Secure Programming via Game-based Synthesis

By

William R. Harris

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2014

Date of final oral examination: 17 Dec. 2014

The dissertation is approved by the following members of the Final Oral Committee:

Somesh Jha, Professor, Computer Sciences
Thomas Reps, Professor, Computer Sciences
Thomas Ristenpart, Assistant Professor, Computer Sciences
Rajeev Alur, Professor, Computer Sciences
Xinyu Zhang, Assistant Professor, Electrical & Computer Engineering

Dedicated to my mother and father

Acknowledgments

Lately it occurs to me what a long, strange trip it's been.

— Robert Hunter

It does, as they say, take a village to raise a graduate student. The work presented in this thesis owes its existence largely to the work and guidance of my advisors, Somesh Jha and Thomas Reps; their unique mixture of dedication to the field and rigorous irreverence will be forever unmatched. It has also been aided directly by the hard work and keen insights of Jonathan Anderson, Sagar Chaki, Nicholas Kidd, Roman Manevich, Mooly Sagiv, Robert Watson, and Nickolai Zeldovich. It could only exist in the first place due to the creative insights and robust engineering of several independent groups of researchers that included Robert Watson, Jonathan Anderson, Nickolai Zeldovich, and an evaluation team at MIT Lincoln Laboratory led by Michael Zhivich. The DARPA CRASH program was also instrumental in fostering many of the collaborations that produced the presented work.

Over the time that I performed the work described in this thesis, I also grew tremendously as a researcher over the course of internships at NEC Labs America under the mentorship of Sriram Sankaranarayanan, at Microsoft Research India under the mentorship of Aditya Nori and Sriram Rajamani, and at Microsoft Research Redmond under the mentorship of Sumit Gulwani.

What part of it has come from me, I was only in position to produce due to the influence of the unreasonably talented students from my advisors' research groups, and the camaraderie of the friends that I have met in Madison and abroad over the last several years, and the unconditional support of my family.

My dissertation research was supported, in part, by the National Science Foundation under grant CCF-0524051; by DARPA and AFRL under contract FA8650-10-C-7088; and by a Microsoft Graduate Fellowship. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of DARPA, AFRL, NSF, or Microsoft.

Contents

Part IV Conclusion

Figures, Tables, and Listings

Abstract

Interactive security systems provide powerful *security primitives* (i.e., security-oriented system calls) that an application can invoke at various moments during execution to control accesses to its sensitive information. Prior to the work described in this thesis, an application developer was forced to explicitly write imperative code that executes security primitives. Moreover, a developer could only reason informally about whether the code satisfied the developers intuitive notions of security and correctness.

This dissertation describes the design of *policy weavers* for interactive-security systems. A policy weaver allows a programmer to specify desired functionality and security guarantees of an application, and automatically obtain a modified application that satisfies such guarantees when executed on an interactive-security system. Each policy weaver consists of (i) a policy language in which the developer expresses desired guarantees, and (ii) a program instrumenter that takes as input an uninstrumented program and a policy in the language, and outputs a program that satisfies the specified policy.

We have designed and evaluated policy weavers for the Capsicum capability system and the HiStar decentralized information-flow control (DIFC) system by designing and applying a *policy-weaver generator*, which takes as input the semantics of the primitives of each system and outputs a weaver for the system.

1 Introduction

Developing practical but secure programs remains a difficult, important, and open problem. A significant portion of the security vulnerabilities in widely-used applications allow an attacker who can control inputs to the program to use the program to perform actions on system state not intended by the application programmer or user, or the system administrator. An attacker can use a vulnerable application to violate the secrecy or integrity of information stored on the system on which the application is executed (i.e., the application's *host system*). Such vulnerabilities include "Improper neutralization of special elements used in OS command ('OS Command Injection')" and "Buffer copy without checking size of input ('Classic Buffer Overflow')," which, in a recent audit of security-critical applications [19], were classified in the Common Weakness Enumeration (CWE) by the SysAdmin, Audit, Networking, and Security (SANS) Institute as the second and third most prevalent classes of vulnerabilities. Such vulnerabilities can be found in network utilities that typically read inputs directly from an untrusted network and execute with the privilege to access arbitrary system resources [8, 10], and in file utilities and language interpreters that are often deployed to process untrusted data or execute untrusted programs [9, 11–14].

Even programs that do not contain vulnerabilities typically must share sensitive information with other programs executing on their host (i.e., the application's *environment*). In such situations, the goal is that cooperative programs should be able to carry out desired functionality using the sensitive information, but malicious programs should not be able to violate the secrecy or integrity of the sensitive information. For example, a trusted logging service may maintain a log file of important events—with the desired behavior being that each program in the logging service's environment can read the log, but can only modify the log by appending to it (and cannot corrupt entries previously added to the log).

Conventional system-level security mechanisms can enforce security guarantees for sensitive information throughout a system, but do not provide mechanisms that an application run by an unprivileged user can use to enforce the security of its sensitive information. *Multi-level secure systems* [44] and SELinux [59] implement *mandatory access control (MAC)*, which allows a trusted user, typically an administrator, to specify an access-control policy that the operating system enforces throughout the system by mediating each access of a resource by a process. For example, an administrator of a MAC system can specify a policy that enforces that if an untrusted user u reads information from a sensitive file, then u can never write information to a public directory. However, such systems do not enable a program executed by an unprivileged user to guarantee the security of its information. For example, the logging service described above, executed by an unprivileged user on a MAC system, cannot prevent other untrusted programs from directly modifying the log file that the service creates.

Programming languages, program analyses, and program rewriters can enforce that a given program does not violate the security of sensitive information that is used only by that program. However, they cannot enforce security guarantees about information shared by the application with other programs on a system. In particular, information-flow languages (i) analyze a program statically to determine that no execution of the program can violate security [41, 55], or (ii) monitor each program execution at runtime [26, 33] to determine that the monitored execution does not violate security. An *Inline Reference Monitor* (IRM) [24] is instrumentation code, inserted into a program by an IRM rewriter, that checks throughout each execution of the instrumented program that the instrumented program satisfies a given security policy. Such tools may be used, e.g., to check that a program that accesses a user's credit-card number does not leak any information about the credit-card number to a publicly-readable output channel. However, such tools cannot be used to enforce that if an application creates a sensitive resource (e.g., the log file described above) and transfers control to an unmonitored program in its environment, then the unmonitored program does not leak information from or corrupt information in the sensitive resource.

However, recent work [7, 22, 36, 58, 61] has produced new operating systems that allow a program that executes on behalf of an unprivileged user to protect the security of the program's sensitive information, even when the program executes a vulnerable program module or transfers control to an untrusted program. Such operating systems extend the set of system calls provided by a conventional operating system with security-specific system calls. (We refer to such operating systems as *interactive-security* systems, and refer to the system calls that they provide as *security primitives*.) At various points during a program's execution, it invokes security primitives to direct the system to protect the security of the program's sensitive information before transferring control to an untrusted program module or to the program's environment.

One example of an interactive-security system on which applications can enforce strong security guarantees is the capability operating system Capsicum [58], now included in FreeBSD 9 [25]. For each process, Capsicum tracks (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 authority to grant to itself more capabilities (i.e., open more files). Capsicum provides to each process a set of system calls that the process uses to limit its capabilities and its authority. Thus, a process executing trusted code in a program can first access system resources unrestricted by Capsicum, and then invoke primitives to limit itself to have only the capabilities that it requires while executing an untrusted program module. Thus, even if an attacker exploits a vulnerability in an untrusted module that allows the attacker to attempt to perform arbitrary system operations, the attacker will only be able to successfully carry out operations allowed by the limited capabilities set by the trusted code.

The Capsicum primitives are sufficiently powerful that a programmer can rewrite a practical program to satisfy a strong security guarantees by inserting only a few calls to Capsicum primitives [58]. Unfortunately, a programmer who writes a program for Capsicum must explicitly write code that executes imperative operations on capabilities, and reason informally that the rewritten program satisfies the programmer's implicit notion of correct behavior. In practice, it is difficult for programmers to reason about the subtle, temporal effects of the primitives. In fact, even Capsicum's own developers have rewritten programs, such as `tcpdump`, in a way that they tentatively thought was correct, only to discover later that the program was incorrect and required a different rewriting [58]. Often, as in the case of `tcpdump`, the difficulty results from the conflicting demands of (i) using low-level primitives, (ii) ensuring that the program satisfies a strong, high-level security requirement, and (iii) preserving the core functionality of the original program.

Whereas a program that executes on a capability system invokes primitives to restrict the operations that can be performed by untrusted program modules executed by the program, a program on *Decentralized Information-Flow Control (DIFC)* operating system invokes primitives to protect the secrecy and integrity of its information from untrusted programs that execute in the program's environment. A DIFC system maps each object on the system (e.g., a process or file) to a label in a partially-ordered set, mediates the flow of information between objects during an execution, and only allows information to be transferred if the labels of the objects satisfy an ordering condition [20, 22, 36, 46, 61]. Such systems provide primitives that a program can invoke to update the labels of objects, according to a label semantics.

A program executing on a DIFC system can invoke primitives that enable it to enforce strong information-flow guarantees; for example, the login service on the HiStar DIFC system enforces that the password that a client provides to even an untrusted authenticator is not leaked by the authenticator. Unfortunately, a programmer who writes a program for a DIFC system must explicitly write a program that uses imperative label operations, and informally reason that the program uses such operations correctly to (i) to carry out desired functionality when interacting with a cooperative environment, but (ii) protects the secrecy and integrity of its information when interacting with a malicious environment. Previous research [38, 39, 57] has shown that programmers have difficulty using labels in the context of DIFC languages to verify that a program does not leak information, or to rewrite a program that maintains labels to enforce information-flow security. There has been almost no previous work on writing programs that maintain labels on a DIFC system to preserve the security of information shared with untrusted programs. (The limitations of previous work [21] are discussed in detail in Chapter 13.)

Developing applications for a given interactive-security system is thus a significant challenge. A second challenge is to develop methods so that techniques and tools for programming for a given interactive-security system can be easily adapted to another system. Capability systems provide security guarantees different from those provided by DIFC systems. Moreover, systems with primitives and guarantees different from both capability and DIFC systems continue to be developed, such as tagged memory systems [7]. Finally, even within a single class of interactive-security systems, different systems can have important, but subtle differences. Asbestos [22], HiStar [61], and Flume [36] are each DIFC systems that allow applications to enforce information-flow guarantees, but provide to applications primitives with subtle differences with which to enforce such guarantees. A developer of an interactive-security system thus faces a significant challenge to deploying his system, in that after designing and developing the primitives of the system, he must then design and develop the application programming environment (i.e., programming libraries) for the system from scratch.

The thesis of the work presented in this dissertation is that *practical programs can be instrumented automatically from declarative security policies to use imperative interactive-security primitives to satisfy the policies; instrumenters can be generated automatically from declarative specifications of the semantics of interactive-security primitives.* The work presented in this dissertation introduces techniques that address the above challenges faced by application and system developers for interactive-security systems. To address the challenge of programming applications for two of the interactive-security systems described above, the Capsicum capability system and the HiStar DIFC system, we have designed languages of *security policies* with which a programmer can explicitly specify the operations that untrusted program modules and the program's environment should and should not be able to perform on sensitive resources. Along with each policy language, we have created a program instrumenter that takes from the programmer a program that invokes no security primitives and a security policy for the program, and automatically instruments the program to execute security primitives so that it satisfies the policy. We refer to the process of instrumenting a program to satisfy a policy as *weaving* the policy into the program (or simply "weaving," for short), and refer to a program instrumenter that implements the weaving process as a *policy weaver*.

To address the challenge of designing and developing a programming

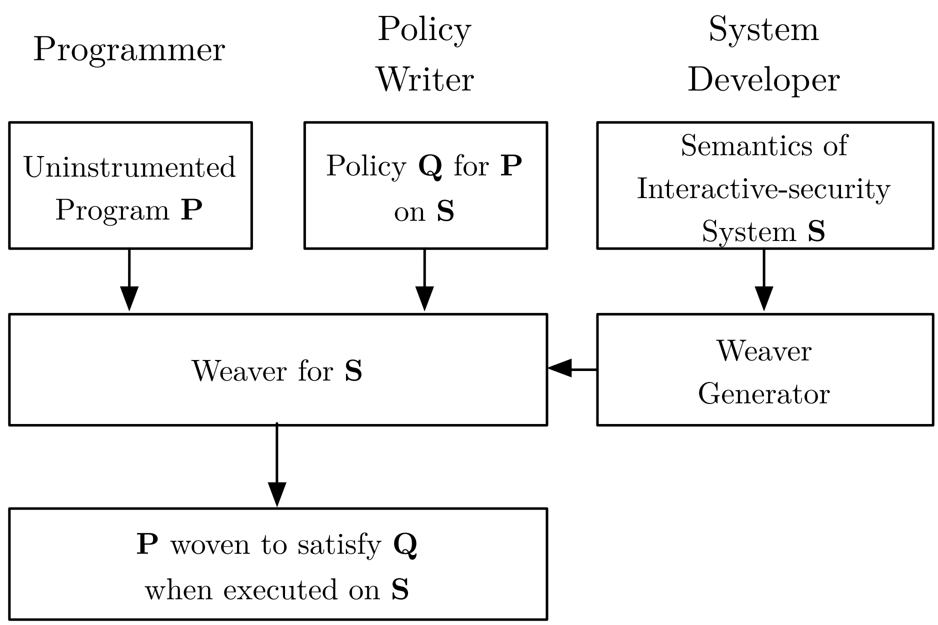

Figure 1.1: Workflow of the weaver generator.

environment for an interactive-security system, we have developed a *weaver generator* that takes as input a semantics of an interactive security system, and generates a weaver for the system automatically; the workflow of the weaver generator is depicted graphically in Fig. 1.1. A system developer provides to the generator a definition of (1) the state space of their system, (2) the set of primitives provided by the system, and (3) a semantics of each primitive that describes how each primitive transforms the system state. The developer then obtains a policy weaver for their system from the generator automatically. We have obtained the policy weavers for the Capsicum and HiStar in particular by applying the weaver generator.

There are three key, closely-related challenges to developing weavers for interactive-security systems. The first challenge is to design policy languages that can express the security requirements of practical applications for interactive security systems independent of the primitives that a program must invoke to satisfy the policy. The second challenge is to design a weaving algorithm that can reason about the semantics of programs on interactive security systems; such programs are difficult to reason about because interactive-security systems typically allow a program to generate security-relevant state consisting of an unbounded set of objects, with intricate relationships between objects. The third challenge is to design a weaving algorithm that is parameterized on the semantics of a given interactive-security system and its policies, and can thus be applied to generate weavers for different interactive-security systems.

To address the above challenges, we developed an approach that combines techniques for synthesizing programs that satisfy temporal policies (namely, game-based synthesis) with techniques that compute sound approximations of the infinite set of states that a program may reach (namely, analysis of structure transition systems). In particular:

- We defined an operational semantics for the Capsicum and HiStar primitives as transformers of logical structures. We modeled the state space of each system as the space of first-order structures in a logical vocabulary, and modeled the semantics of each system primitive as transformers for each predicate in the vocabulary. (We refer to the transition system defined by such a state space and its transformers as a *structure transition system*; structure transition systems were originally explored in previous work on shape analysis [47] and as a platform for emulating sequential algorithms [28].)

- We designed policy languages as *classes of automata over conditions on state*. Policies for Capsicum applications describe what capabilities a trusted program module must ensure that an untrusted program module possesses when the trusted module transfers controls to the untrusted module, and what capabilities the trusted program must ensure that the untrusted program never obtains. Policies for HiStar applications describe which files a program's environment must be able to read from or write to when the program transfers control to the environment, and which files the program's environment should never be able to read from or write to.

- We designed an algorithm that takes (1) a program that invokes no security primitives and (2) an application policy, and weaves the program to invoke system primitives so that it satisfies the policy. The algorithm reduces the problem of correctly weaving the program to solving a *two-player safety game* [3]. Such a game is played in sequential turns by two competing players: an *Attacker*, who attempts to drive the game to a particular state, and a *Defender*, who attempts to thwart the attacker. A *winning Defender strategy for a game* is a procedure that chooses a Defender action in response to a play such that if the Defender always chooses his next action according to the strategy, then the Defender always wins the game. The weaver constructs a game in which the Attacker models untrusted program modules and the program's environment, and every play won by the Attacker corresponds to an execution of the program and its environment that violates the input application policy. Under this model, a winning Defender strategy corresponds to a weaving of the program such that all executions of the woven program satisfy the application policy.

  To construct from a program P a finite game (so that it can be solved efficiently via a classical algorithm), we create a program P' that may execute the sequences of operations executed by multiple possible instrumentations of P, and then create a finite abstraction $\mathsf{P}'^{\#}$ of P'. To construct $\mathsf{P}'^{\#}$ so that it still retains enough information about the executions of P' to differentiate executions that result in a violation of the policy from runs that result in satisfaction of the policy, we perform an analysis of P' that computes a finite approximation of the set of structures that may be reached by a structure-transition system [47] that models P'. From $P'^{\#}$, we construct a game for which each winning Defender strategy defines a satisfying weaving of P.

We believe that the technique proposed in this work, which combines game-based synthesis with shape analysis, is particularly well-suited to address the problems that arise in weaving programs for interactive-security systems. Before performing the work described in this thesis, we performed preliminary work on automatically instrumenting programs for the Flume DIFC system [31]. The approach in our preliminary work reduced the problem of instrumenting a program for a DIFC system to solving a system of constraints using an SMT solver. Our experience led us to conclude that approaches based on constraint solving were useful for instrumenting programs that operated over a bounded set of security-sensitive objects, but could not be naturally applied to instrument programs that operated over an unbounded set of objects, including programs that create capabilities over unbounded sets of descriptors or assign labels to an unbounded set of objects on a filesystem.

We thus developed a weaver for the Capsicum capability system that constructed a game by exploring an abstract state space of a program, using a system semantics and abstract semantics represented as operational code in the weaver [32]. While we were able to apply this approach to weave practical programs for Capsicum, we found that implementing even fixed abstractions for Capsicum's state space was a burdensome and error-prone task, and that such abstractions could not be generalized naturally to construct useful abstractions for the relatively-complex states of HiStar programs.

To develop a weaver generator that could be instantiated to obtain weavers for both Capsicum and HiStar, we observed that the state spaces of both Capsicum and HiStar, while having apparently fundamentally different properties, can both be modeled accurately as classes of relational structures, and the semantics of primitives can be modeled as predicate transformers specified using formulas in first-order logic with transitive closure. Existing work on shape analysis [47] has developed accurate but scalable abstract domains for such classes of structures, even when the size of structures in a class is unbounded. The work presented in this thesis thus overcomes the restrictions on the size of states and the difficulties of developing abstractions that are inherent to previous approaches.

If we step back a bit and consider the trajectory of this work, we switched from thinking about the problem as one that was best encoded as a constraint-satisfying problem to a kind of synthesis problem: the goal of policy weaving is to synthesize the code (and its placement in the program) that enforces the desired policy. The use of games falls out naturally from this perspective. We also found that some techniques often used in other work on synthesis—particularly the use of advice "templates"—were crucial to scaling up the game-based synthesis approach to policy weaving (see §5.5.3 and §9.5.3).

We developed a weaver-generator `wag` that weaves programs in the LLVM intermediate language, applied `wag` to generate weavers for Capsicum and HiStar, and applied the weavers to weave security-critical applications for Capsicum and HiStar. We found that the weavers could both weave code that was functionally equivalent to code written manually for such applications in previous work by the system developers, and could weave programs that had not be instrumented in previous work.

We applied the weaver for Capsicum, `capweave`, to rewrite several Unix utilities for Capsicum that contain security vulnerabilities. The woven programs included programs that were previously instrumented manually by the Capsicum development team, programs suggested through discussion with the Capsicum development team, and the PHP CGI interpreter, whose policy was defined by an independent group at MIT

Lincoln Laboratory. We applied `capweave` to weave programs to satisfy application policies with no more than three to four transitions. Each policy not only ruled out known exploits in each program, but restricted the capabilities of significant segments of the program, potentially ruling out a large class of future vulnerabilities. Programs woven by `capweave` executed with behavior equivalent to programs instrumented manually by an expert, and incurred sufficiently low runtime overhead that they are still deployable: only 4% runtime overhead over unwoven programs on realistic workloads.

We applied our weaver for HiStar, `hiweave`, to weave programs instrumented for HiStar in previous work, including a wrapper for the ClamAV virus scanner and a login service for a mutually-untrusting client and authenticator, which consists of four independent but tightly-interacting programs [61], and to weave the applications woven for Capsicum. The manually-instrumented versions of the programs were non-trivial: although small (only a few hundred lines of code), they contain dozens of operations that use information-flow labels, and typically must use labels to protect program modules that can be invoked later in an execution by an arbitrary environment to operate on sensitive objects. `hiweave` wove such programs automatically from application policies that contain between 2–7 transitions.

Outline: This dissertation is organized as follows. Chapter 2 reviews previous work in program analysis and automata. Part I and Part II discuss the design and development of weavers for the Capsicum and HiStar operating systems, respectively. Part III discusses the design of a policy-weaver generator. Part IV concludes with a discussion of related work and possible directions for future work.

2 Transition Systems

In this chapter, we review definitions of several automata-theoretic concepts that we will use to define and solve the policy-weaving problem for Capsicum and HiStar. In particular, we review definitions of abstractions (§2.1), two-player safety games (§2.2), and transition systems whose states are logical structures (§2.3).

2.1 Simulation, refinement, and abstraction

A transition system consists of a state machine, a space of actions, and a transition relation between states on actions.

Definition 1. Let a *transition system* be $T = (Q, \Sigma, \rightarrow)$, where:

- Q is the *state space* of T.
- Σ is the *action space* of T.
- $\rightarrow \subseteq Q \times \Sigma \times Q$ is the *transition relation* of T.

A *run* of T is an alternating sequence of states and actions that respect the transition relation of T. That is, a run is a sequence $q_0, a_0, \ldots, q_{n-1}, a_{n-1}, a_n$ such that for $0 \leqslant i < n$, $(q_i, a_i q_{i+1}) \in \rightarrow$. A *trace* of T is a sequence of actions $a_0, \ldots, a_n$ such that there is some sequence of states $q_0, \ldots, q_{n+1}$ for which $q_0, a_0, \ldots, q_n, a_n, q_{n+1}$ is a run of T.

For transition system T and T', T' at state q' simulates T at state q if each transition that T can take from q is matched by a transition that T' can take from q'.

Definition 2. Let $T = (Q, \Sigma, \rightarrow)$ and $T' = (Q', \Sigma, \rightarrow')$ be transition systems. A *simulation relation* $\sim \subseteq Q \times Q'$ from T to T' is a relation from the states of T to the states of T' such that for all states $q_0, q_1 \in Q$ and $q'_0 \in Q$ and each action $a \in \Sigma$, if $(q_0, a, q_1) \in \rightarrow$ and $q_0 \sim q'_0$, then there is some $q'_1 \in Q$ such that (q'_0, a, q'_1) and $q_1 \sim q'_1$. If there is a simulation relation $\sim$ from T to T' that contains a pair of states (q, q'), then we say that T' *simulates* T from (q, q'), or, alternatively that T *refines* T' *from* (q, q').

If $\sim$ is a function, then we refer to it as an *abstraction function*, and we refer to T' as an abstraction of T.

2.2 Two-player safety games

A conventional automaton may be viewed as a transition system that, in each step of a run, takes as its next input a symbol from one agent, typically referred to as the *environment*. A two-player safety game is a transition system that, in each step, takes as its next input a symbol from one of two competing agents, or *players*, called the *Attacker* and the *Defender*.

Definition 3. A *turn-based two-player safety game* is a six-tuple $G = (A, D, \iota, F, \Sigma, \tau)$, where:

- A is the set of *Attacker states*.
- D, which does not overlap with A, is the set of *Defender states*. $Q_G = A \cup D$ are the *states* of G.
- $\iota \in D$ is the *initial state*.
- $F \subseteq Q_G$ is the set of *Attacker-winning states*.

- Σ is the *alphabet*.
- $\tau : Q_G \times \Sigma \rightarrow Q_G$ is the *transition function*.

A *play* is a sequence of symbols in Σ. A play that drives G from ι to a state in A (D) is an *Attacker (Defender) play*, and a play that drives G from ι to a state in F is an *Attacker-winning play*.

A strategy for a game player is a procedure that takes as input the current play and outputs an action for the player to choose. A winning strategy is a strategy that a player can follow to always win when a play begins in the initial state. In general, many practical problems in synthesis can be reduced to synthesizing winning strategies for both the Attacker and Defender. In this work, we focus only on synthesizing and using winning strategies for the Defender.

Definition 4. Let $G = (A, D, \iota, F, \Sigma, T)$ be a game. A *Defender strategy of* G is a function $S : \Sigma^* \rightarrow \Sigma$ from each Defender play of G to a game action. The *plays of* S are all plays in which the Defender chooses as the next symbol of the play the symbol output by S for the current play. I.e., $p \in \Sigma^*$ is a play of S if for each prefix p' of p that is a Defender play of G, p' appended with $S(p)$ (i.e., $p'\ S(p')$) is a play of S.

A *winning* Defender strategy is a strategy for which each play is not an Attacker-winning play.

A *positionless Defender strategy* is one that chooses the next action using only the current state of the game. I.e., a positionless strategy $S : D \rightarrow \Sigma_G$ is a function from each Defender state to an action; a play $p \in \Sigma^*$ is a play of S if for each prefix p' of p, if p' drives G to state $q \in D$, then $p'\ S(q)$ is a prefix of p.

There are known algorithms that take a finite two-player game G and either (1) determine that G has no winning Defender strategy, or (2) construct a winning Defender strategy [3]. Such algorithms execute in time linear in the size of G, and construct a positionless strategy whose representation uses space linear in the size of G.

For a finite-state acceptor A of words over an alphabet Σ and a game G, the game-automaton product of A and G is a game for which each Attacker winning play is an Attacker-winning play of G that is also accepted by A.

Definition 5. Let $M = (Q_M, \iota_M, \Sigma, F_M, T_M)$ be a deterministic finite-state acceptor, and let $G = (A_G, D_G, \iota_G, F_G, \Sigma, T_G)$ be a two-player safety game. The *game-acceptor product* $G \times_{G,A} M = (A_P, D_P, \iota_P, F_P, \Sigma, T_P)$, where

- The Attacker states are the product space of the Attacker states of G and the states of M. I.e., $A_P = A_G \times Q_M$.
- The Defender states are the product space of the Defender states of G and the states of M. I.e., $D_P = D_G \times Q_M$.
- The initial state is initial state of G paired with the initial state of M. I.e., $\iota_P = (\iota_G, \iota_M)$.
- The alphabet is the alphabet of G and M, Σ.
- The transition function of P defined analogously to the transition function of a product of automata. I.e., if for $q_G \in Q_G$, $q_M \in Q_M$, and $a \in \Sigma$, $\tau_P((q_G, q_M), a) = (\tau_G(q_G, a), \tau_M(q_M, a))$.

2.3 Structure Transition Systems

Previous work in program analysis and model checking has focused on techniques that take a program P that defines a transition relation over a large—potentially infinite—set of states and construct a finite program that simulates P. In particular, previous work [47] on analyzing programs that operates on linked data structures showed that shape-analysis problems can often by solved by analyzing programs that operate on logical structures. The shape-analysis problem is to determine if the program heaps reached over all runs of a given program satisfy a given property. Structure programs consist of a control-flow graph labeled by predicate transformers, which define guarded transformers over a state space of logical structures. A *structure-analysis* problem is to determine if all of the logical structures generated by a given structure program satisfy a given formula in first-order logic plus transitive closure (denoted by $\mathsf{FOL[TC]}$). We denote the set of $\mathsf{FOL[TC]}$ formulas over a vocabulary $\mathcal{V}$ as $\mathsf{FORMS}[\mathcal{V}]$.

Definition 6. Let $\mathcal{V}$ be a first-order relational vocabulary (i.e., a first-order vocabulary with predicate symbols, but no constant or function symbols) that contains the unary-predicate symbol new. A *predicate transformer for $\mathcal{V}$* is a triple of (1) a Boolean flag, (2) an $\mathsf{FOL[TC]}$ formula over vocabulary $\mathcal{V}$, and (3) a function from each predicate symbol in $\mathcal{V}$ of arity n to an $\mathsf{FOL[TC]}$ formula over $\mathcal{V}$ with the indexed set of n free variables $\{x_i\}_i$. We denote the space of logical structures, $\mathsf{FOL[TC]}$ formulas, and predicate transformers over vocabulary $\mathcal{V}$ as $\mathsf{STRUCTS}[\mathcal{V}]$, $\mathsf{FORMS}[\mathcal{V}]$, and $T_{\mathcal{V}} = \mathbb{B} \times \mathsf{FORMS}[\mathcal{V}] \times (\mathcal{V} \to \mathsf{FORMS}[\mathcal{V}])$. For structures S_0 and S_1, let the *union* of S_0 and S_1, denoted $S_0 \cup S_1$ be the structure whose universe is the union of the universes of S_0 and S_1, and whose relations are the unions of the relations of S_0 and S_1.

For vocabulary $\mathcal{V}$, a predicate transformer $\rho = (\nu, \varphi, T) \in T[\mathcal{V}]$ defines a partial function over logical structures $\tau_\rho : \mathsf{STRUCTS}[\mathcal{V}] \to \mathsf{STRUCTS}[\mathcal{V}]$, as follows. Let $S = \langle U, \iota \rangle$ be a logical structure with universe U and interpretation ι of the relational symbols in V in universe U. $\tau_\rho(S) = \langle U', \iota' \rangle$ is the logical structure in which

- The universe U' contains a fresh individual not in U if and only if the Boolean flag ν designates that τ introduces a fresh individual. I.e., if $\nu = \mathsf{False}$, then $U' = U$, and if $\nu = \mathsf{True}$, then $U' = U \cup \{o\}$ for some $o \notin U$.

- Only the fresh individual satisfies the unary relation new. I.e., for an individual $o \in U'$, $\mathsf{new}(o)$ holds only if $o \in U' \setminus U$. For every other predicate $P \in \mathcal{V}$ of arity k, P holds for a k-tuple of individuals I if I satisfies the update-formula bound to P in τ_ρ. I.e, I satisfies $T(P)$ in a variable context that maps each free variable x_i to the ith component of I.

A structure program is a control-flow graph over a space of operations that transform logical structures.

Definition 7. Let a *structure program* be a six-tuple $(N, \iota, \mathtt{O}, E, \mathcal{V}, \tau)$, where:

- N is the set of *control nodes*.
- $\iota \in N$ is the *initial control node*.
- $\mathtt{O}$ is a space of operations.
- $E \subseteq N \times \mathtt{O} \times N$ is a set of control edges annotated with operations.
- $\mathcal{V}$ is a first-order vocabulary.
- $\tau : \mathtt{O} \to T[V]$ are the predicate transformers of P.

A structure program P defines a transition system over logical structures $T_P = (Q, \Sigma, \to)$, where:

- A state is a control-flow node paired with a structure over $\mathcal{V}$: $Q = N_P \times \mathsf{STRUCTS}[\mathcal{V}]$.
- The actions are the space of operations: $\Sigma = \mathtt{O}_P$.
- The transition relation $\to \subseteq Q \times \Sigma \times Q$ updates the control location of a state according to the control edges of P, and updates the structure of a state according to the predicate transformers of P's edges. For control locations $n, n' \in N$, operation $o \in \mathtt{O}$, and logical structures $S \in \mathsf{STRUCTS}[V]$, if $(n, o, n') \in E$, then $((n, S), o, (n', \tau[o](S))) \in \to$.

We denote the class of structure programs over vocabulary V and operations $\mathtt{O}$ as $\mathsf{StructProgs}(V, \mathtt{O})$.

For each structure program P, a solution to the *structure-abstraction program* $\mathsf{STRUCT\_ABS}(P)$ is a pair $(P^{\#}, N_A)$, where A is a finite abstraction of P, and $N_A : \mathsf{LOC}_{P^{\#}} \to N_P$ maps each abstract state $q^{\#} \in Q_{P^{\#}}$ to the control-location in $P^{\#}$ of all states abstracted by $q^{\#}$.

A structure simulation is a simulation in which the state space of the simulating transition system is a space of logical structures.

Definition 8. For transition system $(\mathsf{Stores}, \mathtt{O}, \rightarrow)$, a *structure simulation* is a triple consisting of:

- A relational vocabulary V.
- A class of predicate transformers for V-structures over the operations $\mathtt{O}$.
- A function $\mathsf{StoreToStruct} : \mathsf{Stores} \rightarrow \mathsf{STRUCTS}[V]$ that defines a simulation from the transition system $(\mathsf{Stores}, \mathtt{O}, \rightarrow)$ to the transition system $(\mathsf{STRUCTS}[V], \mathtt{O}, T)$.

Part I

Weaving for a Capability System

In this part of the dissertation, we introduce the weaving problem for the Capsicum capability system, and describe our technique for the solving the weaving problem. In Chapter 3, we review a simplified version of Capsicum with which we describe the Capsicum weaving problem. In Chapter 4, we illustrate the Capsicum weaving problem and our approach by example. In Chapter 5, we explain our approach in technical detail. In Chapter 6, we present case studies of applying our Capsicum policy weaver to weave programs for Capsicum.

3 The Capsicum Capability System

Capsicum [58] is a capability system that extends the system objects and operations of the UNIX operating system. Capsicum provides primitives that an application can invoke to restrict under what conditions application modules can open descriptors to resources (i.e., files and network connections) and perform operations on descriptors (e.g., read, write, and seek). In particular, the Capsicum kernel maps each process $\mathtt{p}$ to a Boolean flag that denotes whether $\mathtt{p}$ has *ambient authority*, and maps each process $\mathtt{p}$ and descriptor $\mathtt{d}$ to the set of operations that $\mathtt{p}$ can perform on $\mathtt{d}$; if $\mathtt{p}$ can perform operation $\mathtt{o}$ on $\mathtt{d}$, then $\mathtt{p}$ holds the *access right* for $\mathtt{o}$ on $\mathtt{d}$. A descriptor paired with an access right is a *capability*.

If a process $\mathtt{p}$ requests to open a descriptor $\mathtt{d}$ to a resource or perform an operation $\mathtt{o}$ on $\mathtt{d}$, the Capsicum kernel only grants the request if $\mathtt{p}$, $\mathtt{o}$, and $\mathtt{d}$ satisfy particular constraints. In particular, $\mathtt{p}$ can only open a new descriptor if $\mathtt{p}$ holds ambient authority. $\mathtt{p}$ can only perform operation $\mathtt{o}$ on descriptor $\mathtt{d}$ if $\mathtt{p}$ holds the capability $(\mathtt{d}, \mathtt{o})$.

A process $\mathtt{p}$ can allow another process $\mathtt{q}$ (which in practice, typically executes an untrusted program module) to perform fixed operations with authority or capabilities held by $\mathtt{p}$ but not held by $\mathtt{q}$ via a *remote-procedure call (RPC) service*. In particular, if $\mathtt{p}$ holds capabilities C, then $\mathtt{p}$ can create an RPC service $\mathtt{s}$ consisting of (1) a program module $\mathtt{M}$, (2) a Boolean flag A denoting whether $\mathtt{s}$ holds ambient authority, which may only hold if $\mathtt{p}$

holds ambient authority, and (3) a set of capabilities $\mathtt{C}' \subseteq \mathtt{C}$. A process $\mathtt{q}$ distinct from $\mathtt{p}$ can invoke $\mathtt{s}$ to execute $\mathtt{M}$ with (1) ambient authority if A holds and, (ii) with capabilities $\mathtt{C}'$.

The actual implementation of Capsicum is more complex than the system described above. In particular,

- Capsicum extends the semantics of each Unix operation on processes, in particular forking a process, to affect how ambient authority and capabilities are propagated across processes.
- Capsicum maps some operations to a *set* of access rights; a process $\mathtt{p}$ can only perform an operation $\mathtt{o}$ on a descriptor $\mathtt{d}$ if $\mathtt{p}$ holds *each* access right for $\mathtt{d}$ required to perform $\mathtt{o}$.
- A process on Capsicum can relinquish ambient authority or relinquish a capability at any point in its execution, not only when calling an RPC service.

To simplify the presentation of our approach, we do not consider programs whose state consists of multiple processes and RPC services, but consider programs whose state instead consists of a *memory* and RPC services, each of which may hold ambient authority and capabilities. We omit descriptions of such features to simplify the presentation of the approach described in this dissertation, but such features are supported by the actual implementation of the approach.

4 Overview

In this section, we illustrate by example the capability-instrumentation problem, and our technique to solve the problem. In §4.1, we introduce a version of the `gzip` compression utility written in a simple language with capability features. In §4.2, we present a policy in our policy language that describes the security and functionality requirements of `gzip`. In §4.3, we describe an instrumented version of `gzip` that satisfies the policy and that is generated by our technique. We also summarize the key challenges for instrumenting `gzip`, and how they are addressed by our technique.

4.1 `gzip`: a compression utility

Fig. 4.1 contains pseudocode for a version of the `gzip` compression utility written in `cap`, a simple language with capability features. For now, ignore the lines highlighted in gray: these are the instrumentation code introduced by our technique, and are described in §4.3.

`gzip` consists of three modules: an entry module `gzip`, a driver-loop module `loop`, and a compression module `cmp`. `gzip` immediately transfers control (i.e, "jumps") to `loop` (line `L0`). `loop` iterates over the sequence of input filenames (line `L1`). In each iteration, `loop` loads the next input file name (`L1-L2`), opens a new descriptor `i` for input and binds `i` to descriptor variable `in` (line `L3`), opens a new descriptor `o` for output and binds `o` to descriptor variable `out` (line `L4`), and then jumps to `cmp`.

