Title

Program synthesis for interactive-security systems

Permalink

https://escholarship.org/uc/item/8h27h8sb

Journal

Formal Methods in System Design, 51(2)

ISSN

0925-9856

Authors

Harris, William R
Jha, Somesh
Reps, Thomas W
et al.

Publication Date

2017-11-01

DOI

10.1007/s10703-017-0296-5

Peer reviewed

Form Methods Syst Des (2017) 51:362–394
DOI 10.1007/s10703-017-0296-5

Program synthesis for interactive-security systems

William R. Harris[1] · **Somesh Jha**[2] ·
Thomas W. Reps[2,3] · **Sanjit A. Seshia**[4]

Published online: 16 September 2017

Abstract Developing practical but secure programs remains an important and open problem. Recently, the operating-system and architecture communities have proposed novel systems, which we refer to as *interactive-security systems*. They provide primitives that a program can use to perform security-critical operations, such as reading from and writing to system storage by restricting some modules to execute with limited privileges. Developing programs that use the low-level primitives provided by such systems to correctly ensure end-to-end security guarantees while preserving intended functionality is a challenging problem. This paper describes previous and proposed work on techniques and tools that enable a programmer to generate programs automatically that use such primitives. For two interactive security systems, namely the Capsicum capability system and the HiStar information-flow system, we developed languages of policies that a programmer can use to directly express security and functionality requirements, along with synthesizers that take a program and policy in the language and generate a program that correctly uses system primitives to satisfy the policy. We propose future work on developing a similar synthesizer for novel architectures that enable an application to execute different modules in *Secure Isolated Regions* without trusting any other software components on a platform, including the operating system.

Keywords Computer security · Program synthesis · Information flow · Capabilities · Secure isolated regions

✉ William R. Harris
wharris@cc.gatech.edu

[1] Georgia Institute of Technology, Atlanta, GA, USA

[2] University of Wisconsin–Madison, Madison, WI, USA

[3] GrammaTech Inc., Ithaca, NY, USA

[4] University of California, Berkeley, Berkeley, CA, USA

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 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 [62], 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 [15,16], and in file utilities and language interpreters that are often deployed to process untrusted data or execute untrusted programs [9,10,17,66,67].

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* [64] and SELinux [69] 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 [42,59], or (ii) monitor each program execution at runtime [23,34] to determine that the monitored execution does not violate security. An *Inline Reference Monitor* (IRM) [21] 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 [7,20,37,68,71] 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.

The goals of this paper are threefold. The first goal is to review previous work that we have performed on developing automatic program instrumenters and synthesizers for interactive-security operating systems, namely the Capsicum capability system [68] and the HiStar Decentralized Information Flow system [71]; the synthesizer for HiStar has not been described in previous work. The second goal is to review previous work that we have performed on verifying programs that execute on interactive-security architectures that provide Secure Isolated Regions [5,35], and to propose future work that will result in automatic program rewriters and instrumenters for such architectures. The third goal is to relate both our previous work and the proposed work to work performed by Veith et. al. [33].

Programming with capabilities on Capsicum. A primary goal of this paper is to review previous work that we have performed on developing techniques and tools to make it easier to program such systems. One example of an interactive-security system on which applications can enforce strong security guarantees is the capability operating system Capsicum [68] (starting with FreeBSD 9) [22]. 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 guarantee by inserting only a few calls to Capsicum primitives [68]. 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 [68]. Often, as in the case of `tcpdump`, the difficulty results from satisfying the conflicting demands of ensuring—using a low-level set of system primitives—that a module that provides capabilities to its environment that are (i) sufficient for a benign environment to perform desired functionality but (ii) insufficient for a malicious environment to violate desired security properties.

In previous work [30], we addressed the challenge of developing applications for Capsicum. To do so, we designed a language 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 the policy language, we created a program instrumenter that takes from the programmer (i) a program that invokes no capability primitives, and (ii) a security policy for the program, and automatically instruments the program to execute security primitives so that the resulting program 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*.

Programming with information-flow labels on HiStar. 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 [18,20,37,45,71]. 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 (i) to perform desired functionality when interacting with a cooperative environment, but (ii) to protect the secrecy and integrity of its information when interacting with a malicious environment. Previous research [39,40,65] 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 little previous work on writing programs that maintain labels on a DIFC system to preserve the security of information shared with untrusted programs [45], and such approaches typically require the programmer to reason directly about the intricate semantics of label operations.

