $\varphi_{\mathcal{P}}$ corresponds to a winning modular defender strategy to $\mathsf{G}_{\mathcal{P}}$. A model m of $\varphi_{\mathcal{P}}$ defines a strategy as a restriction G_m of $\mathsf{G}_{\mathcal{P}}$ to a particular set of states and transitions. When G_m reads an action chosen by an attacker, it transitions to one of its defender states according to its transition function. By construction, each defender state of G_m is assigned a unique defender action. G_m outputs the defender action assigned to its current defender state, transitions from the state accordingly, and then reads the next action chosen by the attacker. Let the strategy defined in this way from a model m be S_m. The approach is analogous to the one given in [26] for symbolically searching for winning attacker strategies.

To construct $\varphi_{\mathcal{P}}$, we assume that $\mathsf{G}_{\mathcal{P}}$ is represented symbolically. In other words, the stack alphabet and attacker and defender state sets, internal actions, call actions, and return actions are all represented as domains in an SMT theory. The formula $A(x)$ has one free variable x, and is true exactly when x is an accepting state of $\mathsf{G}_{\mathcal{P}}$. The attacker and defender internal, call, and transition functions are represented as interpreted functions.

$\varphi_{\mathcal{P}}$ is defined as the conjunction of two formulas $\varphi_{\mathcal{P}}^w$ and $\varphi_{\mathcal{P}}^R$. Any model of $\varphi_{\mathcal{P}}^w$ corresponds to a winning defender strategy. $\varphi_{\mathcal{P}}^w$ is defined as follows. First, let there be a fixed set of constants $\{q_i^a\}_i$ and $\{q_j^d\}_j$. The initial state of the game must be a state of the strategy:

$$\bigvee_x q_x^a = i$$

Each attacker state of the strategy is not an accepting state of the game (by the definition of a game, no defender state is an accepting state):

$$\bigwedge_x \neg A(q_x^a)$$

From each attacker state of the strategy, each internal action that the attacker chooses causes the game to transition to some defender state of the strategy:

$$\bigwedge_{x,y} \bigvee_z \tau_i^a(q_x^a, \mathsf{ins}_y) = q_z^d$$

From each defender state of the strategy, the defender chooses a defender internal action on which the game transitions to an attacker state of the strategy. For a $\delta_i : S^d \to \Sigma_i^d$ a function that maps each defender state to the internal action that the defender chooses when in the state:

$$\bigwedge_x \bigvee_y \tau_i^d(q_x^d, \delta_i(q_x^d)) = q_y^a$$

For each attacker state of the strategy, each call that the attacker chooses causes the game to transition to some defender state of the strategy:

$$\bigwedge_{x,y} \bigvee_z \tau_i^a(q_x^a, c_y^a) = q_z^d$$

For each defender state of the strategy, the defender chooses a defender call action on which the game transitions to an attacker state of the strategy. For $\delta_c : S^d \to \Sigma_c^d$ a function that maps each defender state to the call action that the defender chooses when in the state:

$$\bigwedge_x \bigvee_y \tau_c^d(q_x^d, \delta_c(q_x^d)) = q_y^a$$

We introduce analogous constraints to ensure that strategy states always transition on return actions to strategy states.

Any model of $\varphi_{\mathcal{P}}^R$ corresponds to a modular strategy. $\varphi_{\mathcal{P}}^R$ is defined using bisimulation relation over attacker states $\cong_a \subseteq S^a \times S^a$, and a bisimulation relation over defender states $\cong_d \subseteq S^d \times S^d$. If two states defender states are in $\cong$, then the defender must choose the same defender action from each state. We require that any two states that are the destination of the same RPC call must be in the bisimulation relation to ensure that a strategy is modular for every RPC call.