When `cmp` executes correctly, it reads uncompressed data from `i`, com-

```
gzip:
// Create RPC service for loop.
C0a: s0 := create_service(loop, mem_amb(), mem_caps());
C0b: set_mod_service(loop, s0);
L0:  jump loop;

// Create input, output descriptors from the next filename.
loop:
L1:  has_next := has_next_file();
L2:  br has_next ? L3 : L6;
L3:  in = open(IN, next_in_path());
L4:  out = open(OUT, next_out_path());
// Create a service with which to execute the compression  module.
C5a: s1 := create_service(cmp, no_amb(),
             rem(in, WR, rem(out, RD, mem_caps())));
C5b: set_mod_service(cmp, s1);
L5:  jump cmp;
```

Figure 4.1: `gzip`: a compression utility in the capability language `cap`. `gzip` consists of an entry module (`gzip`), a loop driver (`loop`), and a compression module (`cmp`, not defined). Operations on capabilities, which are generated by our technique, and accompanying comments are highlighted with a gray background. Our technique does not actually generate comments that accompany capability operations.

presses the data read, writes the compressed data to o, and then jumps to `loop`. However, in previous versions of the UNIX utility `gzip`, `cmp` contained vulnerabilities that an attacker who could control the inputs to `gzip` could exploit to execute arbitrary program operations within `cmp`. To represent the fact that the possible executions of `cmp` are unknown, Fig. 4.1 contains no definition of `cmp`, and we refer to `cmp` as the *environment* of `gzip`.

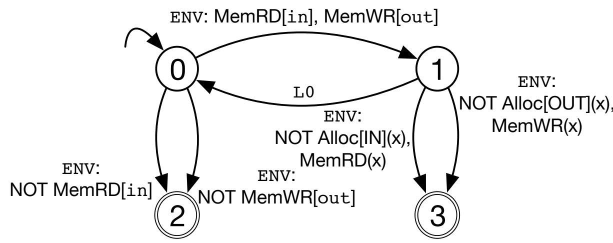

Figure 4.2: `gzip_pol`: a capability policy for `gzip`. Each transition is annotated with a description of a set of `cap` states; the semantics of each description is described in Ex. 1.

4.2 `gzip_pol`: a capability policy for `gzip`

Our goal is to automatically instrument `gzip` to use capability operations so that the environment of `gzip` can carry out only necessary operations on a restricted set of descriptors. In particular:

- When `loop` jumps to `cmp`, memory should hold the (1) the `RD` access right for the descriptor stored in variable `in` and (2) the `WR` access right for the descriptor stored in variable `out`.

- When the program executes `cmp`, memory should only hold the `RD` access right for descriptors allocated at `IN` and the `WR` access right for descriptors allocated at `OUT`.

In §5.3.2, we define a language of policies for `cap` programs as *capability policies*. A capability policy is a finite-state machine over an alphabet of conditions on the capabilities held by a program. A trace of `cap` states `t` violates a capability policy C if the states of `t` satisfy a trace of conditions that is accepted by C.

Example 1. Fig. 4.2 contains a capability policy `gzip_pol` that explicitly expresses the requirements for the modules of `gzip`. `gzip_pol` is an automaton over an alphabet in which each symbol represents a set of `cap` states. Each symbol is represented as (1) the control location of each state in the set and (2) a (potentially-empty) sequence of comma-separated clauses. Each clause is a literal over a predicate that describes whether memory can read from or write to particular objects. Objects are represented as either `in` or `out`, which represent the objects stored in variables `in` or `out`, or the variable `x`, which is existentially quantified over all objects. A comma-separated sequence of clauses represents the conjunction of all clauses in the sequence.

Each execution of `gzip` begins in the initial state 0, and remains in 0 until it executes a state in the environment of `gzip`. The execution satisfies `gzip_pol` if it completes `loop` in a state in which the environment can read from the descriptor stored in `in` and can write to the descriptor stored in `out`, on which actions `gzip_pol` transitions from state 0 to state 1. Otherwise, the execution violates `gzip_pol`; i.e., `gzip_pol` transitions from state 0 to state 2. While the execution is in the environment, if the program can read from a descriptor not allocated at `IN` or write to a descriptor not allocated at `OUT`, then the execution violates `gzip_pol` (i.e., `gzip_pol` transitions from state 1 to state 3). Otherwise, when the execution reenters `loop` (i.e., executes the operation at control location `L0`), `gzip_pol` transitions from state 1 to state 0.

4.3 Instrumenting `gzip`

The complete `gzip` in Fig. 4.1, including the capability operations highlighted in gray, satisfies the capability policy `gzip_pol`. The semantics of the capability operations used by `gzip` are described briefly in Chapter 3, and in detail in §5.1. In lines `C0a-C0b`, `gzip` binds to `loop` an RPC service s_0 with ambient authority, and jumps to `loop`, updating its ambient authority and capabilities to those of s_0. In lines `C5a-C5b` `gzip` binds to `cmp` an RPC service without (1) ambient authority, (2) the `WR` access right for the descriptor stored in `in`, and (3) the `RD` access right for the descriptor stored in `out`. `gzip` then jumps to the undefined module `cmp` (represented as the control location ENV, which denotes the environment). The result of executing the instrumented capability operations is that program memory can hold only the capabilities to read from the descriptors allocated at allocation site `IN` and write to descriptors allocated at allocation site `OUT`, and cannot obtain any other capabilities. `gzip_pol` thus remains in state 1 while the environment executes, and remains in state 0 when a module of `gzip` of executes.

The instrumentation algorithm implemented in our policy weaver for Capsicum, `capweave`, can take as input the version of `gzip` that executes no capability operations (i.e., `gzip` in Fig. 4.1 with the capability operations in gray removed), and the capability policy `gzip_pol`, and can automatically instrument `gzip` to execute the capability operations depicted in Fig. 4.1. The primary programming challenge addressed by `capweave` in the context of `gzip` is to model soundly all possible executions of the untrusted `cmp` module of `gzip`, which may include (1) cooperating executions in which `cmp` attempts to only read from the descriptor stored in `in` and write to the descriptor stored in `out` and (2) malicious executions in which `cmp` attempts to open arbitrary descriptors and perform arbitrary operations on the descriptors that it holds. The technique applied by `capweave` to address this challenge is: (1) define a program `gzip`′ whose executions are the executions of multiple possible instrumentations of `gzip`; (2) construct a finite over-approximation `gzip`′# of the language of executions of `gzip`′ that violate `gzip_pol`; (3) use `gzip`′# to construct a game G (defined in §2.2) for which each play models an execution of `gzip`′, and each Attacker-winning play models an execution of `gzip`′# that may result in a violation of `gzip_pol`; (4) try to find a winning Defender strategy D of G; (5) from D, instrument `gzip` to execute capability operations throughout each execution e that correspond to the actions chosen by D through the

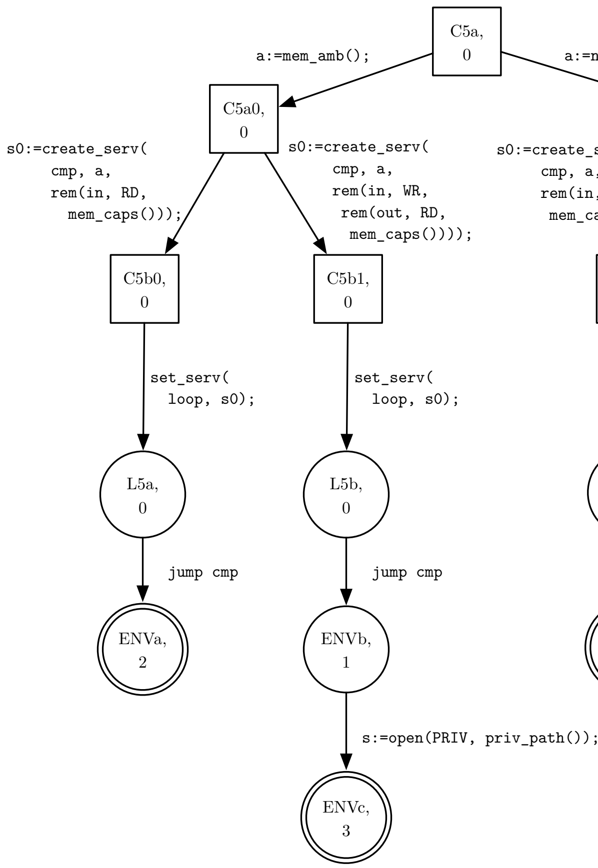

Figure 4.3: Fragment of the game modeling the problem of instrumenting `gzip` to satisfy `gzip_pol` immediately before executing control location `L5`. Defender states are depicted as squares, Attacker states are depicted as circles, and Attacker-winning states are depicted as doubled circles.

play that models *e*.

A fragment of the game constructed by `capweave` to weave `gzip` to satisfy `gzip_pol` is depicted in Fig. 4.3. Each game state consists of a pair of a `gzip`′ state and a `gzip_pol` state, and is depicted in Fig. 4.3 as a node annotated with (1) the control location of the state of `gzip` extended with a distinguishing extension character in the range ′a′—′e′, and (2) the state of `gzip_pol` that it models. States in which `gzip`′ executes `cmp` are annotated with a control location of the form ENVi to denote that the environment of gzip′ executes. Each edge between states is annotated with the cap operation on which the game transitions. Variations of the capability operation to create an RPC service at C5a in Fig. 4.1 are modeled in Fig. 4.3 as sequences of capability operations chosen at control location C5a, followed by either control location C5a0 or C5a1.

Capweave actually constructs a game from a finite over-approximation gzip′# of the language of executions of gzip′. Such an abstraction will, for example, merge "similar" states that, e.g., differ only in the *number* of descriptors allocated at each allocation site, but not in the capabilities assigned to each descriptor. To simplify the discussion, in Fig. 4.3, we have depicted a fragment of the game related to the one that would actually be used, in this case constructed directly from gzip′.

The game fragment in Fig. 4.3 depicts four plays that start from a state "C5a, 0", which models an execution at control location C5a that has driven gzip_pol to state 0. The game states starting from state "C5a, 0" model states reached after gzip executes the operations in Fig. 4.1, lines C0a–L4.

Along each play from "C5a, 0", the Defender chooses a sequence of actions that model an instrumentation of gzip that chooses (1) an ambient authority and (2) a set of capabilities with which to create the RPC service that it invokes to execute cmp. The ambient authority and capabilities chosen in each play are distinct. On the play from state "C5a, 0" to state "ENVa, 2", the Defender chooses actions that model an instrumentation that executes cmp with the ambient authority and capabilities of memory, without the RD access right for the descriptor stored in descriptor variable out. "ENVa, 2" is an Attacker-winning state because it models a program state in which memory does not hold the RD access right for the descriptor stored in in when gzip completes execution of loop, driving gzip_pol to the accepting state 2.

On the play from "C5a, 0" to "ENVb, 1" the Defender chooses actions that model an instrumentation that executes `cmp` with the ambient authority held by memory, the `RD` access right for the descriptor stored in `in`, and the `WR` access right for the descriptor stored in `out`. "ENVb, 1" is not an Attacker-winning state, but the Attacker may transition from "ENVb, 1" to the state "ENVc, 3" by opening a new file descriptor with arbitrary access rights. "ENVc, 3" is an Attacker-winning state because it models a program state in which the program executes an untrusted module and memory holds a capability not opened at allocation sites `IN` or `OUT`, driving `gzip_pol` to the accepting state 3.

On the play from "C5a, 0" to "ENVd, 2", the Defender chooses actions that model an instrumentation that executes `cmp` without ambient authority, and with the capabilities of memory, except for the `RD` access right for the descriptor stored in descriptor variable `in`. "ENVd" is an Attacker-winning state for a reason analogous to the reason that "ENVa, 2" is an Attacker-winning state.

On the play from "C5a, 0" to "ENVe, 1", the Defender chooses actions that model an instrumentation that executes `cmp` without ambient authority, and with the capabilities held by memory, except for the `RD` access right for the descriptor stored in descriptor variable `out` and the `WR` access right for the descriptor stored in descriptor variable `in`. "ENVe, 1" is not an Attacker winning state, and the Attacker cannot choose any sequence of actions from "ENVe, 1" that will drive the game to an Attacker-winning state. The trace of actions from "C5a, 0" to "ENVe, 1" is the trace of each execution of the instrumented `gzip` in Fig. 4.1 from control location to `C5a` to `L5`.

5 Technical Approach

In this chapter, we describe the technical details of our approach to solving the Capsicum weaving problem. In §5.1, we define the syntax and semantics of a capability-based programming language `cap`. In §5.2, we formulate the conditions under which one `cap` program is a valid instrumentation of another `cap` program. In §5.3.2, we define a language of policies for `cap` programs. In §5.4, we define the problem of instrumenting a `cap` program to satisfy a capability policy. In §5.5, we describe our technique for solving the instrumentation problem.

5.1 `cap`: a language of capability programs

In this section, we first define a core language `capcore` of imperative programs without capability features. We then use `capcore` to define the syntax (§5.1.2) and semantics (§5.1.3) of the capability language, `cap`.

5.1.1 `capcore`: a core language

A `capcore` program opens descriptors to a filesystem and performs operations on descriptors and values. The syntax of a `capcore` program is given in Fig. 5.1, and is defined over fixed finite sets of module symbols (MODSYMS), control locations (LOC), allocation sites (ALLOCS), data variables (DATAVARS), and descriptor variables (DESCVARS). A `capcore` program is a sequence of bindings from a module symbol to a sequence

$$\mathsf{Prog} := (\mathsf{MODSYMS} : (\mathsf{LOC} : \mathtt{Op})^*)^* \tag{5.1}$$
$$\mathtt{Op} := \mathsf{DATAVARS} := \mathsf{DATAOP}(\overline{\mathsf{DATAVARS}}) \tag{5.2}$$
$$\mid \mathtt{jump}\ \mathsf{MODSYMS} \tag{5.3}$$
$$\mid \mathsf{DATAVARS}\ ?\ \mathsf{LOC} : \mathsf{LOC} \tag{5.4}$$
$$\mid \mathsf{DESCVARS} := \mathtt{open}(\mathsf{ALLOCS}, \mathsf{DATAVARS}) \tag{5.5}$$
$$\mid \mathsf{DATAVARS} := \mathsf{DESCOP}(\mathsf{DESCVARS}) \tag{5.6}$$

Figure 5.1: Syntax of `capcore`, a language of programs that operate on descriptors.

of operations, each annotated with a control location (Eqn. (5.1)); each location may annotate at most one operation. An operation may compute a value from values in data variables (Eqn. (5.2)), transfer control to a particular control location (Eqn. (5.3)), change control based on the value in a data variable (Eqn. (5.4)), open a descriptor to a system object (Eqn. (5.5)), or perform an operation on a descriptor (Eqn. (5.6)).

A `capcore` program $\mathtt{P}$ can be represented as an annotated control-flow graph over control locations and operations. Each `capcore` program $\mathtt{P}$ defines (1) an initial module $\mathsf{M}_0^{\mathtt{P}}$ (by convention, the first module in $\mathtt{P}$), (2) a function $\mathsf{LocMod}_{\mathtt{P}} : \mathsf{LOC} \to \mathsf{MODSYMS}$ from each control location $\mathtt{L}$ to the module that contains an operation annotated with $\mathtt{L}$, (3) a function $\mathsf{ModInit}_{\mathtt{P}} : \mathsf{MODSYMS} \to \mathsf{LOC}$ from each module $\mathtt{M}$ to the location that annotates the initial operation of $\mathtt{M}$ in $\mathtt{P}$, and (4) a control-flow graph $(\mathsf{LOC}, \mathtt{E}_{\mathtt{P}})$. The control nodes are the control locations LOC, and the control edges $\mathtt{E}_{\mathtt{P}} \subseteq \mathsf{LOC} \times \mathtt{Op} \times \mathsf{LOC}$ are the control-flow edges defined by each operation that is not a jump (i.e., each *intra-module operation*).

$$\text{intra}\frac{(\mathtt{L},\mathtt{o},\mathtt{L}') \in \mathsf{E}'_{\mathsf{P}} \qquad \langle \sigma, \mathtt{o} \rangle \rightarrow_{\mathtt{cc}} \sigma'}{\langle (\mathtt{L}, \sigma), \mathtt{o} \rangle \rightarrow_{\mathsf{P}} (\mathtt{L}', \sigma')}$$

$$\text{jump-P}\frac{\mathtt{L}' = \mathsf{ModInit}_{\mathsf{P}}(\mathtt{M})}{\langle (\mathtt{L}, \sigma), \mathtt{jump\ M} \rangle \rightarrow_{\mathsf{P}} (\mathtt{L}', \sigma)}$$

$$\text{jump-non-P}\frac{\mathsf{ModInit}_{\mathsf{P}}(\mathtt{M}) =\uparrow}{\langle (\mathtt{L}, \sigma), \mathtt{jump\ M} \rangle \rightarrow_{\mathsf{P}} (\mathsf{ENV}, \sigma)}$$

Figure 5.2: Inference rules that define the transition relation $\rightarrow_{\mathsf{P}}$ of a `capcore` program P.

`capcore` semantics

A `capcore` program P defines a transition relation over `capcore` states. Let the set of control locations LOC be the set of control locations, which contains a distinguished control location ENV that models the environment of P. Each `capcore` state consists of a control location in $\mathsf{Locs}_{\mathsf{NT}}$ and a value store, which is a valuation of data variables and a set of descriptors. Let D^* be an infinite universe of descriptors. A *value store* $(\mathsf{V}, \mathsf{F}, \mathsf{D}, \mathsf{V}_{\mathsf{D}}, \alpha)$ is a five-tuple of (1) a valuation of data variables $\mathsf{V} : \mathsf{DATAVARS} \rightarrow \mathbb{Z}$, (2) the value stored in the filesystem $\mathsf{F} \in \mathbb{Z}$, (3) a set of descriptors $\mathsf{D} \subseteq \mathsf{D}^*$, (4) a valuation of descriptor variables $\mathsf{V}_{\mathsf{D}} : \mathsf{DESCVARS} \rightarrow \mathsf{D}$, and (5) a map from each descriptor to its allocation site $\alpha : \mathsf{D} \rightarrow \mathsf{ALLOCS}$. We denote the components of a value store σ as V^{σ}, F^{σ}, D^{σ}, $\mathsf{V}^{\sigma}_{\mathsf{D}}$ and α^{σ}, respectively, and denote the space of all value stores as $\mathsf{CapCoreStores}$. A `capcore` *state* $(\mathtt{L}, \sigma)$ is a control location $\mathtt{L}$ and a value store σ. We denote the space of all `capcore` states as $\mathsf{Q}_{\mathsf{CC}} = \mathsf{Locs}_{\mathsf{NT}} \times \mathsf{CapCoreStores}$.

The transition relation $\rightarrow_{\mathsf{P}} \subseteq \mathsf{Q}_{\mathsf{CC}} \times \mathtt{Op} \times \mathsf{Q}_{\mathsf{CC}}$ of a `capcore` program is defined by the control structure of the modules of P and a transition relation over stores. Let $\mathsf{E}'_{\mathsf{P}} \subseteq \mathsf{Locs}_{\mathsf{NT}} \times \mathtt{Op} \times \mathsf{Locs}_{\mathsf{NT}}$ be the control-flow edges E_{P} extended to contain an edge from ENV to itself on each intra-

$$\mathtt{Op} := \mathsf{SVAR} \coloneqq \mathtt{create\_serv}(\mathsf{LOC}, \mathsf{AuthExpr}, \mathsf{CapsExpr}) \tag{5.7}$$
$$\mid \mathtt{set\_serv}(\mathsf{LOC}, \mathsf{SVAR}) \tag{5.8}$$
$$\mathsf{AuthExpr} := \mathtt{mem\_auth}() \tag{5.9}$$
$$\mid \mathtt{no\_amb}() \tag{5.10}$$
$$\mathsf{CapsExpr} := \mathtt{mem\_caps}() \tag{5.11}$$
$$\mid \mathtt{rem}(\mathsf{DESCVARS}, \mathsf{DESCOP}, \mathsf{CapsExpr}) \tag{5.12}$$

Figure 5.3: Capability operations of cap that extend Op.

module operation, and let $\rightarrow_{\mathtt{cc}} \subseteq \mathsf{CapCoreStores} \times \mathtt{Op} \times \mathsf{CapCoreStores}$ be a transition relation over capcore stores. Inference rules that define $\rightarrow_{\mathtt{P}}$ for each operation $\mathtt{o}$ using $\mathtt{E}'_{\mathtt{P}}$ and $\rightarrow_{\mathtt{cc}}$ are given in Fig. 5.2. If $\mathtt{o}$ is an intra-module operation (Rule intra), then pre-state $(\mathtt{L}, \sigma)$ transitions to post-state $(\mathtt{L}', \sigma')$ on $\mathtt{o}$ if $\mathtt{L}'$ is a control successor of $\mathtt{L}$ on $\mathtt{o}$ ($(\mathtt{L}, \mathtt{o}, \mathtt{L}') \in \mathtt{E}_{\mathtt{P}}$) and pre-store σ transitions to post-store σ' on $\mathtt{o}$ ($\langle \sigma, \mathtt{o} \rangle \rightarrow_{\mathtt{cc}} \sigma'$). The definition of $\rightarrow_{\mathtt{cc}}$ is straightforward from the informal definitions of each intra-module operation, and we omit a full description.

If $\mathtt{o}$ is a jump operation whose target is a module $\mathtt{M}$ of $\mathtt{P}$ (Rule jump-P), then pre-state $(\mathtt{L}, \sigma)$ transitions to a state whose control location is the initial location of $\mathtt{M}$, and whose store is σ. If $\mathtt{o}$ is a jump operation whose target is a module $\mathtt{M}$ not in $\mathtt{P}$ (Rule jump-non-P, where $\mathsf{ModInit}_{\mathtt{P}}(\mathtt{M}) = \uparrow$ denotes that $\mathsf{ModInit}_{\mathtt{P}}$ is not defined at $\mathtt{M}$), then pre-state $(\mathtt{L}, \sigma)$ transitions to a state whose control location is ENV, and whose store is σ.

5.1.2 cap syntax

A cap program is a capcore program whose operations are the capcore operations extended with a set of capability operations, given in Fig. 5.3; we refer to the space of all operations of cap programs as $\mathsf{O}_{\mathtt{c}}$. A capability operation may create an RPC service using a control location, ambient-authority expression, and a capability expression (Eqn. (5.7)) or set the RPC service for a location (Eqn. (5.8)).

An ambient-authority expression may be either `mem_auth()` (Eqn. (5.9)), which evaluates to the ambient authority of memory, or `no_amb()` (Eqn. (5.10)), which evaluates to no ambient authority. A capability expression may be either `mem_caps()` (Eqn. (5.11)), which evaluates to the capabilities of memory, or $\texttt{rem}(\mathsf{E}, \texttt{d}, \texttt{R})$, which evaluates to the value of E without the capability consisting of the descriptor bound to `d` and access right `R`.

5.1.3 `cap` semantics

A `cap` program defines a transition relation over `cap` stores.

`cap` states

A `cap` program defines a transition relation over the space of `cap` states, denoted $\mathsf{CapStates}$. A `cap` state consists of a control location, a value store, and a *capability store*. Let S^* be an infinite universe of *service identifiers*, and let SVAR be a set of *service variables*. Let a capability be a descriptor paired with a descriptor operation; we denote the space of capabilities as $\mathsf{Caps} = \mathsf{D} \times \mathsf{DESCOP}$. A capability store (A, C, S, V_S, R, μ) is a six-tuple of (1) an *ambient-authority flag* $A \in \mathbb{B}$; (2) *capabilities* $C \subseteq \mathsf{Caps}$; (3) *service identifiers* $S \subseteq \mathsf{S}^*$, (4) a *valuation of service variables* $V_S : \mathsf{SVAR} \hookrightarrow S$, (5) a *service-identifier map* $R : S \hookrightarrow \mathsf{MODSYMS} \times \mathbb{B} \times \mathsf{Caps}$ from each store service identifier to its module, ambient-authority flag, and capabilities, and (6) a *module-service map* $\mu : \mathsf{MODSYMS} \hookrightarrow \mathbb{B} \times \mathsf{Caps}$. We denote the ambient-authority flag, capabilities, service identifiers, valuation of service variables, service-identifier map, and module-service map of a capability store κ as A^κ, C^κ, S^κ, ${V_S}^\kappa$, R^κ, and μ^κ, respectively. We denote the space of all capability stores as $\mathsf{CapStores}$. A *`cap` store* is a `capcore` store paired with

$$\text{open}\;\frac{\langle V, \mathtt{d} \coloneqq \mathtt{open}(\mathtt{S}, \mathtt{x})\rangle \to_{\mathsf{cc}} V' \qquad A^{\kappa} = \mathsf{True}}{\langle (V, \kappa), \mathtt{d} \coloneqq \mathtt{open}(\mathtt{S}, \mathtt{x})\rangle \to_{\mathsf{c}} (V', \kappa)}$$

$$\text{desc-op}\;\frac{\langle V, x \coloneqq \mathtt{op}(X)\rangle \to_{\mathsf{cc}} V' \qquad \mathtt{d} = D^{V}(\mathtt{op}(\mathtt{X})) \qquad (\mathtt{d}, \mathtt{op}(\mathtt{X})) \in C^{\kappa}}{\langle (V, \kappa), x \coloneqq \mathtt{op}(\mathtt{d})\rangle \to_{\mathsf{c}} (V', \kappa)}$$

$$\text{jump-rpc}\;\frac{\mu^{\kappa}(\mathtt{M}) = (A', C') \qquad \kappa' = (A', C', S^{\kappa}, {V_S}^{\kappa}, R^{\kappa}, \mu^{\kappa})}{\langle (V, \kappa), \mathtt{jump}\ \mathtt{M}\rangle \to_{\mathsf{c}} (V, \kappa')}$$

$$\text{create-rpc}\;\frac{\begin{array}{ccc} \langle \mathtt{E_A}, \kappa\rangle \to_{\mathsf{c}}^{\mathsf{Amb}} A' & & A' \implies A^{\kappa} \\ \langle \mathtt{E_C}, \kappa\rangle \to_{\mathsf{c}}^{\mathsf{cap}} C' & C' \subseteq C^{\kappa} & s \notin S^{\kappa} \\ \multicolumn{3}{c}{\kappa' = (A^{\kappa}, C^{\kappa}, s \cup \{S^{\kappa}\}, {V_S}^{\kappa}[\mathtt{s} \mapsto s], R^{\kappa}[s \mapsto (\mathtt{M}, A', C')], \mu^{\kappa})} \end{array}}{\langle (V, \kappa), \mathtt{s} \coloneqq \mathtt{create\_serv}(\mathtt{M}, \mathtt{E_A}, \mathtt{E_C})\rangle \to_{\mathsf{c}} (V, \kappa')}$$

$$\text{set-rpc}\;\frac{\begin{array}{c} R^{\kappa}({V_S}^{\kappa}(\mathtt{s})) = (\mathtt{M}, A', C') \qquad \mu' = \mu^{\kappa}[M \mapsto (A', C')] \\ \kappa' = (A^{\kappa}, C^{\kappa}, S^{\kappa}, {V_S}^{\kappa}, R^{\kappa}, \mu') \end{array}}{\langle (V, \kappa), \mathtt{set\_serv}(\mathtt{M}, \mathtt{s})\rangle \to_{\mathsf{c}} (V, \kappa')}$$

Figure 5.4: Inference rules that define the transition relation $\to_{\mathsf{c}}$ over cap stores.

a capability store; i.e., capStores $=$ CapCoreStores $\times$ CapStores. A cap state is a control location paired with a cap store; i.e., CapStates $=$ LOC $\times$ capStores.

cap **transitions**

A cap program P defines a transition relation $\to_{\mathsf{P}} \subseteq$ CapStates $\times$ O_{c} $\times$ CapStates. $\to_{\mathsf{P}}$ is defined by semantic inference rules identical to the rules given in Fig. 5.2, using a transition relation $\to_{\mathsf{c}} \subseteq$ capStores $\times$ O_{c} $\times$ capStores over cap stores in place of the transition relation $\to_{\mathsf{cc}}$ used in Fig. 5.2 to define the semantics of capcore.

Inference rules that define $\to_{\mathsf{c}}$ for a selection of cap operations are given in Fig. 5.4. For cap store $(V, \kappa) \in$ capStores and operation $\mathtt{o} \in O_{\mathsf{c}}$:

- If $\mathtt{o}$ is an operation $\mathtt{d} \coloneqq \mathtt{open}(\mathtt{S}, \mathtt{x})$ that opens a descriptor (Rule open),

V transitions to value store V' under the transition relation for `capcore` operations ($V \rightarrow_{cc} V'$), and κ holds ambient authority (A^{κ}), then (V, κ) transitions to (V', κ).

- If o is an operation $x \coloneqq \mathtt{op}(\mathtt{d})$ that operates on a descriptor (Rule desc-op), V transitions to value store V' under the transition relation for `capcore` operations ($V \rightarrow_{cc} V'$), and κ holds the capability of the descriptor bound to $\mathtt{d}$ paired with o ($d = D^{V}(\mathtt{d})$ and $(d, o) \in C^{\kappa}$), then (V, κ) transitions to (V', κ).

- If o is an operation `jump M` and an RPC service is defined for `M` in κ (Rule jump-rpc), then (V, κ) transitions to a `cap` store with value store V and a capability store that contains the ambient authority and capabilities of the service bound to `M` in κ.

- If o is an operation $\mathtt{s} \coloneqq \mathtt{create\_serv}(\mathtt{L}, \mathtt{E_A}, \mathtt{E_C})$ where $\mathtt{E_A}$ is an ambient-authority expression and $\mathtt{E_C}$ is a capability expression (Rule create-rpc), $\mathtt{E_A}$ evaluates to ambient-authority A' in κ ($\langle \kappa, \mathtt{E_A} \rangle \rightarrow_{c}^{\mathsf{Amb}} A'$), A' implies the ambient authority of memory ($A' \implies A^{\kappa}$), $\mathtt{E_C}$ evaluates to C' in κ ($\langle \kappa, \mathtt{E_C} \rangle \rightarrow_{c}^{\mathsf{cap}} C'$), and C' is contained by the capabilities of memory in κ ($C' \subseteq C^{\kappa}$), then (V, κ) transitions to a `cap` store whose value store is V and whose capability store is κ updated to bind `s` to a service consisting of `M`, A', and C'. The definitions of the evaluation relation for ambient-authority expressions, $\rightarrow_{c}^{\mathsf{Amb}}$, and the evaluation relation for capability expressions, $\rightarrow_{c}^{\mathsf{cap}}$, are straightforward from their informal definitions. We thus omit a full description.

- If o is an operation $\mathtt{set\_serv}(\mathtt{M}, \mathtt{s})$ where `M` is a program module and `s` is a service-identifier variable (Rule set-rpc), and in pre-store (V, κ), `s` is bound in κ to a service s whose module is `M`, then (V, κ) transitions to a `cap` store whose value store is V and whose capability store is κ updated to bind module `M` to s.

The evaluation relations $\rightarrow_A$ for ambient-authority expressions and $\rightarrow_C$ for capability expressions, used in Rule create-rpc, are straightforward from their informal definitions, and are omitted.

5.1.4 Program runs

A run of a cap program P is a sequence of cap states in which each pair of adjacent states are in the transition relation of P.

Definition 9. For cap program P, let $\Rightarrow_P \subseteq \mathsf{CapStates} \times \mathsf{CapStates}$ contain states $q, q' \in \mathsf{CapStates}$ if there is some operation $o \in \mathtt{Op}$ such that $(q, o) \rightarrow_P q'$. Then a sequence of program states $q_0, q_1, \ldots, q_n$ is a *run of* P if for $0 \leqslant i < n$, $q_i \Rightarrow_P q_{i+1}$.

For module symbol $\mathtt{M} \in \mathsf{MODSYMS}$, program state $q_i = (\mathtt{L}, \sigma) \in \mathsf{CapStates}$ is an M-*state* if $\mathsf{LocMod}_P(\mathtt{L}) = \mathtt{M}$. The union of M states over all module symbols M are the *module states* of P. If r is a run of P, then the subsequence of all module states of r is a *module run* of P. The sequence of all operations executed in a module state of r is a *module trace* of P.

5.2 Instrumentation as capability refinement

We formulate a valid instrumentation of a cap program by adapting definitions of simulation and refinement (defined in Chapter 2, Defn. 2), which are used to define the correctness of semantics-preserving transformations in a compiler [42, 45]. Unfortunately, neither simulation nor refinement formulate our intuitive notion of a valid instrumentation, as demonstrated by gzip (introduced in §4.1).

Example 2. Formulating a valid instrumentation of a cap program P as a simulation of P disallows cap programs that satisfy our intuitive notion of a valid instrumentation. E.g., let no_cap_gzip be a version of gzip with the capability operations removed. Intuitively, we wish to allow gzip as an instrumentation of no_cap_gzip, but gzip is not a simulation of no_cap_gzip from any pair of cap states at the initial location of no_cap_gzip. In particular, consider a state q at the initial location of no_cap_gzip. From q, no_cap_gzip may execute (1) the operations in gzip, then (2) the operations in loop, and then (3) an open operation while executing cmp to transition to a state with a fresh descriptor whose allocation site is neither IN nor OUT. However, no state can execute any number of transitions of gzip to reach a state with such a descriptor.

Conversely, formulating a valid instrumentation of a cap program as a refinement allows an instrumentation of a cap program P to be a trivial program that reproduces none of the behaviors of P. E.g., let halt be a trivial cap program that contains a control location $\mathtt{L_h}$ and no control edges. halt is a refinement of no_cap_gzip.

Intuitively, cap program P′ is an instrumentation of P if P′ can match any sequence of value stores "chosen" over an execution of P, but P′ can choose the capability store paired with each value store in the sequence. We formulate this intuition by defining that under such a condition, P′ is a *capability refinement* of P.

Definition 10. For cap programs P and P′, a *capability-refinement relation* $\sim \subseteq \mathsf{CapStates} \times \mathsf{CapStates}$ is a relation over cap states that satisfies the following conditions:

1. $\sim$ only relates states with equal value stores. I.e., for $q = (\mathtt{L}, (V, C)) \in \mathsf{CapStates}$ and $q' = (\mathtt{L}', (V', C')) \in \mathsf{CapStates}$, if $q \sim q'$, then $V = V'$.
2. If a pair of states (q, q') is in $\sim$, then each successor of q on one step of P is paired with a successor of q' over multiple steps of P′. I.e., for $q_0, q_0' \in \mathsf{CapStates}$ such that $q_0 \sim q_0'$, if $q_0 \Rightarrow_{\mathtt{P}} q_1$, then there is some cap state q_1' such that $q_0' \Rightarrow^*_{\mathtt{P}'} q_1'$ and $q_1 \sim q_1'$.

P' is a *capability refinement* of P if there is a capability-refinement relation $\sim$ for P and P' such that for each cap store $\sigma \in \mathsf{capStores}$, $(\mathsf{ModInit}_{\mathsf{P}}(M_0^{\mathsf{P}}), \sigma) \sim (\mathsf{ModInit}_{\mathsf{P}}(M_0^{\mathsf{P}}), \sigma))$.

Capability refinement may be viewed as a special case of alternating refinement [2].

5.3 Capability policies

In §4.2, we presented a policy for `gzip`. In this section, we define the syntax and semantics of capability policies in general. In §5.3.1, we define a space of conditions on capability stores. In §5.3.2, we use capability-store conditions to define a space of capability policies, and define under which conditions a `cap` program satisfies a capability policy.

5.3.1 Conditions on `cap` stores

Store conditions are formulas over a first-order relational vocabulary V_c whose predicates model properties of `cap` stores. V_c is the union of two vocabularies V_{cc} and V_c'; V_{cc} describes features of `capcore` states.

Definition 11. Each `capcore` store $\sigma \in \mathsf{CapCoreStores}$ defines a model $m_\sigma^{cc} = \langle U_\sigma, \iota_\sigma \rangle$ over a first-order relational vocabulary V_{cc}. The universe U_σ contains the descriptors in σ. The vocabulary V_{cc} and the interpretation ι_σ of each predicate symbol in V_{cc} in the universe U_σ are defined as follows.

- V_{cc} contains the set of descriptor variables DESCVARS. If in σ there is a descriptor d in descriptor variable $\mathtt{d}$, then $\iota_\sigma(\mathtt{d})(d)$ holds.
- For each allocation site $\mathtt{A} \in \mathsf{ALLOCS}$, V_{cc} contains a unary predicate symbol $\mathtt{alloc[A]}$. $\iota_\sigma(\mathtt{alloc[A]})(d)$ holds if descriptor d was allocated by a invocation of the `open` operation at allocation site `A`.

The vocabulary V'_c describes features of `cap` states that are not features of `capcore` states.

Definition 12. Each `cap` store $\sigma \in \mathsf{CapStores}$ defines a model $m^c_\sigma = \langle U_\sigma, \iota_\sigma \rangle$ over a first-order relational vocabulary V'_c. The universe U_σ is the set of descriptors, service identifiers, and the object Mem. The vocabulary V'_c and the interpretation ι^c_σ of each predicate symbol in V'_c in the universe U_σ are defined as follows.

- V'_c contains a unary predicate symbol HasAmb. $\iota_\sigma(\mathsf{HasAmb})(x)$ holds if memory or RPC service x has ambient authority.

- V'_c contains a unary predicate symbol IsMem. IsMem is a singleton relation; in particular, $\iota_\sigma(\mathsf{IsMem})(\mathsf{Mem})$ holds.

- For each operation $\mathtt{o} \in \mathtt{O}$, V'_c contains a binary predicate symbol $\mathsf{HasOp}[\mathtt{o}]$. $\iota_\sigma(\mathsf{HasOp}[\mathtt{o}])(x, d)$ holds if memory or RPC service x holds the access right to operation $\mathtt{o}$ for descriptor d.

- For each module $\mathtt{M} \in \mathsf{MODSYMS}$, V'_c contains a unary predicate symbol $\mathsf{ServMod}[\mathtt{M}]$. $\iota_\sigma(\mathsf{ServMod}[\mathtt{M}])(s)$ holds if $\mathtt{M}$ is the module for RPC service s.

- For each module $\mathtt{M} \in \mathsf{MODSYMS}$, V'_c contains a unary predicate symbol $\mathsf{IsServ}[\mathtt{M}]$. $\mathsf{IsServ}[\mathtt{M}]$ holds for at most one individual; in particular, $\iota_\sigma(\mathsf{IsServ}[\mathtt{M}])(s)$ holds if service s is the RPC service bound to $\mathtt{M}$.