In this paper, we review a framework that we proposed for modular instrumentation that enables a programmer to scalably instrument multiple mutually-untrusting modules to use DIFC labels (in particular, the labels provided by HiStar) so that they satisfy end-to-end security guarantees and functionality requirements. Our framework enables global security and functionality requirements to be decomposed soundly into local guarantees for individual program modules. Such guarantees are then discharged using verification and synthesis techniques developed in previous work [28,42].

Programming with secure isolated regions. The second primary goal of this paper is to highlight further opportunities for automatically synthesizing secure programs for emerging interactive-security systems. A key common feature of the interactive security systems considered in previous work is that each is implemented at the level of the *operating system*, which provides powerful primitives for isolating the memory spaces of distinct modules. However, modern computer users often wish to perform resource-intensive computations on their sensitive data on networked cloud servers, which may run operating systems that the user cannot vet or trust.

Novel architectures provide special instructions that an application invokes to contain segments of code and data in hardware-level *Secure Isolated Regions* (SIRs). In particular, the SGX enclave feature supported by recent Intel processors [35] and the TrustZone feature supported by many modern ARM processors [5] support an extended instruction-set architecture (ISA) that a user-level application can use to ensure confidentiality and integrity. Code that executes in a SIR cannot be tampered with by other agents on a system. Data in a SIR is encrypted when written to main memory, ensuring that even an adversary with complete control over hardware connected to the core processor cannot learn information about the application's data or tamper with the data without being detected.

While instructions used to create and maintain SIRs are powerful, they are difficult to use in practice. The difficulty stems from the fact that a programmer must rewrite their application to (i) use machine instructions that each protect individual memory regions, (ii) sanitize untrusted inputs read directly from its environment, and (iii) ensure that only an acceptable amount of information about sensitive information is directly released to its environment. In this paper, we describe previous work that we have performed on verifying that programs that use SIRs satisfy the above properties. We propose further work on *synthesizing* programs automatically that use SIRs to satisfy the above properties.

This paper thus describes problems in program instrumentation under three significantly different contexts: an operating system that provides capabilities, and operating system that provides information-flow labels, and architectures that provide SIRs. However, a common theme throughout the paper is that while each of the systems was designed to address a distinct problem and does so by providing a distinct set of mechanisms, each system can be viewed as a runtime environment that requires its applications to implement logic that interacts with it actively, rather than simply being monitored passively. Our previous work established that problems of instrumenting programs for capability and DIFC systems can be addressed as instances of a general approach that reduces the problems to instances of game-based synthesis. Adapting and extending such an approach to instrument programs to correctly use SIRs remains an open problem.

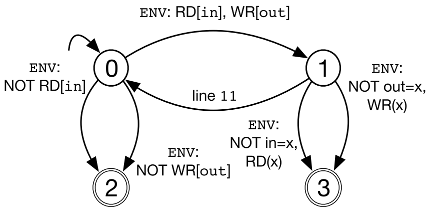
Organization. This paper is organized as follows. Section 2 summarizes our previous work on synthesizing programs that use capabilities. Section 3 presents our previous work on synthesizing mutually-untrusting programs that use DIFC labels. Section 4 proposes future work on synthesizing secure programs that use Secure Isolated Regions. Section 5 compares our previous work to related work on secure programming, including work performed by Veith et al.; Sect. 6 concludes.

2 Synthesizing programs for the capsicum capability system

In this section, we describe the problem of instrumenting programs for the Capsicum capability system, and our solution to the problem. In Sect. 2.1, we introduce a simplified version of the `gzip` compression utility. In Sect. 2.2, we present a simple policy for the capabilities that modules of `gzip` should provide to their environment. In Sect. 2.3, we describe how `capweave`, our program instrumenter for Capsicum, instruments `gzip` to satisfy its policy.

```
1  gzip:
2  // Create RPC service for loop.
3  C0: s0 :=
4    create_service(
5      loop, mem_amb(), mem_caps());
6  C1: set_mod_service(loop, s0);
7    jump loop;
8
9  /* Create input, output descriptors from
10  * the next filename. */
11 loop:
12   has_next := has_next_file();
13   br has_next EXIT;
14   in = open(IN, next_in_path());
15   out = open(OUT, next_out_path());
16   /* Create a service with which to
17    * execute the compression module. */
18 C2: s1 :=
19     create_service(cmp, no_amb(),
20       rem(in, WR,
21           rem(out, RD, mem_caps())));
22 C3: set_mod_service(cmp, s1);
23   jump cmp;
```