The bisimulation relations are defined as follows. If two attacker states are the destination of a transition on the defender call action rpc, then they must be bisimilar:

$$\bigwedge_{x,y} \tau_r^d(q_x^a, \mathsf{rpc}) \cong_a \tau_r^d(q_y^a, \mathsf{rpc})$$

Intuitively, we must constrain that bisimilar states transition on the same actions to bisimilar states. Each attacker state is bisimilar to itself.

$$\bigwedge_x q_i^a \cong_a q_x^a$$

Each defender state is also bisimilar to itself.

$$\bigwedge_x q_x^d \cong_d q_x^d$$

If two attacker states are bisimilar, then on each attacker internal action, the states transition to states that are bisimilar. For each attacker internal action k:

$$\bigwedge_{x,y} (q_x^a \cong_a q_y^a \implies \tau_i^a(q_x^a, k) \cong_d \tau_i^a(q_y^a, k))$$

If two defender states are bisimilar, then on each internal defender action, they transition to states that are bisimilar. For each defender internal action k:

$$\bigwedge_{x,y} (q_x^d \cong_a q_y^d \implies \tau_i^d(q_x^d, k) \cong_d \tau_i^d(q_y^d, k))$$

If two attacker states are bisimilar, then on each attacker call action, the states transition to states that are bisimilar. For each attacker call action k:

$$\bigwedge_{x,y} (q_x^a \cong_a q_y^a \implies \tau_c^a(q_x^a, k) \cong_d \tau_c^a(q_y^a, k))$$

If two defender states are bisimilar, then the defender call actions for each must be equal, and the defender states must transition on

the call action to bisimilar states.

$$\bigwedge_{x,y}(q_x^d \cong_d q_y^d \implies \tau_c^d(q_x^d, \delta_c(q_x^d)) \cong_a \tau_c^d(q_y^d, \delta_c(q_y^d)))$$

Bisimilarity is constrained analogously for return transitions.

Lemma 3. *For a policy-weaving problem $\mathcal{P}$, let m be a model of $\varphi_{\mathcal{P}}$, as defined above. Let G_m be $\mathsf{G}_{\mathcal{P}}$ restricted to all attacker states that are values in m of some constant q_i^a, defender states that are values of some constant q_j^d, defender internal actions in the range of δ_i, defender call actions in the range of δ_c, and defender return actions in the range of δ_r, in the model σ. Let S_m be the strategy constructed from G_m as described above. Then S_m is a winning, modular defender strategy for $\mathsf{G}_{\mathcal{P}}$.*

Proof. Follows directly from the correctness proofs given in [5, 26]. □

Theorem 1. *For a policy-weaving problem $\mathcal{P}$, let m be a model of $\varphi_{\mathcal{P}}$, let S be the strategy defined by m. Then I_S solves $\mathcal{P}$.*

Proof. Follows directly from Lem. 1, Lem. 2, and Lem. 3. □

3.4 Discussion

Completeness: The reduction of §3.3 is complete for finding modular strategies. We have found that practical weaving problems can be solved by searching only for such strategies (see §4). However, in principle, there may be weaving problems for which the corresponding safety game has no modular winning defender strategy. Such problems fall into one of two cases.

First, some problems may have a winning attacker strategy, which defeats any defender strategy. We can search for a winning attacker strategy in parallel to searching for a winning defender strategy, and if one is found, provide it to the programmer. However, the problem of finding any global (i.e., not necessarily modular) attacker strategy is EXPTIME-complete [9].

Second, some problems may have a winning global defender strategy, but not have a modular winning defender strategy. We cannot use such a strategy to instrument a program, as an instrumentation function defined by it is not modular.

In both cases, while a given VPA abstraction of the program may be too coarse to allow for a winning modular defender strategy, there might be some more precise, yet sound, abstraction of the program that has a winning modular winning defender strategy. It may be possible to automatically refine a VPA abstraction of a program from a failed search for a winning modular defender strategy, perhaps by extending CEGAR-style automatic refinement [6, 11]. We leave this as future work.

Injected code: The reduction of §3.3 searches for an instrumentation that invokes a Capsicum primitive after each program instruction. However, if a program is compromised, then it may execute arbitrary code that is not instrumented with Capsicum primitives. The reduction can be extended to model this threat by allowing the attacker to execute a special "injected code" instruction at states of the game that correspond to vulnerable points of the program. After the attacker executes the injected-code instruction, they can execute any instruction or invoke any Capsicum primitive, while the defender can only invoke the `noop` primitive.