For `cap` store σ, the model of σ over the vocabulary V_c is the union of the models (defined in §2.3, Defn. 6) $m_\sigma = m^{cc}_\sigma \cup m^c_\sigma$.

A *capability-store condition* is a closed first-order V_c formula; we denote the space of all store conditions as $\mathsf{CapStoreConds}$. A store σ satisfies capability-store condition φ if m_σ is a model of φ.

We now illustrate capability-store conditions used to express properties of stores in `gzip_pol`.

Example 3. Conditions that must hold when modules of gzip complete execution can be represented as store conditions. For clarity, these store conditions are depicted in Fig. 4.2 as a set of derived V_c predicates. The derived nullary predicate $\mathtt{RD}(\mathtt{in})$ denotes the store condition:

$$\forall m, o.\ \mathsf{IsMem}(m) \land \mathtt{in}(o) \implies \mathtt{RD}(m, o)$$

The derived nullary predicate $\mathtt{WR}(\mathtt{out})$ denotes the store condition:

$$\forall m, o.\ \mathsf{IsMem}(m) \land \mathtt{out}(o) \implies \mathtt{WR}(m, o)$$

The derived unary predicate $\mathtt{MemRD}(x)$ denotes the store condition:

$$\forall m.\ \mathsf{IsMem}(m) \implies \mathtt{RD}(m, x)$$

The derived unary predicate $\mathtt{MemWR}(x)$ denotes the store condition:

$$\forall m.\ \mathsf{IsMem}(x) \implies \mathtt{WR}(m, x)$$

5.3.2 Capability policies

A capability policy is a finite-state automaton in which each alphabet symbol is a control location paired with a store condition.

Definition 13. A *capability policy* is a finite-state automaton whose alphabet is a finite subset of $\Sigma_c = \mathsf{LOC} \times \mathsf{CapStoreConds}$. We denote the space of all capability policies as $\mathsf{CapPols}$.

Each capability policy A defines a language of cap traces as violations such that each state in a trace satisfies a corresponding condition in some trace of conditions accepted by A.

Definition 14. Let $t = (\mathtt{L}_0, \sigma_0), \ldots, (\mathtt{L}_n, \sigma_n) \in \mathsf{CapStates}^*$ be a trace of cap states, and let $C \in \mathsf{CapPols}$ be a capability policy. If $t_C = a_0, \ldots, a_n \in \Sigma_c^*$ is such that for each $0 \leqslant i \leqslant n$ and $a_i = (L'_i, \varphi_i)$, (1) $L_i = L'_i$ and (2) $\sigma_i \models \varphi_i$, then t_C is a *condition trace* of t. t *violates* C if C accepts some condition trace of t. For `difc` program P, if each trace t of P does not violate A, then P *satisfies* A.

5.4 The capability-instrumentation problem

The capability-instrumentation problem is to take a `cap` program P and a capability policy C, and instrument P to satisfy C.

Definition 15. Let P be a `cap` program and let C be a capability policy. A solution to the capability-instrumentation problem $\mathsf{CAP}(P, C)$ is a `cap` program P' such that (1) P' is a capability refinement of P and (2) P' satisfies C.

5.5 Capability instrumentation as game solving

In this section, we describe a sound, but incomplete, procedure `capweave` for solving the capability-instrumentation problem.

5.5.1 Overview

In principle, a solution to a capability-instrumentation problem $\mathsf{CAP}(P, C)$ can be any instrumentation of P which, at particular control locations, checks predicates of its current state and chooses an appropriate capability operation to execute next in order to satisfy C. The problem of synthesizing a set of predicates to be checked and acted on at runtime raises daunting challenges. In particular, a `cap` state is defined by an unbounded set of descriptors and a filesystem. Thus, checking many properties of a `cap` state at runtime could be expensive, and in the worst case impossible if

```
Input   : A cap program P and capability policy C.
Output  : A solution to CAP(P, C).
G_{P,Π} := CapProgPolicyGame(P, Π) ;
if HasWinningDefenderStrategy(G_{P,Π}) then
    D := FindWinningDefenderStrategy(G_{P,Π}) ;
    return CapCodeGen(D) ;
else
    Fail () ;
```

Algorithm 5.5: `capweave`: a sound solver for the capability-instrumentation problem.

the value of the predicates depends on the value of the filesystem and memory does not hold ambient authority.

Instead of attempting to synthesize a `cap` program that chooses the next capability operation to execute based on properties of its *execution state*, we attempt to synthesize a program that chooses the next capability operation to execute based on properties that summarize aspects of its *history of executed operations*. We reduce the problem of searching for a valid instrumentation to finding a winning Defender strategy to a game, where the symbols of the game model `cap` operations in module traces of executions. This approach has several advantages:

- `capweave` can be parameterized on advice "templates" from an expert user, which specify restricted languages of operations for an instrumented program to potentially execute.
- `capweave` can apply any analysis that builds a sound abstraction of the transition relation of a `cap` program, independent of the representation of states in the abstraction.
- `capweave` can use standard automata-theoretic language operations to model the instrumented program's inability to observe the actions of the environment.

capweave attempts to solve a capability-instrumentation problem $\mathsf{CAP}(\mathsf{P}, \mathsf{F})$ in three main steps, presented in pseudo-code as Alg. 5.5. capweave first constructs (line [1]) from P and Π a finite two-player game $\mathsf{G}_{\mathsf{P},\Pi}$ whose alphabet is the space of operations of P, such that for any Defender strategy D that wins $\mathsf{G}_{\mathsf{P},\Pi}$, the plays of D are the traces of a solution to $\mathsf{CAP}(\mathsf{P}, \Pi)$ (for brevity, we say that D *defines* a solution to $\mathsf{CAP}(\mathsf{P}, \Pi)$). This step is described in more detail in §5.5.2.

capweave then applies a classical algorithm HasWinningDefenderStrategy [3] to determine if $\mathsf{G}_{\mathsf{P},\Pi}$ has a winning Defender strategy (line [2]). If $\mathsf{G}_{\mathsf{P},\Pi}$ has a winning Defender strategy, then capweave applies a classical algorithm FindWinningDefenderStrategy [3] to construct a winning Defender strategy D (line [3]), Otherwise, capweave aborts (line [6]); we discuss this limitation, and possible extensions of our work that could overcome it, in Chapter 14.

If capweave finds a Defender strategy D that wins $\mathsf{G}_{\mathsf{P},\mathsf{C}}$, then capweave instruments P to form a new cap program P' that is a solution to $\mathsf{CAP}(\mathsf{P}, \mathsf{C})$ (line [4]). During each execution, P' stores two tables that represent the transition function of D. One table, $\mathtt{T_A}$, represents the transition function of D from Attacker states. $\mathtt{T_A}$ is indexed by an Attacker pre-state and a cap operation, and maps each index-pair to a post-state. As P' executes, it stores the current state of D in a variable cur. When P' executes a capcore operation o, it updates cur to store the value in $\mathtt{T_A}$ indexed by the current value in cur and o.

The second table, $\mathtt{T_D}$, represents the transition function of D from Defender states. $\mathtt{T_D}$ is indexed by a Defender pre-state, and maps each index to a pair of a capability operation and state $(\mathtt{o}, \mathtt{q}')$. As long as the state stored by cur is a Defender state, P' performs the capability operation o and updates cur to store $\mathtt{q}'$. When cur stores an Attacker state, P' executes the next operation of P.

In the remainder of this section, we describe in detail how capweave takes an input `cap` program P and capability policy C and constructs a finite game $G_{P,C}$ that it solves in order to solve $\mathsf{CAP}(P, C)$.

5.5.2 From a program and policy to a finite game

From an input program P and input capability policy C, `capweave` constructs a finite two-player game $G_{P,C}$ such that each winning Defender strategy of $G_{P,C}$ defines an instrumentation of P that satisfies C. To construct $G_{P,C}$, `capweave` performs the following steps:

1. Let $T = (Q_T, \iota_T, F_T, O_c, \Delta_T)$ be a finite acceptor of traces of `cap` operations that serves as a *template* of potential sequences of capability operations that an instrumented version of P may execute before each operation of P. Using T, `capweave` constructs a finite two-player game G_T such that each play of G_T not won by the Attacker is a sequence of `cap` operations accepted by T chosen by the Defender, followed by a `cap` operation chosen by the Attacker. We describe our experience designing capability-operation templates in §5.5.3.

2. From P and T, `capweave` constructs a structure program (defined in §2.3) $S_{P,T}$ such that each trace of $S_{P,T}$ is a trace t of P with a sequence of operations accepted by T injected before each operation of t.

3. From $S_{P,T}$, `capweave` constructs a finite-state acceptor $A^{\#}_{P}$ of traces of `cap` operations such that each module trace of a run of P is accepted by $A^{\#}_{P,T}$.

4. From capability policy C, `capweave` constructs a structure program S_C such that each `cap` trace of a run that does not satisfy C drives S_C to an error control location.

5. From structure program S_C, `capweave` constructs a finite-state acceptor $A^{\#}_{C}$ of traces of `cap` operations such that each module trace of a run that drives S_C to an error control location is accepted by $A^{\#}_{C}$.

6. capweave constructs $G_{P,C}$ as the product of G_T, $A^{\#}_{P,T}$, and $A^{\#}_{C}$.

We now describe each step of the construction of $G_{P,C}$ in more detail.

Constructing a game of template instrumentations

From template T, capweave constructs a two-player safety game G_T in which the Defender is restricted to play only a sequence of operations accepted by T. In particular, each play of G_T is an unbounded sequence of phases, in which each phase consists of (1) a sequence of capability operations chosen by the Defender that are accepted by T, followed by (2) any cap operation, chosen by the Attacker. The construction of G_T is straightforward from its informal description, and we omit a fill definition.

From cap program P to a structure-program model

From the input program P and template T, capweave constructs a structure program $S_{P,T} = (LOC_S, \iota_S, O_S, E_S, V_S, T_S)$ such that each trace of $S_{P,T}$ is a trace t of P with a sequence of operations accepted by T injected before each operation in t. The components of $S_{P,T}$, i.e., the control locations LOC_S, initial control location ι_S, operations O_S, control edges E_S, logical vocabulary V_S, and predicate transformers T_S of $S_{P,T}$, constructed by capweave are as follows.

Control locations of $S_{P,T}$ The control locations LOC_S of $S_{P,T}$ contain "copies" of the states of T for each control location of P and a control location at which $S_{P,T}$ models the environment of P. I.e., LOC_S contains the following:

- For each control location $L \in LOC_S$ and each state $q \in Q_T$, a control location (L, q).
- The control location ENV.

Initial control location of $S_{P,T}$ The initial control location of $S_{P,T}$ is the initial control location of the initial module of P.

Operations of $S_{P,T}$ The operations of $S_{P,T}$ are the `cap` operations O_c.

Control edges of $S_{P,T}$ The control edges of $S_{P,T}$ induce $S_{P,T}$ throughout each execution to execute a sequence of operations accepted by T and then execute the next operation executed by P. In particular, E_P contains the following edges:

- For each transition $(q, o, q') \in \Delta_T$ of T, $S_{P,T}$ may transition on o from each copy of q. I.e., for each control location $L \in \mathsf{LOC}$ and each transition $(q, o, q') \in \Delta_T$, E_P contains a control edge $((L, q), o, (L, q'))$.
- If P transitions from control location L to control location L' on an intra-module operation o, then $S_{P,T}$ transitions on o from each copy of a final state of T for L to the copy of the initial state of T for L'. I.e., for each intra-module control edge $(L, o, L') \in E_P$ and each final state $q \in F_T$, E_S contains a control edge $((L, q), o, (L', \iota_T))$.
- If P in control location L executes operation $\mathtt{jump}\ M$ to jump to a module M not in P, then $S_{P,T}$ transitions on operation $\mathtt{jump}\ M$ to control location ENV. I.e., for each operation $\mathtt{jump}\ M$ at location L with M in the environment and $q \in F_T$ a final state of T, E_S contains a control edge $((L, q), \mathtt{jump}\ M, \mathsf{ENV})$.
- If $S_{P,T}$ is in a state that models the environment, then $S_{P,T}$ may transition on any intra-module operation and continue to model an environment module. I.e., for each `cap` operation o, E_S contains a control edge $(\mathsf{ENV}, o, \mathsf{ENV})$.

- For each module $\mathtt{M} \in \mathsf{P}$, E_S contains an edge from ENV to the copy of the initial state of T for the initial location of $\mathtt{M}$. I.e., E_S contains a control edge $(\mathsf{ENV}, \mathtt{jump}\ \mathtt{M}, (\mathsf{ModInit}_\mathsf{P}(\mathtt{M}), \iota_\mathsf{T}))$.

Vocabulary of $\mathsf{S}_{\mathsf{P},\mathsf{T}}$ The logical vocabulary of $\mathsf{S}_{\mathsf{P},\mathsf{T}}$ is V_c, the logical vocabulary over which capability-store conditions are defined in §5.3.1, Defn. 12.

Predicate transformers of $\mathsf{S}_{\mathsf{P},\mathsf{T}}$ For each `cap` operation, `capweave` defines a predicate transformer over the vocabulary V_c. We now describe how the semantics of each `cap` operation, in particular each condition over ambient authority, capabilities, and RPC services in the premise of an operation and each update of a capability store in Fig. 5.4, is modeled as a predicate transformer over V_c structures.

We define the space of predicate transformers as the union of transformers T_{cc} that describe how predicates in V_{cc} are updated, and transformers T_c that describe how the predicates in V'_c are updated. The T_{cc} transformer for an operation $\mathtt{d} \coloneqq \mathtt{open}(\mathtt{A}, \mathtt{x})$ (1) stores any newly-allocated individual in descriptor variable `d` and (2) stores that the allocation site of any new individual is `A` (in the predicate updates given below, as well as several assertions and predicate updates of other transformers, we have annotated a subformula with $(\mathtt{i})$ if it models a premise or update in the semantics itemized with (i)).

$$
\begin{aligned}
\mathtt{d}(d) &\coloneqq \mathsf{new}(d) && (1)\\
\mathtt{alloc}[A](d) &\coloneqq \mathtt{alloc}[A](d) \lor \mathsf{new}(d) && (2)
\end{aligned}
$$

The predicate transformers for `capcore` on each other operation `o` do not place any constraint on the pre-structure of `o`, and do not update the values of any predicates in the pre-structure.

The predicate transformers T_c update the predicates in V'_c for each operation as follows:

- A cap operation $\mathtt{d} \coloneqq \mathtt{open}(\mathtt{A}, \mathtt{x})$ checks that in pre-store σ, memory has ambient authority. The predicate transformer $\tau[\mathtt{d} \coloneqq \mathtt{open}(\mathtt{A}, \mathtt{S})]$ asserts that its pre-structure satisfies the following V_c formula:

$$\forall m.\ \mathsf{IsMem}(m) \implies \mathsf{HasAmb}(m)$$

If σ passes the check of $\mathtt{d} \coloneqq \mathtt{open}(\mathtt{A}, \mathtt{x})$, then $\mathtt{d} \coloneqq \mathtt{open}(\mathtt{A}, \mathtt{x})$ allocates a fresh descriptor d and extends the capabilities of memory to contain d paired with each operation on descriptors. If a pre-structure S satisfies the assertion of $\tau[\mathtt{d} \coloneqq \mathtt{open}(\mathtt{L}, \mathtt{S})]$, then $\tau[\mathtt{d} \coloneqq \mathtt{open}(\mathtt{L}, \mathtt{S})]$ updates the universe and predicates of S by introducing a new individual and applying the following predicate updates :

$$\mathtt{O}(m, d) \coloneqq \mathtt{O}(m, d) \lor (\mathsf{IsMem}(m) \land \mathsf{new}(d))$$

- A cap descriptor-operation $\mathtt{x} \coloneqq \mathtt{o}(\mathtt{d})$ checks that in pre-store σ, memory holds the access right for $\mathtt{o}$ at the descriptor bound to descriptor variable $\mathtt{d}$. The predicate transformer $\tau[\mathtt{x} \coloneqq \mathtt{o}(\mathtt{d})]$ asserts that its pre-structure satisfies the following V_c formula:

$$\forall m, d.\ \mathsf{IsMem}(m) \land \mathtt{d}(d) \implies \mathtt{o}(m, d)$$

- A cap operation $\mathtt{s} \coloneqq \mathtt{create\_serv}(\mathtt{L}, \mathsf{E_A}, \mathsf{E_C})$ checks that in pre-store σ, (1) the ambient-authority expression $\mathsf{E_A}$ evaluates to a Boolean value that implies the ambient authority of memory and (2) the capability expression $\mathsf{E_C}$ evaluates to a set of capabilities contained by the capabilities of memory. The predicate transformer $\tau[\mathtt{s} \coloneqq \mathtt{create\_serv}(\mathtt{L}, \mathsf{E_A}, \mathsf{E_C})]$ asserts that its pre-structure S satisfies the following V_c formula:

$$\begin{aligned} \forall m, d.\ \mathsf{IsMem}(m) \implies & (\varphi_A() \implies \mathsf{HasAmb}(p)) && (1) \\ & \wedge (\textstyle\bigwedge_{o \in O_c} \varphi[o](d) \implies \mathsf{o}(p, d)) && (2) \end{aligned}$$

Where φ_A and φ_C are defined below.

If σ passes the check of $\mathtt{s} \coloneqq \mathtt{create\_serv}(\mathtt{M}, E_A, E_C)$, then $\mathtt{s} \coloneqq \mathtt{create\_serv}(\mathtt{M}, E_A, E_C)$ creates a fresh RPC service s, (1) binds s to $\mathtt{s}$, (2) gives s module $\mathtt{M}$, (3) gives s ambient authority if E_A evaluates to True in σ, and (4) gives s the capabilities in the evaluation of E_C in σ. If S satisfies the assertion of $\tau[\mathtt{s} \coloneqq \mathtt{create\_serv}(\mathtt{M}, \mathtt{A}, \mathtt{C})]$, then $\tau[\mathtt{s} \coloneqq \mathtt{create\_serv}(\mathtt{M}, \mathtt{A}, \mathtt{C})]$ introduces a fresh individual into the universe of S and updates the predicates of S according to the following predicate updates:

$$\begin{aligned} \mathsf{s}(s) &\coloneqq \mathsf{new}(s) && (1) \\ \mathsf{ServMod}[\mathtt{M}](x) &\coloneqq \mathsf{ServMod}[\mathtt{M}](s) \vee \mathsf{new}(s) && (2) \\ \mathsf{HasAmb}'(s) &\coloneqq \mathsf{HasAmb}(s) \vee (\mathsf{new}(s) \wedge \mathsf{E}_A()) && (3) \\ \mathsf{o}(s, d) &\coloneqq \mathsf{o}(s, d) \vee (\mathsf{new}(s) \wedge \mathsf{E}_C[\mathsf{o}](d)) && (4) \end{aligned}$$

Fig. 5.6 depicts the predicate transformer for the operation

$$\begin{aligned} \mathtt{o} \equiv \mathtt{s1} \coloneqq \mathtt{create}(&\mathtt{cmp}, \mathtt{no\_amb}(), \\ &\mathtt{rem}(\mathtt{in}, \mathtt{WR}, \mathtt{rem}(\mathtt{out}, \mathtt{RD}, \mathtt{mem\_caps}))) \end{aligned}$$

contained in the module `loop` introduced in Chapter 4, applied to a pre-structure that models a store σ at control location `C5a`. The pre-structure contains three individuals that model (1) program memory (annotated "m"), (2) the input file descriptor (annotated "i"), and (3) the output file descriptor (annotated "o"). For individual m, the unary predicates IsMem and HasAmb hold, which models the facts

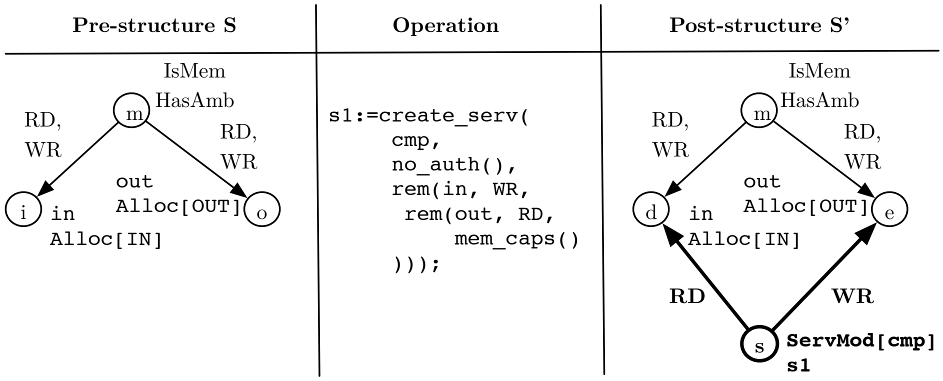

Figure 5.6: A graphical depiction of the predicate transformer that models the `create_serv` operation o executed by `gzip` at location `C5a` (introduced in Chapter 8). The pre-structure S is depicted on the left, and the resulting post-structure S' is depicted on the right. Each structure is depicted as a graph in which each node depicts an individual, and each edge depicts a binary relation between nodes. Each node n depicting an individual i_n is annotated with a name inside n, and unary predicate symbols to the side of n that hold for i_n, and each edge from node m to node n is annotated with the binary relation that holds for (m, n). In the post-structure S', nodes and edges depicting individuals and relations created by o are highlighted in bold.

that m models the program memory and has ambient authority. For individual i, the unary predicates `in` and `alloc[IN]` hold, which models the facts that i is stored in the descriptor variable `in` and was allocated at allocation site `IN`. For individual o, the unary predicates `out` and `alloc[OUT]` hold, which models the facts that o is stored in `out` and was allocated at allocation site `OUT`. The edges from m to i and from m to o annotated `RD` and `WR` model the fact that memory has the `RD` and `WR` access rights for descriptors i and o.

The post-structure S' in Fig. 5.6, obtained by applying the predicate transformer $\tau[\text{o}]$ to the pre-structure S, is S extended with an additional individual (annotated "s") that models the RPC service created by executing $\mathtt{o}$. Individual s is annotated with unary predicates $\mathsf{ServMod}[\mathtt{cmp}]$ and $\mathtt{s}$, which models the fact that in S', the service s has module $\mathtt{cmp}$ and is stored in service variable $\mathtt{s}$. S' contains (1) an edge from s to $\mathtt{i}$ that models the fact that in S', s has access right $\mathtt{RD}$ for descriptor $\mathtt{i}$ and (2) an edge from s to $\mathtt{o}$ that models the fact that in S', s has access right $\mathtt{WR}$ for $\mathtt{o}$.

- A $\mathtt{cap}$ operation $\mathtt{set\_serv}(\mathtt{s}, \mathtt{M})$ sets the service for module $\mathtt{M}$ to be the service bound to service variable $\mathtt{s}$. The predicate transformer $\tau[\mathtt{set\_serv}(\mathtt{s}, \mathtt{M})]$ updates the predicates of a pre-structure according to the following predicate update:

$$\mathsf{IsServ}[\mathtt{M}]'(x) \coloneqq \mathtt{s}(x)$$

- A $\mathtt{cap}$ operation $\mathtt{jump}\ \mathtt{M}$ updates the control location of pre-state σ to be $\mathtt{M}$, (1) updates the ambient authority of memory to be the ambient authority of the RPC service s bound to $\mathtt{M}$, and (2) updates the capabilities of memory to be the capabilities of s. For formulas φ_g, φ_t, and φ_f, let the *if-then-else* formula, denoted $\mathsf{ITE}(\varphi_g, \varphi_t, \varphi_f)$, denote the formula $(\varphi_g \implies \varphi_t) \land (\neg\varphi_g \implies \neg\varphi_f)$. The predicate transformer $\tau[\mathtt{jump}\ \mathtt{M}]$ updates the predicates of a pre-structure S according to the following predicate updates:

$$\mathsf{HasAmb}'(m) \coloneqq \mathsf{ITE}(\mathsf{IsMem}(m), \exists s.\ \mathsf{IsServ}[\mathtt{L}](s) \land \mathsf{HasAmb}(s), \mathsf{HasAmb}(m)) \tag{1}$$

$$\mathtt{o}'(m, d) \coloneqq \mathsf{ITE}(\mathsf{IsMem}(m), \exists s.\ \mathsf{IsServ}[\mathtt{L}](s) \land \mathtt{o}(s, d), \mathtt{o}(m, d)) \tag{2}$$

For an ambient-authority expression $\mathtt{E}$, which may be a component of $\mathtt{create\_serv}$ operation, the nullary V_c formula $\varphi_{\mathtt{E}}$ used in the predicate updates for $\mathtt{create\_serv}$ operations is defined as follows:

- If E is the ambient-authority expression `mem_auth`, then $\varphi_E \equiv \exists m.\mathsf{IsMem}(m) \wedge \mathsf{HasAmb}(m)$.
- If E is the ambient-authority expression `no_amb`, then $\varphi_E \equiv \mathsf{False}$.

For a capability expression E, which is a component of a `create_serv` operation, and descriptor operation $o \in \mathsf{DESCOP}$, the unary V_c formula $\varphi_E[o](d)$ used in the predicate updates for `create_serv` operations is defined as follows:

- If E is the capability expression `mem_caps`, then $\varphi_E[o](d) \equiv \exists m.\ \mathsf{IsMem}(m) \wedge o(m, d)$.
- If E is a capability expression $\mathtt{rem}(d, o, E')$, then $\varphi_E[o](d) \equiv \mathsf{False}$, and for $o' \in \mathsf{DESCOP}$ such that $o \neq o'$, $\varphi_E(d) \equiv \varphi_{E'}(d)$.

From a structure-program model to a finite abstraction

To construct a finite over-approximation of the language of module traces of executions of $S_{P,T}$, `capweave` applies a procedure $\mathsf{AbsStruct}$ that solves the structure-abstraction problem (described in Chapter 2) $\mathsf{STRUCT\_ABS}(S_{P,T})$. Let $(S^{\#}_{P,T}, \mathsf{AbsNode}) = \mathsf{AbsStruct}(S_{P,T})$ be a solution produced by $\mathsf{AbsStruct}$ for $\mathsf{STRUCT\_ABS}(S_{P,T})$. Then from $S^{\#}_{P,T} = (Q^{\#}, \Sigma, \Delta^{\#})$ and $\mathsf{AbsNode}$, `capweave` constructs the finite acceptor $A^{\#}_P = (Q_P, I_P, F_P, \Sigma_P, \Delta_P)$, where

- The states Q_P are the states $Q^{\#}$.
- The initial states I_P are the states of $S^{\#}_{P,T}$ that abstract states at the initial control location of $S_{P,T}$. I.e., $I_P = \{\iota \mid \iota \in Q^{\#}, \mathsf{AbsNode}(\iota) = \iota_S\}$.
- The final states F_P are the states of $S^{\#}_{P,T}$ that abstract states whose control location is ERR.
- The alphabet Σ_P is the space O_c of `cap` operations.

- The transition relation Δ_P is the transition relation of $S^{\#}_{P,T}$, with each operation that $S_{P,T}$ executes to model the environment replaced with an ϵ transition. I.e.,

$$\begin{aligned}\Delta_P =& \{(q, o, q') \mid (q, o, q') \in \Delta^{\#}, \mathsf{AbsNode}(q) \neq \mathsf{ENV}\} \\ & \cup \{(q, \epsilon, q') \mid (q, o, q') \in \Delta^{\#}, \mathsf{AbsNode}(q) = \mathsf{ENV}\}\end{aligned}$$

From capability policy C to a structure-program model

From the input capability policy $C = (Q_C, \iota_C, F_C, \Sigma_c, \Delta_C)$, `capweave` constructs a structure program $S_C = (\mathsf{LOC}^S_C, \iota^S_C, \mathtt{O}^S_C, E^S_C, V^S_C, T^S_C)$ such that each trace of `cap` operations that violates C drives S_C to an error control location. The components of S_C are defined as follows.

Control locations of S_C The control locations LOC^S_C store the state of C inhabited by the `cap` run simulated by S_C. In particular, for each state $q \in Q_C$, LOC_S contains control locations q and q'. LOC_S also contains an error location ERR.

Initial control location of S_Π The initial control location ι^S_C of S_Π is the initial state of the policy Π.

Operations of S_Π The operations $\mathtt{O}_S$ of S_C are the `cap` operations extended with a set of operations of the form $\mathtt{assume}[\varphi]$, where $\varphi \in \Sigma_C$ is a store condition in any symbol in a state condition in the alphabet of C.

Control edges of S_Π The control edges E_S of S_C define how S_C maintains the state of C inhabited by its current execution. In particular, E_S contains the following edges:

- For each pair of states $q_0, q_1 \in Q_C$ and store condition φ such that C transitions from q_0 to q_1 on φ (i.e., $(q_0, \varphi, q_1) \in \Delta_C$), E_S contains a control edge $(q_0, \texttt{assume}[\varphi], q_1')$.

- For each policy state $q \in Q_C$ and each `difc` operation $\texttt{o}$, E_S contains a control edge $(q', \texttt{o}, q)$.

Vocabulary of S_C The vocabulary of S_C is V_c, defined in §5.3.1, Defn. 12.

Predicate transformers of S_C The predicate transformers of each `cap` operation $\texttt{o}$ in S_C is the predicate transformer of $\texttt{o}$ in $S_{P,T}$. For each state condition $\texttt{L} : \varphi \in \Sigma_C$, the predicate transformer checks that its pre-structure satisfies φ.

From a structure-program model of C to a finite abstraction

To construct a finite over-approximation of the module-traces of `cap` executions that violate C, `capweave` applies the procedure AbsStruct (described in §5.5.2) to construct a finite abstraction of S_C, and replaces each transition that models a step of execution of the environment with an ϵ transition. Let $(S_C^\#, \mathsf{AbsNode}) = \mathsf{AbsStruct}(S_C)$ be a solution to the structure-program-abstraction problem $\mathsf{STRUCT\_ABS}(S_C)$ produced by AbsStruct. Then from $S_C^\# = (Q^\#, \Sigma, \Delta^\#)$ and AbsNode, `capweave` constructs the finite acceptor $A_C^\# = (Q, I, F, \Sigma, \delta)$ of `cap` operations, where:

- The states Q are the states $Q^\#$.

- The initial states I are the states of $S_C^\#$ that abstract states at the initial control location of S_C. I.e., $I = \{\iota \mid \iota \in Q, \mathsf{AbsNode}(\iota) = \iota_C\}$.

- The final states F are the states of $S_C^\#$ that abstract states of S_C whose control location is ERR. I.e.

$$F = \{q \mid q \in Q^\#, \mathsf{AbsNode}(q) = \mathsf{ERR}\}$$

- The alphabet Σ is the space of `cap` operations O_c.
- The transition relation Δ is the transition relation of $S^{\#}_{C}$, with transitions from environment locations replaced with ϵ transitions:

$$\begin{aligned}\Delta =&\{(q, o, q') \mid (q, \mathtt{L} : o, q') \in \Delta^{\#}, \mathtt{L} \neq \mathsf{ENV}\}\\ &\cup \{(q, \epsilon, q') \mid (q, \mathtt{L} : o, q') \in \Delta^{\#}, \mathtt{L} = \mathsf{ENV}\}\end{aligned}$$

From template game, finite program and policy models to a game

`capweave` constructs $G_{P,C}$ as a product of the template game G_T, the over-approximation $A^{\#}_{P}$ of module-traces of P, and the over-approximation $A^{\#}_{C}$ of violations of C. In particular, $G = G_T \times_{G,A} (\det(A^{\#}_{P}) \times \det(A^{\#}_{C}))$, where for a non-deterministic finite-state acceptor A, $\mathtt{det}(A)$ is a deterministic acceptor that accepts the same language as A, and $\times_{G,A}$ is the game-automaton product defined in §2.2.

5.5.3 Designing capability-operation templates

The capability-operation template T used by `capweave` directly affects both the space of instrumentations considered by `capweave`, as well as the size of the game constructed by `capweave`. We found that templates for many practical programs that run on Capsicum can be defined as regular languages; these languages accept sequences of capability operations that create a fresh RPC service s by (1) choosing the program module of s to be some module `M` that appears in the input capability policy C, (2) choosing the ambient authority of s to either be no authority or the ambient authority of memory, (3) choosing the capabilities of s by choosing a bounded set of descriptor variables V and access rights `R` that appear in C, and (4) removing some subset of capabilities defined by descriptors stored in V paired with access rights in `R`. The template then sets the RPC service for `M` to be s. We discuss the effect of using templates of varying sophistication in Chapter 6.

6 Evaluation

We carried out a series of experiments, designed to answer the following questions about weaving technique:

1. Can practical policies for program capabilities be expressed as capability policies?
2. Can our weaving algorithm efficiently instrument practical programs to satisfy a policy represented as a capability policy?
3. Do programs woven by our weaving algorithm perform comparably to programs instrumented by hand?

To answer the above questions, we implemented our weaving algorithm as a tool, `capclang`, that performs a source-to-source translation in the LLVM intermediate language [37] to instrument programs to be run on the Capsicum capability system. The steps of the `capweave` algorithm described in §5.5 are implemented in `capclang` as follows:

1. From an input program $\mathtt{P}$, `capweave` constructs a structure program $\mathrm{S}_{\mathtt{P}}$ that simulates the executions of $\mathtt{P}$. `capclang` constructs $\mathrm{S}_{\mathtt{P}}$ using the API provided by LLVM.
2. From an input capability policy C, `capweave` constructs a structure program S_{C} whose executions violate C. `capclang` constructs S_{C} by parsing C using a custom parser for capability policies.

3. `capweave` constructs finite abstractions of the language of traces of S_P and S_C by applying a solver for the structure-program-abstraction problem. `capclang` constructs finite abstractions $S^{\#}_{P}$ and $S^{\#}_{C}$ of S_P and S_C by applying the TVLA logic-analysis engine [40].

4. `capweave` constructs a game G from $S^{\#}_{P}$ and $S^{\#}_{C}$, and attempts to find a winning Defender strategy to G by applying a classical algorithm for solving two-player games. `capclang` attempts to find a winning Defender strategy to G by applying a the game-solving algorithm implemented in the GOAL tool [56].

5. If `capweave` determines that the game G has a winning Defender strategy D, then from D, `capweave` instruments P to satisfy C. `capclang` checks if GOAL found a winning Defender strategy D, and if so, uses the LLVM API to (1) generates multi-dimensional arrays that represent D in the LLVM intermediate language, and instruments P with LLVM functions calls that invoke a fixed runtime-library function that updates program state and executes capability operations.

To determine if practical policies can be expressed as capability policies (item 1), we collected a set of benchmark programs that had known security vulnerabilities, including programs that had been instrumented manually for Capsicum by the Capsicum developers in previous work [58], as well as programs that had not been instrumented previously. For each benchmark program, we wrote a capability policy.

To determine if `capclang` could instrument practical programs to satisfy their policies (item 2), we applied `capclang` to each benchmark program and its policy. We ran `capclang` on a server running Linux kernel version 2.6.32-431.3.1.el6.x86_64, with 16 2.4-GHz cores, and 32 GB of RAM, although `capclang` executes in a single thread.

To determine if programs instrumented automatically by `capclang` perform comparably to programs instrumented manually by an expert developer (item 3), we ran versions of each benchmark program written manually and instrumented automatically by `capclang` on representative workloads for the program. We ran each program in a Capsicum virtual machine on the same server on which we ran `capclang`.

In short, we found that `capclang` often allowed us to instrument existing programs completely from capability policies. In some cases, we found that it was infeasible in practice to instrument programs completely from policies without first manually editing the programs. However, these manual edits do not themselves contain capability primitives: instead, they are used to signify key events in the execution of program, which we used to adapt our original desired capability policies; such cases are described in detail in §6.1.

We found that programs that could be instrumented completely from policies incurred acceptable runtime overhead, often even compared to their uninstrumented versions. Programs that had to be edited before `capclang` could instrument them successfully incurred significant runtime overhead compared to their uninstrumented versions. We discuss such cases in §6.2.

Partitioning programs for Capsicum One difference between instrumenting `capcore` programs with `cap` primitives and instrumenting actual LLVM programs with Capsicum primitives is that `capcore` programs are structured as independent modules, whereas LLVM programs are structured as a collection of functions that pass data through parameters and global variables. We developed an analysis that determines which functions in an LLVM program can potentially be extracted from the program and placed in an independent module. Our analysis accounts for the possibility of using different primitives—with different inherent costs—for executing trusted modules with particular capabilities, such as forking a process or creating an RPC service. Our analysis implements standard techniques for partitioning programs [60], but does not itself choose which modules will be partitioned using which Capsicum primitives. Instead, the results of the analysis are used to constrain the game constructed by `capweave` so that a valid instrumentation can only invoke partitioning primitives allowed according to the analysis.

6.1 Benchmark programs, policies, and instrumentation

In this section, we describe each benchmark program, describe the policy that we wrote for the program, and describe the capability operations that `capweave` instrumented each program to execute in order to satisfy the policy.

6.1.1 Compression utilities `bzip2` and `gzip`

The compression utilities `gzip`, its capability policy, and `capweave`'s instrumentation of `gzip` to satisfy the policy were discussed in the overview of `capweave` (Chapter 4). The structure of `bzip2` concerning how it manages descriptors, and thus its capability policy and instrumentation to satisfy the policy, are directly analogous to those of `gzip`.

6.1.2 `tcpdump`

`tcpdump` is a widely-used network-facing application that historically has been the target of many exploits. `tcpdump` takes as input a Berkeley Packet Filter (BPF), and a device from which to read packets. In a correct execution, it reads packets from the device, matches them against the input BPF, and if the packet matches, prints the packet to standard output. Unfortunately, the packet-matching code in `tcpdump` is complex; in previous versions of `tcpdump`, an attacker who controls the network input to `tcpdump` can craft a packet that allows him to take control of the process executing `tcpdump` [10].