(a)



(b)

Fig. 1 Pseudocode for the gzip compression utility, and its security policy. **a** gzip: a compression utility consisting of two code segments, labeled gzip and loop. **b** gzip_pol: a capability policy for gzip, represented as an automaton that accepts disallowed traces of states. Each state is described by the set of capabilities that the program holds in the state

2.1 The gzip compression utility

Implementing efficient but secure programs that run on conventional operating systems has proven to be a significant challenge. Figure 1a contains pseudocode for a version of the gzip compression utility written in a simple low-level, intermediate imperative language. For now, ignore the lines with control labels beginning with C; these lines are instrumentation code introduced by our synthesizer, and are described below. When the compression module of gzip, cmp, executes correctly, it reads uncompressed data from a file descriptor bound to in, compresses the data read, writes the compressed data to a file descriptor bound to out, and then jumps to loop. However, in previous versions of gzip, cmp contained vulnerabilities that an attacker who could control the inputs to gzip could exploit to access arbitrary system resources with the privileges of the user who executed gzip.

2.2 A policy for gzip

Ideally, a programmer would implement a version of gzip that is provably free of such vulnerabilities; in practice, this goal has proven to be an intractable challenge because the compression module performs several, heavily-optimized operations that are difficult to prove preserve memory safety. However, the security guarantees of gzip would be strengthened significantly if it could be implemented so that (1) when loop jumps to cmp, the executing process should hold (a) the rd access right for the descriptor stored in variable in (i.e., it should hold the capability (in, rd)) and (b) the wr access right for the descriptor stored in variable out. (2) When gzip executes cmp, the executing process should only hold the rd access right for descriptors allocated at in and the wr access right for descriptors allocated at out. In previous work [30], we defined a language of *capability policies* as finite-state machines over the control locations and access rights of the states of each run of a program.

A policy for gzip that formalizes the security guarantee stated above is given as the automaton gzip_pol in Fig. 1b. gzip_pol defines a language of runs, defined to be sequences of states that constitute policy violations. I.e., each symbol in the alphabet is a projection of a program state to the capabilities held by the program. The alphabet is finite because each symbol is defined over state projected to only capabilities bound to local program variables. In Fig. 1b, sets of symbols are represented by properties that they satisfy.

gzip_pol accepts two classes of runs as violations. (1) Each run in which gzip transfers control to its environment without the capability to read from the descriptor bound to in or write to the descriptor bound to out corresponds to a run of gzip_pol that reaches accepting state 2. (2) Each run in which the environment of gzip reaches a state in which it can read from a descriptor that is not bound to in or write to a descriptor that is not bound to out corresponds to a run of gzip_pol that reaches accepting state 3.

2.3 Instrumenting **gzip**

The complete gzip in Fig. 1a, including the capability operations in lines with labels beginning with C, satisfies the capability policy gzip_pol, show in Fig. 1b. In lines 3–6, 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 18–22, 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 module cmp (represented in the policy automaton as the control location ENV, which denotes the program's environment). The result of executing the instrumented capability operations is that program memory can hold only the capabilities to read from the descriptor stored in variable in and write to the descriptor stored in 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. 1a with the capability operations removed), and the capability policy gzip_pol, and can automatically instrument gzip to execute the capability operations depicted in Fig. 1a. 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 $\texttt{gzip}'$ whose executions are the executions of multiple possible instrumentations of gzip; (2) construct a finite over-approximation $\texttt{gzip}'^{\#}$ of the language of executions of $\texttt{gzip}'$ that violate gzip_pol; (3) use $\texttt{gzip}'^{\#}$ to construct a safety game G for which each play models an execution of $\texttt{gzip}'$, and each Attacker-winning play models an execution of $\texttt{gzip}'^{\#}$ 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 play that models e.