Primitives per instruction: The reduction of §3.3 only searches for an instrumentation that invokes exactly one primitive after each instruction. This does not fundamentally limit the number of primitives that the instrumentation can invoke, as we can either (1) redefine the space of primitives Prims to be all sequences of k Capsicum primitives for some fixed integer k, or (2) inject a block of "noop" program instructions that have no effect on the state of policies, and allow a primitive to be invoked after each.

Runtime overhead: We do not address the problem of finding instrumentations that minimize runtime overhead. The runtime overhead of Capsicum primitives has been previously studied in-depth [34] by the Capsicum developers. The developers found that each invocation of the primitives `enter_cm` and `limit_fd` induces only a small overhead, on the order of microseconds or nanoseconds [34]. However, partitioning a program to execute in multiple processes can induce overheads on the order of milliseconds, which is observed when executing the rewritten `gzip` on some workloads.

We can extend our algorithm in various ways to minimize runtime overhead. Some redundant or inefficient invocations of `enter_cm` and `limit_fd`, can be removed post-hoc by a simple peephole optimizer. We may be able to search for instrumentations that optimize a performance metric by extending the constraint-solving problem defined in §3.3 to a constraint-optimization problem. We leave this as future work.

Weaving for other operating systems: We have presented an algorithm that takes a program, and policies defined over the capabilities provided by Capsicum, and instruments the program to correctly invoke the primitives provided by Capsicum. However, the algorithm can be generalized naturally to instrument programs for other operating systems that provide security primitives, such as decentralized information flow control (DIFC) [17, 23, 36] and tagged-memory systems [7]. Intuitively, all such systems are similar in that they decide the *privileges* with which a program executes (e.g., capabilites) by monitoring a separate set of *primitives* that the program invokes (e.g., `enter_cm`). Moreover, all such systems are similar in that the relationship that they define between privileges and primitives can be modeled as a VPA, similar to the Capsicum VPA introduced in §3.2.1. Given such a model of the system, we can automatically define a weaving algorithm for the system that reduces the problem of instrumenting a program to solving a safety game.

The Capsicum weaving algorithm of §3.3 may be generalized to weaving algorithms for other systems by replacing the Capsicum model introduced §3.2.1 with a VPA model of the target system. The language of the Capsicum model relates an execution of the program instrumented with invocations to `limit_fd` and `enter_cm` to the capability traces induced by the instrumented execution, where a capability trace is a trace of program instructions paired with capabilities. In general, an operating system model relates an execution of the program instrumented with *system primitives* to the *privilege traces* induced by the instrumented execution, where a privilege trace is a trace of program instructions paired with privileges defined by the system. For DIFC operating systems, the system primitives manage the labels of each process. The privileges are the ability of one process to send and receive information to another. For tagged memory systems, the system primitives manage the labels of memory objects. The privileges are the ability of one process to read from or write to memory. Given a VPA model of an operating system, we can apply the reduction of §3.3 as a weaving algorithm for the system.

3.5 Weaving over Abstractions of Programs

When defining our weaving algorithm in §3.2, we assumed that the set of executions of a program is represented as a VPA. But in general, the executions of a program cannot be described exactly by a VPA. However, the executions of every program can be over-approximated by a VPA. For a given program, several natural over-approximating VPA's can be defined which model the control flow of the program, along with a finite set of facts about its data.

To verify that a program satisfies a given safety property, it suffices to verify that an over-approximation of the program satisfies the property. A similar result holds in policy weaving. In particular, a correct instrumentation for a policy-weaving problem defined

by an over-approximation of a program P is a correct instrumentation for the policy-weaving problem defined over P and the same policies.

Lemma 4. *For a set of instructions* Instrs*, set of Capsicum primitives* Prims*, and capabilities* Caps*, let* $L(P) \subseteq L(P^\#) \subseteq \mathsf{Instrs}^*$*, let* $S \subseteq (\mathsf{Instrs} \times \mathsf{Caps})^*$*, and let* $F \subseteq (\mathsf{Instrs} \times \mathsf{Caps})^*$*. If* $I^\# : \mathsf{Instrs}^* \to (\mathsf{Instrs} \times \mathsf{Prims})^*$ *is a correct instrumentation for* $\mathcal{P}^\# = \mathsf{Prob}(P^\#, S, F)$*, then* $I^\#$ *is a correct instrumentation for* $\mathcal{P} = \mathsf{Prob}(P, S, F)$*.*

Proof. We show that $I^\#$ is a solution for $\mathcal{P}$ by showing that $I^\#$ is secure, functional, and modular, as defined in §3.2.2.