Policy

We defined a capability policy for `tcpdump` that strictly limits the power of an attacker who is able to compromise `tcpdump`. The policy assumes that the only trusted modules in `tcpdump` are its `main` function, which executes before matching packets to filters, and a small function that only opens temporary network connections to resolve DNS queries. The policy treats the majority of `tcpdump` as an untrusted environment, which executes during or after `tcpdump` executes its vulnerable packet-matching code. Our policy specifies that when `main` transfers control to the environment, the environment should only hold the capabilities to write to standard output and read from the packet descriptor opened by `main`. The environment may only ever hold another capability whose descriptor component is a socket when it executes the DNS resolver. Our policy was directly inspired by previous work on manually rewriting programs for Capsicum [58].

Instrumentation

`capweave` successfully instrumented `tcpdump` to satisfy the above policy. In particular, `capweave` instrumented `main` to create an RPC service s whose module was the DNS resolver, such that s holds ambient authority, and no other capabilities. `capweave` instrumented `main` to transfer control to its environment with exactly the capabilities required to read from the packet descriptor opened in `main` and write to standard output, and with access to the RPC service for the DNS resolver.

6.1.3 `php-cgi`

Executing programs written in web scripting languages, such as `php`, raises multiple security issues. First, it is inherently difficult to analyze, monitor, and restrict the behavior of a program written in a scripting language. Second, a maliciously-crafted web program can potentially compromise the interpreter that executes it, and then perform any action on its host system that is allowed for the user who launched the interpreter [14].

Because `php-cgi` may be run to execute valid `php` programs that attempt to perform arbitrary sequences of operations on descriptors, writing a single policy for `php-cgi` that could ensure non-trivial security guarantees while still protecting the functionality of the program is, for all practical purposes, impossible. Thus, before writing a capability policy for `php-cgi`, we extended `php-cgi` to attempt to open each file descriptor and socket by calling *shim* functions. Each shim function checks if each filepath satisfies particular key security properties, such as if the filepath belonged to configurable whitelists; the manually-written shim functions then disregards the results of the checks, and returns a descriptor or socket with complete capabilities.

Policy

We wrote a policy that assumes that the only trusted modules of `php-cgi` are a small fragment of initialization code at the beginning of the `main` function of `php-cgi` and the shim functions for opening descriptors and sockets, and that the rest of `php-cgi` is an untrusted environment. We wrote a policy that specifies that if the environment holds a descriptor or socket with a particular access right, then the filepath of the resource for the descriptor must satisfy the checks associated with the access right.

Instrumentation

When we applied `capweave` to the version of `php-cgi` manually extended with shim functions and the capability policy, `capweave` found an instrumentation of `php-cgi` that satisfies the policy. When the instrumented version of `php-cgi` executes `main`, it creates RPC services with the shim functions as modules. Each service holds ambient authority, and no capabilities. Each time an instrumented shim function is entered, then function monitors the execution history of filepath property checks. When the instrumented shim function transfers control to its environment, it provides to the environment only the exact capabilities appropriate for the filepath checks that succeeded.

6.1.4 `tar`

The `tar` archiving utility allows a user to maintain archive files. In particular, a user can run `tar` to create a new archive from *source files* or all files in a *source directory*, list the contents of an archive, update the contents of an archive, and delete entries in an archive. Unfortunately, past versions of `tar` have demonstrated vulnerabilities that allow an attacker who controls the inputs to `tar` to run injected code with the privileges of the user who invoked `tar` [9, 12]. One key feature of `tar` is that it opens files that are the *sources* of a copy not by invoking the `open` system call directly, but instead by only invoking the `openat` system call on descriptors that it already holds for directories of source files.

Policy

We defined a capability policy that strictly limits the abilities of an attacker who compromises `tar`. The policy assumes that the only trusted module of `tar` is the `main` function that opens descriptors to filepaths provided as arguments; the policy treats as an untrusted environment the majority of the code in `tar`, in particular the functions that actually create, read from, and update archives. It is particularly challenging to instrument `tar` to satisfy strong safety guarantees while assuming that such functions are in the environment, because the function for creating an archive must be able to open new descriptors for children of input arguments that are directories. The policy for `tar` specifies the following:

- If `main` chooses to call a module to create an archive file, then the environment should hold capabilities to read from descriptors for source files, and should hold the capability to write to the target archive file. However, the environment should never hold a capability for any file that is not a descendent in the filesystem of a source directory.

- If `main` chooses to call a module to list the contents of an archive, then the environment should only hold capabilities to read from the archive and print to standard output.

- If `main` chooses to call a module to update or delete members of an archive, then the environment should only hold the capability to write to the target archive file.

Instrumentation

`capweave` successfully instrumented `tar` to satisfy the above policy. The instrumented `tar` correctly restricts the capabilities that it holds for file descriptors opened in `main` before transferring control to the environment. If `main` chooses to transfer control to a module that will create an archive, then the instrumented `tar` ensures that the environment has the `openat` capability for each descriptor for each source file. As a result, the environment can open new descriptors, but only to files that are descendents of source directories, by invoking the `openat` operation.

6.1.5 wget

The wget downloader is a command-line utility that takes as input a list of URL's. For each URL, wget attempts to download the data addressed by the URL and write the data in the file system of wget's host. wget is a mature, sophisticated tool that supports the HTTP, HTTPS, and FTP protocols. Once wget determines the protocol required for a download, it runs protocol-specific functions to (i) open a socket to the server holding the URL, (ii) download the data addressed by the URL over the socket, and (iii) write the data to a file to the file system.

Unfortunately, versions of wget through v.1.12 demonstrate a vulnerability that allows an attacker who controls a server with which wget interacts to write data to any file on the host file system that can be written by the user who runs wget. The vulnerability is exposed when wget processes a particular HTTP response from the server. In particular, wget may receive from a server a redirect response, which directs wget to download data from a different network address. When wget receives such a response, it determines the path on its host file system to which it will write data directly from the information provided by the redirect server. A malicious server can exploit this behavior to craft a redirect response that causes wget to write data chosen by the attacker to a path in the file system chosen by the attacker. A server can exploit such a vulnerability to execute code on the host system by directing wget to write data to an appropriate startup or configuration file [8].

Policy

Our original desired policy for wget assumed that the main function of wget is trusted, and that all other functions, including the functions that implement the client for each protocol, constitute an untrusted environment. The policy specified that if wget holds the capability to write to a file, then the file should be a descendent of a fixed sandbox directory. However, wget should be able to hold read capabilities for arbitrary files, and hold arbitrary capabilities for sockets. Such a policy is analogous to the policy that bounds the files from which tar can read when it creates an archive. The policy was inspired by discussion on the Capsicum-developer mailing list and the known vulnerabilities of wget [1, 8].

Unfortunately, capweave was not able to instrument wget to satisfy the desired policy because unlike tar, wget attempts to open new file descriptors by invoking open directly on filepaths, not invoking openat on file descriptors held for directories. As a result, capweave was not able to find an instrumentation that restricted the ambient authority of wget's environment so that the environment was prevented from creating arbitrary write capabilities while ensuring that the environment could still open arbitrary read and socket capabilities.

However, we found that by introducing a small shim function analogous to the one created for php-cgi, we were able to apply capclang to instrument wget to satisfy a strong security policy. In particular, we wrote a shim function that checks the filepath f of each file before open is invoked to open a descriptor to the file at f, and determines whether f is a child of the sandbox directory. We then adapted our policy to specify that the environment of wget should be able to hold arbitrary read and socket capabilities, but should only be able to hold write capabilities to files that satisfy the check in the shim open function.

Instrumentation

capweave successfully instrumented the manually-extended version of wget to satisfy the adapted policy. In particular, capweave instrumented the main function to create RPC services for both the shimmed open function and the operation to open sockets; when created, each such service holds ambient authority and no other capabilities. capweave instrumented the open shim to provide to its environment a write capability only if the

Program			Policy	
Name	KLoC	Descriptor sites	LoC	States
`bzip2`	8	2	39	3
`gzip`	9	2	42	3
`php-cgi`	852	2	23	13
`tar`	108	5	51	8
`tcpdump`	87	2	31	3
`wget`	64	2	41	3

Table 6.1: Features of benchmark programs and policies to which we applied `capweave`. Under the "Program" header, "Name" contains the name of the program, "LoC" contains the number of lines of code of the program, measured with the `cloc` utility (which does not count white space or comments); "Descriptor Sites" contains the number of sites in the program that allocate a descriptor that is relevant to the policy. Under the "Policy" header, "LoC" contains the number of lines of code of the capability policy; "States" contains the number of states in the capability policy.

filepath provided to the `open` shim is a child of the current directory when `wget` was executed. `capweave` instrumented `socket` to provide a full socket capability to its environment unconditionally.

Comparing our experience instrumenting `tar` and `wget` illustrates the advantage of instrumenting programs already structured to manage system resources via descriptors, such as `tar`. Such programs can be instrumented with little manual effort by applying `capclang` directly. However, programs that are not written in that style, such as `wget`, can still be instrumented to be secure by expending some manual effort to modify the program, and weakening an ideal security policy to assume as trusted a slightly larger codebase.

Benchmark	capweave				Inst. Prog.
Name	Time	Mem. (MB)	Structures	Game States	Slow-down
bzip2	0m43.17s	38	1,530	30	1.189
gzip	0m55.821s	35	1,137	43	1.201
php-cgi	1m58.92s	141	1,787	91	1.938
tar	1m23.30s	126	1,693	86	1.085
tcpdump	0m43.74s	40	2,896	52	1.153
wget	0m41.92s	60	2,047	75	2.031

Table 6.2: Results of applying capweave to the benchmarks described in Tab. 6.1. The "Benchmark" header contains the name of each benchmark program. "Time" contains the execution time of capweave; "Mem." contains the peak memory usage of capweave; "Structures" contains the number of store structures constructed by the structure analysis applied by capweave; "Game States" contains the number of states in the minimal game constructed by capweave from the transition graph over structures. Under the "Instrumented Program" header, "Slowdown" contains the running time of the instrumented program expressed as a multiple of the running time of the original program.

6.2 Performance

Tabs. 6.1 and 6.2 contain the results of our experience applying capweave. Tab. 6.1 contains data describing features of the benchmarks to which we applied capweave. The columns of Tab. 6.1 are divided into (1) features of the input program and (2) features of the input policy. The columns of Tab. 6.2 are divided into (1) identification of the benchmark program described in Tab. 6.1, (2) data concerning the performance of capweave in instrumenting the benchmark, and (3) data concerning the runtime performance of the version of the program instrumented by capweave.

Tabs. 6.1 and 6.2 contain the data concerning the performance of capweave. Tab. 6.1 contains data describing features of the benchmarks to which we applied capweave. The columns of Tab. 6.1 are divided into (1) features of the input program and (2) features of the input policy. Tab. 6.2 contains data describing features of the performance of applying `capweave`. The columns of Tab. 6.2 are divided into (1) identification of the benchmark program, (2) data concerning the performance of `capweave` in instrumenting the benchmark, and (3) data concerning the runtime performance of the version of benchmark instrumented by `capweave`.

The results indicate that `capweave` can be used to efficiently instrumenting even large programs. In particular,

- The instrumentation time of `capweave` scales well with code size. We believe that this is due to the fact that `capweave` applies a sequence of basic program optimizations to succinctly represent the untrusted code in each program, which in each case consists of only a few thousand lines. Performance instead appears to depend more directly on the size of the policies, which grows slowly with the size of the program.

- The set of structures in the structure transition system generated by the structure analysis is often significantly larger than the set of states in the minimal automaton that accepts the same language of traces. This phenomenon indicates that "local" decisions that the structure analysis makes for distinguishing structures based on a fixed abstraction tend to cause the analysis to maintain distinct structures that are equivalent in terms of which traces executed from states abstracted by the structures violate the capability policy.

- Runtime overhead is significant only for the programs `php-cgi` and `wget`, which `capweave` instrumented to execute an expensive RPC call before performing common operations (opening file descriptors and sockets). We inspected the code of both `php-cgi` and `wget` manually, and believe that there is no instrumentation of `php-cgi` and `wget` that satisfies the given policies that will not require RPC calls to be executed frequently.

For each benchmark, the runtime overhead of the benchmark instrumented by `capweave` compared to the overhead of the benchmark instrumented manually was negligible.

Part II

Weaving for a DIFC System

In this part of the dissertation, we introduce the weaving problem for the HiStar DIFC system, and describe our technique for solving the weaving problem. In Chapter 7, we review a simplified version of HiStar with which we describe the HiStar weaving problem. In Chapter 8, we illustrate the HiStar weaving problem and our approach by example. In Chapter 9, we explain our approach in technical detail. In Chapter 10, we present case studies of applying our HiStar policy weaver to weave programs for HiStar. The structure of Chapters 7–10 for describing the HiStar policy weaver parallels the structure of Chapters 3–6 for describing the Capsicum policy weaver. The weaver generator generalizes both weavers by exploiting this parallel structure, and is described in Part III.

7 Background on the HiStar DIFC System

HiStar, a Decentralized Information-Flow Control (DIFC) operating system [61], provides primitives that an application can invoke to protect the secrecy and integrity of its sensitive information. In particular, the HiStar kernel maps each process and object on the system to a label. Each time a process p attempts to access an object o, HiStar interposes the access, and only allows the access if the labels of p and o satisfy particular constraints. A process can use the primitives provided by HiStar to update the labels of system objects, subject to particular constraints. We now discuss these constraints in detail.

HiStar maintains a rooted graph of objects, i.e., processes and files, where each object is bound to a *label*. A label is a map from each element in the space of *categories* maintained by HiStar to one of three levels: Low, Mid, and High, ordered as $\mathsf{Low} < \mathsf{Mid} < \mathsf{High}$. A label L_0 *flows to* label L_1 over categories C if each category $c \in C$ has a level in L_0 less than or equal to its level in L_1. The HiStar kernel maps each object o to a label L_o, and maps each process p to a *declassification* D_p. D_p holds a set of categories that HiStar ignores when determining whether or not to allow an access attempted by p. A process p can read from (write to) a file f if L_p flows from (to) the label of L_f over all categories not in D_p. p can create a file with label L_f linked from directory d if (1) L_p flows to L_f and (2) p can write to d. A process may at any time create a fresh category c, at which point it is the only process that declassifies c.

A process can create a *gate*, which is a program module that another process p can execute to perform fixed operations on sensitive information. A process p can create a gate g with label L_g and declassification D_g if both (1) L_p flows to L_g over categories not in D_p and (2) D_g is contained in D_p. A process q can then execute g with declassification $D \subseteq D_q \cup D_g$. Moreover, q can execute gate g with a label L such that L_q and L_g flow to L over all categories not in D.

The complete design of HiStar is more complex than the system described above: it includes *four* levels, additional *clearance and verify labels* for processes and gates, and primitives that a process can invoke to update its label throughout its execution (not just when calling a gate). We omit descriptions of these features to simplify the presentation of our approach, but the actual implementation of the approach described in this dissertation supports such features.

8 Overview

In this section, we illustrate by example the instrumentation problem for HiStar, and our technique to solve the problem. In §8.1, we introduce a DIFC program `auth_log` that we use as a running example. In §8.2, we present a policy in our policy language that describes the non-interference and functionality requirements of `auth_log`. In §8.3, we describe an instrumentation of `auth_log` that satisfies the policy.

8.1 `auth_log`: an append-only logging service

Fig. 8.1 contains pseudocode for a program `auth_log`, which uses HiStar label operations to maintain a log file and provide a gate that `auth_log`'s environment can use to append to the log file. For now, ignore the lines highlighted in gray: these are the instrumentation code introduced by our technique, and are described in §8.3.

`auth_log` consists of two modules: `log_init` and `logger`. Program modules are called asynchronously by an environment, which by convention provides to a module a *return gate* linked from the object symbol `RET`. By convention, when the module completes execution, it calls the return gate to return control to its environment. `log_init` loads the root object into an object variable (line `L0`), creates a log file linked from the root at symbol `LOG` (line `L1`), creates a gate linked from the root at symbol `LOGGER` and bound to the `logger` module (lines `L2`), loads the return gate into object variable `ret` (line `L3`), and returns control to its environment

```
log_init:
        rt := ld_root();
        // Create category c to protect integrity of LOG.
LBL1a:  c := create_cat();
LBL1b:  set_op_label(upd_lv(mem_label(), c, LOW));
        // Create log as child of root.
L1:     create(rt, LOG);
        // Create LOGGER gate that owns c (owned by memory)
LBL2a:  set_op_declass(mem_declass());
L2:     creategate(rt, LOGGER, logger);
L3:     ret := ld(rt, RET);
        // Return to the environment unable to modify log
        // (higher than LOW at c and not owning c)
LBL4a:  set_op_label(upd_lv(mem_label(), c, MID));
LBL4b:  set_op_declass(rem_declass_cat(mem_declass(), c));
L4:     gate_call(ret);

logger:
        // Append the message to LOG
L5:     msg := ld(rt, MSG);
L6:     txt := read(msg);
L7:     log := ld(rt, LOG);
L8:     append(log, txt);
L9:     ret := ld(rt, RET);
        // Call the environment without ownership of c
LBL9a:  set_op_label(mem_label());
LBL9b:  set_op_declass(rem_declass_cat(mem_declass(), c));
L10:    gate_call(ret);
```

Figure 8.1: `auth_log`: a collection of `difc` modules, `log_init` and `logger`, that implement an append-only log service. `log_init` initializes a log object at link `LOG` and a gate whose module is `logger`. `logger` reads a message from the object at link `MSG`, and appends the value read to `LOG`. Operations on labels and declassifications, which are generated by our technique, and accompanying comments are highlighted with a gray background. Our technique does not actually generate comments that accompany label and declassification operations.

```
log_client:
C0: rt := ld_root();
C1: msg := ld(rt, MSG);
C2: write(msg, ‘‘new login attempted’’);
C3: log_gt := ld(rt, LOGGER);
    set_op_label(mem_label());
    set_op_declass(mem_declass());
C4: gate_call(log_gt);

mal_client:
M0: write(LOG, ‘‘no logins ever attempted’’);
```

Figure 8.2: Pseudocode for `log_client`, a cooperative client of `auth_log`, and `mal_client`, a malicious client of `auth_log`.

by calling the return gate (line `L4`). `logger` appends the message value in `msg` to the log file (lines `L5`–`L8`), and returns control to its environment by calling the return gate (lines `L9`–`L10`).

Fig. 8.2 contains pseudocode for two clients of `auth_log`: `log_client` and `mal_client`. `log_client` cooperates with `logger` to append to `LOG`: `log_client` writes a new message to `MSG` (lines `C0`–`C2`) and calls the `logger` gate to append the message to `LOG` (lines `C3`–`C4`). `mal_client` attempts to violate the integrity of `LOG` by writing a message directly to `LOG` (line `M0`).

8.2 Policies for `auth_log`

Our goal is to automatically instrument `auth_log` to use label operations to provide sufficient access rights when interacting with a cooperating environment, but satisfy non-interference when interacting with a malicious environment. In particular:

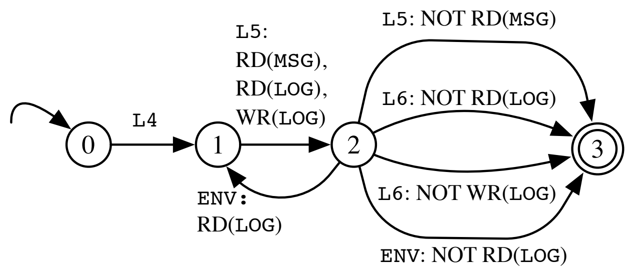

Figure 8.3: `log_ar`: a DIFC policy for `auth_log`. `log_ar` accepts traces of `difc` states in which `auth_log` executes operations on objects or transfers control to its environment with insufficient access rights.

1. After `auth_log`'s environment executes `log_init`, the environment should be able to read from `LOG`.

2. If `logger` is entered in a state in which it can read from `MSG` and read from and write to `LOG`, then it successfully reads from `MSG` and appends to `LOG`. If `logger` then returns control to its environment, then the environment can read from `LOG`.

3. Information does not flow from the environment to `LOG`, except through the value in `MSG` read by `auth_log`.

The requirements for `auth_log` can be expressed as a pair of policy automata. The policy specifying the *access rights* to read from and write to files that the program must hold when `auth_log` transfers control to its environment (items 1 and 2) is represented as a finite-state automaton over an alphabet of conditions on `difc` states. A trace of `difc` states `t` violates a DIFC policy `F` if the states of `t` satisfy a trace of conditions that is accepted by `F`.

Example 4. Fig. 8.3 contains a DIFC policy `log_ar` that explicitly expresses the desired access-rights policy for `auth_log`. `log_ar` is an automaton over

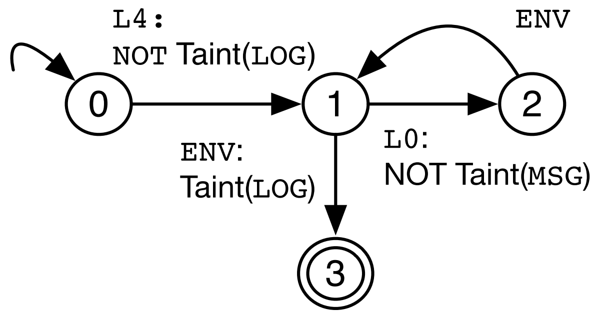

Figure 8.4: `log_ni`: a taint policy for `auth_log`.

an alphabet in which each symbol is a control location paired with a set of conditions on a DIFC store. `log_ar` accepts sequences of conditions that represent an execution of `auth_log` in which (1) `log_init` completes execution (by executing the operation at control location `L4`, on which `log_ar` transitions from state 0 to state 1); (2) `logger` is entered and memory has the rights to (a) read from `MSG`, (b) read from `LOG`, and (c) write to `LOG` (by executing the operation at control location `L5`, on which `log_ar` transitions from state 1 to state 2); (3) `logger` attempts (a) to read from `MSG` at `L5` without the right to read from `MSG` or (b) to append to `LOG` at `L6` without either the right to read from or write to `LOG` (on which `log_ar` transitions from state 2 to state 3).

A temporal non-interference policy defines undesired flows from a set of source objects to sink objects.

Example 5. The non-interference policy for `auth_log` can be expressed as an automaton `log_ni` over an alphabet in which each symbol is a control location paired with a condition on a DIFC store. In particular, each condition is defined over predicates of the form `Taint(X)` that stores whether the program execution may have influenced the value stored in the object stored in object variable `X`. As in existing work on DIFC languages [26, 41], the information stored in the taint predicate over-approximates information about what objects may store different values over different executions of a program. We describe the Taint predicate and its relationship to non-interference properties of a difc program in more detail in App. A.1. For now, its intuitive meaning suffices to understand the policy specified by log_ni.

In particular, log_ni accepts sequences of conditions that represent an execution of auth_log in which (1) log_init completes execution with LOG untainted (by executing the operation at control location L4, on which log_ni transitions from state 0 to state 1); (2) log_ni executes logger an unbounded number of times with MSG untainted (by executing the operation at control location L0, on which log_ni transitions from state 1 to state 2, and then transferring control to the environment, on which log_ni transitions from state 2 to state 1); (3) LOG is tainted.

8.3 Instrumenting auth_log

The complete auth_log in Fig. 8.1, including label operations highlighted in gray, satisfies the DIFC policy log_ar and non-interference policy log_ni. The semantics of the label operations used by auth_log are described briefly in Chapter 7, and in detail in §9.1.3. The instrumented log_init creates a fresh category c (LBL1a), and log_init and logger use c to satisfy log_ar and log_ni. In particular, when log_init returns control to its environment, the label of memory (chosen at LBL4b) is higher at c than the label of the log file at c (chosen at line LBL1b), and the declassification of memory (chosen at line LBL4a) does not contain c; thus, no matter what label operations the environment executes, including the operations in mal_client (Fig. 8.2), the environment cannot write to the log file, ensuring that log_ni remains in state 2. However, because memory can read from an object with a lower label, the environment can read from

LOG, which ensures that when logger exits, log_ar transitions to state 1, not to state 3.

The instrumentation algorithm implemented in our policy weaver for HiStar, hiweave, takes as input the version of auth_log that executes no label operations (i.e., auth_log in Fig. 8.1 with the label operations in gray removed), the DIFC policy log_ar, and the non-interference policy log_ni, and automatically instruments auth_log to execute the label operations depicted in Fig. 8.1. The primary programming challenge addressed by hiweave in the context of auth_log is to model soundly all possible behaviors of auth_log's environment, which may include arbitrary traces of label operations that the environment executes to either cooperatively attempt to call the logger gate (e.g., log_client, Fig. 8.2) or maliciously attempt to directly write to LOG (e.g., mal_client, Fig. 8.2). The technique applied by hiweave to address these challenges is: (1) define a program auth_log′ whose runs are the runs of multiple possible instrumentations of auth_log; (2) compute a finite over-approximation auth_log# of the language of runs of auth_log′ that violate log_ar or log_ni; (3) use auth_log# to construct a game G for which each play models a run of auth_log′, and each Attacker-winning play models a run of auth_log# that may result in a violation of log_ar or log_ni; (4) try to find a winning Defender strategy D of G; (5) from D, instrument auth_log to execute label operations throughout each run r that correspond to the actions chosen by D throughout the play that models r.

A fragment of the game constructed by hiweave to weave auth_log to satisfy log_ar and log_ni is depicted in Fig. 8.5. Each game state models a triple consisting of a state of auth_log′, a state of log_ar, and a state of log_ni. Each state is depicted in Fig. 8.5 as a node annotated with (1) the control location of the state of logger, (2) the state of log_ar, and (3) the state of log_ni that it models. The game fragment illustrates that in general, the weaver can instrument programs to satisfy multiple policy

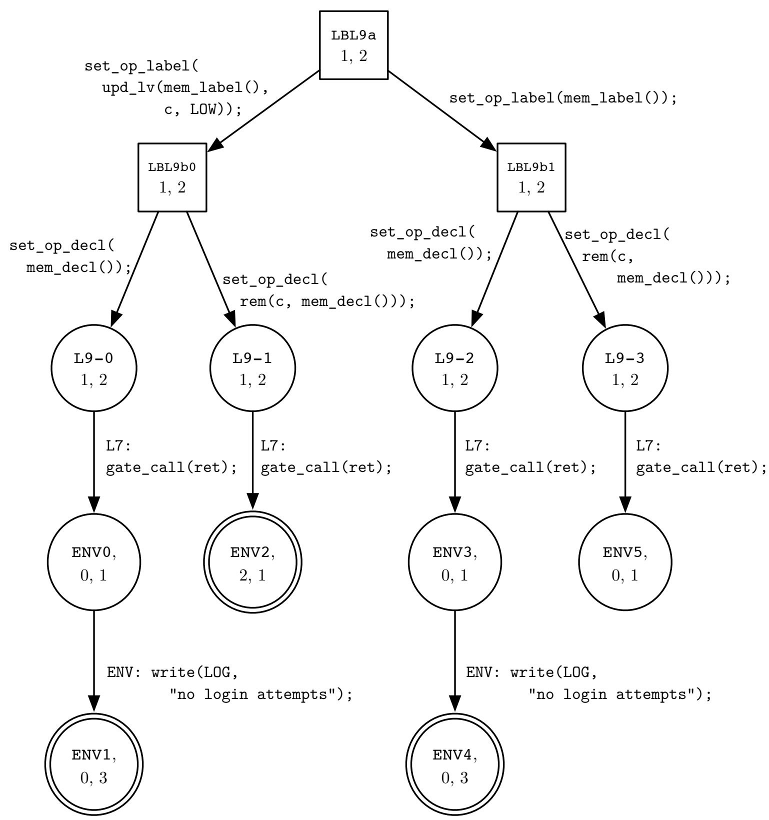

Figure 8.5: Fragment of the game modeling the problem of instrumenting `auth_log` to satisfy `log_ar` and `log_ni` from location `L10`. Defender states are depicted as squares, Attacker states are depicted as circles, and Attacker-winning states are depicted as doubled circles.

automata by constructing a game whose states simultaneously track the states of each policy automaton. However, for simplicity, we describe the weaver as taking as input only a single policy automaton. Each edge between states is annotated with the program operation on which the game transitions.

`Hiweave` actually constructs a game from a finite over-approximation `auth_log`$^{\#}$ of the language of executions of `auth_log`′. Such an abstraction will, for example, merge "similar" states that, e.g., differ only in the *number* of categories allocated, but not in the levels that the labels of objects hold at each category. In Fig. 8.5, we have depicted a fragment of the game constructed directly from `auth_log`′, for simplicity.

Each play of the game fragment depicts a potential instrumentation of `logger` immediately before `logger` invokes the gatecall operation at control location `L9`. The game fragment is reached by executing the label operations in the instrumentation of `log_init` depicted in Fig. 8.1, which create a category c stored in category variable `c`. The play $\mathtt{p}_0$ in which the Defender chooses label operations that drive the game to the state with control location `L9-0` models an execution of an instrumentation of `logger` that returns control to its environment with the label of memory updated to map c to level Low and with the declassification of memory. $\mathtt{p}_0$ ends in an Attacker winning state, which models the fact that if `logger` returns to its environment after executing the modeled label operations, then the environment can write to `LOG`, which violates `log_ni`.

The play $\mathtt{p}_1$ in which the Defender chooses label operations that drive the game to the state with control location `L9-1` models an execution of an instrumentation of `logger` that returns control to its environment with the label of memory updated with category c set to Low and with a declassification that does not contain category `c`. $\mathtt{p}_1$ ends in an Attacker winning state, which models the fact that if `logger` returns to its environment after executing the modeled label operations, then the environment will not be able to read from LOG, which would violate log_ar.

The play p_2 in which the Defender chooses label operations that drive the game to the state with control location L9-2 models an execution of an instrumentation of logger in which logger returns control to its environment with the label and declassification of memory. p_2 ends in a winning state for the Attacker, for a reason analogous to the reason that play p_0 ends in a winning Attacker state.

The play p_3 in which the Defender chooses label operations that drive the game to the state with control location L9-3 models an execution of an instrumentation of logger in which logger returns control to its environment with the label of memory and with the declassification of memory with the category c removed. $p_{r,r}$ ends in a state q that is not a winning state for the Attacker. Furthermore, there is a winning Defender strategy from q for the complete game constructed by hiweave for auth_log, log_ar, and log_ni. This models the fact that the modeled label operations satisfy the log_ar and log_ni, as explained in detail in §8.3.

9

Technical approach

In this chapter, we describe the technical details of our approach to solving the HiStar weaving problem. In §9.1, we define the syntax and semantics of a DIFC programming language `difc`. In §9.2, we formulate the conditions under which one `difc` program is a valid instrumentation of another `difc` program. In §9.3.2, we define a language of DIFC policies for `difc` programs. In §9.4, we define the problem of instrumenting a `difc` program to satisfy a DIFC policy. In §9.5, we describe our technique for solving the instrumentation problem.

9.1 `difc`: a language of DIFC programs

In this section, we first define a core language `difccore` of imperative programs (§9.1.1) without DIFC features. We then use `difccore` to define the syntax (§9.1.2) and semantics (§9.1.3) of our subject DIFC language, `difc`.

9.1.1 `difccore`: a core language

`difccore` syntax

A `difccore` program reads values from a rooted graph of objects into memory, computes operations on the loaded values, and writes the computed values to objects. The syntax of a `difccore` program is given in Fig. 9.1, and is defined over fixed finite sets of module symbols (MODSYMS), con-

$$\mathsf{Prog} := (\mathsf{MODSYMS} : (\mathsf{LOC} : \mathsf{Op})^*)^* \tag{9.1}$$
$$\mathsf{Op} := \mathsf{DATAVARS} \coloneqq \mathsf{OP}(\overline{\mathsf{DATAVARS}}) \tag{9.2}$$
$$\mid \mathsf{DATAVARS} \mathbin{?} \mathsf{LOC} : \mathsf{LOC} \tag{9.3}$$
$$\mid \mathsf{DATAVARS} \coloneqq \mathtt{RD}(\mathsf{OBJVARS}) \tag{9.4}$$
$$\mid \mathtt{WR}(\mathsf{OBJVARS}, \mathsf{DATAVARS}) \tag{9.5}$$
$$\mid \mathsf{OBJVARS} \coloneqq \mathtt{ld\_root}() \tag{9.6}$$
$$\mid \mathsf{OBJVARS} \coloneqq \mathtt{ld}(\mathsf{OBJVARS}, \mathtt{LINKS}) \tag{9.7}$$
$$\mid \mathsf{OBJVARS} \coloneqq \mathtt{create}(\mathsf{OBJVARS}, \mathtt{LINKS}) \tag{9.8}$$
$$\mid \mathsf{g} \coloneqq \mathtt{creategate}(\mathsf{OBJVARS}, \mathtt{LINKS}, \mathsf{MODSYMS}) \tag{9.9}$$
$$\mid \mathtt{gatecall}(\mathsf{OBJVARS}) \tag{9.10}$$

Figure 9.1: Syntax of `difccore`, a core programming language that operates over data values.

trol locations (LOC), link symbols (`LINKS`), object variables (OBJVARS), and data variables (DATAVARS). A `difccore` program is a sequence of bindings from a module symbol to a sequence of operations, each annotated with a control location (Eqn. (9.1)); each location may annotate at most one operation. An operation may compute a value from values in data variables and store the result in a data variable (Eqn. (9.2), where OP is a set of standard arithmetic operations over integers), or may branch control flow based on the value in a data variable (Eqn. (9.3)). An operation also may read a value from an object to a data variable (Eqn. (9.4)), may write a value in a data variable to an object (Eqn. (9.5)), may load the root object into an object variable (Eqn. (9.6)), may load an object linked from an object in an object variable into an object variable (Eqn. (9.7)), may create an object (Eqn. (9.8)), may create a gate, (Eqn. (9.9)), or may call a gate (Eqn. (9.10)).

A `difccore` program P can be represented as an annotated control-flow graph. Each `difccore` program P defines (1) a function $\mathsf{LocMod}_\mathsf{P} : \mathsf{LOC} \rightarrow$

$\mathsf{MODSYMS}$ from each control location L to the module that contains an operation annotated with L, (2) a function $\mathsf{ModInit}_\mathsf{P} : \mathsf{MODSYMS} \rightarrow \mathsf{LOC}$ from each module `M` to the location that annotates the initial operation of `M` in P, and (3) a control-flow graph $(\mathsf{N}_\mathsf{P}, \mathsf{E}_\mathsf{P})$. The control nodes N_P are the control locations LOC, and the control edges $\mathsf{E} \subseteq \mathsf{N} \times \mathsf{Op} \times \mathsf{N}$ are defined by the structure of each module in P.

Example 6. Let `nolbl_log` be the `difccore` program formed by removing all label and declassification operations from `auth_log` (§8.1, Fig. 8.1). The `log_init` module of `nolbl_log` is a sequence of operations that load the root object (`rt := ld_root()`), create a log object as a child of the root (`create(rt, LOG)`), create a gate as a child of the root (`creategate(rt, LOGGER, logger)`), load the return gate (`ret := ld(rt, RET)`), and call the return gate (`gatecall(ret)`).

`difccore` semantics

A `difccore` program P defines a transition relation over `difccore` states. Let the set of control locations $\mathtt{LOC}_\mathsf{E}$ be the set of control locations extended with a distinguished control location ENV that models the program's environment. Each `difccore` state consists of a control location in $\mathtt{LOC}_\mathsf{E}$ and a value store, which is a graph of objects in which each object is bound to a data value and each edge between objects is annotated with a link symbol. Let O^* be an infinite universe of objects, containing elements Mem and Root. A *value store* $\sigma = (\mathsf{D}, \mathsf{O}, \delta, \mu, \rho, \nu)$ is a six-tuple of (1) a valuation of data variables $\mathsf{D} : \mathsf{DATAVARS} \rightarrow \mathbb{Z}$, (2) a set of objects $\mathsf{O} \subseteq \mathsf{O}^*$ containing distinguished elements Mem and Root, (3) a map from each object to a data value $\delta : \mathsf{O} \rightarrow \mathbb{Z}$, (4) a partial map from objects to module symbols $\mu : \mathsf{O} \rightarrow \mathsf{MODSYMS}$, (5) an evaluation of object variables $\rho : \mathsf{OBJVARS} \hookrightarrow \mathsf{O}$, and (6) a partial map from objects and link symbols to objects $\nu : \mathsf{O} \times \mathtt{LINKS} \hookrightarrow \mathsf{O}$. We denote the components of a value store σ as D^σ, O^σ, δ^σ, μ^σ, ρ^σ, and ν^σ, respectively, and denote the space of all

$$\text{intra}\,\frac{(\texttt{L},\texttt{o},\texttt{L'}) \in E'_P \qquad \langle \sigma, \texttt{o}\rangle \rightarrow_{\texttt{dc}} \sigma'}{\langle (\texttt{L},\sigma),\texttt{o}\rangle \rightarrow_P \langle \texttt{L'},\sigma'\rangle}$$

$$\text{gatecall-P}\,\frac{\texttt{L}' = \mathsf{ModInit}_P(\mu^{\sigma}(g))}{\langle (\texttt{L},\sigma),\texttt{gatecall}(g)\rangle \rightarrow_P (\texttt{L}',\sigma)}$$

$$\text{gatecall-non-P}\,\frac{\mathsf{ModInit}_P(\mu^{\sigma}(g)) = \uparrow}{\langle (\texttt{L},\sigma),\texttt{gatecall}(g)\rangle \rightarrow_P (\mathsf{ENV},\sigma)}$$

Figure 9.2: Inference rules that define transition relation $\rightarrow_P$ of a `difccore` program P.

value stores as DIFCCoreStores. A `difccore` *state* $(\texttt{L},\sigma)$ is a control location $\texttt{L} \in \texttt{LOC}_E$ and value store σ. We denote the space of all `difccore` states as $\mathsf{CoreStates} = \texttt{LOC}_E \times \mathsf{DIFCCoreStores}$.

Example 7. A `difccore` state that can be reached by executing `nolbl_log` (Ex. 6) is $(\texttt{L9},\sigma)$, where σ is the following value store:

- The set of objects contains memory, the root object, the log object, the message object, the `logger` gate, and a return gate bound to object variable `ret`.
- There are links from the root object to (1) the log object on link symbol `LOG`, (2) the message object on link symbol `MSG`, and (3) the logger gate on link symbol `LOGGER`.
- The object linked from root on symbol `LOGGER` is bound to module `logger`.