Given program P, policy automaton A, and budget $k \in \mathbb{N}$ for the maximum number of security primitives that an instrumented program can execute consecutively, capweave constructs a game $G_{P,A,k} = (Q_0, Q_1, \Sigma_0, \Sigma_1, \tau_0, \tau_1, Q_F)$ in which each component is defined as follows. Each Attacker state (i.e., each element in Q_0) is a state of P abstracted to its control location and state of capabilities, paired with the current state of A. Each Defender state (i.e., each element in Q_1) is a tuple consisting of the current control location of P, the current state of A, and the number of remaining primitives that may be executed, set to k immediately after the program executes an instruction. Each Attacker action (i.e., each element in Σ_0) is an instruction in P, and each Defender action (i.e., each element in Σ_1) is a Capsicum primitive. The Attacker transition function τ_0 and Defender transition function τ_1 are defined by the effect of each program instruction and Capsicum primitive on capability state, along with the transition relation of A. The accepting states Q_F are all states constructed from an accepting state of A.

capweave thus requires information from its user in addition to a program and policy, namely a budget for the number of primitives that an instrumented program may executed consecutively. In practice, we have found that a user does not have to spend significant effort to choose a sufficient budget, because practical programs that can be instrumented to satisfy a policy typically can be instrumented to do so under a small budget. Extending capweave with an *autotuner* that chooses such a budget or proves its absence automatically is a promising direction for future work.

A fragment of the game constructed by capweave to weave gzip to satisfy gzip_pol is depicted in Fig. 2. Each game state consists of a pair of a gzip' state and a gzip_pol state, and is depicted in Fig. 2 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 a Capsicum operation on which the game transitions. Variations of the capability operation to create an RPC service at line 18 in Fig. 1 are modeled in Fig. 2 as sequences of capability operations chosen at control location 18a, followed by either control location 18a0 or 18a1.

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. 2, 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. 2 depicts four plays that start from a state (18a, 0), which models an execution at control location 18a that has driven gzip_pol to state 0. The game states starting from state (18a, 0) model states reached after gzip completes execution of line 15.

Along each play from (18a, 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 (18a, 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.

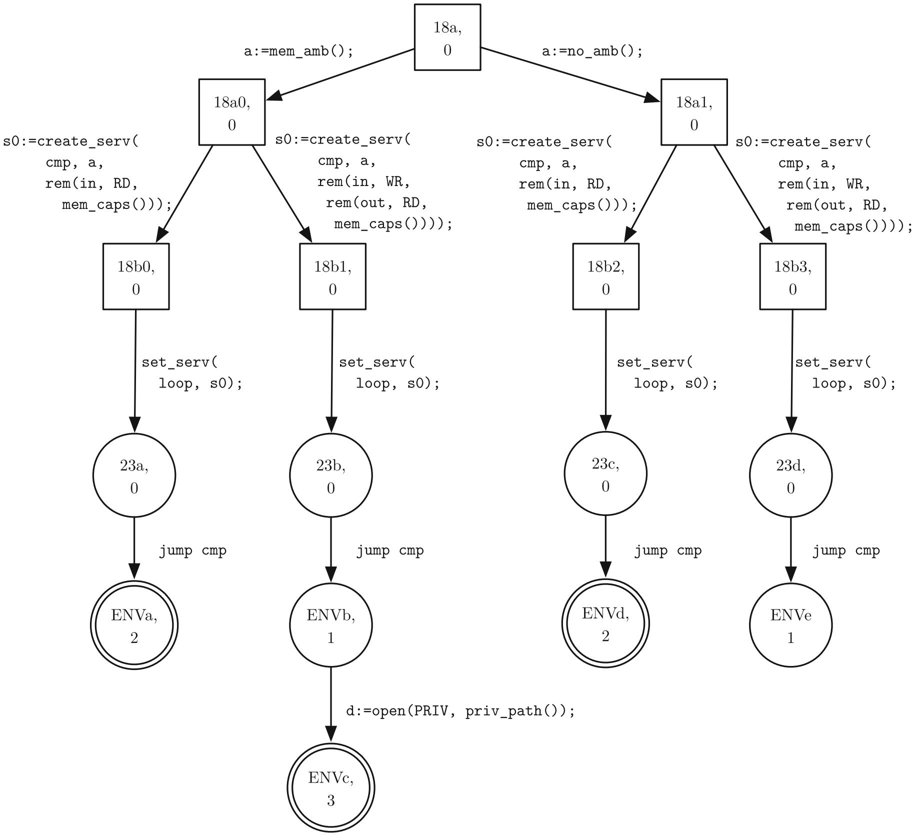


Fig. 2 Fragment of the game modeling runs of different possible instrumentations of `gzip` immediately before executing line 23. Defender states are depicted as squares, Attacker states are depicted as circles, and Attacker-winning states are depicted as doubled circles. Edges from Defender states are annotated with potential capability operations. Edges from Attacker states are annotated with instructions of the original program