Secure and Functional: Suppose, for a proof by contradiction, that $I^\#$ is not secure or not functional for $\mathcal{P}$. Then by Lem. 1, there is some sequence $s \in (\mathsf{Instrs} \times \mathsf{Prims})^*$ in $L_\mathcal{P}$ (Eqn. (3)). Let s_{Instrs} be the Instrs-components of s. By the definition of $L_\mathcal{P}$, $s_{\mathsf{Instrs}} \in P$, and thus $s_{\mathsf{Instrs}} \in P^\#$. Thus $s \in L_{\mathcal{P}^\#}$, and ins is not secure or not functional for $\mathcal{P}^\#$. But this contradicts the assumption that I is a solution of $\mathcal{P}^\#$.

Modular: This is immediate, as the definition of "Modular" in §3.2.2 does not depend on the $P^\#$. □

3.6 Hardness of Capsicum Policy Weaving

In §3.3.3, we defined, for a given policy weaving problem $\mathcal{P}$, a set of constraints $\varphi_\mathcal{P}$ such that from a model to $\varphi_\mathcal{P}$, we can construct a solution to $\mathcal{P}$ in polynomial time. The size of $\varphi_\mathcal{P}$ is polynomial in the size of $\mathcal{P}$, and $\varphi_\mathcal{P}$ is a formula for a theory whose decision problem is in NP. Thus the Capsicum policy-weaving problem is in NP.

We now show that the Capsicum policy-weaving problem is NP-hard by reduction from 3-SAT. Combined with the fact that Capsicum policy-weaving is in NP, this shows that the policy-weaving problem is NP-complete.

Lemma 5. *For a set of instructions* Instrs*, Capsicum primitives* Prims*, and capabilities* Caps*, let* $P \subseteq \mathsf{Instrs}$*,* $S \subseteq (\mathsf{Instrs} \times \mathsf{Caps})^*$*,* $F \subseteq (\mathsf{Instrs} \times \mathsf{Caps})^*$*, and* $\mathcal{P} = \mathsf{Prob}(P, S, F)$*. Solving* $\mathcal{P}$ *is NP-hard in the size of the transition functions of* P*,* S*, and* F*.*

Proof. We will show that the Capsicum-policy-weaving problem is NP-hard by reduction from 3-SAT, similar to a hardness proof given in [5]. For an instance of 3-SAT φ, we construct a weaving problem $\mathcal{P}_\varphi$ such that a solution to $\mathcal{P}_\varphi$ corresponds to a solution to φ. For each variable x that occurs in φ, we introduce an instruction in Instrs that executes with capability Env iff x is true. For each conjunct in φ, we allow the program P to choose a corresponding instruction in Instrs, and for each disjunct in the conjunct chosen by the program, we allow the instrumentation to choose whether some instruction must or must not execute with capability Env.

We construct $\varphi_\mathcal{P}$ as follows. Let φ be a 3-SAT formula in conjunctive normal form (CNF), and let $\mathsf{Vars}(\varphi)$ be the propositional variables that occur in φ. Let the set of program internal actions Intras include: two "initial" instructions $\mathsf{ins}^0_m, \mathsf{ins}^1_m$; for each $x \in \mathsf{Vars}(\varphi)$, instructions ins^0_x and ins^1_x; and for each conjunct c in φ, instructions ins^0_c and ins^1_c. Let the set of program call actions Calls include for each $x \in \mathsf{Vars}(\varphi)$ a call call_x, and let the set of return actions Rets include a return actions ret_x. Let the set of program descriptors include a single descriptor desc. Assume that Capsicum defines exactly three rights, r_0, r_1, and r_2. Thus, the set of capabilities for P is $\mathsf{Caps} = \{\mathsf{Triv}, \mathsf{Env}, (\mathsf{desc}, r_0), (\mathsf{desc}, r_1), (\mathsf{desc}, r_2)\}$.