For `difccore` program P, the transition relation $\rightarrow_P \subseteq \mathsf{CoreStates} \times \mathsf{Op} \times \mathsf{CoreStates}$ of a `difccore` program P is defined by semantic inference rules given in Fig. 9.2. Let $E'_P \subseteq \texttt{LOC}_E \times \mathsf{OP} \times \texttt{LOC}_E$ be E_P extended to contain an edge from ENV to ENV on each operation that is not a gatecall (i.e.,

$$\begin{aligned}
\mathsf{Op} &\coloneqq \mathsf{CVAR} \coloneqq \texttt{create_cat}() && (9.11)\\
&\mid \texttt{set_op_label}(\mathsf{LExpr}) && (9.12)\\
&\mid \texttt{set_op_declass}(\mathsf{DExpr}) && (9.13)\\
\mathsf{LExpr} &\coloneqq \texttt{mem_label}() && (9.14)\\
&\mid \texttt{upd_lv}(\mathsf{LExpr}, \mathsf{CVAR}, \mathsf{LEVELS}) && (9.15)\\
\mathsf{DExpr} &\coloneqq \texttt{mem_decl}() && (9.16)\\
&\mid \texttt{rem_decl_cat}(\mathsf{DExpr}, \mathsf{CVAR}) && (9.17)
\end{aligned}$$

Figure 9.3: Label operations of difc that extend Op.

each *intra-module* operation). If $\mathtt{o}$ is an intra-module operation (Rule intra), then pre-state $(\mathtt{L}, \sigma)$ transitions to post-state $(\mathtt{L}', \sigma')$ on $\mathtt{o}$ if $\mathtt{L}'$ is a control-successor of $\mathtt{L}$ on $\mathtt{o}$ ($(\mathtt{L}, \mathtt{o}, \mathtt{L}') \in \mathtt{E}'_{\mathtt{P}}$) and pre-store σ transitions to post-store σ' on $\mathtt{o}$ ($\langle \sigma, \mathtt{o} \rangle \rightarrow_{\mathtt{dc}} \sigma'$). The definition of $\rightarrow_{\mathtt{dc}}$ is straightforward from the informal definitions of each intra-module operation, and we omit a full description.

For an operation $\mathtt{gatecall}(\mathtt{g})$, if the gate $\mathtt{g}$ bound to $\mathtt{g}$ is bound to a module $\mathtt{M}$ in $\mathtt{P}$ (Rule gatecall-P), then pre-state $(\mathtt{L}, \sigma)$ transitions to a post-state whose control location is the initial location of $\mathtt{M}$, and whose store is σ. If $\mathtt{g}$ is not bound to any module (Rule gatecall-P), then pre-state $(\mathtt{L}, \sigma)$ transitions to a post-state whose control location is ENV (the control location of the environment), and whose store is σ.

9.1.2 difc syntax

A difc program is a difccore program whose operations are the difccore operations extended with a set of label operations, given in Fig. 9.3; we refer to the space of all operations of difc programs as Op. Op is defined over the space of category variables (CVAR), and the space of levels LEVELS, ordered $\mathsf{Low} < \mathsf{Mid} < \mathsf{High}$. A label operation may create a fresh category

(Eqn. (9.11)), set the value of a label expression as the label to be used by the next operation (i.e., the *operation label*; Eqn. (9.12)) or set the value of a declassification expression to be the declassification used by the next operation (i.e., the *operation declassification*; Eqn. (9.13)).

A label expression is either the memory label (Eqn. (9.14)) or a label expression updated to map a category in a variable to a level (Eqn. (9.15)). A declassification expression is either the memory declassification (Eqn. (9.16)) or a category removed from a declassification expression (Eqn. (9.17)).

Example 8. `auth_log` contains the label operation $\mathtt{o} \equiv \mathtt{set\_op\_label}(\mathtt{upd\_lv}(\mathtt{mem\_label}(), \mathtt{c}, LOW))$. `o` sets the operation label to be the label of memory, updated to map the category in category variable `c` to level Low.

9.1.3 `difc` semantics

In this section, we define a semantics for `difc` by defining a space of `difc` states (§9.1.3), and, for each `difc` program $\mathtt{P}$, a transition relation over `difc` states (§9.1.3).

`difc` states

A `difc` program defines a transition relation over the space of `difc` states. A `difc` state consists of a control location, a value store, and a *label store*. Let C^* be an infinite set of *categories* that do not overlap with the set of objects (i.e., $O^* \cap C^* = \emptyset$). Let a *label* $L \in \mathcal{L}$ be a function that maps each category to a level (i.e., $\mathcal{L} = C^* \to \mathsf{LEVELS}$), and let a *declassification* $D \in \mathcal{D}$ be a set of categories (i.e., $\mathcal{D} = \mathcal{P}(C^*)$, where for a set S, $\mathcal{P}(S)$ denotes the power-set of S). A label store $(C, \lambda, \kappa, L_{\mathtt{op}}, D_{\mathtt{op}})$ is a five-tuple consisting of (1) *categories* $C \subseteq C^*$ created by the program, (2) a *store label* $\lambda : O \to \mathcal{L}$, (3) a *store declassification* $\kappa : O \hookrightarrow \mathcal{D}$, (4) the operation label $L_{\mathtt{op}} \in \mathcal{L}$, and (5) the operation declassification $D_{\mathtt{op}} \in \mathcal{D}$. The operation label and operation declassification store the value of the next label and declassification to be used by an operation.

Labels and declassifications are notions that should only be visible at the `difc` level, not the `difccore` level. Thus, `difccore` operations, such as `create`, `creategate`, and `gatecall`, should not require any label-valued or declassification-valued parameters. To avoid having to redefine the signatures of `create`, etc. at the `difc` level, we introduced the operation label (i.e., (4)) and the operation declassification (i.e., (5)) in the `difc` state, along with the operations `set_op_label` and `set_op_declass` in the `difc` syntax to manipulate them.

We denote the space of label stores as $\mathsf{LabelStores}$. For label store σ, we denote the categories, environment categories, store label, store declassification, operation label, and operation declassification of σ as C^{σ}, λ^{σ}, κ^{σ}, $L^{\sigma}_{\mathtt{op}}$, and $D^{\sigma}_{\mathtt{op}}$, respectively.

For label store σ, we refer to $\lambda^{\sigma}(\mathsf{Mem})$, and $\kappa^{\sigma}(\mathsf{Mem})$ as the *memory label* and *memory declassification*, respectively, in σ.

A *`difc` store* is a value store paired with a label store; we denote the space of `difc` stores as $\mathtt{difc}\mathsf{Stores} = \mathsf{DIFCCoreStores} \times \mathsf{LabelStores}$. A *`difc` state* is a control location paired with a `difc` store; we denote the space of `difc` states as $Q_{\mathtt{d}} = \mathtt{LOC}_{\mathrm{E}} \times \mathtt{difc}\mathsf{Stores}$.

Example 9. Before `auth_log` executes the operation at control location `L9` (§8.1, Fig. 8.1), it can reach the `difc` state $(\mathtt{L9}, (\sigma_V, \sigma_L))$, where σ_V is the value store introduced in Ex. 7 and σ_L consists of the following components:

1. The set of categories contains the category c created at location `LBL1a`.

2. The store label maps each object other than `LOG` to a label that maps c to Mid. The label of `LOG` maps c to Low.

3. The store declassification maps memory and the logger gate to a declassification that contains only c, and maps the return gate to a declassification that is the empty set of categories.

4. The operation label is equal to the label of memory.

5. The operation declassification is the empty set of categories.

The premises of many rules that define the semantics of `difc` operations use a flows-to relation over labels, which defines when information may flow from one object to another.

Definition 16. For labels L_0 and L_1 and categories C, L_0 *flows to* L_1 *over* C (denoted by $L_0 \sqsubseteq_C L_1$) if the level of L_0 is at least as low as the level of L_1 at each category in C. That is, $L_0 \sqsubseteq_C L_1$ if and only if $\forall c \in C.\ L_0(c) \leqslant L_1(c)$.

The semantics of `difc` operations often will be defined using the flows-to relation over the set of categories not declassified by memory. For label store Λ, we use $\sqsubseteq^{\Lambda}_{\mathsf{Mem}}$ to denote $\sqsubseteq_{C^* \setminus \kappa^{\Lambda}(\mathsf{Mem})}$. Typically, a label store maps objects to labels with a labeling function λ, and the premises of a semantic rule or property of stores in a policy compares the labels of two objects o and p under λ. When λ is clear from context, for simplicity, we will simply say that "o flows to p," rather than saying that "the label of o under λ flows to the label of p under λ."

`difc` transitions

A `difc` program P defines a transition relation $\rightarrow_P \subseteq Q_{\mathtt{d}} \times \mathtt{Op} \times Q_{\mathtt{d}}$. $\rightarrow_P$ is defined by the transition relation $\rightarrow_{\mathtt{d}} \subseteq \mathtt{difc}\mathsf{Stores} \times \mathtt{Op} \times \mathtt{difc}\mathsf{Stores}$ that relates `difc` pre-states, operations, and post-stores. Inference rules that define $\rightarrow_{\mathtt{d}}$ for a selection of `difc` operations are given in Fig. 9.4 and Fig. 9.5. The rules in Fig. 9.4 define the semantics of operations that check, but do not update, the label store of a pre-store. For `difc` stores

$$\text{read}\ \frac{\langle V, \texttt{x} \coloneqq \texttt{RD}(\texttt{o})\rangle \rightarrow_{\texttt{dc}} V' \qquad \lambda^{\Lambda}(\rho^{V}(\texttt{o})) \sqsubseteq^{\Lambda}_{\mathsf{Mem}} \lambda^{\Lambda}(\mathsf{Mem})}{\langle (V, \Lambda), \texttt{x} \coloneqq \texttt{RD}(\texttt{o})\rangle \rightarrow_{\texttt{d}} (V', \Lambda)}$$

$$\text{write}\ \frac{\langle V, \texttt{WR}(\texttt{o}, \texttt{x})\rangle \rightarrow_{\texttt{dc}} V' \qquad \lambda^{\Lambda}(\mathsf{Mem}) \sqsubseteq^{\Lambda}_{\mathsf{Mem}} \lambda^{\Lambda}(\rho^{V}(\texttt{o}))}{\langle (V, \Lambda), \texttt{WR}(\texttt{o}, \texttt{x})\rangle \rightarrow_{\texttt{d}} (V', \Lambda)}$$

$$\text{load}\ \frac{\langle V, \texttt{o} \coloneqq \texttt{ld}(\texttt{d}, \texttt{l})\rangle \rightarrow_{\texttt{dc}} V' \qquad \lambda^{\Lambda}(\rho^{V}(\texttt{d})) \sqsubseteq^{\Lambda}_{\mathsf{Mem}} \lambda^{\Lambda}(\mathsf{Mem})}{\langle (V, \Lambda), \texttt{o} \coloneqq \texttt{ld}(\texttt{d}, \texttt{L})\rangle \rightarrow_{\texttt{d}} (V', \Lambda)}$$

Figure 9.4: Semantic inference rules for `difc`. The rules partially define a transition relation $\rightarrow_{\texttt{d}} \subseteq \texttt{difc}\mathsf{Stores} \times \texttt{Op} \times \texttt{difc}\mathsf{Stores}$ from a `difc` pre-store and operation to a post-store.

$\sigma, \sigma' \in \texttt{difc}\mathsf{Stores}$ and operation $\texttt{op} \in \texttt{Op}$, σ transitions to σ' on `op` under the following conditions:

- If `op` is an operation on data variables or a control branch, then σ' is identical to σ.

- If `op` is an operation $\texttt{x} \coloneqq \texttt{RD}(\texttt{o})$ and the object o bound to object variable `o` flows to memory, then σ' is σ with a value store updated according to the `difccore` semantics (Rule read).

- If `op` is an operation $\texttt{WR}(\texttt{o}, \texttt{x})$ and memory flows to the object o bound to object variable `o`, then σ' is σ with a value store updated according to the `difccore` semantics (Rule write).

- If `op` is an operation $\texttt{x} \coloneqq \texttt{ld}(\texttt{d}, \texttt{l})$ and the object d bound to object variable `d` flows to memory, then σ' is σ with a value store updated according to the `difccore` semantics (Rule load).

The rules in Fig. 9.4 define the semantics of operations that check and update the label store of a pre-store:

$$\text{create}\ \frac{\begin{array}{c}\langle V, \mathtt{o} \coloneqq \mathtt{create}(\mathtt{d},\mathtt{L})\rangle \to_{\mathtt{dc}} V' \qquad \mathtt{o} \in O^{V'} \setminus O^{V} \\ \lambda^{\wedge}(\mathsf{Mem}) \sqsubseteq^{\wedge}_{\mathsf{Mem}} \lambda^{\wedge}(\rho^{V}(\mathtt{d})) \\ \lambda^{\wedge}(\mathsf{Mem}) \sqsubseteq^{\wedge}_{\mathsf{Mem}} L^{\wedge}_{\mathtt{op}} \qquad \lambda' = \lambda^{\wedge}[\mathtt{o} \mapsto L^{\wedge}_{\mathtt{op}}] \\ \Lambda' = (C^{\wedge}, \lambda', \kappa^{\wedge}, L^{\wedge}_{\mathtt{op}}, D^{\wedge}_{\mathtt{op}})\end{array}}{\langle (V, \Lambda), \mathtt{o} \coloneqq \mathtt{create}(\mathtt{d},\mathtt{L})\rangle \to_{\mathtt{d}} (V', \Lambda')}$$

$$\text{create-gate}\ \frac{\begin{array}{c}\langle V, \mathtt{g} \coloneqq \mathtt{creategate}(\mathtt{d},\mathtt{L},\mathtt{M})\rangle \to_{\mathtt{dc}} V' \qquad \mathtt{o} \in O^{V'} \setminus O^{V} \\ \lambda^{\wedge}(\mathsf{Mem}) \sqsubseteq^{\wedge}_{\mathsf{Mem}} \lambda^{\wedge}(\rho^{V}(\mathtt{d})) \\ \lambda^{\wedge}(\mathsf{Mem}) \sqsubseteq^{\wedge}_{\mathsf{Mem}} L^{\wedge}_{\mathtt{op}} \qquad \lambda' = \lambda^{\wedge}[\mathtt{o} \mapsto L^{\wedge}_{\mathtt{op}}] \\ D_{\mathtt{op}} \subseteq \kappa^{\wedge}(\mathsf{Mem}) \qquad \kappa' = \kappa^{\wedge}[\mathtt{o} \mapsto D_{\mathtt{op}}] \\ \Lambda' = (C^{\wedge}, \lambda', \kappa', L^{\wedge}_{\mathtt{op}}, D^{\wedge}_{\mathtt{op}})\end{array}}{\langle (V, \Lambda), \mathtt{g} \coloneqq \mathtt{creategate}(\mathtt{d},\mathtt{L},\mathtt{M})\rangle \to_{\mathtt{d}} (V', \Lambda')}$$

$$\text{gatecall}\ \frac{\begin{array}{c}g = \rho^{V}(\mathtt{g}) \qquad D^{\wedge}_{\mathtt{op}} \subseteq \kappa^{\wedge}(\mathsf{Mem}) \qquad D = D^{\wedge}_{\mathtt{op}} \cup \kappa^{\wedge}(g) \\ \lambda^{\wedge}(\mathsf{Mem}) \sqsubseteq_{C^{*} \setminus D} L^{\wedge}_{\mathtt{op}} \qquad \lambda^{\wedge}(g) \sqsubseteq_{C^{*} \setminus D} L^{\wedge}_{\mathtt{op}} \\ \lambda' = \lambda^{\wedge}[\mathsf{Mem} \mapsto L^{\wedge}_{\mathtt{op}}]] \qquad \kappa' = \kappa^{\wedge}[\mathsf{Mem} \mapsto D] \\ \Lambda' = (C^{\wedge}, \lambda', \kappa', L^{\wedge}_{\mathtt{op}}, D^{\wedge}_{\mathtt{op}})\end{array}}{\langle (V, \Lambda), \mathtt{gatecall}(\mathtt{g}))\rangle \to_{\mathtt{d}} (V', \Lambda')}$$

$$\text{createcat}\ \frac{\begin{array}{c}c \in C^{*} \setminus C^{\wedge} \qquad \kappa' = \kappa^{\wedge}[\mathsf{Mem} \mapsto \kappa^{\wedge}(\mathsf{Mem}) \cup \{c\}] \\ \Lambda' = (C^{\wedge} \cup \{c\}, \lambda^{\wedge}, \kappa', L^{\wedge}_{\mathtt{op}}, D^{\wedge}_{\mathtt{op}})\end{array}}{\langle (V, \Lambda), \mathtt{c} \coloneqq \mathtt{create\_cat}()\rangle \to_{\mathtt{d}} (V, \Lambda')}$$

Figure 9.5: Inference rules that define the transition relation $\to_{\mathtt{d}}$ over `difc` stores.

- If $\texttt{op}$ is an operation $\texttt{o} \coloneqq \texttt{create}(\texttt{d}, \texttt{L})$ (Rule create), then (1) memory flows to the object bound to object variable $\texttt{d}$ ($\lambda^{\wedge}(\mathsf{Mem}) \sqsubseteq^{\wedge}_{\mathsf{Mem}} \lambda^{\wedge}(\rho^{\wedge}(\texttt{d}))$) and (2) memory flows to the operation label $L^{\wedge}_{\mathrm{op}}$ ($\lambda^{\wedge}(\mathsf{Mem}) \sqsubseteq L^{\wedge}_{\mathrm{op}}$).

  Let o be the fresh object created according to the $\texttt{difccore}$ semantics for a $\texttt{create}$ operation ($\langle \texttt{o} \coloneqq \texttt{create}(\texttt{d}, \texttt{L}), V \rangle \to_{\texttt{d}} V'$ and $o \in O^{V'} \setminus O^{V}$). σ' is σ with a label store that maps o to $L^{\wedge}_{\mathrm{op}}$ ($\lambda' = \lambda^{\wedge}[o \mapsto L^{\wedge}_{\mathrm{op}}]$).

- If $\texttt{op}$ is an operation $\texttt{g} \coloneqq \texttt{creategate}(\texttt{l}, \texttt{L}, \texttt{M})$ (Rule create-gate) then the conditions on σ, σ', and $\texttt{op}$ for $\texttt{op} \equiv \texttt{o} \coloneqq \texttt{create}(\texttt{d}, \texttt{L})$ apply. Furthermore, the operation declassification $D^{\wedge}_{\mathrm{op}}$ must be contained by the declassification of memory ($D^{\wedge}_{\mathrm{op}} \subseteq \kappa^{\wedge}(\mathsf{Mem})$).

  σ' is σ with a label store that maps o to declassification $D^{\wedge}_{\mathrm{op}}$.

- If $\texttt{op}$ is an operation $\texttt{gatecall}(\texttt{g})$ (Rule gatecall), then let g be the object bound to object variable $\texttt{g}$ ($g = \rho^{\wedge}(\texttt{g})$) and let D be the union of the operation declassification and the declassification of g ($D = D^{\wedge}_{\mathrm{op}} \cup \kappa^{\wedge}(g)$). Then (1) the operation declassification is contained by the declassification of memory ($D^{\wedge}_{\mathrm{op}} \subseteq \kappa^{\wedge}(\mathsf{Mem})$); (2) & (3) memory and g flow to the operation label over all categories not in D ($\lambda^{\wedge}(\mathsf{Mem}) \sqsubseteq_{C^* \setminus D} L$ and $\lambda^{\wedge}(o) \sqsubseteq_{C^* \setminus D} L^{\wedge}_{\mathrm{op}}$).

  σ' is σ with a label store updated to map memory to label $L^{\wedge}_{\mathrm{op}}$ and declassification $D^{\wedge}_{\mathrm{op}}$.

- If $\texttt{op}$ is an operation $\texttt{c} \coloneqq \texttt{create\_cat}()$, then σ' (1) contains a fresh category c not in σ and (2) memory declassifies c.

The semantics of the label operations that set the operation label and operation declassification are straightforward from their informal descriptions (§9.1.2), and so we omit a full description.

Example 10. Each execution of `auth_log` that completes `logger` contains a transition on a `gatecall` operation. Let σ be the `difc` store introduced in Ex. 9. When `logger` executes the operation `gatecall(ret)` from σ, the program successfully takes a step of execution. In particular, the union of the operation declassification $D^{\sigma}_{\mathtt{op}} = \emptyset$ and the declassification of the return gate $\mathtt{r}$ stored in `ret` is $D = \{d\}$. The operation declassification is (trivially) contained by the memory declassification; thus, σ satisfies premise (1) of a `gatecall` operation. Memory and $\mathtt{r}$ flow to the operation label over all categories not in D; thus, σ satisfies premises (2) and (3).

The post-store σ' that results from executing `gatecall(ret)` is σ updated as follows: the label of memory maps $\mathtt{c}$ and $\mathtt{d}$ to Mid; the declassification of memory contains only $\mathtt{d}$.

For `difc` program $\mathtt{P}$ and `difc` states $q, q' \in Q_{\mathtt{d}}$, if there is some operation $\mathtt{op} \in \mathtt{Op}$ for which $\langle q, \mathtt{op} \rangle \rightarrow_{\mathtt{P}} q'$, then q *reaches* q', denoted $q \Rightarrow_{\mathtt{P}} q'$. The reflexive transitive closure of $\Rightarrow_{\mathtt{P}}$ is denoted as $\Rightarrow^{*}_{\mathtt{P}} \subseteq Q_{\mathtt{d}} \times Q_{\mathtt{d}}$.

9.1.4 Program runs

A run of a `difc` program $\mathtt{P}$ is a sequence of `difc` states in which each adjacent pair of states are in the transition relation of $\mathtt{P}$.

Definition 17. Let $\mathtt{P}$ be a `difc` program. Then a sequence of `difc` states $q_0, q_1, \ldots, q_n$ is a *run of* $\mathtt{P}$ if for $0 \leqslant i < n$, $q_i \Rightarrow_{\mathtt{P}} q_{i+1}$.

For module symbol $\mathtt{M} \in \mathsf{MODSYMS}$, `difc` state $q_i = (\mathtt{L}, \sigma) \in Q_{\mathtt{d}}$ is an $\mathtt{M}$-*state* if $\mathsf{LocMod}_{\mathtt{P}}(\mathtt{L}) = \mathtt{M}$. The union of $\mathtt{M}$ states over all module symbols $\mathtt{M}$ are the *module states* of $\mathtt{P}$. If $\mathtt{r}$ is a run of $\mathtt{P}$, then the subsequence of all module states of $\mathtt{r}$ is a *module run* of $\mathtt{P}$. The sequence of all operations executed in a module state of $\mathtt{r}$ is a *module trace* of $\mathtt{P}$.

9.2 Valid instrumentation as label refinement

We formulate a valid instrumentation of a `difc` program by adapting definitions of simulation and refinement (defined in Chapter 2, Defn. 2). Unfortunately, neither simulation nor refinement formulate our intuitive notion of a valid instrumentation, as demonstrated by `auth_log` (introduced in §8.1).

Example 11. Formulating a valid instrumentation of a `difc` program $\mathtt{P}$ as a simulation of $\mathtt{P}$ disallows `difc` programs that satisfy our intuitive notion of a valid instrumentation. E.g., let `nolbl_log` be a version of `auth_log` with the label operations (highlighted in gray in Fig. 8.1) removed. Intuitively, we wish to allow `auth_log` as an instrumentation of `nolbl_log`, but `auth_log` is not a simulation of `nolbl_log` from any pair of `difc` states. In particular, consider a state q at control location $\mathtt{L1}$ of `nolbl_log`. q transitions on operation $\mathtt{log} := \mathtt{create}(\mathtt{d}, \mathtt{l})$ to a state with a fresh log object o, and in which the set of categories is the set of categories in σ. No state of `auth_log` can simulate q, because any such state would transition over the operations $\mathtt{c} := \mathtt{create\_cat}()$; $\mathtt{set\_op\_label}(\mathtt{upd\_lv}(\mathtt{mem\_label}(), \mathtt{c}, \mathtt{LOW}))$;, $\mathtt{log} := \mathtt{create}(\mathtt{rt}, \mathtt{LOG})$ to a state with a store in which there is some category c that is not a category in σ.

Conversely, formulating a valid instrumentation of a `difc` program as a refinement allows an instrumentation of a `difc` program $\mathtt{P}$ to be a trivial program that reproduces none of the behaviors of $\mathtt{P}$. E.g., let `halt` be a trivial `difc` program that contains a control location $\mathtt{L_h}$, and no control edges. `halt` is a refinement of `nolbl_log`.

Intuitively, `difc` program $\mathtt{P}'$ is an instrumentation of $\mathtt{P}$ if $\mathtt{P}'$ can match any sequence of value stores "chosen" over a run of $\mathtt{P}$, but $\mathtt{P}'$ can choose the label stores paired with each value store in the sequence. We formulate this intuition by defining that under such a condition, P' is a *label refinement* of P.

Definition 18. For `difc` programs P and P', a *label-refinement relation* $\sim \subseteq Q_{\mathtt{d}} \times Q_{\mathtt{d}}$ is a relation over `difc` states that satisfies the following conditions:

1. $\sim$ only relates states with equal value stores. I.e., for $q_0 = (\mathtt{L}_0, (V_0, L_0)) \in Q_{\mathtt{d}}$ and $q_1 = (\mathtt{L}_1, (V_1, L_1)) \in Q_{\mathtt{d}}$, if $q_0 \sim q_1$, then $V_0 = V_1$.

2. If a pair of states (q, q') is in $\sim$, then each successor of q on one step of P is paired with a successor of q' over multiple steps of P'. I.e., for $q_0, q'_0 \in Q_{\mathtt{d}}$ such that $q_0 \sim q'_0$, if $q_0 \Rightarrow_{\mathsf{P}} q_1$, then there is some state q'_1 such that $q'_0 \Rightarrow^*_{\mathsf{P}'} q'_1$ and $q_1 \sim q'_1$.

P' is a *label refinement* of P if there is a label-refinement relation $\sim$ for P and P' such that for each store $\sigma \in$ `difc`Stores, $(\mathsf{ENV}, \sigma) \sim (\mathsf{ENV}, \sigma)$.

Label refinement, like capability refinement (§5.2), may be viewed as a special case of alternating refinement [2].

9.3 DIFC policies

In §8.2, we presented a policy for `auth_log` as an automaton whose symbols were conditions on DIFC states and an automaton whose symbols were conditions on pairs of DIFC states. In this section, we define the structure and semantics of DIFC automata in general. In §9.3.1, we define a space of conditions on `difc` stores. In §9.3.2, we use store conditions to define a space of DIFC policy automata, and define under what conditions a `difc` program satisfies a DIFC policy automaton. We describe how each non-interference automaton can be compiled into a DIFC policy automaton over an extended store vocabulary in App. A.1.

9.3.1 Conditions on `difc` stores

Store conditions are used in DIFC policies to define assumed and required conditions on `difc` states. Store conditions are formulas over a first-order relational vocabulary $V_{\mathtt{d}}$ whose predicates model properties of `difc` stores. $V_{\mathtt{d}}$ is the union of two vocabularies $V_{\mathtt{dc}}$ and $V'_{\mathtt{d}}$; $V_{\mathtt{dc}}$ describes features of `difccore` states.

Definition 19. Each `difccore` store $\sigma \in \mathsf{DIFCCoreStores}$ defines a model $m^{\mathtt{dc}}_{\sigma} = \langle U_{\sigma}, \iota_{\sigma} \rangle$ over a first-order relational vocabulary $V_{\mathtt{dc}}$. The universe U_{σ} of m_{σ} contains the objects of σ. The vocabulary $V_{\mathtt{dc}}$ and the interpretation ι_{σ} of each predicate symbol in $V_{\mathtt{dc}}$ in the universe U_{σ} are defined as follows.

- $V_{\mathtt{dc}}$ contains a unary predicate symbol IsRoot. IsRoot holds for exactly one individual; in particular, $\iota_{\sigma}(\mathsf{IsRoot})(\mathsf{Root})$ holds.
- $V_{\mathtt{dc}}$ contains the set of module names $\mathsf{MODSYMS}$ as unary predicate symbols. At most one module-name predicate can hold for a given object. If in σ the program executes module $\mathtt{M} \in \mathsf{MODSYMS}$, then $\iota_{\sigma}(\mathtt{M})(\mathsf{Mem})$ holds. If in σ a gate g is bound to $\mathtt{M}$, then $\iota_{\sigma}(\mathtt{M})(g)$ holds.
- $V_{\mathtt{dc}}$ contains the set of object variables $\mathsf{OBJVARS}$ as unary predicate symbols. Each object-variable predicate can hold for at most one object. If in σ there is an object o in object variable $\mathtt{o}$, then $\iota_{\sigma}(\mathtt{o})(o)$ holds.
- $V_{\mathtt{dc}}$ contains the set of links $\mathtt{LINKS}$ as binary predicate symbols. Each link-symbol predicate is a partial function over objects. If in σ there are objects o and p such that there is a link from o to p on link symbol $\mathtt{l} \in \mathtt{LINKS}$, then $\iota_{\sigma}(l)(o, p)$ holds.

Definition 20. Each `difc` store σ defines a model $m^{\mathtt{d}}_{\sigma} = \langle U_{\sigma}, \iota_{\sigma} \rangle$ over a first-order relational vocabulary $V'_{\mathtt{d}}$. The universe U_{σ} contains the objects and categories of σ and an individual i_o that models the operation label and operation declassification. ι_σ of each predicate symbol in V'_d in the universe U_σ is defined as follows.

- V_d contains a unary predicate symbol IsObj. For each object o, $\mathsf{IsObj}(o)$ holds.
- V_d contains a unary predicate symbol IsMem. $\iota_\sigma(\mathsf{IsMem})(\mathsf{Mem})$ holds.
- V_d contains a unary predicate symbol IsCat. If in σ, c is a category ($c \in C^\sigma$), then $\iota_\sigma(\mathsf{IsCat})(c)$ holds.
- V_d contains the set of category variables CVAR. If in σ there is a category c bound to c, then $\iota_\sigma(\mathsf{c})(c)$ holds.
- V_d contains a unary predicate symbol IsOp. $\mathsf{IsOp}(i_o)$ holds.
- For each level $lv \in \mathsf{LEVELS}$, V_d contains a binary predicate symbol $\mathsf{label}[lv]$. If in σ the label of object o has level lv at category c, then $\iota_\sigma(\mathsf{label}[lv])(o, c)$ holds.
- V_d contains a binary predicate symbol $\mathsf{Declassifies}$. If in σ the declassification of object o contains category c, then $\iota_\sigma(\mathsf{Declassifies})(o, c)$ holds.

For `difc` store σ, the model of σ over the vocabulary V_d is the union of the models (defined in §2.3, Defn. 6) $m_\sigma = m_\sigma^{dc} \cup m_\sigma^{d}$.

A *store condition* is a closed first-order V_d-formula; we denote the space of all store conditions as $\mathsf{StoreCond}$. A store σ satisfies store condition φ if m_σ is a model of φ.

The store conditions used to define `log_ar` (§8.2) are defined over a set of derived `difc` predicates. The derived ternary predicate $\mathsf{LeqLv}(x, y, c)$

holds if the label of object x is less than or equal to the label of object y at category c:

$$\begin{aligned}\mathsf{LeqLv}(x, y, c) \equiv\ & \mathit{label}[\mathsf{Low}](x, c) \\ & \vee(\mathit{label}[\mathsf{Mid}](x, c) \\ & \quad \wedge(\mathit{label}[\mathsf{Mid}](y, c) \vee \mathit{label}[\mathsf{High}](y, c)) \\ & \vee(\mathit{label}[\mathsf{High}](x, c) \wedge \mathit{label}[\mathsf{High}](y, c))\end{aligned}$$

The derived unary predicate $\mathsf{CanRead}(x)$ holds if memory can read from object x:

$$\begin{aligned}\mathsf{CanRead}(x) \equiv \forall m, c.\ \mathsf{IsMem}(m) \wedge \mathsf{IsCat}(c) \implies & \\ \mathsf{LeqLv}(x, m, c) \vee \mathsf{Declassifies}(m, c) &\end{aligned}$$

The derived unary predicate $\mathsf{CanWrite}(x)$ holds if memory can write to object x:

$$\begin{aligned}\mathsf{CanWrite}(x) \equiv \forall m, c.\ \mathsf{IsMem}(m) \wedge \mathsf{IsCat}(c) \implies & \\ \mathsf{LeqLv}(m, x, c) \vee \mathsf{Declassifies}(m, c) &\end{aligned}$$

Example 12. In `log_ar`, assumptions on the access rights held when each module of `auth_log` is entered and assertions on the access rights held when each module exits can be represented as store conditions; these store conditions were depicted for clarity in Fig. 8.3 as a set of derived V_d nullary predicates. The derived nullary store predicate `RD[MSG]` denotes the store condition:

$$\forall r, l.\ \mathsf{IsRoot}(r) \wedge \mathtt{MSG}(r, l) \implies \mathsf{CanRead}(l)$$

The derived nullary store predicate `RD[LOG]` denotes the store condition:

$$\forall r, l.\ \mathsf{IsRoot}(r) \wedge \mathtt{LOG}(r, l) \implies \mathsf{CanRead}(l)$$

The derived nullary store predicate $\texttt{WR[LOG]}$ denotes the store condition:

$$\forall r, l.\ \mathsf{IsRoot}(r) \land \texttt{LOG}(r, l) \implies \mathsf{CanWrite}(l)$$

9.3.2 Policy automata

A DIFC policy is a finite-state automaton in which each alphabet symbol is a control location paired with a store condition.

Definition 21. A *DIFC policy* is a finite-state automaton whose alphabet Σ is a finite set in which each element is a control location paired with a store condition. I.e., $\Sigma_d = \texttt{LOC}_E \times \mathsf{StoreCond}$. The class of DIFC policies is denoted $\mathsf{DIFCPols}$.

Each DIFC policy defines a language of traces of `difc` states to be policy violations. A trace t is in the language if each state in t satisfies a corresponding store condition in some trace of conditions accepted by A.

Definition 22. Let $t = (L_0, \sigma_0), \ldots, (L_n, \sigma_n) \in Q_d^*$ be a trace of `difc` states, and let $D \in \mathsf{DIFCPols}$ be a DIFC policy. If the trace of state conditions $t_A = a_0, \ldots, a_n \in \Sigma_d^*$ is such that for each $0 \leqslant i \leqslant n$ and $a_i = (L_i', \varphi_i)$, (1) $L_i = L_i'$ and (2) $\sigma_i \models \varphi_i$, then t_A is a *store-condition trace* of t. t *violates* A if A accepts some store-condition trace of t. For `difc` program P, if each trace t of P does not violate A, then P *satisfies* A (denoted $P \models A$).

9.4 The DIFC labeling problem

The DIFC labeling problem is to take a program P and a DIFC policy Π, and instrument P to satisfy Π.

Definition 23. Let P be a `difc` program and let Π be a DIFC policy. A solution to the DIFC instrumentation problem $\mathsf{LABEL}(P, \Pi)$ is a `difc` program P' such that (1) P' is a label refinement of P and (2) P' satisfies Π.

9.5 DIFC labeling as game-solving

In this section, we describe a sound, but incomplete, procedure `hiweave` for solving the DIFC labeling problem.

9.5.1 Overview

In principle, a solution to a labeling problem $\mathsf{LABEL}(\mathsf{P}, \Pi)$ can be any instrumentation of P which, at particular control locations, checks predicates of its current state and chooses an appropriate label operation to execute next in order to satisfy Π. The problem of synthesizing a set of predicates to be checked and acted on at runtime raises daunting challenges. In particular, a `difc` state is defined by an unbounded set of objects and categories, and thus checking many properties of a `difc` state at runtime could be expensive, and in the worst case impossible if components of the state are created by the environment to be unreadable when instrumented modules of the program execute.

Instead of attempting to synthesize a `difc` program that chooses label operations based on properties of its *execution state*, `hiweave` attempts to synthesize a program that chooses label operations based on properties of the *history of executed operations*. We reduce the problem of searching for a valid instrumentation to finding a winning Defender strategy to a game, where the symbols of the game model `difc` operations. This approach has several advantages:

- `hiweave` can be parameterized on advice "templates" from an expert user, which specify restricted languages of operations for an instrumented program to potentially execute.
- `hiweave` can apply any analysis that builds a sound abstraction of the transition relation of a `difc` program, independent of the representation of states in the abstraction.

Input : A `difc` program P and DIFC policy Π.
Output: A solution to $\mathsf{LABEL}(\mathsf{P}, \Pi)$.
1 $\mathsf{G}_{\mathsf{P},\Pi} \coloneqq$ `DIFCProgPolicyGame`(P, Π) ;
2 **if** `HasWinningDefStrategy`$(\mathsf{G}_{\mathsf{P},\Pi})$ **then**
3 $\quad \mathsf{D} \coloneqq$ `FindWinningDefStrategy`$(\mathsf{G}_{\mathsf{P},\Pi})$;
4 $\quad$ **return** `DIFCCodeGen`(D) ;
5 **else**
6 $\quad$ `Fail` () ;

Algorithm 9.6: `hiweave`: a sound solver for the DIFC labeling problem.

- `hiweave` can use standard automata-theoretic language operations to model the instrumented program's inability to directly observe the actions of the environment.

`hiweave` attempts to solve a labeling problem $\mathsf{LABEL}(\mathsf{P}, \mathsf{F})$ in three main steps, presented in pseudo-code as Alg. 9.6. `hiweave` first constructs (line [1]) from P and Π a finite two-player game $\mathsf{G}_{\mathsf{P},\Pi}$ whose alphabet is the space of operations of P, such that for any Defender strategy D that wins $\mathsf{G}_{\mathsf{P},\Pi}$, the plays of D are the traces of a solution to $\mathsf{LABEL}(\mathsf{P}, \Pi)$ (in such a case, we say that D *defines* a solution to $\mathsf{LABEL}(\mathsf{P}, \Pi)$). This step is described in more detail in §9.5.2.