On the play from $(\texttt{18a}, 0)$ to $(\mathsf{ENV}b, 1)$ 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`. $(\mathsf{ENV}b, 1)$ is not an Attacker-winning state, but the Attacker may transition from $(\mathsf{ENV}b, 1)$ to the state $(\mathsf{ENV}c, 3)$ by opening a new file descriptor with arbitrary access rights. $(\mathsf{ENV}c, 3)$ is an Attacker-winning state because it models a program state in which the program executes an untrusted module and memory holds a capability for a descriptor not bound to `in` or `out`, driving `gzip_pol` to the accepting state 3.

On the play from $(\texttt{18a}, 0)$ to $(\mathsf{ENV}d, 2)$, 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`. $\mathsf{ENV}d$ is an Attacker-winning state for a reason analogous to the reason that $(\mathsf{ENV}a, 2)$ is an Attacker-winning state.

On the play from $(\texttt{18a}, 0)$ to $(\mathsf{ENV}e, 1)$, 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`. $(\textsf{ENV}e, 1)$ is not an Attacker winning state, and the Attacker cannot choose any sequence of actions from $(\textsf{ENV}e, 1)$ that will drive the game to an Attacker-winning state. The trace of actions from (`18a`, 0) to $(\textsf{ENV}e, 1)$ is the trace of each execution of the instrumented `gzip` in Fig. 1 from line 18 to the jump to `cmp` at line 23.

Given program P, policy automaton A, and instrumentation budget k, `capweave` solves the game $G_{P,A,k}$ using a standard algorithm for solving two-player safety games [54]. The algorithm, given game G, computes the states of G from which an Attacker can always win, i.e., the set of *attractors* of G using an iterative reachability algorithm.

2.4 Key properties

The key correctness of property of `capweave` is that for each program P and policy automaton A, `capweave`, given P and A, generates program P', the P' satisfies A. One proof of correctness defines a game $G_{P,A}$ whose plays won by an attacker are exactly the runs of some instrumentation of P that violate A. The proof then establishes that for each k, each winning Defender strategy of $G_{P,A,k}$ (Sect. 2.3) is a winning strategy of $G_{P,A}$. In previous work [26], we gave precise definitions of language semantics, policy satisfaction, and the constructions of games used by `capweave` and in its proof of correctness in previous work.

We performed an experimental evaluation of `capweave` to determine if it could be used to instrument practical applications to satisfy policies that formalize practical security requirements. In particular, we formulated the requirements of the `gzip` and `bzip2` compression utilities, `tar` archiving utility, `tcpdump` and `wget` network utilities, and `php` language interpreter, and used `capweave` to instrument each to satisfy its policy when run on Capsicum. We found that `capweave` could consistently instrument the programs successfully at scale, although in some cases it generated programs with sub-optimal performance. A full description of our evaluation can be found in a previous presentation of our work [30].

3 Synthesizing programs for the HiStar DIFC system

3.1 Overview

This section illustrates, by means of an example, the DIFC instrumentation problem and our modular instrumenter, MODLIN. Section 3.1.1 reviews the design of a standard DIFC system. Section 3.1.2 introduces a DIFC program `user_info`, which we use as a running example, and its desired global flow policy, **Global Flow**. Section 3.1.3 describes how MODLIN instruments `user_info` to satisfy **Global Flow**.

3.1.1 Background: a generic DIFC system

We now describe a DIFC language `difc` as a simplification of the HiStar DIFC system [71]; a more detailed description of `difc` is given in Sect. 3.2.1, and a more detailed description of HiStar is given in Sect. 3.4.1. A program execution is an iterative process. In each iteration, the program's *environment* non-deterministically chooses a *module*, which executes atomically until completion.