Let the program P be as follows. From the initial state of P, let it execute ins^0_m followed by ins^1_m, and then choose to execute $\mathsf{ins}^0_c, \mathsf{ins}^1_c, \mathsf{call}_x, \mathsf{ins}^0_x, \mathsf{ins}^1_x \mathsf{ret}_x$ for exactly one conjunct c and variable $x \in \mathsf{Vars}(c)$.

Let the security policy $S \subseteq (\mathsf{Instrs} \times \mathsf{Caps})^*$ accept all strings except for the following:

- Any string that contains $(\mathsf{ins}^1_m, \mathsf{Env})$.
- If $\neg x$ is the jth disjunct of conjunct c, any string that contains $(\mathsf{ins}^1_c, (\mathsf{desc}, r_j))$ followed eventually by $(\mathsf{ins}^1_x, \mathsf{Env})$.

Let the functionality policy $F \subseteq (\mathsf{Instrs} \times \mathsf{Caps})^*$ accept exactly the strings:

$$\pi^j_c = (\mathsf{ins}^0_m, \mathsf{Triv}), (\mathsf{ins}^1_m, \mathsf{Triv}), (\mathsf{ins}^0_c, \mathsf{Triv}), (\mathsf{ins}^1_c, (\mathsf{desc}, r_j)), (\mathsf{call}_x, \mathsf{Triv}), (\mathsf{ins}^0_x, \mathsf{Env})$$

for conjunction c whose jth disjunct $l \equiv \neg x$ for some $x \in \mathsf{Vars}(\varphi)$, along with the strings

$$\pi^j_c.(\mathsf{ins}^1_x, \mathsf{Env})$$

for conjunction c whose jth disjunct $l \equiv x$ for some propositional variable x.

Let $\mathcal{P}_\varphi = \mathsf{Prob}(P, S, F)$. Suppose that $\mathcal{P}_\varphi$ has a solution instrumentation I. From I, we can efficiently construct a satisfying assignment σ_I for φ. For each $x \in \mathsf{Vars}(\varphi)$, let $s \in \mathsf{Instrs}^*$ be any execution of P that contains $\mathsf{call}_x, \mathsf{ins}^0_x$. Let $\sigma_I(x) = \mathsf{True}$ if and only if in $(\mathsf{ins}^0_x, \texttt{enter\_cm}) \in I(s)$.

σ_I is a satisfying assignment of φ. To see this, c be an arbitrary conjunct of φ. Let l be the jth disjunct of c. Let $s \in P$, and let I limit the program to have only capability (desc, r_j) when it executes ins^1_c in s.

Remark 3. *For* $x \in \mathsf{Vars}(\varphi)$*, the program has capability* Env *when it executes* ins^1_x *in some capability trace induced by* $I(s)$ *if and only if* $(\mathsf{ins}^0_x, \texttt{enter\_cm}) \notin I(s)$*. To see this, first suppose that* $l \equiv x$*. Then some capability trace induced by* $I(s)$ *must contain* $(\mathsf{ins}^1_x, \mathsf{Env})$ *by the definition* F*. Thus* $(\mathsf{ins}^0_x, \texttt{enter\_cm}) \notin I$*. If* $l \equiv \neg x$ *for some variable* x*, then no trace induced by* $I(s)$ *contains* $(\mathsf{ins}_x, \mathsf{Env})$*, by the definition of* S*. But by the definition of* F*, it must be that* $(\mathsf{call}_x, \mathsf{rpc}) \in I(s)$*. Thus* $(\mathsf{ins}^0_x, \texttt{enter\_cm}) \in I(s)$*.*

We now show that $\sigma_I(l) = \mathsf{True}$. If $l \equiv x$ for some variable x, then by the definition of F and Remark 3, I must not invoke `enter_cm` in response to ins^0_x. Then by the definition of σ_I, $\sigma_I(l) = \sigma_I(x) = \mathsf{True}$. If $l \equiv \neg x$ for some variable x, then by the definition of S and Remark 3, I must invoke `enter_cm` in response to ins^0_x. Then by the definition of σ_I, $\sigma_I(l) = \sigma_I(\neg x) = \mathsf{True}$. In either case, l is satisfied under σ_I.

We have shown that for each conjunct c, there is some literal that is true under σ_I. Thus, σ_I is a satisfying assignment of φ.