`hiweave` then applies a classical algorithm `HasWinningDefStrategy` to determine if $\mathsf{G}_{\mathsf{P},\Pi}$ has a winning Defender strategy (line [2]). If $\mathsf{G}_{\mathsf{P},\Pi}$ has a winning Defender strategy, then `hiweave` applies a classical algorithm `FindWinningDefStrategy` to construct a winning Defender strategy D (line [3]). Otherwise, `hiweave` aborts (line [6]); we discuss this limitation, and possible extensions of our work to overcome it, in Chapter 14.

If `hiweave` finds a winning Defender strategy D for $\mathsf{G}_{\mathsf{P},\Pi}$, then `hiweave` instruments P to form a new `difc` program P' that is a solution to $\mathsf{LABEL}(\mathsf{P}, \Pi)$ (line [4]). During each run, P' stores two tables that represent the transition function of D. One table, $\mathtt{T_A}$, represents the transition function of D from Attacker states. $\mathtt{T_A}$ is indexed by an Attacker pre-state and a `difc` operation, and maps each index-pair to a post-state. As P' executes, it stores the current state of D in a variable `cur`. When P' executes a `difccore` operation `o`, it updates `cur` to store the value in $\mathtt{T_A}$ indexed by the current value in `cur` and `o`.

The second table, $\mathtt{T_D}$, represents the transition function of D from Defender states. $\mathtt{T_D}$ is indexed by a Defender pre-state, and maps each index to a label operation-state pair $(\mathtt{o}, \mathsf{q}')$. As long as the state stored by `cur` is a Defender state, P' performs the label operation `o` and updates `cur` to store q'. When `cur` stores an Attacker state, P' executes the next operation of P.

In the remainder of this section, we describe in detail how `hiweave` takes an input `difc` program P and DIFC policy Π, and constructs a finite game $\mathsf{G}_{\mathsf{P},\Pi}$ that it solves in order to solve $\mathsf{LABEL}(\mathsf{P}, \mathsf{Q})$.

9.5.2 From a program and DIFC policy to a game

From an input program P and input DIFC policy Π, `hiweave` constructs a finite two-player game $\mathsf{G}_{\mathsf{P},\Pi}$ such that each winning Defender strategy of $\mathsf{G}_{\mathsf{P},\Pi}$ defines an instrumentation of P that satisfies Π. To construct $\mathsf{G}_{\mathsf{P},\Pi}$, `hiweave` performs the following steps.

1. Let $\mathsf{T} = (\mathsf{Q}_\mathsf{T}, \iota_\mathsf{T}, \mathsf{F}_\mathsf{T}, \mathtt{Op}, \Delta_\mathsf{T})$ be a finite acceptor of traces of `difc` operations that serves as a *template* of potential traces of label operations that an instrumented version of P may execute before each operation of P. From T, `hiweave` constructs a finite two-player game G_T such that each play of T is a sequence of `difc` operations accepted by T chosen by the Defender, followed by a `difc` operation chosen by the Attacker. We describe our experience designing operation templates in §9.5.3.

2. From P and T, `hiweave` constructs a structure program (defined in §2.3) $\mathsf{S}_{\mathsf{P},\mathsf{T}}$ such that each execution of $\mathsf{S}_{\mathsf{P},\mathsf{T}}$ models an execution E of

P with a sequence of operations accepted by T injected before each operation of E.

3. From $S_{P,T}$, `hiweave` constructs a finite-state acceptor $A^{\#}_{P,T}$ of traces of `difc` operations such that each trace that drives $S_{P,T}$ to an error location is accepted by $A^{\#}_{P,T}$.

4. From Π, `hiweave` constructs a structure program S_{Π} such that each `difc` trace of a run that does not satisfy Π drives S_{Π} to an error control location.

5. From S_{Π}, `hiweave` constructs a finite-state acceptor $A^{\#}_{\Pi}$ of traces of `difc` operations such that each trace that drives S_{Π} to an error control location is accepted by $A^{\#}_{\Pi}$.

6. `hiweave` constructs $G_{P,\Pi}$ as the product of G_T, $A^{\#}_P$, and $A^{\#}_{\Pi}$.

We now describe each step of the construction of $G_{P,\Pi}$ in more detail.

Constructing a game of template instrumentations

From template T, `hiweave` constructs a two-player safety game G_T in which the Defender is restricted to play only sequences of operations accepted by T. In particular, each play of G_T not won by the Attacker is an unbounded sequence of phases, in which each phase consists of (1) a sequence of label operations chosen by the Defender that are accepted by T, followed by (2) any `difc` operation, chosen by the Attacker. The construction of G_T is straightforward from its informal description, and we omit a full definition.

From a `difc` program to a structure program

From the input program P and template T, `hiweave` constructs a structure program $S_{P,T} = (\mathsf{LOC}_S, \iota_S, \mathsf{O}_S, \mathsf{E}_S, \mathsf{V}_S, \mathsf{T}_S)$ such that each trace of $S_{P,T}$ is a trace $\mathtt{t}$ of P with a sequence of operations accepted by T injected before each operation in $\mathtt{t}$. `hiweave` constructs $\mathsf{S}_{\mathsf{P},\mathsf{T}}$ from the following components:

Control locations of $\mathsf{S}_{\mathsf{P},\mathsf{T}}$ The control locations LOC_S of $\mathsf{S}_{\mathsf{P},\mathsf{T}}$ contain "copies" of the states of T for each control location of P, and a control location at which $\mathsf{S}_{\mathsf{P},\mathsf{T}}$ models the environment of P. I.e., LOC_S contains the following:

- For each control location $\mathtt{L} \in \mathsf{LOC}$ and each state $\mathsf{q} \in \mathsf{Q}_\mathsf{T}$, a control location $(\mathtt{L}, \mathsf{q})$. We refer to $(\mathtt{L}, \mathsf{q})$ as the *copy of* q *for* $\mathtt{L}$.
- The control location ENV.

Initial control location of $\mathsf{S}_{\mathsf{P},\mathsf{T}}$ The initial control location of $\mathsf{S}_{\mathsf{P},\mathsf{T}}$ is the control location ENV.

Operations of $\mathsf{S}_{\mathsf{P},\mathsf{T}}$ The operations of $\mathsf{S}_{\mathsf{P},\mathsf{T}}$ are the `difc` operations $\mathtt{Op}$.

Control edges of $\mathsf{S}_{\mathsf{P},\mathsf{T}}$ The control edges E_S of $\mathsf{S}_{\mathsf{P},\mathsf{T}}$ induce $\mathsf{S}_{\mathsf{P},\mathsf{T}}$ to execute a trace of operations accepted by T and then execute the next operation to be executed by P. In particular, E_S contains the following edges:

- For each transition $(\mathsf{q}, \mathtt{o}, \mathsf{q}') \in \Delta_\mathsf{T}$ of T, $\mathsf{S}_{\mathsf{P},\mathsf{T}}$ may execute $\mathtt{o}$ from each copy of q. I.e., for each control location $\mathtt{L} \in \mathsf{LOC}$ and each transition $(\mathsf{q}, \mathtt{o}, \mathsf{q}') \in \Delta_\mathsf{T}$, E_S contains a control edge $((\mathtt{L}, \mathsf{q}), \mathtt{o}, (\mathtt{L}, \mathsf{q}'))$.
- If P transitions from control location $\mathtt{L}$ to control location $\mathtt{L}'$ on an intra-module operation $\mathtt{o}$, then $\mathsf{S}_{\mathsf{P},\mathsf{T}}$ transitions on operation $\mathtt{o}$ from each copy of a final state of T for $\mathtt{L}$ to the copy of the initial state of T for $\mathtt{L}'$. I.e., for each intra-module control edge $(\mathtt{L}, \mathtt{o}, \mathtt{L}') \in \mathsf{E}_\mathsf{P}$ and each final state $\mathsf{q} \in \mathsf{F}_\mathsf{T}$, E_S contains a control edge $((\mathtt{L}, \mathsf{q}), \mathtt{o}, (\mathtt{L}', \iota_\mathsf{T}))$.

- If P contains a control location $\mathtt{L}$ with a gatecall operation, and $\mathtt{L}_0$ is the initial control location of a module or ENV, then $\mathsf{E_S}$ contains a control edge from the copy of each final state of T for $\mathtt{L}$ to the copy of the initial state of T for $\mathtt{L}_0$. I.e., for operation $\mathtt{gatecall}(g)$ at location $\mathtt{L} \in \mathsf{LOC}$, $\mathtt{L}_0 \in \mathsf{LOC}$ an initial location of a module in P, and $q \in \mathsf{F_T}$ a final state of T, $\mathsf{E_S}$ contains a control edge $((\mathtt{L}, q), \mathtt{gatecall}(g), (\mathtt{L}_0, \iota_F))$.

- If P models the program environment, then P can execute any intra-module operation and continue to model the environment. I.e., for each intra-module operation $\mathtt{o}$, $\mathsf{E_S}$ contains a control edge $(\mathsf{ENV}, \mathtt{o}, \mathsf{ENV})$.

Vocabulary of $\mathsf{S_{P,T}}$ The vocabulary of $\mathsf{T_{P,F}}$ is the DIFC vocabulary $\mathsf{V_d}$, defined in §9.3.1, Defn. 20.

Predicate transformers of $\mathsf{S_{P,T}}$ For each `difc` operation $\mathtt{o}$, `hiweave` defines a predicate transformer over the vocabulary $\mathsf{V_d}$ that models the semantics of $\mathtt{o}$. We now describe how each condition over labels in the premise of an operation $\mathtt{o}$ and each update of a label store in Figs. 9.4 and 9.5 is modeled as a predicate transformer $\tau[\mathtt{o}]$ over $\mathsf{V_d}$ structures. We present the predicate transformers of `difc` as the union of predicate transformers that model the semantics of `difccore`, denoted $\mathsf{T_{dc}}$, and predicate transformers that model the semantics of `difc`, denoted $\mathsf{T_d}$. The transformers $\mathsf{T_{dc}}$ that update the values of $\mathsf{V_{dc}}$ predicates are defined as follows:

- A `difccore` operation $\mathtt{o} := \mathtt{ld\_root}()$ loads the root object into the object variable $\mathtt{o}$. The transformer for the operation $\mathtt{o} := \mathtt{ld\_root}()$ updates $\mathsf{V_{dc}}$ predicates as follows:

$$\mathtt{o}(o) := \mathsf{IsRoot}(o)$$

- A $\texttt{difccore}$ operation $\texttt{o} \coloneqq \texttt{ld}(\texttt{d}, \texttt{L})$ loads into object-variable $\texttt{o}$ the object linked from the object stored in object-variable $\texttt{d}$ at link symbol $\texttt{L}$. The transformer for $\texttt{o} \coloneqq \texttt{ld}(\texttt{d}, \texttt{L})$ updates $V_{\texttt{dc}}$ predicates as follows:

$$\texttt{o}(o) \coloneqq \exists d.\ \texttt{d}(d) \land \texttt{L}(d, o)$$

- A $\texttt{difccore}$ operation $\texttt{o} \coloneqq \texttt{create}(\texttt{d}, \texttt{l})$ (1) binds a fresh object o to object variable $\texttt{o}$ and (2) links the object bound to $\texttt{d}$ to o on link symbol $\texttt{L}$. The transformer for $\texttt{o} \coloneqq \texttt{create}(\texttt{d}, \texttt{l})$ updates $V_{\texttt{dc}}$ predicates according to the follow predicate updates:

$$\begin{aligned} \texttt{o}(o) \coloneqq &\quad \mathsf{new}(o) && (1) \\ \texttt{L}(d, o) \coloneqq &\quad \mathsf{ITE}(\texttt{d}(d), \mathsf{new}(o), \texttt{L}(d, o)) && (2) \end{aligned}$$

- A $\texttt{difccore}$ operation $\texttt{g} \coloneqq \texttt{creategate}(\texttt{d}, \texttt{L}, \texttt{M})$, updates pre-store σ analogously to how $\texttt{o} \coloneqq \texttt{create}(\texttt{d}, \texttt{L})$ updates its pre-store, and in addition, sets the module of the freshly-allocated object g to $\texttt{M}$. The transformer for $\texttt{g} \coloneqq \texttt{creategate}(\texttt{d}, \texttt{L}, \texttt{M})$ updates $\texttt{difccore}$ predicates according to the predicate updates for operation $\texttt{g} \coloneqq \texttt{create}(\texttt{d}, \texttt{L})$ and the predicate update:

$$\texttt{M}(g) \coloneqq \texttt{M}(g) \lor \mathsf{new}(g)$$

The transformers $T_{\texttt{d}}$ that update the values of $V'_{\texttt{d}}$ predicates are defined as follows:

- A $\texttt{difc}$ operation $\texttt{x} \coloneqq \texttt{ld}(\texttt{d}, \texttt{L})$ checks that in pre-store σ, the object stored in object variable $\texttt{d}$ flows to memory. For the derived unary formula $\mathsf{CanRead}$ defined in §9.3.1, the predicate transformer $\tau[\texttt{x} \coloneqq \texttt{ld}(\texttt{o})]$ asserts that its pre-structure satisfies the following $V_{\texttt{d}}$ formula:

$$\forall o.\ \texttt{d}(o) \implies \mathsf{CanRead}(o)$$

- A $\texttt{difc}$ operation $\texttt{x} \coloneqq \texttt{RD}(\texttt{o})$ checks that in pre-store σ, the object o bound to object-variable $\texttt{o}$ flows to memory over categories not declassified by memory. The predicate transformer $\tau[\texttt{x} \coloneqq \texttt{RD}(\texttt{o})]$ asserts that its pre-structure satisfies the following V_d formula:

$$\forall o.\ \texttt{o}(o) \implies \mathsf{CanRead}(o)$$

- A $\texttt{difc}$ operation $\texttt{WR}(\texttt{o}, \texttt{x})$ checks that in pre-store σ, memory flows to the object o bound to object-variable $\texttt{o}$. For the derived unary formula $\mathsf{CanWrite}$ defined in §9.3.1, the predicate transformer $\tau[\texttt{WR}(\texttt{o}, \texttt{x})]$ asserts that its pre-structure satisfies the following V_d formula:

$$\forall o.\texttt{o}(o) \implies \mathsf{CanWrite}(o)$$

  If σ passes the check of $\texttt{WR}(\texttt{o}, \texttt{x})$, then $\texttt{WR}(\texttt{o}, \texttt{x})$ binds to o the data value bound to data variable $\texttt{x}$. $\tau[\texttt{WR}(\texttt{o}, \texttt{x})]$ does not change its pre-structure.

- A $\texttt{difc}$ operation $\texttt{o} \coloneqq \texttt{create}(\texttt{d}, \texttt{L})$ checks that in pre-store σ, (1) memory flows to the object bound to $\texttt{d}$ over categories C not declassified by memory and (2) memory flows to the operation label over C. The predicate transformer $\tau[\texttt{o} \coloneqq \texttt{create}(\texttt{d}, \texttt{L})]$ asserts that its pre-structure satisfies the following V_d formula:

$$\begin{aligned} \forall m, d, o, c.\ \mathsf{IsMem}(m) \land \texttt{d}(d) \land \mathsf{IsOp}(o) \land \mathsf{IsCat}(c) \implies& \\ (\mathsf{Declassifies}(m, c)& \\ \lor ((1)\mathsf{LeqLv}(m, d, c) \land (2)\mathsf{LeqLv}(m, o, c)))& \end{aligned}$$

  If σ passes the check of $\texttt{o} \coloneqq \texttt{create}(\texttt{d}, \texttt{l})$, then $\texttt{o} \coloneqq \texttt{create}(\texttt{d}, \texttt{l})$ (1) allocates a fresh object o and (2) sets the label of o to be the operation label. If a pre-structure S satisfies the assertion of $\tau[\texttt{o} \coloneqq \texttt{create}(\texttt{d}, \texttt{L})]$,

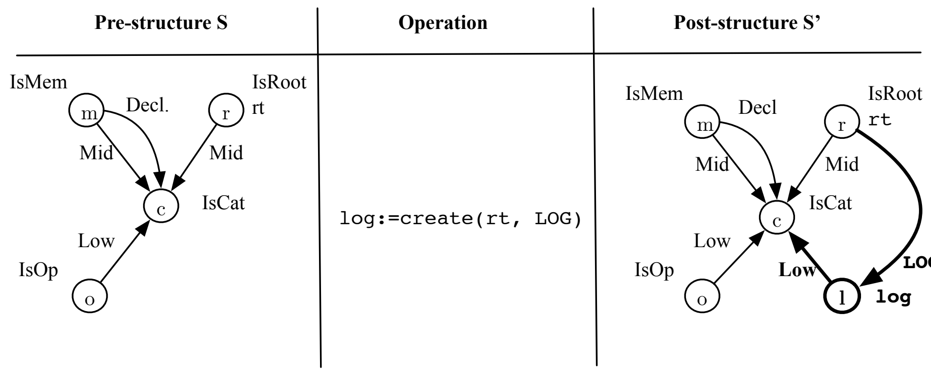

Figure 9.7: A graphical depiction of the predicate transformer that models the `create` operation executed by `log_init` (see Chapter 8). The pre-structure S is depicted on the left, and the resulting post-structure S' is depicted on the right. Each structure is depicted as a graph in which each node depicts an individual, and each edge depicts a binary relation between nodes. Each node n depicting an individual i_n is annotated with a name inside n, and unary predicate symbols to the side of n that hold for i_n, and each edge from node m to node n is annotated with the binary relation that holds for (m, n). In the S', nodes and edges depicting individuals and relations created by $\texttt{log} \coloneqq \texttt{create}(\texttt{d}, \texttt{LOG})$ are highlighted in bold.

then $\tau[\texttt{o} \coloneqq \texttt{create}(\texttt{d}, \texttt{L})]$ updates the universe and predicates of S by introducing a new individual and applying the following predicate updates:

$$
\begin{aligned}
&\text{for } l\nu \in \mathsf{LEVELS}:\\
&\mathsf{label}[l\nu]'(o, c) \coloneqq \ \mathsf{label}[l\nu](o, c)\\
&\qquad\qquad\qquad\quad \vee(\mathsf{new}(o) \wedge \exists p.\mathsf{IsOp}(p) \wedge \mathsf{label}[l\nu](p, c)) \quad (4)
\end{aligned}
$$

Fig. 9.7 depicts the predicate transformer for the operation $\texttt{o} \equiv \texttt{log} \coloneqq \texttt{create}(\texttt{rt}, \texttt{LOG})$ contained in the `difc` module `log_init` introduced in Chapter 8, applied to a pre-structure that models a store σ of `log_init` when `log_init` executes the operation $\texttt{o} \equiv$

$\texttt{log} \coloneqq \texttt{create}(\texttt{rt}, \texttt{LOG})$. The pre-structure contains four individuals that model (1) program memory (annotated "m"), (2) the root object (annotated "r"), (3) the operation label (annotated "o"), and (4) a category created by `auth_log` (annotated "c"). For individual $\mathtt{m}$, the unary predicate IsMem holds, for individual $\mathtt{r}$ the unary predicate IsRoot holds, and for individual $\mathtt{o}$, the unary predicate IsOp holds, which model the facts that $\mathtt{m}$, $\mathtt{r}$, and $\mathtt{o}$ model memory, root, and the operation label, respectively. Individual $\mathtt{c}$ is in the unary predicate IsCat, which models that fact that $\mathtt{c}$ is a category. The edges from $\mathtt{m}$ to $\mathtt{c}$ and from $\mathtt{r}$ to $\mathtt{c}$ annotated $\mathsf{label}[\mathsf{Mid}]$ model the fact that the labels of the memory and root objects have level Mid at $\mathtt{c}$. The edge from $\mathtt{o}$ to $\mathtt{c}$ annotated $\mathsf{label}[\mathsf{Low}]$ models the fact that the operation label has level Low at $\mathtt{c}$. The edge from $\mathtt{m}$ to $\mathtt{c}$ annotated $\mathsf{Declassifies}$ models the fact that in σ, memory declassifies $\mathtt{c}$.

The post-structure S' in Fig. 9.7, obtained by applying the predicate transformer $\tau[\texttt{log} \coloneqq \texttt{create}(\texttt{rt}, \texttt{LOG})]$ to the pre-structure S, is S extended with an additional individual (annotated "l") that models the log file created by executing $\texttt{log} \coloneqq \texttt{create}(\texttt{rt}, \texttt{LOG})$. Individual $\mathtt{l}$ is annotated with a unary predicate `log`, which models the fact that in S', the log object is stored in object variable `log`. S' contains (1) an edge from $\mathtt{l}$ to $\mathtt{c}$ which models the fact that in σ', the label of the log object has level Low at $\mathtt{c}$ and (2) an edge annotated `LOG` from $\mathtt{r}$ to $\mathtt{l}$ which models the fact that in σ', there is a link with symbol `LOG` from the root to the log.

- A `difc` operation $\texttt{g} \coloneqq \texttt{creategate}(\texttt{d}, \texttt{L}, \texttt{M})$ checks that in pre-store σ, (1) over all categories not declassified by memory, (2) memory flows to the operation label; and (3) the operation declassification is contained by the memory declassification. The predicate transformer $\tau[\texttt{g} \coloneqq \texttt{creategate}(\texttt{d}, \texttt{l}, \texttt{M})]$ asserts that a pre-structure S satisfies the following formula:

$$\begin{aligned}\forall m, d, o, c.\quad & \mathsf{IsMem}(m) \land d(d) \land \mathsf{IsOp}(o) \land \mathsf{IsCat}(c) \implies \\ & \land((1)\mathsf{Declassifies}(m, c) \\ & \quad \lor((2)\mathsf{LeqLv}(m, d, c) \land (3)\neg\mathsf{Declassifies}(o, c))\end{aligned}$$

If σ satisfies the check of $\mathtt{g} \coloneqq \mathtt{creategate}(\mathtt{d}, \mathtt{L}, \mathtt{M})$, then $\mathtt{g} \coloneqq \mathtt{creategate}(\mathtt{d}, \mathtt{L}, \mathtt{M})$ updates σ analogously to how $\mathtt{o} \coloneqq \mathtt{create}(\mathtt{d}, \mathtt{L})$ updates its pre-store, and in addition sets the declassification of g to the operation declassification. If pre-structure S satisfies the assertion of $\tau[\mathtt{creategate}(\mathtt{d}, \mathtt{l}, \mathtt{M})]$, then $\tau[\mathtt{creategate}(\mathtt{d}, \mathtt{L}, \mathtt{M})]$ adds a new individual to the universe of S, and updates the predicates of S according to the following predicate updates:

$$\begin{aligned}& \text{for } lv \in \mathsf{LEVELS}: \\ & \quad \mathsf{label}[lv]'(x, c) \coloneqq \; \mathsf{label}[lv](x, c) \\ & \qquad\qquad\qquad\qquad \lor(\mathsf{new}(x) \land \exists o.\mathsf{IsOp}(o) \land \mathsf{label}[lv](o, c)) \\ & \mathsf{Declassifies}'(x, c) \coloneqq \; \mathsf{Declassifies}(x, c) \\ & \qquad\qquad\qquad\qquad \lor \quad (\mathsf{new}(x) \\ & \qquad\qquad\qquad\qquad\qquad \land \exists o.\ \mathsf{IsOp}(o) \land \mathsf{Declassifies}(o, c))\end{aligned}$$

- A `difc` operation $\mathtt{gatecall}(\mathtt{g})$ checks that in pre-store σ, over all categories not in the set of D of categories declassified by memory or the gate object g bound to g, (1) the operation declassification contains the memory declassification and over all categories not declassified by the operation declassification or the gate declassification, (2) memory flows to the operation label, and (3) g flows to the operation label. The predicate transformer $\tau[\mathtt{gatecall}(\mathtt{g})]$ asserts that pre-structure S satisfies the following condition:

$$\begin{aligned}\forall m, g, o, c.\quad & \mathsf{IsMem}(m) \wedge \mathsf{g}(g) \wedge \mathsf{IsOp}(o) \wedge \mathsf{IsCat}(c) \implies \\ & ((1)\mathsf{Declassifies}(o, c) \implies \mathsf{Declassifies}(m, c)) \\ & \wedge(\mathsf{Declassifies}(o, c) \vee \mathsf{Declassifies}(g, c) \\ & \quad \vee((2)\mathsf{LeqLv}(m, o, c) \wedge (3)\mathsf{LeqLv}(g, o, c)\end{aligned}$$

If pre-store σ satisfies the check of `gatecall(g)`, then `gatecall(g)` updates σ so that (1) the module of memory is the module of the gate object g bound to `g` in σ, (2) the memory label is the operation label, and (3) the memory declassification is the operation declassification. If a pre-structure S satisfies the assertion of τ[`gatecall(g)`], then τ[`gatecall(g)`] updates S according to the following predicate updates.

$$\begin{aligned}&\text{for } M \in \mathsf{MODSYMS}: \\ &\qquad M'(x) \coloneqq\ \mathsf{ITE}(\mathsf{IsMem}(x), \exists g.\ \mathsf{g}(g) \wedge M(g), M(x)) && (1) \\ &\text{for } lv \in \mathsf{LEVELS}: \\ &\qquad \mathsf{label}[lv]'(x, c) \coloneqq\ \mathsf{ITE}(\mathsf{IsMem}(x), \\ &\qquad\qquad \exists o.\ \mathsf{IsOp}(o) \wedge \mathsf{label}[lv](c), \mathsf{label}[lv](x, c)) && (2) \\ &\mathsf{Declassifies}'(x, c) \coloneqq\ \mathsf{ITE}(\mathsf{IsMem}(x), \\ &\qquad\qquad \exists o.\ \mathsf{IsOp}(o) \wedge \mathsf{Declassifies}(o, c), \\ &\qquad\qquad \mathsf{Declassifies}(x, c)) && (3)\end{aligned}$$

- A `difc` operation `c` $\coloneqq$ `create_cat()` updates pre-store σ by (1) allocating a fresh category c, (2) binding c to category variable `c`, and (3) extending the declassification of memory to contain c. (4) The label of each object has level Mid at c. The predicate transformer τ[`c` $\coloneqq$ `create_cat()`] adds a new individual to the universe of pre-structure S and updates the predicates of S according to the following predicate updates:

$$\begin{aligned}
\mathsf{IsCat}(c) &:= \mathsf{IsCat}(c) \vee \mathsf{new}(c) && (1)\\
\mathsf{c}'(c) &:= \mathsf{new}(c) && (2)\\
\mathsf{Declassifies}'(o,c) &:= \mathsf{Declassifies}(o,c) \vee (\mathsf{IsMem}(o) \wedge \mathsf{new}(c)) && (3)\\
\text{for } lv \in \mathsf{LEVELS}&:\\
\mathsf{label}[\mathsf{Mid}]'(o,c) &:= \mathsf{label}[\mathsf{Mid}](o,c) \vee (\mathsf{IsObj}(o) \wedge \mathsf{new}(c)) && (4)
\end{aligned}$$

- A `difc` operation `set_op_label`(E) with label expression E updates pre-store σ to hold an operation label with the value of E in σ. The predicate transformer $\tau[$`set_op_label`$(E)]$ updates the predicates of its pre-structure according to the following predicate updates:

  $$\begin{aligned}
  &\text{for } lv \in \mathsf{LEVELS}:\\
  &\mathsf{label}[lv]'(o,c) := \mathsf{ITE}(\mathsf{IsOp}(o), E[lv](c), \mathsf{label}[lv](o,c))
  \end{aligned}$$

  $E[lv]$ is a unary derived predicate defined below.

- A `difc` operation `set_op_declass`(E) with declassification expression E updates pre-store σ by setting the operation declassification to the value of E in σ. The predicate transformer $\tau[$`set_op_declass`$(E)]$ updates the predicates of its pre-structure according to the following predicate updates:

  $$\begin{aligned}
  &\text{for } lv \in \mathsf{LEVELS}:\\
  &\mathsf{Declassifies}'(o,c) := \mathsf{ITE}(\mathsf{IsOp}(o), E(c), \mathsf{Declassifies}(o,c))
  \end{aligned}$$

E is a unary derived predicate defined below.

For a label expression E and a level predicate lv, the derived V_d unary predicate $E[lv](c)$ used in the predicate updates for operation `set_op_label` is defined as follows:

- If E is the label expression `mem_label`, then:

$$\mathsf{E}[lv](c) \equiv \exists y.\mathsf{IsMem}(y) \wedge label[lv](y, c)$$

- If E is a level-update expression $\texttt{upd\_lv}(\mathsf{E}_0, c, lv_0)$, then:
  - If $lv = lv_0$, then $\mathsf{E}[lv](x) \equiv \mathsf{E}_0[lv](x) \vee \mathtt{c}(x)$.
  - Otherwise, if $lv \neq lv'$, then $\mathsf{E}[lv](x) \equiv \mathsf{E}_0[lv](x) \wedge \neg \mathtt{c}(x)$.

For a declassification expression E, the derived V_{d} unary predicate $\mathsf{E}(c)$ used in the predicate update for `set_op_declass` is defined as follows:

- If E is the declassification expression $\texttt{mem\_decl}()$, then:

$$\mathsf{E}(c) \equiv \exists y.\mathsf{IsMem}(y) \wedge \mathsf{Declassifies}(y, c)$$

- If E is the declassification expression $\texttt{rem\_decl\_cat}(\mathsf{E}_0, \mathtt{c})$ for declassification expression E_0, then: $\mathsf{E}'(c) \equiv \mathsf{E}_0'(c) \wedge \neg \mathtt{c}(c)$.

From a structure-program model of P to a finite abstraction

To construct a finite over-approximation of the language of module traces of executions of the structure program $S_{P,T}$, `hiweave` applies a procedure $\mathsf{AbsStruct}$ that solves the structure-abstraction problem (defined in §2.3) $\mathsf{STRUCT\_ABS}(S_{P,T})$. Let $(S^{\#}_{P,T}, \mathsf{AbsNode}) = \mathsf{AbsStruct}(S_{P,T})$ be a solution produced by $\mathsf{AbsStruct}$ to the structure-abstraction problem $\mathsf{STRUCT\_ABS}(S_{P,T})$. Then from $S^{\#}_{P,T} = (Q^{\#}, \Sigma, \Delta^{\#})$ and $\mathsf{AbsNode}$, `hiweave` constructs the finite acceptor $A^{\#}_{P,T} = (Q_P, I_P, F_P, \Sigma_P, \Delta_P)$, where

- The states Q_P are the states of the abstraction $S^{\#}_{P,T}$. I.e., $Q_P = Q^{\#}$.
- The initial states I_P are the states of $S^{\#}_{P,T}$ that abstract states at the initial control location of $S_{P,T}$. I.e., $I_P = \{\iota \mid \iota \in Q^{\#}, \mathsf{AbsNode}(\iota) = \iota_S\}$.

- The alphabet Σ_P is the space of `difc` operations.
- The transition relation Δ_P is the transition relation of $S^{\#}_{P,T}$, with each operation that $S_{P,T}$ executes to model the environment replaced with an ϵ transition. I.e.,

$$\begin{aligned}\Delta_P = &\{(q, o, q') \mid (q, o, q') \in \Delta^{\#}, \mathsf{AbsNode}(q) \neq \mathsf{ENV}\} \\ &\cup \{(q, \epsilon, q') \mid o \in \mathtt{Op}, (q, o, q') \in \Delta^{\#}, \mathsf{AbsNode}(q) = \mathsf{ENV}\}\end{aligned}$$

- The final states F_P are all states of $S^{\#}_{P,T}$. I.e., $F_P = Q^{\#}$.

From DIFC policy Π to a structure program

From the input DIFC policy $\Pi = (Q_\Pi, \iota_\Pi, A_\Pi, \Sigma_\Pi, \Delta_\Pi)$, `hiweave` constructs a structure program $S_\Pi = (\mathsf{LOC}_S, \iota_S, \mathtt{O}_S, E_S, V_S, T_S)$ such that each trace of `difc` operations that violates Π drives S_Π to the error control location ERR. The components of S_Π are defined as follows.

Control locations of S_Π The control locations LOC_S of S_Π store the state of Π inhabited by the `difc` run simulated by S_Π. In particular, for each state $q \in Q_\Pi$, LOC_S contains control locations q and q'. LOC_S also contains an error location ERR.

Initial control location of S_Π The initial control location ι_S of S_Π is the initial state of Π, ι_Π.

Operations of S_Π The operations $\mathtt{O}_S$ of S_Π are the `difc` operations extended with a set of operations of the form $\mathtt{assume}[\varphi]$, where $\varphi \in \Sigma_F$ is a store condition in the alphabet of Π.

Control edges of S_Π The control edges E_S of S_Π define how S_Π maintains the state of Π inhabited by its current execution. In particular, E_S contains the following edges:

- For each pair of states $q_0, q_1 \in Q_F$ and store condition φ such that Π transitions from q_0 to q_1 on φ (i.e., $(q_0, \varphi, q_1) \in \Delta_F$), E_S contains a control edge $(q_0, \texttt{assume}[\varphi], q_1')$.

- For each policy state $q \in Q_F$ and each `difc` operation $\texttt{o}$, E_S contains a control edge $(q', \texttt{o}, q)$.

Vocabulary of S_Π The vocabulary of S_Π is the `difc` vocabulary V_{d}, defined in §9.3.1, Defn. 20.

Predicate transformers of S_Π The predicate transformers in S_Π of the `difc` operations are the predicate transformers of the `difc` operations in $S_{P,T}$. For each store condition $\varphi \in \mathsf{StoreCond}$, the predicate transformer for operation $\texttt{assume}[\varphi]$ checks that its pre-structure satisfies φ.

From a structure-program model of Π to a finite abstraction

To construct a finite over-approximation of the module traces of runs that violate Π, `hiweave` applies the procedure $\mathsf{AbsStruct}$ for solving a structure-program-abstraction problem (described in §9.5.2) to construct a finite abstraction $S_\Pi^\#$ of S_Π, and replaces each transition of $S_\Pi^\#$ that does not model a step of execution of a policy transition of Π with an ϵ transition. Let $(S_\Pi^\#, \mathsf{AbsNode}) = \mathsf{AbsStruct}(S_\Pi)$ be a solution produced by $\mathsf{AbsStruct}$ to the structure-abstraction problem $\mathsf{STRUCT\_ABS}(S_\Pi)$. Then from $S_\Pi^\# = (Q^\#, \Sigma, \Delta^\#)$ and $\mathsf{AbsNode}$, `hiweave` constructs the finite acceptor $A_\Pi^\# = (Q, I, F, \Sigma, \Delta)$, where:

- The states Q of $A_\Pi^\#$ are the states of the abstraction $S_\Pi^\#$. I.e., $Q = Q^\#$.

- The initial states I of $A^{\#}_{\Pi}$ are the states of $S^{\#}_{\Pi}$ that abstract states at the initial control location of S_{Π}. I.e., $I = \{\iota \mid \iota \in Q^{\#}, \mathsf{AbsNode}(\iota) = \iota_{\Pi}\}$.

- The final states F of $A^{\#}_{\Pi}$ are the states of $S^{\#}_{\Pi}$ that abstract states of S_{Π} whose control location is ERR. I.e., $F = \{q \mid q \in Q^{\#}, \mathsf{AbsNode}(q) = \mathsf{ERR}\}$.

- The alphabet Σ of $A^{\#}_{\Pi}$ is the space of `difc` operations `Op`.

- The transition relation Δ of $A^{\#}_{\Pi}$ is the transition relation of $S^{\#}_{\Pi}$, with transitions from locations in which S_{Π} does not execute a policy transition replaced with ϵ transitions:

$$\begin{aligned}\Delta =&\{(q, \mathtt{L} : \mathtt{o}, q') \mid (q, \mathtt{L} : \mathtt{o}, q') \in \Delta^{\#}, \mathtt{L} \neq \mathsf{ENV}\}\\ &\cup \{(q, \epsilon, q') \mid (q, \mathtt{L} : \mathtt{o}, q') \in \Delta^{\#}, \mathtt{L} = \mathsf{ENV}\}\end{aligned}$$

From template game, program approximation, and policy to a game

`hiweave` constructs $G_{P,\Pi}$ as a product of the template game G_T, the over-approximation $A^{\#}_{P}$ of the module traces of P, and the over-approximation $A^{\#}_{\Pi}$ of violations of Π. In particular $G_{P,\Pi} = G_T \times_{G,A} (\mathsf{det}(A^{\#}_{P}) \times \mathsf{det}(A^{\#}_{\Pi}))$, where for a non-deterministic finite-state acceptor A, $\mathsf{det}(A)$ is a deterministic acceptor that accepts the same language as A, and $\times_{G,A}$ is the game-automaton product defined in §2.2.

9.5.3 Designing label-operation templates

The label-operation template given to `hiweave` directly affects both the space of instrumentations considered by `hiweave`, as well as the size of the game constructed by `hiweave`. We found that templates for many practical programs on a DIFC system can be defined as regular languages that accept traces of (i) category creations and (ii) updates to the level of the operation label and operation declassification at categories stored in a bounded set of category variables. In §10.2, we discuss the effect of using templates of varying sophistication to instrument practical programs.

10 Evaluation

We carried out a series of experiments, designed to answer the following questions about our instrumentation technique:

1. Can practical information-DIFC policies be written as DIFC policies?
2. Can our instrumentation algorithm efficiently instrument practical programs to satisfy a policy represented as a DIFC policy?
3. Do programs instrumented by our algorithm perform comparably with programs instrumented by hand?

To answer the above questions, we implemented our weaving algorithm as a tool, `hiclang`, that performs a source-to-source translation in the LLVM intermediate language [37] to instrument programs to be run on HiStar. The steps of the `hiweave` algorithm described in §9.5 are implemented in `hiclang` as follows:

1. From an input program P, `hiweave` constructs a structure program S_P that simulates the executions of P. `hiclang` constructs S_P using the API provided by LLVM.
2. From an input DIFC policy F, `hiweave` constructs a structure program S_F whose executions violate F. `hiclang` constructs S_F by parsing F using a custom parser for DIFC policies.
3. `hiweave` constructs finite abstractions of the language of traces of S_P and S_F by applying a solver for the structure-program-abstraction

problem. hiclang constructs finite abstractions $S^\#_P$ and $S^\#_F$ of S_P and S_F by applying the TVLA logic-analysis engine [40].