The state of a `difc` program consists of memory and a set of objects, which model persistent storage and communication channels, and an environment; each is associated with

```
 1 cat_t c;
 2
 3 void add_ssn(int uid, int ssn) {
 4   /* If the SSN directory
 5    *  isn't initialized, */
 6   if (/SSNS == NULL) {
 7     // then create it.
 8   LBLS0: c = create_cat();
 9   LBLS1: set_op_label(c->HIGH);
10     create(/SSNS);
11   }
12   // Add a uid/ssn entry to the directory.
13   add_entry(/SSNS, uid, ssn);
14  LBLS2: LBLset_op_label(c->MID);
15   return;
16 }
```

(a)

```
 1 cat_t c;
 2
 3 int get_addr(int uid) {
 4   /* If the address directory
 5    * isn't initialized, */
 6   if (/ADDRS == NULL) {
 7     // then create it and populate it.
 8   LBLA0: c = create_cat();
 9   LBLA1: set_op_label(c->LOW);
10     create(/ADDRS);
11     populate(/ADDRS);
12   }
13   // Get a uid's entry in the address book.
14   int addr = get_entry(/ADDRS, uid);
15  LBLA2: set_op_label(c->MID);
16   return addr;
17 }
```

(b)

Fig. 3 `user_info`: a DIFC program that contains two modules, `add_ssn` and `get_addr` that enable users to maintain persistent key-value bindings storing persistent information. **a** `add_ssn`: takes a user ID (stored in `uid`) and social-security number (stored in `ssn`) and binds the user ID to the social-security number in a persistent dictionary. **b** `get_addr`: given a user's ID (stored in `uid`), returns the user's address, which is publicly observable, but should not be corruptible

a *label*. When any module of the program attempts to read data from an object into memory, the `difc` runtime only allows the read if the label of the object has the proper relationship to the label of memory. In particular, 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 Low $<$ Mid $<$ High. A label L_0 *flows to* label L_1, denoted by $L_0 \sqsubseteq L_1$, if each category c has a level in L_0 lower than or equal to its level in L_1. A program can write to or create a file f with a label L_f if the label of memory flows to L_f.

A program can create a fresh category, which its memory *owns* until the program returns control to its environment. When any module of the program returns control to its environment, it can choose for the runtime system to update the label of the environment to any label L such that the label of memory flows to L over all categories not owned by memory.

3.1.2 A simple information-management system

The module `add_ssn` (Fig. 3a) and module `get_addr` (Fig. 3b) can be used to store and load sensitive information to a key-value store, respectively. For now, ignore the lines with control labels: these are the instrumentation code introduced by our technique, and are described in Sect. 3.1.3. When `add_ssn` begins execution, it checks if there is an SSN-dictionary object bound to symbol `/SSNS` (line 6); if no such object is found, `add_ssn` binds `/SSNS` to a freshly-created object (line 10). In either case, `add_ssn` adds to the object at `/SSNS` a dictionary entry that binds the user identifier in `uid` to the SSN-value in `ssn` (line 13), and returns (line 15).

When `get_addr` begins execution, it checks if there is an address-directory object bound to symbol `/SSNS` (line 6); if no object is bound, then `get_addr` binds `/ADDRS` to a freshly-created object (line 10), and populates the dictionary with the public addresses of all users (line 11). In either case, `get_addr` loads the address of the user identifier stored in `uid` bound in the object at `/ADDRS` (line 14) and returns the result (line 16).

The problem that we address, in the context of `user_info`, is to instrument `user_info` to satisfy the following informal information-flow policy **Global Flow**. After `add_ssn` creates an object at /SSNS, no information may flow from the object at /SSNS to the public output channel-object at /PUB_OUT. After get_addr creates an object at /ADDRS, no information may flow from the public input channel-object at /PUB_IN to the object at /ADDRS.

We will discuss the problem of instrumenting user_info to satisfy **Global Flow** by invoking the label operations described in Sect. 3.1.1.

3.1.3 Instrumentation

A correct instrumentation of user_info. The complete version of add_ssn in Fig. 3a and get_addr in Fig. 3b, including label operations, satisfies **Global Flow**. The complete add_ssn (Fig. 3a), in addition to performing the operations on objects described in Sect. 3.1.2, performs the following label operations. Before add_ssn creates an object at /SSNS, it creates a fresh category that it stores in category variable c (LBL1S0) and sets c to be HIGH in the label used to create the fresh object (set in the *operation label* at LBLS1). When add_ssn returns control to its environment, it updates the label of memory so that c is set to MID (see LBLS2). Thus, no matter what operations on objects and labels the environment executes, it cannot read information from /SSNS.