Conversely, if φ has a satisfying assignment, then $\mathcal{P}$ has a solution. Then from a satisfying assignment for φ, we can efficiently construct a solution to $\mathcal{P}_\varphi$, and vice versa. To see this, first, suppose that φ has a satisfying assignment σ. Let I_σ be an instrumentation that does the following:

- When the program executes ins^0_m, it invokes `enter_cm`.
- When the program executes ins_c, it chooses some disjunct j of conjunct c that is true under σ, and responds to ins_c by limiting the capabilities of the program to only (desc, r_j). Because σ is a satisfying assignment of φ, there is always some disjunct j for each conjunct c.
- When the program executes call_x, responds with rpc.
- When the program executes ins^0_x, responds by invoking `enter_cm` if and only if $\sigma(x)$ is false.

I_σ satisfies $\mathcal{P}_\varphi$. To see this, first, let s be an execution of P that contains the instruction ins_c for the ith conjunct of φ. We first show

Name	LoC	S	F	RPC			Time
bzip2-1.0.6	8,399	4	3	2	2	676	0m 03s
fetchmail-6.3.19	49,370	3	4	1	1	3,468	1m 05s
gzip-1.2.4	9,076	3	3	2	1	543	0m 23s
tar-1.25	108,723	3	4	0	0	5,135	2m 51s
tcpdump-4.1.1	87,593	4	3	2	1	10,075	0m 28s
wget-1.12	64,443	3	7	3	0	4,329	1m 00s

Table 1. Performance data for the policy-weaving tool. Column "LoC" contains lines of C source code (including blank lines and comments), "S" contains the number of states in the security policy, "F" contains the number of states in the functionality policy, "RPC" contains the number of calls marked for RPC vs. the number that appear to be needed, and "Time" contains the time taken to find an instrumentation, in minutes and seconds.

that I_σ is secure. By the definition of S, $I_\sigma(s)$ can only violate S if one of the following:

1. The program executes ins_m^1 with capability Env. But I_σ does not allow such an execution, as it responds to ins_m^0 with `enter_cm`.
2. $l \equiv \neg x$, and the program executes ins_x^1 with capability Env. But I_σ invokes `enter_cm` before the program executes ins_x.

Thus I_σ is secure.

We now show that I_σ is functional. By the definition of F, for $s \in P$, $I_\sigma(s)$ can only violate F if $l \equiv x$ and the program executes ins_x without capability Env. But $(\mathsf{call}_x, \mathsf{rpc}) \in I_\sigma(s)$, and $(\mathsf{ins}_x^0, \texttt{enter\_cm}) \notin I_\sigma(s)$, by the definition of I_σ. Thus I_σ is functional.

To see that I_σ is modular, observe that the primitives that I invokes after each call call_x depend only on $\sigma(x)$, not on any of the instructions chosen by the program before call_x. I_σ is secure, functional, and modular, and thus is a solution of $\mathcal{P}_\varphi$. □

4. Experiments

We experimentally evaluated the policy-weaving algorithm presented in §3. In particular, we designed a set of experiments to answer the following questions: (1) Can policies for practical programs be expressed naturally as policy automata? and (2) Can the weaving algorithm efficiently instrument programs to satisfy their policies?

To answer these questions, we collected a set of real-world system programs with known past security vulnerabilities, and informal policies for each program. Some of the programs were found through interaction with the Capsicum developers [12], while others were chosen as popular system utilities with well-known vulnerabilities [22, 31]. We specified a policy for each program as security and functionality policy automata. We implemented the algorithm described in §3 as a tool, and applied to the tool to each program and its policies to rewrite the program.

The experiments indicate that our policy-weaving algorithm is practical. We were able to express desired policies for each of the programs as automata, and the tool found an instrumentation for each program and policies in minutes.

We now discuss the program and policies used, and discuss the results of applying our tool to the programs and policies.

4.1 Programs and Polices

`tcpdump` We described the structure of `tcpdump` and its required security and functionality policies in §2.

`gzip, bzip2, tar` The `gzip` compression tool has exhibited vulnerabilities in the past, due to its complicated compression and decompression code [33]. `gzip` was previously rewritten manually by the Capsicum developers to execute securely on Capsicum [34].