4. hiweave constructs a game G from $S^\#_P$ and $S^\#_F$, and attempts to find a winning Defender strategy to G by applying a classical algorithm for solving two-player games. hiclang attempts to find a winning Defender strategy to G by applying a the game-solving algorithm implemented in the Goal tool [56].

5. If hiweave determines that the game G has a winning Defender strategy D, then from D, hiweave instruments P to satisfy C. hiclang checks if Goal found a winning Defender strategy D, and if so, uses the LLVM API to (1) generates multi-dimensional arrays the LLVM intermediate language that represent D, and instruments P with LLVM functions calls that invoke a fixed runtime-library function that updates program state and executes label operations.

To determine if practical policies can be expressed as DIFC policies (item 1), we collected a set of benchmark programs that were manually instrumented to be label programs by HiStar's developers [61]. For each benchmark program, we wrote a policy as a DIFC policy.

To determine if hiweave could instrument practical programs to satisfy their policies (item 2), we applied hiclang to each benchmark program and its policy. For each benchmark, we provided to hiweave multiple label-operation templates, which (1) directed hiweave to consider using either all levels or a restricted subset of levels derived from applying a simple heuristic to the policy,[1] and (2) either directed hiweave to choose in which states to allocate a fresh category or fixed a control location at which to allocate a category. We ran hiclang on a server with 16 2.4-GHz cores, and 32 GB of RAM, although hiclang executes in a single thread.

[1]If the policy specified a non-interference policy for the program's resources, then the template directed hiweave to use only middle-to-high levels, and if the policy specified an integrity policy, then the template directed hiweave to use only low-to-middle levels.

To determine if programs instrumented automatically by hiclang perform comparably to programs instrumented manually by an expert developer (item 3), we ran versions of each benchmark written manually and instrumented automatically by hiclang on representative workloads for the program. We ran each program in a HiStar virtual machine on the same server on which we ran hiclang.

In short, we found that policies for practical programs could be expressed as DIFC policies. We also found that hiweave could instrument relatively small, simple programs with trivial guidance from a label-operation template, and could instrument larger, more complex programs from label-operation templates that partially narrowed the search for an instrumentation using simple heuristics. The runtime-performance overhead of programs generated by hiweave compared to those written manually was negligible.

10.1 Benchmark Programs and Policies

In this section, we describe each benchmark program, describe the policy that we defined for the program, and describe the label operations that hiweave instrumented each program to execute.

10.1.1 A mutually-untrusting `login` service

The HiStar login service [61] allows a client with a username and password to request ownership of the user's private category upriv, while controlling to which objects on the system the client's password may flow. The login service is implemented as four distinct programs: a logging service auth_log, a login directory auth_dir, a user-authenticator auth, and an authentication client clnt. In a login session between a cooperating auth, auth_dir, clnt, and auth_log, clnt obtains a pointer to auth from auth_dir, and provides a client's password to auth. If the client's pass-

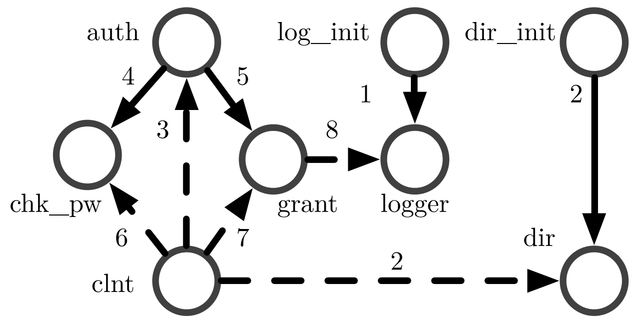

Figure 10.1: Gates created during an authentication session of the mutually-untrusting login service. Each node denotes a gate; a solid edge $\mathtt{g} \to \mathtt{h}$ denotes that a state executing gate $\mathtt{g}$ creates gate $\mathtt{h}$; a dashed edge $\mathtt{g} \to \mathtt{h}$ denotes that a state executing $\mathtt{g}$ calls $\mathtt{h}$.

word matches the user's password, then `auth` grants the client ownership of `upriv` and calls `auth_log` to log the event.

In each login session, `auth` creates multiple gates to check a password provided by a client, grant the client the user's access rights, and log the completion of a login session. The key interactions between gates in an authentication session are depicted in Fig. 10.1. Each node in Fig. 10.1 denotes a gate. A solid line from gate $\mathtt{g}$ to gate $\mathtt{h}$ annotated with $\mathtt{n}$ denotes that in step $\mathtt{n}$ of the session, a program executing gate $\mathtt{g}$ creates gate $\mathtt{h}$; a dashed line denotes that while executing gate $\mathtt{g}$, the program calls gate $\mathtt{h}$.

Before a login session occurs, users who need not be trusted by `auth` with the user's access rights or by `clnt` with the client's password initialize services used by the authenticator and client. In particular, some user executes `log_init`, which creates a log file and a gate bound to the module `logger`, which any program in `log_init`'s environment can invoke to append a message to the log (arc (1)); this step serves as the running example in Chapter 8. A potentially-distinct user executes `dir_init`, which creates a *login directory*, which is a map from each username to the user's authentication gate (which a user will typically choose to bind to the module `auth`), and a directory gate `dir_entry` bound to module `dir`, which a client can invoke to obtain a pointer to the user's authenticating gate.

`clnt` initiates a login session by calling the directory gate to obtain a pointer to the user's authenticating gate (arc (2)), which `clnt` then calls (arc (3)). In response, `auth` creates a gate bound to a module `chk_pw` (arc (4)) and a gate bound to a module `grant` (arc (5)). `clnt` then calls the `chk_pw` gate with the client password (arc (6)). If `chk_pw` determines that the client's password matches the user's password, then `chk_pw` allows the client to call the `grant` gate, and `clnt` does so (arc (7)). `grant` calls the `logger` gate to log that `clnt` has provided the user's password (arc (8)), and exits with a store in which the client owns upriv.

The key challenge in instrumenting the four programs that constitute login is to instrument the programs so that (1) if they each cooperate, then they can carry out the above session successfully, but (2) each program can ensure that if the other programs with which it interacts are malicious, then the security guarantees of the program are still upheld. In particular:

- `auth_log` ensures that a malicious program can only modify the log file by calling the `logger` gate, as described in Chapter 8.
- `auth_dir` ensures that a malicious program cannot directly modify the login directly.
- `auth` ensures that a malicious client cannot obtain the user's access rights unless the client provides a correct password and allows `auth` to log that it has granted the user's access rights.
- `clnt` ensures that a malicious authenticator cannot leak the password that it provides to any file on the system, including the log file.

The guarantees desired for the authentication directory are analogous to the guarantees desired for the logging service (i.e., an untrusted user should be able to read from but not directly modify the login directory), and so we do not describe the DIFC policy or the instrumentation of dir_init in further detail. The policies and instrumentation of auth and clnt, however, do differ significantly from the policies and instrumentation of auth_log and auth_dir, and we discuss them further below.

auth policy The policy for auth can be represented as a simple DIFC policy that ensures that (1) auth's environment can call a gate g_c whose module is chk_pw, (2) if auth's environment calls g_c with a valid password, then the environment can call a gate g_r whose module is grant, and (3) if auth's environment calls g_r, then it owns upriv. However, if auth's environment does not call g_c with the user's password and does not call g_r, the environment does not own upriv.

auth instrumentation hiweave instrumented a version of auth that implements the described DIFC policy. In particular, hiweave uses clearance labels in a non-trivial way to instrument auth, chk_pw, and grant to satisfy the above policy. The key invariant on labels maintained by auth, chk_pw, and grant is that the environment can only own upriv after calling grant, but the environment cannot call grant until it owns a *session category* sesh_cat. The environment can only own sesh_cat if it provides a client password to chk_pw that matches the user's password. To maintain the above invariants, the instrumented auth, chk_pw, and grant execute the following label operations:

1. auth creates a category sesh_cat.

2. auth creates the chk_pw gate so that it owns sesh_cat, and so that its clearance is high at sesh_cat.

3. auth creates the grant gate so that it does not own sesh_cat and its clearance is low at sesh_cat.

4. `auth` exits to its environment in a state in which memory has level Mid at `sesh_cat`. Thus the environment of `auth` is not able to call the `grant` gate directly, but the environment can call the `chk_pw` gate.

5. If the environment calls the `chk_pw` gate with a client password that matches the user's password, then `chk_pw` exits to the environment in a state in which memory owns `sesh_cat`, and thus the environment can call the `grant` gate. Otherwise, `chk_pw` exits in a state in which memory does not own `sesh_cat`.

6. If the `grant` gate is executed, then `grant` exits in a state in which memory owns the category upriv.

`clnt` policy The policy for `clnt` is a simple flow policy that ensures that (1) when `clnt` calls the `chk_pw` gate with `clnt`'s password, information cannot flow from the environment to any other object and (2) `clnt` calls the `grant` gate with a label such that `grant` can call the `logger` gate.

`clnt` instrumentation `hiweave` instrumented a version of `clnt` that satisfies the above policy. The secure version of `clnt` executes the following label operations during an execution:

1. After `clnt` calls the `auth` gate to create the session's `chk_pw` and `grant` gates, `clnt` creates a category `pw_cat`.

2. `clnt` calls the `chk_pw` gate to execute with memory that is high at `pw_cat`.

3. If the `chk_pw` gate determines that the client provided a password that matched the user's password, then `clnt` calls the `grant` gate with memory whose label has level Mid at `pw_cat`.

Instrumenting a practical login service Modules in the actual implementation of HiStar's login service perform several additional interactions not discussed above. In particular, `auth_dir` logs each request by a client to access the login directory, and `auth` logs the beginning of each login session, before the client provides a password. Furthermore, `auth` and `clnt` cooperate to construct a *retry file* such that `chk_pw` can maintain, in the retry file, a persistent count of the number of authentication attempts made by `clnt`, `chk_pw` ensures that the client cannot corrupt the retry file, and the client ensures that `chk_pw` can leak the client password only to the retry file. We omit a full description of these features for simplicity, but `hiweave` instruments `auth_dir`, `auth`, and `clnt` to satisfy the described policies.

10.1.2 `clamwrap`: a wrapper for ClamAV

The `clamav` virus scanner checks if the files on a filesystem match signatures from a database of viruses. Virus scanners are themselves targets of security attacks, because they must execute with the right to observe all files on a system, but execute large, complex code that can be exploited by a maliciously-crafted file, potentially to execute arbitrary code [15, 61]. In previous work [61], the HiStar developers wrote a wrapper program, `clamwrap`, that runs the ClamAV virus scanner so that ClamAV can only leak information to a personal temporary directory that cannot be read by any other process.

We expressed the requirements of `clamwrap` as a simple DIFC policy that specifies that:

1. When `clamwrap` creates a process executing `clamav`, the `clamav` process can write to its standard input/output file descriptors and a temporary directory allocated as a scratch space.

2. No matter what label operations the `clamav` process performs, it cannot modify any file other than its standard input/output file descriptors or files that are descendants of its temporary directory.

3. No matter what operations any process in the environment of `clamwrap` perform, the process cannot read from the standard input/output file descriptors for the `clamav` process, and cannot read from any file that is a descendant of the `clamav` process's temporary directory.

We applied `hiweave` to a version of `clamwrap` that performed no label operations and the above DIFC policy. `hiweave` instrumented `clamwrap` to perform the following label operations to satisfy the above DIFC policy:

1. `clamwrap` creates a fresh category `clamcat`.

2. `clamwrap` creates the standard file descriptors and temporary directory of `clamav` to be high at `clamcat`.

3. `clamwrap` creates the process that executes `clamav` to execute with a label that is high at `clamcat`, and with a declassification that does not contain `clamcat`.

This instrumentation of `clamcat` is semantically equivalent to the version of `clamwrap` written manually in previous work by the HiStar developers.

10.2 Results

Tabs. 10.2 and 10.3 contain the results of our experience applying `hiweave`. Tab. 10.2 contains data describing features of the benchmarks that we used when we applied `hiweave`. The columns of Tab. 10.2 are divided into (1) features of the input program, (2) features of the input policy, and (3) features of the templates to which we applied `hiweave`. Tab. 10.3 contains

Program			Policy		Template	
Name	LoC	Label sites	LoC	Trans.	Levels	Cat. creation
`auth_log`	54	5	21	4	Low Low All All	Fixed Choose Fixed Choose
`auth_dir`	157	6	34	6	Low Low All All	Fixed Choose Fixed Choose
`auth`	281	19	58	4	Low Low All All	Fixed Choose Fixed Choose
`clnt`	254	15	47	7	High High All All	Fixed Choose Fixed Choose
`clamwrap`	83	5	22	2	High High All All	Fixed Choose Fixed Choose

Table 10.2: Features of benchmark programs and policies to which we applied `hiweave`. Under the "Program" header, "Name" contains the name of the program, "LoC" contains the number of lines of code of the program, measured with the `cloc` utility (which does not count white space or comments); "Label Sites" contains the number of sites in the program that use a label when run on HiStar (e.g., when creating an object). Under the "Policy" header, "LoC" contains the number of lines of code of the DIFC policy; "Trans." contains the number of transitions in the flow policy. Under the "Template" header, "Levels" contains which levels the template directed `hiweave` to consider; "Cat creation" contains whether the template allowed `hiweave` to choose at which control locations an execution may create a category.

Benchmark		hiweave				Inst. Prog.
Name	T.	Time	Mem. (MB)	Structs	Game States	Slow-down
auth_log	Low, Fix.	0m33s	4,509	441	55	1
	Low, Ch.	0m38s	4,524	998	71	1
	All, Fix.	0m39s	4,510	1428	55	1
	All, Ch.	1m05s	7,116	4113	71	1
auth_dir	Low, Fix.	1m18s	7,009	599	60	1
	Low, Ch.	1m42s	7,142	2796	93	1
	All, Fix.	2m05s	7,307	3429	60	1
	All, Ch.	23m23s	10,359	20332	122	1.1
auth	Low, Fix.	29m57s	15,981	16389	345	1.1
	Low, Ch.	MEM	-	-	-	-
	All, Fix.	MEM	-	152,155	-	-
	All, Ch.	MEM	-	-	-	-
clnt	High, Fix.	9m22s	8,033	1,380	200	1.1
	High, Ch.	9m54s	9,462	3,221	459	1.2
	All, Fix.	MEM	-	44,072	-	-
	All, Ch.	TIME	-	93,451	-	-
clamwrap	High, Fix.	0m50s	1,754	564	116	1
	High, Ch.	1m03s	4,582	943	151	1.1
	All, Fix.	9m00s	11,826	11,720	241	1.1
	All, Ch.	32m18s	15,496	20,674	281	1.1

Table 10.3: Results of applying hiweave to the benchmarks described in Tab. 10.2. The "Benchmark" header contains the name of each benchmark program and the template that we provided to hiweave (whether hiweave fixes the location at which a category is created or directs hiweave to choose is abbreviated as "Fix." and "Ch.," respectively. The hiweave header contains data about the runtime behavior of the program instrumented under a given template. "Time" contains the execution time of hiweave; "Mem." contains the peak memory usage of hiweave; "Structures" contains the number of structures constructed by the solver for structure-analysis problems that was applied by hiweave; "Game States" contains the number of states in the minimal game constructed by hiweave from the transition graph over structures. Under the "Instrumented Program" header, "Slowdown" contains the running time of the instrumented program expressed as a multiple of the running time of the original, manually-instrumented program.

data describing features of the performance of applying `hiweave`. The columns of Tab. 10.3 are divided into (1) identification of the benchmark program and template described in Tab. 10.2, (2) data concerning the performance of `hiweave` in instrumenting the benchmark, and (3) data concerning the runtime performance of the version of the benchmark instrumented by `hiweave`.

The results indicate that `hiweave` can be used to efficiently instrument small programs that perform multiple, complex operations on the system store. In particular, we observe the following:

- We were able to write succinct DIFC policies that described the information-flow components of the login service. Relatively larger modules of the login service that used labels in more operations did not require proportionately larger policies.

- `hiweave` could instrument `clamwrap` and the smaller, simpler components of the login service using each template that we provided, even templates that gave few directives for searching for a correct instrumentation. However, `hiweave` could not instrument the larger, more complex modules of the login service unless it was provided with a template that gave non-trivial directives for searching for an instrumentation. In particular, we found that templates that limited the set of levels considered by `hiweave` were effective in directing `hiweave`'s search, even if the templates did not give any direction as to where an instrumented program should allocate a category. We conjecture that this is because the abstraction used by `hiweave`'s structure analysis distinguishes cells that model objects with distinct levels, but compactly summarizes multiple categories at which relevant objects have identical levels.

- The set of abstract states in the abstract transition system generated by the structure analysis is often significantly larger than the set of states in the minimal automaton that accepts the same language of traces. This property indicates that “local” decisions that the structure analysis makes for distinguishing structures based on a fixed abstraction tend to cause the analysis to maintain distinct structures that are equivalent in terms of which traces executed from states abstracted by the structures violate the DIFC policy.

- Although the code generated by `hiweave` requires the instrumented program to lookup a post-state and pointer to a label operation and then invoke the label operation, the runtime cost of executing instrumentation code appears to be negligible compared to the cost of other program operations: an effect was only measurable on very small workloads, which we suspect is due to the relatively high fixed cost of initializing the strategy table into program data structures.

Part III

Generating Weavers

In this part of the dissertation, we generalize the designs of the policy weavers for Capsicum and HiStar by describing the design of a policy-weaver generator WeaverGen. WeaverGen takes as input a semantics for both a language and system primitives, and outputs a policy weaver for the input language and system. In Chapter 11, we define the policy-weaving problem parameterized on input language and system semantics. In Chapter 12, we describe the design of WeaverGen.

11 The Parameterized Weaving Problem

In this chapter, we introduce the parameterized weaving problem, which generalizes the weaving problems for Capsicum (§5.4) and HiStar (§9.4). We first define models of languages (§11.1) and systems (§11.2). We then generalize definitions of valid instrumentation (§11.3), policy satisfaction (§11.4), and weaving (§11.5) to be parameterized on given language and system models.

For the rest of the discussion, we fix a space of control locations LOC that contains the distinguished control locations $\mathtt{INIT}$, which represents the initial control location of programs, and and ENV, which represents the environment of programs, a space of operations $\mathtt{O}$, and a space of objects O. O is also the fixed universe of all logical structures discussed.

11.1 Language models

A language model is a language semantics paired with a function that instruments programs in the language to have runs described by a finite state machine. A language semantics defines a transition system for each program in the language.

Definition 24. A *language semantics* $\mathcal{L}$ is a triple consisting of:

- A transition system $(\mathsf{Stores}, \mathtt{O}, \rightarrow)$ over a space of stores Stores, with transition relation $\rightarrow$.
- A space of programs $\mathcal{Q}$.
- A function $\mathsf{ProgCFG} : \mathcal{Q} \rightarrow \mathcal{P}(\mathsf{LOC} \times \mathtt{O} \times \mathsf{LOC})$ that maps each program in $\mathcal{Q}$ to its control-flow graph.

The *state space* of $\mathcal{L}$ is $Q_{\mathcal{L}} = \mathsf{LOC} \times \mathsf{Stores}$. For program $\mathtt{P}$, the transition relation $\rightarrow^{\mathcal{L}}_{\mathtt{P}} \subseteq Q_{\mathcal{L}} \times \mathtt{O} \times Q_{\mathcal{L}}$ of $\mathtt{P}$ is defined by the control-flow graph of $\mathtt{P}$ and the transition relation over $\mathcal{L}$-stores. In particular, for control locations $\mathtt{L}, \mathtt{L}' \in \mathsf{LOC}$ and stores $\sigma, \sigma' \in \mathsf{Stores}$, if $(\mathtt{L}, \mathtt{o}, \mathtt{L}') \in \mathsf{ProgCFG}(\mathtt{P})$ and $(\sigma, \mathtt{o}, \sigma') \in \rightarrow$, then $((\mathtt{L}, \sigma), \mathtt{o}, (\mathtt{L}', \sigma')) \in \rightarrow^{\mathcal{L}}_{\mathtt{P}}$.

Example 13. A language semantics for `capcore` is defined in §5.1.1; a language semantics for `difccore` is defined in §9.1.1.

An instrumentation generator I for language semantics $\mathcal{L}$ is a procedure that takes as input a program $\mathtt{P}$ of $\mathcal{L}$ and transition functions over finite state spaces, which define how $\mathtt{P}$ must be instrumented to execute additional operations. I instruments $\mathtt{P}$ to execute operations as specified by the transition functions.

Definition 25. Let $\mathcal{L}$ be a language semantics, let M be a finite space of *monitor states*, let A be a finite space of *action states* disjoint from M, let $\iota \in M$ be an *initial monitor state*, let $\tau_M : M \times \mathtt{O} \rightarrow (M \cup A)$ be a *monitor function*, and let $\tau_I : A \rightarrow (\mathtt{O} \times (M \cup A))$ be an *operation-injection function*. A *valid instrumentation* of $\mathtt{P}$ under M, A, ι, τ_M, and τ_I is a program $\mathtt{P}' \in \mathcal{Q}_{\mathcal{L}}$ such that for each trace of operations $t = \mathtt{o}_0, \ldots, \mathtt{o}_n \in \mathtt{O}^*$ from a run of $\mathtt{P}'$, there is a sequence of monitor and action states $Q = q_0, \ldots, q_{n+1} \in (M \cup A)^*$ such that:

- The first state of Q is the initial state (i.e., $\iota = q_0$).

- For each $0 \leqslant i \leqslant n$, if $q_i \in M$, then $\tau_M(q_i, o_i) = q_{i+1}$. Otherwise, if $q_i \in A$, then $\tau_A(q_i) = (o_{i+1}, q_{i+1})$.

An *instrumentation generator* of a language semantics $\mathcal{L}$ is a procedure that takes each program $P \in \mathcal{L}$, finite space of monitor states, finite space of action states, and transition functions over the states, and generates a valid instrumentation of P.

Example 14. The instrumentation generators for `capcore` and `difccore` are CapCodeGen and DIFCCodeGen, introduced in §5.5.1 and §9.5.1, respectively.

Definition 26. A *language model* is a triple consisting of (1) a language semantics $\mathcal{L}$, (2) a structure simulation (§2.3, Defn. 8) $\mathcal{R}_{\mathcal{L}}$ of the transition system of $\mathcal{L}$, and (3) an instrumentation-generator $I_{\mathcal{L}}$ for $\mathcal{L}$.

Example 15. Chapter 5 presents a structure simulation $S_{\mathtt{cc}}$ for `capcore`. The state vocabulary of $S_{\mathtt{cc}}$—$V_{\mathtt{cc}}$—and structure-simulation function from `capcore` stores are defined in §5.3.1, Defn. 11, and the predicate transformers for predicates in $V_{\mathtt{cc}}$ are defined in §5.5.2. The `capcore` semantics (Ex. 13), structure simulation $S_{\mathtt{cc}}$, and instrumentation generator CapCodeGen (Ex. 14) form a language model for `capcore`.

Chapter 9 presents a structure simulation $S_{\mathtt{dc}}$ for `difccore`. The state vocabulary of $S_{\mathtt{dc}}$—$V_{\mathtt{dc}}$—and structure-simulation function from `difccore` stores to $V_{\mathtt{dc}}$ structures are defined in §9.3.1, Defn. 19, and the predicate transformers for predicates $V_{\mathtt{dc}}$ are defined in §9.5.2. The `difccore` semantics (Ex. 13), structure simulation $S_{\mathtt{dc}}$, and instrumentation generator DIFCCodeGen (Ex. 14) form a language model for `difccore`.

11.2 System models

A system model $\mathcal{S}$ consists of a system semantics as a transition relation over system stores, a structure simulation for the semantics, and a template language of sequences of operations that the weaver may consider instrumenting before any given operation.

Definition 27. A *system semantics* is a transition system $(\mathsf{Stores}, \mathtt{O}, \rightarrow)$.

Example 16. A system semantics for `cap` is presented in §5.1.3; A system semantics for `difc` is presented in §9.1.3.

Definition 28. A *system model* is a triple of (1) a system semantics S and (2) a structure simulation $\mathcal{R}_S$ for S, and (3) a *template* language T_S of operation sequences, represented as a finite-state acceptor over the alphabet of operations $\mathtt{O}$.

Example 17. Chapter 5 describes a structure simulation $S_{\mathtt{c}}$ for `cap`. The state vocabulary of $S_{\mathtt{c}}$—$V'_{\mathtt{c}}$—and structure simulation function are defined in §5.3.1, Defn. 12, and the predicate transformers for predicates in $S_{\mathtt{c}}$ are defined in §5.5.2. §5.5.3 describes a useful template $T_{\mathtt{c}}$ of `cap` operations. The system semantics for `cap` (Ex. 16), $S_{\mathtt{c}}$, and $T_{\mathtt{c}}$ form a system model for `cap`.

Chapter 9 describes a structure simulation $S_{\mathtt{d}}$ for `difc`. The state vocabulary of $S_{\mathtt{d}}$—$V'_{\mathtt{d}}$—and structure simulation function are defined in §9.3.1, Defn. 20, and the predicate transformers for predicates in $S_{\mathtt{d}}$ are defined in §9.5.2. §9.5.3 describes practical templates of `difc` operations. The system semantics for `difc` (Ex. 16), $S_{\mathtt{d}}$, and the templates form a system model for `difc`.

Each language model $\mathcal{L}$ and system model $\mathcal{S}$ define a transition relation for each $\mathcal{L}$-program that executes on $\mathcal{S}$.

Definition 29. For language model $\mathcal{L}$ and system model $\mathcal{S}$, let an $\mathcal{L}, \mathcal{S}$-*state* be a triple of a control location, a $\mathcal{L}$ store, and a $\mathcal{S}$ store; i.e., the space of $\mathcal{L}, \mathcal{S}$ states is $Q_{\mathcal{L},\mathcal{S}} = \mathsf{LOC} \times \mathsf{Stores}_{\mathcal{L}} \times \mathsf{Stores}_{\mathcal{S}}$.

For $\mathcal{L}$-program P, let the transition relation of P *executing on* $\mathcal{S}$, denoted by $\rightarrow_{\mathsf{P}}^{\mathcal{L},\mathcal{S}} \subseteq Q_{\mathcal{L},\mathcal{S}} \times \mathtt{O} \times Q_{\mathcal{L},\mathcal{S}}$, be a transition relation over $\mathcal{L}, \mathcal{S}$-states, defined as follows. For control locations $\mathtt{L}, \mathtt{L}' \in \mathsf{LOC}$, $\mathcal{L}$-stores L and L', $\mathcal{S}$-stores S and S', and operation $\mathtt{o} \in \mathtt{O}$, if $((\mathtt{L}, L), \mathtt{o}, (\mathtt{L}', L')) \in \rightarrow_{\mathtt{P}}^{\mathcal{L}}$ and $S \rightarrow^{\mathcal{S}} S'$, then $(\mathtt{L}, L, S) \rightarrow_{\mathtt{P}}^{\mathcal{L},\mathcal{S}} (\mathtt{L'}, L', S')$.

For $\mathcal{L}, \mathcal{S}$-states $q, q' \in Q_{\mathcal{L},\mathcal{S}}$, q *reaches* q', denoted by $q \Rightarrow_{\mathtt{P}}^{\mathcal{L},\mathcal{S}} q'$, if there is some operation $\mathtt{o} \in \mathtt{O}$ such that $(q, \mathtt{o}, q') \in \rightarrow_{\mathtt{P}}^{\mathcal{L},\mathcal{S}}$.

Each language model $\mathcal{L}$ and system model $\mathcal{S}$ define a map from each pair of an $\mathcal{L}$-store and $\mathcal{S}$-store to a structure over the union $V_{\mathcal{L},\mathcal{S}}$ of the vocabularies of $\mathcal{L}$ and $\mathcal{S}$. The map from stores to structures defines a map from each run of an $\mathcal{L}$-program on $\mathcal{S}$ to a sequence of control locations paired with structures over vocabulary $V_{\mathcal{L},\mathcal{S}}$.

Definition 30. For language model $\mathcal{L}$ and system model $\mathcal{S}$, let the *store simulation function of $\mathcal{L}$ and $\mathcal{S}$* $\mathsf{StoreToStruct}_{\mathcal{L},\mathcal{S}} : \mathsf{Stores}_{\mathcal{L}} \times \mathsf{Stores}_{\mathcal{S}} \rightarrow \mathsf{STRUCTS}[V_{\mathcal{L}} \cup V_{\mathcal{S}}]$ map each pair of an $\mathcal{L}$-store L and $\mathcal{S}$-store S to the union of the structures that model L and S (the union of structures is defined in §2.3, Defn. 6). I.e., for each $\mathcal{L}$-store $L \in \mathsf{Stores}_{\mathcal{L}}$ and $\mathcal{S}$-store $S \in \mathsf{Stores}_{\mathcal{S}}$, $\mathsf{StoreToStruct}_{\mathcal{L},\mathcal{S}}(L, S) = \mathsf{StoreToStruct}_{\mathcal{L}}(L) \cup \mathsf{StoreToStruct}_{\mathcal{S}}(S)$.

For a run $r = q_0, \ldots, q_n \in (Q_{\mathcal{L},\mathcal{S}})^*$, let the corresponding *structure run of* r be the sequence of control locations paired with structures $r' = (\mathtt{L}_0', \mathsf{StoreToStruct}_{\mathcal{L},\mathcal{S}}(L_0, S_0)), \ldots, (\mathtt{L}_n, \mathsf{StoreToStruct}_{\mathcal{L},\mathcal{S}}(L_n, S_n))$.

11.3 Parameterized valid instrumentation

A valid instrumentation $\mathtt{P}'$ of a program $\mathtt{P}$ in language model $\mathcal{L}$ for system model $\mathcal{S}$ is any program for which the $\mathcal{L}$-store of each successive state in a run is determined by the transition relation of $\mathtt{P}$.

Definition 31. For $\mathcal{L}$ programs $\mathtt{P}$ and $\mathtt{P}'$, an *$\mathcal{L}, \mathcal{S}$-refinement relation* $\sim \subseteq Q_{\mathcal{L},\mathcal{S}} \times Q_{\mathcal{L},\mathcal{S}}$ is a binary relation over $Q_{\mathcal{L},\mathcal{S}}$ such that:

1. $\sim$ only relates $\mathcal{L}, \mathcal{S}$ states with equal $\mathcal{L}$-stores. I.e., for states $q = (\mathtt{L}, \Lambda, S) \in Q_{\mathcal{L},\mathcal{S}}$ and $q' = (\mathtt{L}', \Lambda', S') \in Q_{\mathcal{L},\mathcal{S}}$, if $q \sim q'$, then $\Lambda = \Lambda'$.

2. If a pair of states (q, q') is in $\sim$, then each $\mathcal{L}$-store of a successor of q' in one step of P is paired with a successor over multiple steps of P'. I.e., for $q, q' \in Q_{\mathcal{L},\mathcal{S}}$ such that $q \sim q'$, if $q \Rightarrow_{\mathsf{P}}^{\mathcal{L},\mathcal{S}} q_1$, then there is some $\mathcal{L}, \mathcal{S}$-state q_1' such that $q \Rightarrow_{\mathsf{P}'}^{\mathcal{L},\mathcal{S}*} q_1'$, and $q_1 \sim q_1'$.

P' is an $\mathcal{L}, \mathcal{S}$-*refinement* of P if there is some $\mathcal{L}, \mathcal{S}$ refinement relation $\sim$ such that for each $\mathcal{L}$ store L and $\mathcal{S}$-store S, $(\mathtt{INIT}, \mathsf{L}, \mathsf{S}) \sim (\mathtt{INIT}, \mathsf{L}, \mathsf{S})$.

Example 18. Capability refinement (defined in §5.2) is an instance of an $\mathcal{L}, \mathcal{S}$ refinement relation for the `capcore` language and `cap` system. Label refinement (defined in §9.2) is an instance of an $\mathcal{L}, \mathcal{S}$ refinement relation for the `difccore` language and `difc` system.

11.4 Parameterized Policy Satisfaction

For language model $\mathcal{L}$ and system model $\mathcal{S}$, a policy specifies disallowed runs of states of an $\mathcal{L}$-program executing on $\mathcal{S}$.

Definition 32. For language model $\mathcal{L}$ and system model $\mathcal{S}$, let a *state condition* be a control location paired with a closed formula over the union of the state vocabularies for $\mathcal{L}$ and $\mathcal{S}$; i.e., the space of state conditions is $\mathsf{StateConds} = \mathsf{LOC} \times \mathsf{FORMS}[V_{\mathcal{L}} \cup V_{\mathcal{S}}]$.

A *policy* for $\mathcal{L}$ and $\mathcal{S}$ is a finite-state automaton over the alphabet $\mathsf{StateConds}$; the space of all policies over $\mathcal{L}$ and $\mathcal{S}$ is denoted by $\mathsf{Pols}_{\mathcal{L},\mathcal{S}}$.

Let $r \in (Q_{\mathcal{L},\mathcal{S}})^*$ be a run of $\mathcal{L}$-program P on $\mathcal{S}$, and let $t_S = (\mathtt{L}_0, s_0), \ldots, (\mathtt{L}_n, s_n) \in (\mathsf{LOC} \times \mathsf{STRUCTS}[V_{\mathcal{L}} \cup V_{\mathcal{S}}])^*$ be the structure run of r (Defn. 30). If a sequence of state conditions $r_A = a_0, \ldots, a_n \in \mathsf{StateConds}$ is such that for each $0 \leqslant i \leqslant n$ and $a_i = (\mathtt{L}_i', \varphi_i)$, (1) $\mathtt{L}_i = \mathtt{L}_i'$ and (2) $s_i \models \varphi_i$, then r_A is a *state-condition run* of r. r *violates* policy Π if Π accepts some state-condition run of r. If each run r of P does not violate Π, then P *satisfies* Π.

Example 19. The space of capability policies CapPols (defined in §5.3.2) is an instance of $\mathsf{Pols}_{\mathcal{L},\mathcal{S}}$ for language model `capcore` and system model `cap`. The space of DIFC policies DIFCPols (§9.3.2) is an instance of $\mathsf{Pols}_{\mathcal{L},\mathcal{S}}$ for a language `difccore` and system `difc`.

11.5 Problem definition

For a language model $\mathcal{L}$ and system model $\mathcal{S}$, the policy-weaving problem for $\mathcal{L}$ and $\mathcal{S}$ is to take a program P in $\mathcal{L}$ and a policy $\Pi \in \mathsf{Pols}_{\mathcal{L},\mathcal{S}}$ and instrument P so that it satisfies Π when it executes on $\mathcal{S}$.

Definition 33. Let $\mathcal{L}$ be a language model, let $\mathcal{S}$ be a system model, let P be an $\mathcal{L}$-program, and let $\Pi \in \mathsf{Pols}_{\mathcal{L},\mathcal{S}}$ be a policy for $\mathcal{L}$ and $\mathcal{S}$. A solution to the *policy-weaving problem* for $\mathcal{L}$ and $\mathcal{S}$, denoted $\mathsf{WV\_PROB}_{\mathcal{L},\mathcal{S}}(\mathsf{P}, \Pi)$, is a valid instrumentation of P that satisfies Π. A procedure that solves the policy-weaving problem for $\mathcal{L}$ and $\mathcal{S}$ is a *policy weaver* for $\mathcal{L}$ and $\mathcal{S}$.

Example 20. `capweave` (§5.5) is a policy weaver for language `capcore` and system `cap`. `hiweave` (§9.5) is a policy weaver for language `difccore` and system `difc`.

A *policy-weaver generator* takes as input a language model $\mathcal{L}$ and a system model $\mathcal{S}$, and outputs a policy weaver for $\mathcal{L}$ and $\mathcal{S}$.

12 Design of a Weaver Generator

In this chapter, we describe the design of a policy weaver generator WeaverGen. WeaverGen is a single procedure that takes as input a language model $\mathcal{L}$, a system model $\mathcal{S}$, program P in $\mathcal{L}$, and policy Π for $\mathcal{L}$ and $\mathcal{S}$, and produces an $\mathcal{L}$-program P' that is a valid instrumentation of P that satisfies Π when it executes on $\mathcal{S}$. The design of WeaverGen can be resolved with the definition of a weaver generator in §11.5 by viewing the partial application of WeaverGen applied to only the language model $\mathcal{L}$ and the system model $\mathcal{S}$ as the policy weaver generated by WeaverGen for $\mathcal{L}$ and $\mathcal{S}$.

In §12.1, we give an overview of the design of WeaverGen. In §12.2, we describe in more detail the game construction performed by WeaverGen. In §12.3, we claim the soundness of WeaverGen and, by extension, `capweave` and `hiweave`.