The complete get_addr (Fig. 3b), in addition to performing the operations on objects described in Sect. 3.1.2, performs the following label operations. Before get_addr creates an object at /ADDRS, it creates a fresh category that it stores in category variable c (LBLA0) and sets c to be LOW in the label used to create the fresh object (set in the operation label at LBLA1). When get_addr returns control to its environment, it updates the label of memory so that c is set to MID (see LBLA2). Thus, no matter what operations on objects and labels the environment executes, it cannot write information to /ADDRS. Thus the instrumentation of add_ssn and get_addr satisfies **Global Flow**.

Challenges to instrumenting user_info. There are two significant challenges to instrumenting practical DIFC programs, which are illustrated by the problem of instrumenting user_info. The first challenge is scalability: an approach that monolithically instruments add_ssn and get_addr must reason about all possible labels that may be used to create the objects /SSNS and /ADDRS, as well as the possible labels of memory when either add_ssn and get_addr return control to their environment. While such an approach may feasibly instrument the toy program user_info, it will fail to instrument more than a few modules of a program for a practical DIFC system (see Sect. 3.4).

The second challenge is expressiveness: the labels of a DIFC system only provide guarantees about how information may flow between system objects and program memory. System labels provide coarse granularity in the treatment of memory, in that all of memory is associated with a single label. In the case of get_addr, labels on memory and objects alone cannot be used to ensure the integrity of /ADDRS, because get_addr executes with a label that allows it to write to /ADDRS, and loads data from uid, which may have low integrity. The integrity of /ADDRS can only be ensured by (1) instrumenting add_ssn to use labels so that its environment cannot directly modify /SSNS and (2) analyzing get_addr to determine that while the environment can modify uid, no information flows from uid to /ADDRS. Correctly instrumenting add_ssn requires similar reasoning about both the labels created by add_ssn for /SSNS and how information flows through memory and objects during each execution of add_ssn.

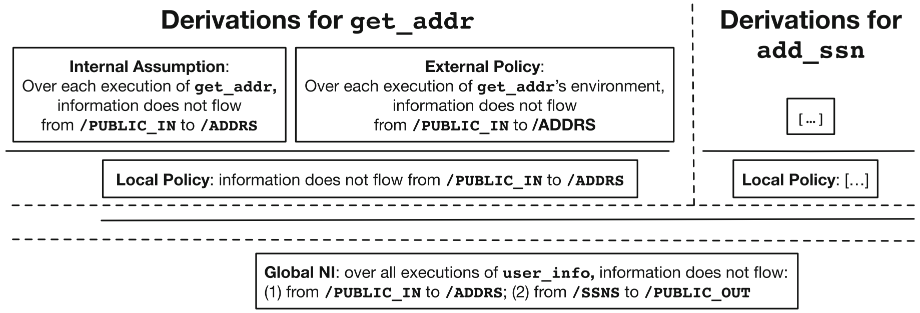

Derivations for get_addr
Internal Assumption: Over each execution of get_addr, information does not flow from /PUBLIC_IN to /ADDRS
External Policy: Over each execution of get_addr's environment, information does not flow from /PUBLIC_IN to /ADDRS
Local Policy: information does not flow from /PUBLIC_IN to /ADDRS
Derivations for add_ssn
[...]
Local Policy: [...]
Global NI: over all executions of user_info, information does not flow: (1) from /PUBLIC_IN to /ADDRS; (2) from /SSNS to /PUBLIC_OUT

Fig. 4 Fragment of a proof that instrumentations of add_ssn and get_addr satisfy **Global Flow**. If (i) add_ssn and get_addr satisfy given internal assumptions add_ssn-Internal and get_addr-Internal, and (ii) instrumentations of add_ssn and get_addr satisfy external policies add_ssn-External and get_addr-External constructed from add_ssn-Internal, get_addr-Internal, and given local policies add_ssn-Local and get_addr-Local, then the composition of instrumentations satisfies **Global Flow**

Compositional DIFC instrumentation. This paper proposes a DIFC instrumenter, Modlin, that can instrument practical programs for a DIFC system at scale under assumptions about individual modules that can be discharged by traditional program analyses used in information-flow languages [6,23,34,42,60,61]. The design of Modlin is based on two key observations about practical DIFC programs. The first observation is that, in practice, each individual module of a DIFC program has a natural *local* policy such that if each instrumentation of a module satisfies its local policy, then the set of all instrumented modules satisfies the *global* policy of the program. The second observation is that in practice, each uninstrumented module *M* of a DIFC program has a natural *internal assumption* on how information flows through *M*'s memory that can be discharged by analyzing only *M*.