`gzip` mainly executes in a loop. In each iteration of the loop, `gzip` processes command-line arguments, configures files, and invokes compression and decompression routines to read input from and write output to the configured files [34]. While the code that processes arguments and configures files is simple and trusted, the compression and decompression routines are complex, and have exhibited vulnerabilties.

Guided by the policy given informally by the Capsicum developers [34], we constructed policy automata that allow `gzip` to access its environment when executing the main loop, but only allow `gzip` to read from an input file and write to an output file when executing the compression and decompression functions, which we assumed could inject arbitrary code.

We applied our tool to `gzip` and these policies. The tool instrumented `gzip` to execute each of `compress` and `decompress` as a separate process. Each such process has capabilities only to read from a specific input file, and write to a specific output file. The tool correctly determined that the compression and decompression functions needed to execute in a separate process so that after they return, `gzip` would again have full permission to access its environment to open new files.

Like `gzip`, the `bzip2` compression utility and `tar` archiving utility have demonstrated security vulnerabilties [22, 31]. We defined policies for `bzip2` and `tar` analogous to the policy described above for `gzip`.

`fetchmail` `fetchmail` downloads mail from a list of servers. In a typical execution, `fetchmail` reads a list of mail servers, and then iteratively downloads mail from each one. As `fetchmail` executes, it thus handles data read from a network connection that may be untrusted. A desirable policy for `fetchmail` is that it should be able to open connections to the network and should always be able to write to a designated output file and log file, but once it reads data from the network, it may not access any other resources in its environment.

`wget` In a typical execution, `wget` first opens and configures output and logging files. It then iterates through a given list of URL's. For each URL, it opens a network connection to the URL, and downloads data from the URL. As `wget` executes, it thus handles data read from a network connection that may be untrusted. A desirable policy for `fetchmail` is that it should be able to open connections to each URL, and should always be able to write to its designated output and log files. However, once it reads data from the URL resource, it should not access any other resource in its environment.

4.2 Results and Analysis

We applied our tool to rewrite each of the above programs to satisfy their policies. Data about the performance of the tool is in Tab. 1. Lines of code ("LoC") are the number of lines of C source code of the program (library code is not included). Each time is the average of three runs on a machine with 16 processors and 32 GB of memory, measured by the UNIX utility `time`. Each processor has four cores and a 12 MB cache. However, our tool does not explicitly exploit parallelism.

The data supports several claims about the effectiveness of the weaving algorithm. The security and functionality policies are small, all with less than ten states, as they are defined over only a small handful of important program actions. In our experience, they are much easier to define and understand than the primitives inserted into their programs, which contain tens of thousands of lines of code. The performance times indicate that the tool can rewrite programs efficiently enough that a developer could feasibly

integrate it into a system that periodically secures a program under development, or perhaps into a compiler toolchain. Note that the performance times are not strongly correlated with the size of the original program. Because each of the policies only concern a small handful of program instructions, our tool can aggressively minimize the full model of a given program to a much smaller model that, intuitively, behaves "equivalently" over instructions relevant to the policies. Thus, the performance of the tool tends to be correlated much more strongly with the location of policy-relevant instructions, in particular if they are located in complex control structures, or frequently-called functions.

As we discussed in §3.4, the only appreciable runtime overhead incurred in rewriting a program for Capsicum is typically due to RPCs. The "RPC" column of Tab. 1 contains the number of RPCs added by our instrumentation vs. the number that we believe are required from a manual inspection, and the total number of callsites in the program. Although our algorithm is not guaranteed to minimize the number of RPCs, the number of RPCs added is extremely low compared to the number of all callsites in the program. This is partly because adding too many RPCs would cause the program to violate its security policy, and partly because our optimizations rule out the vast majority of program callsites as irrelevant to the policy. However, even though the number of RPCs is small, each can add considerable overhead, and so we consider the problem of minimizing them to be an important issue.