12.1 Overview

Alg. 12.1 contains pseudocode for the WeaverGen algorithm. WeaverGen first constructs (line [1]) from $\mathcal{L}$, $\mathcal{S}$, P, and Π a finite two-player game $\mathsf{G}^{\mathcal{L},\mathcal{S}}_{\mathsf{P},\Pi}$ whose alphabet is the space of operations O, such that for any Defender strategy D that wins $\mathsf{G}^{\mathcal{L},\mathcal{S}}_{\mathsf{P},\Pi}$, the plays of D are the traces of a solution to $\mathsf{WV\_PROB}_{\mathcal{L},\mathcal{S}}(\mathsf{P},\Pi)$ (for brevity, we say that D *defines* a solution to $\mathsf{WV\_PROB}_{\mathcal{L},\mathcal{S}}(\mathsf{P},\Pi)$). The construction of $\mathsf{G}^{\mathcal{L},\mathcal{S}}_{\mathsf{P},\Pi}$, represented in Alg. 12.1 as ProgPolicyGame, is a generalization of the constructions of games per-

Input : A language model $\mathcal{L}$, system model $\mathcal{S}$, program $\mathsf{P} \in \mathcal{L}$, policy $\Pi \in \mathsf{Pols}_{\mathcal{L},\mathcal{S}}$
Output : A solution to $\mathsf{WV\_PROB}_{\mathcal{L},\mathcal{S}}(\mathsf{P}, \Pi)$
1 $\mathsf{G}^{\mathcal{L},\mathcal{S}}_{\mathsf{P},\Pi} \coloneqq$ `ProgPolicyGame`$(\mathcal{L}, \mathcal{S}, \mathsf{P}, \Pi)$;
2 **if** `HasWinningDefenderStrategy`$(\mathsf{G}^{\mathcal{L},\mathcal{S}}_{\mathsf{P},\Pi})$ **then**
3     $\mathsf{D} \coloneqq$ `FindWinningDefenderStrategy`$(\mathsf{G}^{\mathcal{L},\mathcal{S}}_{\mathsf{P},\Pi})$;
4   **return** `InstrumentationGen`$_{\mathcal{L}}(\mathsf{P}, \mathsf{D})$;
5 **else**
6   `Fail` () ;

Figure 12.1: Pseudocode for the WeaverGen algorithm. The algorithm is described in detail in §12.1.

formed by `capweave` and `hiweave`, described in §5.5.2 and §9.5.2, and represented as CapProgPolicyGame and DIFCProgPolicyGame in Algs. 5.5 and 9.6. The implementation of `ProgPolicyGame` is described in more detail in §12.2.

WeaverGen then applies a classical algorithm `HasWinningDefenderStrategy` to determine if $\mathsf{G}^{\mathcal{L},\mathcal{S}}_{\mathsf{P},\Pi}$ has a winning Defender strategy (line [2]). If $\mathsf{G}^{\mathcal{L},\mathcal{S}}_{\mathsf{P},\Pi}$ has a winning Defender strategy, then WeaverGen applies a classical algorithm `FindWinningDefenderStrategy` to construct a winning Defender strategy D (line [3]). Otherwise, WeaverGen fails (line [6]); we discuss conditions under which WeaverGen fails, and possible extensions to our work to mitigate such conditions, in Chapter 14.

If WeaverGen finds a winning Defender strategy D for $\mathsf{G}^{\mathcal{L},\mathcal{S}}_{\mathsf{P},\Pi}$, then WeaverGen applies `InstrumentationGen`$_{\mathcal{L}}$ to P and D to generate an instrumentation P' of P that satisfies Π (line [4]). Implementations of `InstrumentationGen`$_{\mathcal{L}}$ for `capcore` and `difccore` are described in Ex. 14.

In the remainder of this section, we describe in more detail how WeaverGen constructs a finite game $\mathsf{G}^{\mathcal{L},\mathcal{S}}_{\mathsf{P},\Pi}$ whose winning Defender strategies define instrumentations of P that satisfy Π (i.e., how WeaverGen implements

ProgPolicyGame in Alg. 12.1).

12.2 From models, program, and policy to a game

From language model $\mathcal{L}$, system model $\mathcal{S}$, program $\mathsf{P} \in \mathcal{L}$, and policy $\Pi \in \mathsf{Pols}_{\mathcal{L},\mathcal{S}}$, WeaverGen constructs a game $\mathsf{G}^{\mathcal{L},\mathcal{S}}_{\mathsf{P},\Pi}$ such that each winning Defender strategy of $\mathsf{G}^{\mathcal{L},\mathcal{S}}_{\mathsf{P},\Pi}$ defines an instrumentation of P for $\mathcal{S}$ that satisfies Π. In this section, we describe the construction, which is a generalization of the game construction for `cap` described in §5.5.2, and the game construction for `difc` described in §9.5.2.

To construct $\mathsf{G}^{\mathcal{L},\mathcal{S}}_{\mathsf{P},\Pi}$, WeaverGen performs the following steps.

1. From the $\mathcal{S}$-template $\mathsf{T} = \mathsf{T}_{\mathcal{S}}$, WeaverGen constructs a finite two-player game G_{T} such that each play of G_{T} won by the Defender is a sequence of operations accepted by $\mathsf{T}_{\mathcal{S}}$ chosen by the Defender, followed by an operation chosen by the Attacker. The construction of G_{T} is an immediate generalization of the constructions described in §5.5.2 for `cap` and in §9.5.2 for `difc`; we thus omit a detailed description.

2. From P and $\mathsf{T}_{\mathcal{S}}$, WeaverGen constructs a structure program (defined in §2.3) $\mathsf{S}_{\mathsf{P},\mathsf{T}}$ such that each run of $\mathsf{S}_{\mathsf{P},\mathsf{T}}$ simulates a run r of P with a sequence of operations accepted by $\mathsf{T}_{\mathcal{S}}$ injected before each operation of r. To construct $\mathsf{S}_{\mathsf{P},\mathsf{T}}$, WeaverGen applies $\mathsf{ProgCFG}_{\mathcal{L}}$ to P to construct the control-flow graph of P. WeaverGen injects a "copy" of $\mathsf{T}_{\mathcal{S}}$ before each non-environment control location in S_{P} to construct $\mathsf{S}_{\mathsf{P},\mathsf{T}}$. The predicate transformers of $\mathsf{S}_{\mathsf{P},\mathsf{T}}$ are defined directly from the union of the predicate transformers of the structure simulation $\mathcal{R}_{\mathcal{L}}$ and $\mathcal{R}_{\mathcal{S}}$. The construction of $\mathsf{S}_{\mathsf{P},\mathsf{T}}$ from S_{P} is an immediate generalization of

the constructions of $S_{P,T}$ for `cap` and `difc` given in §5.5.2 and §9.5.2; we thus omit a detailed description.

3. From $S_{P,T}$, WeaverGen constructs a finite-state acceptor $A^{\#}_{P,T}$ of traces of operations such that each trace that drives $S_{P,T}$ to a designated control location is accepted by $A^{\#}_{P,T}$. To construct $A^{\#}_{P,T}$, WeaverGen applies a procedure $\mathsf{AbsStruct}$ that solves the structure-abstraction problem (described in Chapter 2) $\mathsf{AbsStruct}(S_{P,T})$. The construction of $A^{\#}_{P,T}$ is an immediate generalization of the constructions given in §5.5.2 and §9.5.2; we thus omit a detailed description.

4. From policy Π, WeaverGen constructs a structure program S_{Π} such that each trace of a run that does not satisfy Π drives S_{Π} to an error control location. The intuition behind the construction is that each control location of S_{Π} is a copy of a state of Π, and each transition of S_{Π} checks that the store condition in the state condition of a corresponding transition of Π holds in the current state of S_{Π}. The predicate transformers of S_{Π} are defined directly from the union of the predicate transformers of the structure simulation $\mathcal{R}_{\mathcal{L}}$ and $\mathcal{R}_{\mathcal{S}}$. The construction is an immediate generalization of the constructions given in §5.5.2 and §9.5.2; we thus omit a detailed description.

5. From S_{Π}, WeaverGen constructs a finite-state acceptor $A^{\#}_{\Pi}$ of traces of operations such that each trace that drives S_{Π} to an error control location is accepted by $A^{\#}_{\Pi}$. To construct $A^{\#}_{\Pi}$, WeaverGen applies the procedure $\mathsf{AbsStruct}$ to construct a finite abstraction $S^{\#}_{\Pi}$ of S_{Π}. From $S^{\#}_{\Pi}$, WeaverGen constructs a finite-state acceptor that accepts an over-approximation $A^{\#}_{\Pi}$ of the traces of violations of Π by replacing each operation at an environment location with an ϵ transition. The construction of $A^{\#}_{\Pi}$ is an immediate generalization of the constructions given in §5.5.2 and §9.5.2; we thus omit a detailed description.

6. $\textsc{WeaverGen}$ constructs $G^{\mathcal{L},\mathcal{S}}_{P,\Pi}$ as the product of G_T, $A^{\#}_{P,T}$, and $A^{\#}_{\Pi}$. In particular $G^{\mathcal{L},\mathcal{S}}_{P,\Pi} = G_T \times_{G,A} (\mathsf{det}(A^{\#}_{P,T}) \times \mathsf{det}(A^{\#}_{\Pi}))$, where for a non-deterministic finite-state acceptor A, $\mathsf{det}(A)$ is a deterministic acceptor that accepts the same language as A, and $\times_{G,A}$ is the game-automaton product defined in §2.2.

12.3 Soundness

WeaverGen is sound: if it generates a program, then the program is a valid instrumentation of its input program that satisfies its input policy.

Theorem 12.2. *For each language model $\mathcal{L}$, system model $\mathcal{S}$, $\mathcal{L}$-program P, and policy $\Pi \in \mathsf{Pols}_{\mathcal{L},\mathcal{S}}$, if* WeaverGen$(\mathcal{L}, \mathcal{S}, P, \Pi) = P'$, *then P' is a solution of* $\mathsf{WV\_PROB}_{\mathcal{L},\mathcal{S}}(P, \Pi)$.

Proof. (Sketch) To show that P' satisfies Π, we will show that for each run r of P', if r violates Π, then the trace of r, denoted $\tau(r)$, would be an Attacker-winning play of $G^{\mathcal{L},\mathcal{S}}_{P,\Pi}$, which, combined with the construction of P', implies a contradiction.

For the sake of deriving a contradiction, suppose that r is a run of P' that violates policy Π. For operation template $T = T_{\mathcal{S}}$, the trace of operations of r is a trace of the structure program $S_{P,T}$ by the construction of $S_{P,T}$ (§12.2, item 2)—in particular, the fact that the state space and transformers of $S_{P,T}$ are constructed from structure simulations $\mathcal{R}_{\mathcal{L}}$ and $\mathcal{R}_{\mathcal{S}}$ of $\mathcal{L}$ and $\mathcal{S}$. $\tau(r)$ is accepted by $A^{\#}_{P,T}$ (item 3), by the fact that $A^{\#}_{P,T}$ accepts the traces of a solution to the structure-abstraction problem $\mathsf{AbsStruct}(S_{P,T})$, which is a simulation of $S_{P,T}$ (see §2.3). $\tau(r)$ is a trace of the structure program S_{Π} (item 4) by the assumption that r is a violation of Π and the definition of S_{Π}—in particular, the fact that the structures and predicate transformers of S_{Π} are constructed from $\mathcal{R}_{\mathcal{L}}$ and $\mathcal{R}_{\mathcal{S}}$, which are valid structure simulations of $\mathcal{L}$ and $\mathcal{S}$. $\tau(r)$ is thus accepted by $A^{\#}_{\Pi}$ (item 5), by the fact that $A^{\#}_{\Pi}$ accepts all traces of a solution to the structure-abstraction problem $\mathsf{AbsStruct}(S_\Pi)$, which is a simulation of S_Π (see §2.3). Thus $\tau(r)$ is a winning Attacker play of the game $G^{\mathcal{L},\mathcal{S}}_{P,\Pi}$ (item 6), by the fact that $\mathcal{M}(r)$ is accepted by $A^{\#}_{P,T}$ and $A^{\#}_{\Pi}$.

However, each trace of a run of P' is not a winning play of $G^{\mathcal{L},\mathcal{S}}_{P,\Pi}$, by the fact that $P' = \texttt{InstrumentationGen}_{\mathcal{L}}(P, D)$, where $\texttt{InstrumentationGen}_{\mathcal{L}}$ is a valid instrumentation generator for $\mathcal{L}$ and D is a winning Defender strategy for $G^{\mathcal{L},\mathcal{S}}_{P,\Pi}$. Thus, the existence of a run r of P' that violates Π implies a contradiction, and there can be no such run. □

As a result, `capweave` and `hiweave` are sound policy weavers for their respective languages and systems.

Corollary 12.3. *For each* `capcore` *program* P *and* `cap` *policy* $\Pi \in \mathsf{CapPols}$*, if* `capweave`$(P, \Pi) = P'$*, then* P' *is a solution of* $\mathsf{CAP}(P, \Pi)$.

Proof. (Sketch) `capweave` is an instance of WeaverGen applied to a language model for `capcore`, described in Ex. 15, and a system model for `cap`, described in Ex. 17. The correctness of `capweave` follows from Thm. 12.2. □

Corollary 12.4. *For each* `difccore` *program* P *and* `difc` *policy* $\Pi \in \mathsf{DIFCPols}$*, if* `hiweave`$(P, \Pi) = P'$*, then* P' *is a solution of* $\mathsf{LABEL}(P, \Pi)$.

Proof. (Sketch) `hiweave` is an instance of WeaverGen applied to a language model for `difccore`, described in Ex. 15, and a system model for `difc`, described in Ex. 17. The correctness of `hiweave` follows from Thm. 12.2. □

Part IV

Conclusion

In this chapter, we conclude by comparing the work presented in this dissertation to related work (Chapter 13), and describing potential future work (Chapter 14).

13 Related Work

Capability systems Capabilities were introduced in the MULTICS system [48], and were developed further in the capability systems PSOS [43] and EROS [49]. They provide capabilities as a fine-grained mechanism that mediate each access that an application requests to perform on a system resource, including loading and storing memory pages. The Capsicum operating system [58] provides capabilities that mediate accesses at a coarser granularity than the capabilities of PSOS or EROS: Capsicum capabilities only mediate accesses to file descriptors. However, because Capsicum capabilities only mediate accesses to file descriptors, it was possible for Capsicum to be rapidly developed as an extension to FreeBSD9, a widely-deployed version of UNIX. The work described in this dissertation describes the design and evaluation of a program weaver that automatically instruments programs to use capabilities on Capsicum. We suspect that the instrumentation techniques described in this dissertation could be reapplied to generate program weavers for other capability systems, such as EROS or PSOS, and that the utility of such weavers may in fact be greater for such systems than for Capsicum, given that such systems require applications to use capabilities at a finer granularity.

Programming for capability systems Instrumenting programs for Capsicum encompasses both partitioning a program into modules that execute in separate processes, and instrumenting the program modules that execute in each process to correctly invoke primitives that manage capabilities.

Previous work [16, 17] automatically partitions programs so that high and low confidentiality data are processed by separate processes, or on separate hosts. The SOAP project [27] proposes a semi-automatic technique in which a programmer annotates a program with a hypothetical sandbox, and a program analysis validates that the sandbox does not introduce unexpected program behavior. In contrast, `capweave` automatically instruments a program to invoke system calls that cause the program to execute in different processes (if necessary), and instruments the program executing in each process to use capabilities as necessary to satisfy a security policy.

Skalka and Smith [50] present an algorithm that takes a Java program instrumented with capability security checks, and attempts to show statically that some checks are always satisfied. Our work introduces a technique for instrumenting a program to use capability primitives so that it interacts securely with program modules that are not trusted to execute capability checks, either because the untrusted modules may contain vulnerabilities that can be exploited to violate control-flow integrity, or the modules are provided by an untrusted source.

DIFC systems Information-flow operating systems, such as Asbestos [22], HiStar [61], and Flume [36], explicitly track the flow of information between system objects, such as processes and files. Such systems are designed so that a program can, in principle, implement desired functionality when interacting with cooperative programs, but satisfy strong information-flow properties when interacting with an adversarial environment. In practice, a programmer must (1) write a program to use custom low-level instructions that operate on a persistent information-flow lattice, and (2) informally reason that the rewritten program satisfies desired functionality and information-flow guarantees. Our technique complements information-flow operating systems: we have described a program weaver that takes from a programmer explicit functionality and security policies, and instruments a program to invoke label operations so that it satisfies the policies.

Programming for DIFC systems Prior work on labeling programs for the Flume operating system takes an uninstrumented program and a policy as a conjunction of flow relations and negations of flow relations between threads, and instruments the program to satisfy the policy. Prior work performed by Efstathopoulos and Kohler uses a syntax-directed technique to generate code that initializes the labels of each thread to satisfy such a policy [21]. Preliminary work [30, 31] performed by myself and collaborators used techniques that either (1) verify that a program instrumented with label operations satisfies a given policy using a model-checking algorithm or (2) choose labels that a program can hold at each of its control locations throughout an execution to satisfy such a policy. The technique for choosing program labels allocated a fixed set of label variables to be used by the instrumented program and reduced the problem of determining the labels that should be stored in each label variable to solving a system of set constraints. Because the fragment of set constraints considered was NP-complete, the technique solved such constraints using a SAT-solver.

The technique described in this dissertation instruments programs to satisfy policies described in a policy language more expressive than the languages supported in previous work. In particular, the policy language described in this dissertation can express temporal properties over an execution trace in which each state is a set of labeled objects of unbounded size. Such features are necessary to describe policies of the programs to which we applied `hiweave`. The game-based weaver described in this dissertation relies critically on an engine that can soundly but finitely abstract transition systems over state spaces of unbounded size in order to weave programs to satisfy such policies. We believe that it would be extremely difficult to weave programs to satisfy such policies by extending our previous technique of solving a system of set constraints over a bounded set of variables.

An *inline reference monitor (IRM)* [24] is code inserted by an IRM rewriter into an untrusted program, which monitors the program's behavior at runtime and halts the program immediately before the program carries out an insecure execution. A technique described by Hamlen et al. [29] verifies that programs rewritten by an IRM rewriter are correct. An IRM mediates the operations of the program in which it is instrumented. Our program weavers address a different problem: to rewrite a program so that the program can ensure the security of its resources even after the program transfers control to untrusted, uninstrumented programs in its environment.

Information-flow languages Information-flow languages allow a programmer to ensure that sensitive information flows securely through data objects internal to a program's state. Such languages provide either a type-checker that statically analyzes all program executions [41], or a runtime system that dynamically monitors a single program execution [33]. Our program weaver for HiStar, `hiweave`, instruments a program written in an "ordinary" language to interact with HiStar to control how sensitive resources are accessed by other untrusted programs.

Recent work [6, 55] has proposed languages with value-dependent types that enable programmers to write secure distributed applications. Such languages reduce the problem of checking that a program correctly implements a cryptographic protocol to checking that the program has a type in a type system that includes value-dependent and affine types. `hiweave` enables programmers to write distributed applications that interact with a DIFC system to satisfy information-flow guarantees, which are different from cryptographic guarantees.

Partial synthesis Game-solving has been applied to synthesize finite-state controllers [3], and to "repair" programs to satisfy a specification given in linear-temporal logic (LTL) [35]. The work described in this dissertation applies games to instrument programs automatically for interactive-security systems. Program sketching, as a form of syntax-guided synthesis [4], takes a program with "holes" for program expressions and statements and a specification for a complete program, and synthesizes a complete program by choosing an expression or statement for each hole. Sketching has been applied to synthesize bit-streaming programs [51], finite programs [52, 53], and concurrent data structures [54]. Our technique may be viewed as a variation of sketching, in which expression holes are filled by labels, and statement holes are filled by label operations. Unlike previous applications of sketching, our weaver must maintain an abstract but accurate model of a relational store.

14 Future Work

In this chapter, we discuss current limitations of our technique, and discuss potential future work that could address such limitiations.

Verifying trustworthiness of trusted programs One limitation of our current technique is that policies in our language require a policy writer to completely trust particular program modules to have access to particular resources, but we do not provide techniques with which a policy writer can formally argue that a trusted program module (which, in practice, tends to be small) uses such resources in an acceptably restricted way. As one example, the Capsicum policy for `tcpdump` allows a module executing the DNS resolver (which consists of only 410 lines of code) to execute with ambient authority, but our technique does not provide an analysis that a policy writer can use to verify that the DNS resolver uses ambient authority only to access a network connection to perform DNS resolution (and not, e.g., to write to other files on a host system). As another example, the HiStar policy for `auth_log` allows the `logger` module (which consists of only 17 lines of code) to write to the log file, but does not provide an analysis that a policy writer can use to verify that `auth_log` only writes to the log file by appending a message. We believe that previous work on analyzing information flow within a given program [33, 41] or verifying *shim* programs that mediate all accesses to a sensitive resource [34] can complement our technique to address this limitation. In particular, such verification techniques seem applicable for verifying trusted program modules due to the fact that such modules tend to be small.

Providing diagnostics of failure For programs on any practical interactive-security system, including Capsicum and HiStar, there are many policies—some seemingly feasible—that the program cannot be instrumented to satisfy. Given a program and policy that the program cannot be instrumented to satisfy, the program weavers described in this work abort without providing to the user any information that explains why the weaver could not instrument the program successfully. However, when a weaver fails to find an instrumentation, it constructs a game describing executions of all considered instrumentations of a program, and constructs a winning *Attacker* strategy that, intuitively, explains why the weaver could not successfully instrument the program. In some cases, the winning Attacker strategy explains why the abstraction of the program constructed by the weaver was too coarse to find a winning strategy; we discuss below how weavers can be extended to handle this case. In other cases, such a strategy gives a procedure that describes how the Attacker can always violate the input policy, no matter what primitives are executed as Defender actions. We believe that such strategies can be analyzed to provide to the user a succinct explanation of why the weaver could not instrument an input program to satisfy a policy.

Refining abstractions for instrumentation From an uninstrumented program P and policy, each of the program weavers described in this work constructs a program P' that models all considered instrumentations of the uninstrumented program, and then constructs a fixed abstraction $\mathsf{P}'^{\#}$ of P'. If $\mathsf{P}'^{\#}$ is too coarse an abstraction of P' to distinguish a sufficient set of executions of P' that do not violate the input policy from the set of executions of P' that violate the policy, then from $\mathsf{P}'^{\#}$, the weaver will construct a game with no winning Defender strategy, and will abort execution. We believe that the program weavers described in this work can be improved by iteratively refining $P'^{\#}$ abstractions of P' based on a failure to instrument a given abstraction of P'. In particular, we believe that our program weavers can be extended to attempt to instrument a program under a particular abstraction, and if the attempted instrumentation results in a game G with no winning Defender strategy, use a winning Attacker strategy A of G to refine $P'^{\#}$ so that it defines a game for which A is not a winning Attacker strategy. Such an extension could be founded on a technique that adapts *counterexample-guided abstraction refinement (CEGAR) [5, 18]* from its previous uses in checking that a program has a desired property to synthesizing a program that has a desired property.

Generating optimal instrumentations The program weavers described in this work only attempt to generate a program that is *correct* (i.e., secure and functional), but not necessarily *optimal*: the program may execute unnecessary primitives that degrade its runtime performance. For example, the instrumented version of `tar` produced by our weaver for Capsicum placed a call to a `fork` primitive within a frequently executed loop in `tar`, and as a result, significantly degraded the performance of `tar` compared to an alternative satisfying instrumentation that placed the call to `fork` outside of the loop. We believe that our program weavers can be extended to generate optimal code by reducing an instrumentation problem to solving a *mean-payoff game* [23], which is a turn-based two-player game in which each action is associated with a cost. The goal of one player is to *maximize* the cost of the play of the game, and the goal of the competing player is to *minimize* the cost of the play. An optimal strategy for the cost-minimizing player is a procedure that minimizes the worst-case *mean cost* (i.e., the total cost of a play divided by its length) of all plays. We believe that our weavers can be extended to reduce a given instrumentation problem to a mean-payoff game for which an optimal strategy for the cost-minimizing player of the game defines an optimal program instrumentation.

A Appendix

A.1 Non-interference policies

In this section, we describe how `hiweave` as described in Part II can be extended to instrument a program to satisfy a non-interference policy. In §A.1.1, we give an overview of the extension. In §A.1.2, we describe how we extend `difc` (§9.1.3) to a language `difc-ni` with state predicates that are used to approximate non-interference policies. In §A.1.3, we describe the relationship between properties of a single run `r` of a `difc-ni` program and properties of pairs of runs, one of which is `r`.

A.1.1 Overview

The key challenge in extending `difc` to capture properties relevant for non-interference is extending the state to capture information from a single run `r` that captures useful information about states in other runs, compared to the states of `r`. Existing static analyses accomplish an analogous task by analyzing all possible control paths through a program, and setting the label of a program to be the join of the labels over all paths [41]. Existing dynamic languages accomplish an analogous task during runtime monitoring by extending the state of a `difc` program with a *floating label* [26] that is set before the program chooses a control-flow branch, and enforcing that the label of the program respects a flow relation with the floating label. `difc-ni` does so by extending the `difc` semantics with a *taint* predicate, which stores a sound approximation of how information has flowed between objects during an execution. The design of `difc-ni` combines the approaches implemented by static and dynamic languages by maintaining a *taint* predicate, and requiring a program to choose *label bounds* that the label of memory must stay within over the course of any execution, no matter which control path is taken. The taint predicate approximates the actual possible differences in state between multiple runs; updates of the taint predicate are determined by the label bounds.

We now describe the key features of the extension from `difc` to `difc-ni`. For simplicity, we do not include a complete list of features, which include analogous extensions for declassifications, as well as the labels described below, and only present the key result concerning `difc-ni` (Lem. A.1) without proof.

A.1.2 Extensions to `difc` semantics

In this section, we describe `difc-ni`, an extension of the `difc` semantics that allows non-interference policies to be expressed as automata in which each alphabet symbol is a condition over an extension of a `difc` state. We then describe `difc-ni` as an extension of the `difc` space of stores (§A.1.2), space of operations (§A.1.2), and space of predicate transformers (§A.1.2). In §A.1.2, we describe how the predicates and their transformers are modeled in an extension of the class of structure transition systems used to model `difc` programs (§9.5.2).

Extensions of `difc` stores

The label stores of `difc-ni` are the label stores of `difc`, extended with the following components (the universe of objects O^* and space of labels $\mathcal{L}$ were defined in §9.1).

- A unary predicate Taint over objects.

- A *lower-bound label* $\mathsf{LowerBound} \in \mathcal{L}$ that stores a lower bound on the label of memory allowed over all executions of the module currently being executed by the program.
- An *upper-bound label* $\mathsf{UpperBound} \in \mathcal{L}$ that stores an upper bound on the allowed label of memory over all executions of the module currently being executed by the program.

We denote the space of states constructed from `difc-ni` stores as Q_{NI}.

Extensions of the `difc` operations

The operations of `difc-ni` are the operations of `difc`, extended with the following additional operations:

- `set_lower_bound (E)` sets the lower-bound label to the evaluation of the label expression `E` (see §9.1.2).
- `set_upper_bound (E)` sets the upper-bound label to the evaluation of the label expression `E` (see §9.1.2).

Extensions of `difc` operational semantics

Each of the components in a `difc-ni` state is updated by `difc` operations as follows:

- Each operation updates the Taint predicate non-deterministically, based on the flow relation over system objects. In particular, each operation updates Taint so that for each object o, Taint may hold for o under the following conditions: if o is memory, then o may be tainted if some tainted object flows to $\mathsf{UpperBound}$. Otherwise, o may be tainted if memory is tainted and $\mathsf{LowerBound}$ flows to o.
- When a `difc-ni` program begins executing a module, the program must immediately choose the value of $\mathsf{LowerBound}$ and $\mathsf{UpperBound}$

by invoking the operations `set_lower_bound` and `set_upper_bound`. After setting the bound labels, each program operation asserts that in a given pre-state, LowerBound flows to the label of memory, and the label of memory flows to UpperBound.

Note that both the state and operations unique to `difc-ni` are purely modeling artifacts that model approximate information about multiple runs in a single run. They do not correspond to additional runtime state, or effects on state.

Modeling a `difc-ni` program as a structure-transition system

Each `difc-ni` program can be modeled soundly as a structure program by extending the logical vocabulary V_d (§9.3.1) and extending the structure transformers for the extended logical vocabulary given in §9.5.2. The intuition behind the extension is that Taint is modeled as a unary predicate over objects. LowerBound and UpperBound are modeled analogously to the operation label. The formal definitions of the predicate updates for each operation are straightforward from their informal descriptions given in §A.1.2; we thus omit a full description.

A.1.3 Reasoning about pairs of traces

The key property of `difc-ni` is that each run `r` of a `difc-ni` program `P` soundly approximates information about all pairs of runs of `P` that execute the same sequence of modules. We consider only pairs of runs that execute the same sequence of modules because such pairs naturally generalize the condition on pairs of inputs that establish that an individual function satisfies non-interference. That is, determining if a single function fails to satisfy non-interference is equivalent to determining if there is any single fixed "low" input under the attacker's control that when paired with distinct "high" input values, causes sensitive channels to hold distinct output values [41]. For `difc-ni` programs, the sequence of modules executed is a "low" input in that it is under control of the attacker. Thus, one aspect of generalizing non-interference from the traditional setting, in which a function is executed once, to a reactive setting, in which modules are executed multiple times by an environment, is to consider only pairs of runs in which the sequence of executed modules is fixed.

Example 21. To obtain an intuition as to why it suffices to only consider pairs of executions with a fixed sequence of executed modules as potential violations of non-interference, consider `auth_log` as instrumented in §8.1, Fig. 8.1, which intuitively satisfies a non-interference policy—i.e., ensuring that the value stored in `LOG` is influenced only by the value stored in `MSG` and the previous value stored in `LOG`. However, two executions of `auth_log` that execute `logger` a different number of times can result in distinct values stored in `LOG`.

Because the Taint predicate is updated using labels that bound the actual labels over all possible executions of a module, the Taint predicate can be used to reason about pairs of runs that execute the same sequence of modules.

Lemma A.1. *Let* $\mathtt{r}, \mathtt{r}' \in \mathsf{Q}_{\mathsf{NI}}^*$ *be a pair of runs of a* `difc-ni` *program* P *that each execute the same sequence of modules* $M_0, \ldots, M_n \in \mathsf{MODSYMS}^*$*. For each object* o *and* $\mathsf{q} \in \mathtt{r}$ *and* $\mathsf{q}' \in \mathtt{r}'$ *the final states of* $\mathtt{r}$ *and* $\mathtt{r}'$*, respectively, if the value of* o *in* q *is unequal to the value of* o *in* q'*, then* $\mathsf{Taint}(\mathsf{o})$ *holds in* $\mathtt{q_i}$*.*

Previous approaches to DIFC analysis and monitoring use labels on the program counter as artifacts to prove [41] or enforce [26] that a program satisfies non-interference over pairs of traces. Analogously, `difc-ni` programs instrumented to satisfy policies expressed using the Taint predicate satisfy non-interference policies over pairs of traces.

Example 22. The instrumented version of auth_log satisfies the difc-ni policy given in §8.2, Fig. 8.4. As discussed in Ex. 5, there is thus no run of auth_log in which (i) log_init, with LOG untainted, enters logger with MSG untainted, and (ii) reaches a state in which LOG is tainted. By Lem. A.1, there is no pair of traces of auth_log in which the values stored in MSG are equal each time that the runs enter logger, and in which the runs reach states with unequal values stored in LOG.

Bibliography

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

[2] Rajeev Alur, Thomas A. Henzinger, Orna Kupferman, and Moshe Y. Vardi. Alternating refinement relations. In *CONCUR*, 1998.

[3] Rajeev Alur, Thomas A. Henzinger, and Orna Kupferman. Alternating-time temporal logic. *J. ACM*, 49(5), 2002.

[4] Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *FMCAD*, 2013.

[5] Thomas Ball and Sriram K. Rajamani. The SLAM project: debugging system software via static analysis. In *POPL*, 2002.

[6] Gilles Barthe, Cédric Fournet, Benjamin Grégoire, Pierre-Yves Strub, Nikhil Swamy, and Santiago Zanella Béguelin. Probabilistic relational verification for cryptographic implementations. In *POPL*, 2014.

[7] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. Wedge: Splitting applications into reduced-privilege compartments. In *NSDI*, 2008.

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

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

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

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

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

[13] CVE Editorial Board. Vulnerability note VU#381508. http://www.kb.cert.org/vuls/id/381508, July 2011.

[14] CVE Editorial Board. Vulnerability note VU#520827. http://www.kb.cert.org/vuls/id/520827, May 2012.

[15] CVE Editorial Board. CVE - CVE-2013-2021. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-2021, Feb 2014.

[16] David Brumley and Dawn Xiaodong Song. Privtrans: Automatically partitioning programs for privilege separation. In *USENIX Security Symposium*, 2004.

[17] Stephen Chong, Jed Liu, Andrew C. Myers, Xin Qi, K. Vikram, Lantian Zheng, and Xin Zheng. Secure web application via automatic partitioning. In *SOSP*, 2007.

[18] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50 (5), 2003.

[19] The MITRE Corporation. Cwe - 2011 cwe/sans top 25 most dangerous software errors, 2011. URL http://cwe.mitre.org/top25/.

[20] Dorothy E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5), 1976.

[21] Petros Efstathopoulos and Eddie Kohler. Manageable fine-grained information flow. In *EuroSys*, 2008.

[22] Petros Efstathopoulos, Maxwell N. Krohn, Steve Vandebogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazières, M. Frans Kaashoek, and Robert Morris. Labels and event processes in the Asbestos operating system. In *SOSP*, 2005.

[23] A. Ehrenfeucht and J. Mycielski. Positional strategies for mean payoff games. *International Journal of Game Theory*, 8(2), 1979.

[24] Úlfar Erlingsson and Fred B. Schneider. IRM enforcement of java stack inspection. In *IEEE Symposium on Security and Privacy*, 2000.

[25] freebsd9. FreeBSD 9.0-RELEASE announcement. `http://www.freebsd.org/releases/9.0R/announce.html`, January 2012.

[26] Daniel B. Giffin, Amit Levy, Deian Stefan, David Terei, David Mazières, John C. Mitchell, and Alejandro Russo. Hails: Protecting data privacy in untrusted web applications. In *OSDI*, 2012.

[27] Khilan Gudka, Robert N. M. Watson, Steven Hand, Ben Laurie, and Anil Madhavapeddy. Exploring compartmentalization hypothesis with SOAPP. In *AHANS 2012*, 2012.

[28] Yuri Gurevich. Sequential abstract-state machines capture sequential algorithms. *ACM Trans. Comput. Log.*, 1(1), 2000.

[29] Kevin W. Hamlen, Greg Morrisett, and Fred B. Schneider. Certified in-lined reference monitoring on .NET. In *PLAS*, 2006.

[30] William R. Harris, Nicholas A. Kidd, Sagar Chaki, Somesh Jha, and Thomas Reps. Verifying information flow control over unbounded processes. In *FM*, 2009.

[31] William R. Harris, Somesh Jha, and Thomas W. Reps. DIFC programs by automatic instrumentation. In *CCS*, 2010.

[32] William R. Harris, Somesh Jha, Thomas W. Reps, Jonathan Anderson, and Robert N. M. Watson. Declarative, temporal, and practical programming with capabilities. In *IEEE Symposium on Security and Privacy*, 2013.

[33] Cătălin Hriţcu, Michael Greenberg, Ben Karel, Benjamin C. Pierce, and Greg Morrisett. All your IFCException are belong to us. In *IEEE Symposium on Security and Privacy*, 2013.

[34] Dongseok Jang, Zachary Tatlock, and Sorin Lerner. Establishing browser security guarantees through formal shim verification. In *USENIX Security Symposium*, 2012.

[35] Barbara Jobstmann, Andreas Griesmayer, and Roderick Bloem. Program repair as a game. In *CAV*, 2005.

[36] Maxwell N. Krohn, Alexander Yip, Micah Z. Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. Information flow control for standard OS abstractions. In *SOSP*, 2007.

[37] Chris Lattner. `http://llvm.org/`, November 2011.

[38] Benjamin Livshits and Stephen Chong. Towards fully automatic placement of security sanitizers and declassifiers. In *POPL*, 2013.

[39] V. Benjamin Livshits, Aditya V. Nori, Sriram K. Rajamani, and Anindya Banerjee. Merlin: specification inference for explicit information flow problems. In *PLDI*, 2009.

[40] Roman Manevich. `http://www.cs.tau.ac.il/~tvla`, June 2011.

[41] Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *POPL*, 1999.

[42] George C. Necula. Translation validation for an optimizing compiler. In *PLDI*, 2000.

[43] P. G. Neumann, R. S. Boyer, L. Robinson, K. N. Levitt, R. S. Boyer, and A. R. Saxena. A provably secure operating system. Technical report, Stanford Research Institute, 1980.

[44] United States Department of Defense. Trusted computer system evaluation criteria. *DoD Standard 5200.28-STD*, Dec 1985.

[45] Amir Pnueli, Michael Siegel, and Eli Singerman. Translation validation. In *TACAS*, 1998.

[46] Indrajit Roy, Donald E. Porter, Michael D. Bond, Kathryn S. McKinley, and Emmett Witchel. Laminar: practical fine-grained decentralized information flow control. In *PLDI*, 2009.

[47] Shmuel Sagiv, Thomas W. Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.*, 24(3), 2002.

[48] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

[49] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. EROS: a fast capability system. In *SOSP*, pages 170–185, 1999.

[50] Christian Skalka and Scott F. Smith. Static enforcement of security with types. In *ICFP*, pages 34–45, 2000.

[51] Armando Solar-Lezama, Rodric M. Rabbah, Rastislav Bodík, and Kemal Ebcioglu. Programming by sketching for bit-streaming programs. In *PLDI*, 2005.

[52] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. Combinatorial sketching for finite programs. In *ASPLOS*, 2006.

[53] Armando Solar-Lezama, Gilad Arnold, Liviu Tancau, Rastislav Bodík, Vijay A. Saraswat, and Sanjit A. Seshia. Sketching stencils. In *PLDI*, 2007.

[54] Armando Solar-Lezama, Christopher Grant Jones, and Rastislav Bodík. Sketching concurrent data structures. In *PLDI*, 2008.

[55] Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. Secure distributed programming with value-dependent types. In *ICFP*, 2011.

[56] Ming-Hsien Tsai, Yih-Kuen Tsay, and Yu-Shiang Hwang. GOAL for games, omega-automata, and logics. In *CAV*, 2013.

[57] Jeffrey A. Vaughan and Stephen Chong. Inference of expressive declassification policies. In *IEEE Symposium on Security and Privacy*, 2011.

[58] Robert N. M. Watson, Jonathan Anderson, Ben Laurie, and Kris Kennaway. Capsicum: Practical capabilities for unix. In *USENIX Security Symposium*, 2010.

[59] Chris Wright, Crispin Cowan, Stephen Smalley, James Morris, and Greg Kroah-Hartman. Linux security modules: General security support for the linux kernel. In *USENIX Security Symposium*, 2002.

[60] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. Secure program partitioning. *ACM Trans. Comput. Syst.*, 20 (3), 2002.

[61] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in HiStar. In *OSDI*, 2006.