5. Related Work

Security monitors: This paper describes an algorithm and a tool that automatically rewrite programs for Capsicum, an operating system that provides a set of capability-specific primitives [34]. Operating systems that provide security system calls as primitives allow one to define program-specific policies. In comparison, Mandatory Access Control (MAC) operating systems such as SELinux [25, 29, 35] only support system-wide policies described in terms of standard system calls. Such policies cannot refer to important events in the execution of a particular program, but many practical policies can only be defined in terms of such events [18]. UNIX can monitor programs to ensure that they satisfy policies if the program correctly uses the `setuid` system call, but in general this approach suffers the same shortcomings as MAC systems. In comparison, systems with security primitives allow an application to signal key events in its execution to the operating system.

An *Inline Reference Monitor* (IRM) rewriter takes a policy expressed as an automaton and instruments a target program with an IRM, which executes in the same memory space as the program, and halts the program if it attempts to perform some sequence of actions that would violate the policy [1, 18]. *Edit automata* [24] 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 [34, 36].

Writing programs for security monitors: Prior work in aiding programming for systems with security primitives automatically verifies that a program instrumented to use the Flume OS [23] primitives enforces a high-level policy [20], automatically instruments programs to use the primitives of the HiStar OS [36] to satisfy a policy [16], and automatically instruments programs [20] to use the primitives of the Flume OS [23]. However, the languages of policies used in the approaches presented in [16, 21] are not temporal and cannot clearly be applied to other systems with security primitives, and the proofs of the correctness of the instrumentation algorithms are ad hoc. The instrumentation algorithm presented in this paper is one instance of a general, automata-theoretic algorithm. As a result, the algorithm can be instantiated to generate instrumentation algorithms for a variety of systems with security primitives, including HiStar and Flume, and the tagged memory system Wedge [7].

Previous work [8, 10] automatically partitions programs so that high and low confidentiality data are processed by separate processes, or on separate hosts. We automatically partition programs so that each process of the partitioned program can correctly invoke operating system primitves to satisfy a policy, when a single monolithic may not be able to invoke primitives to satisfy the policy.

Skalka and Smith [30] 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. [19] verify that programs rewritten by an IRM rewriter are correct. Thus, the work in both of those papers concerns analyzing checks of the capabilities in managed programs, whereas our work concerns correctly applying primitives to restrict the capabilities of unmanaged programs.

Safety games: Safety games have been studied as a framework for synthesizing reactive programs and control mechanisms [3, 4, 13, 26]. Previous work describes algorithms that take a safety game represented symbolically, determine which player may always win the game, and sometimes synthesize a winning strategy for the player [13, 26]. One contribution of our work is connecting these game-theoretic problems to the problem of rewriting a program for a capability system. In particular, we extend the known problem of finding modular winning strategies [5] to a game problem that models the problem of instrumenting programs that execute in multiple process spaces. Our algorithm to find winning defender strategies for such games, is a variation on a known symbolic algorithm that searches for winning attacker strategies of a bounded size [26].

References

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

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

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

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

[5] R. Alur, S. L. Torre, and P. Madhusudan. Modular strategies for recursive game graphs. In *TACAS*, pages 363–378, 2003.

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

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

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

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

[10] 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.

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

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

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

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

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

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

[17] 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.

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

[19] K. W. Hamlen, G. Morrisett, and F. B. Schneider. Certified in-lined reference monitoring on .NET. In *PLAS*, pages 7–16, 2006.

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

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

[22] M. Izdebski. bzip2 'BZ2_decompress' function integer overflow vulnerability, Oct. 2011. URL `http://www.securityfocus.com/bid/43331`.

[23] 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.

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

[25] P. Loscocco and S. Smalley. Integrating flexible support for security policies into the Linux operating system. In *USENIX Annual Technical Conference, FREENIX Track*, pages 29–42, 2001.

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

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

[28] R. Naraine. Symantec antivirus worm hole puts millions at risk. *eWeek.com*, May 2006. URL `http://www.eweek.com/article2/0,1895,1967941,00.asp`.

[29] O. S. Saydjari. Lock : An historical perspective. In *ACSAC*, 2002.

[30] C. Skalka and S. F. Smith. Static enforcement of security with types. In *ICFP*, pages 34–45, 2000.

[31] Ubuntu. Ubuntu security notice USN-709-1, Jan. 2009. URL `http://www.ubuntu.com/usn/usn-709-1/`.

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

[33] Vulnerability Note. Vulnerability note VU#381508, July 2011. URL `http://www.kb.cert.org/vuls/id/381508`.

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

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

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