Modlin uses local policies and internal assumptions provided by the programmer to instrument user_info by instrumenting each module of user_info independently. The local and internal policies for get_addr are stated informally in Fig. 4; the local and internal policies for add_ssn are similar. For get_addr, Modlin takes as input a local policy get_addr-Local and internal assumption get_addr-Internal, and for add_ssn, Modlin takes local policy get_addr-Local and internal assumption get_addr-Internal. Modlin first checks that if instrumentations of get_addr and add_ssn satisfy get_addr-Local and add_ssn-Local, respectively, then their composition satisfies **Global Flow**.

Modlin then uses get_addr-Local and get_addr-Internal to construct an *external policy* get_addr-External, which describes only how information should flow through objects when the environment of get_addr and add_ssn executes. If get_addr satisfies get_addr-Internal and an instrumentation get_addr′ of get_addr satisfies get_addr-External, then get_addr′ satisfies get_addr-Local. Modlin performs a similar construction for add_ssn, add_ssn-Local, and add_ssn-Internal.

Modlin then instruments get_addr to satisfy get_addr-External and instruments add_ssn to satisfy add_ssn-External by invoking a game-based instrumenter analogous to capweave (see Sect. 2.3); the game-based instrumenter for information flow is described in more detail in Sect. 3.4.2. The problems of deciding if get_addr and add_ssn satisfy get_addr-Internal and add_ssn-Internal, respectively, can be discharged by a conventional program analysis or type-checker for an information-flow language [23,42].

$$\begin{aligned}
\texttt{core} &:= (\text{MSYMS} : (\texttt{LOC} : \text{Op})^*)^* && (1)\\
\text{Op} &:= \text{DVAR} := \texttt{OP}(\overline{\text{DVAR}}) && (2)\\
&\mid \text{DVAR} \mathbin{?} \texttt{LOC} : \texttt{LOC} && (3)\\
&\mid \text{DVAR} := \texttt{rd}(\text{OSYMS}) && (4)\\
&\mid \texttt{wr}(\text{OSYMS}, \text{DVAR}) && (5)\\
&\mid \texttt{create}(\text{OSYMS}) && (6)\\
&\mid \texttt{ret} && (7)
\end{aligned}$$

(a)

$$\begin{aligned}
\text{Op} &:= \text{CVAR} := \texttt{create\_cat}() && (8)\\
&\mid \texttt{set\_op\_lbl}(\text{LExp}) && (9)\\
\text{LExp} &:= \texttt{empty\_label}() && (10)\\
&\mid \texttt{upd\_lv}(\text{LExp}, \text{CVAR}, \text{LVS}) && (11)
\end{aligned}$$

(b)

Fig. 5 Syntax of **a** `core`, **b** `difc` label operations

Figure 4 shows a fragment of the proof that if (i) MODLIN correctly instruments `add_ssn` and `get_addr` to satisfy the external policies `add_ssn`-External and `get_addr`-External, respectively, and (ii) if `add_ssn` and `get_addr` each satisfy `add_ssn`-Internal and `get_addr`-Internal, respectively, then the composition of instrumentations of `add_ssn` and `get_addr` satisfies **Global Flow**. The inference rules depicted in Fig. 4 are described in detail in Sect. 3.3.

3.2 The instrumentation problem

In this section, we describe the technical details of the DIFC instrumentation problem. Section 3.2.1 defines the syntax and semantics of a DIFC programming language `difc`. Section 3.2.2 defines a policy language for non-interference. We use both definitions to define the problem of instrumenting a `difc` program to satisfy a non-interference policy.

3.2.1 A language of DIFC programs

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

`core`. A `core` program loads values from objects into memory, computes operations on the loaded values, and writes the computed values to objects. The syntax of a `core` program is given in Fig. 5a, and is defined over fixed finite sets of module symbols (MSYMS, which contains a distinguished symbol ENV that models the program's environment), control locations (`LOC`), object symbols (OSYMS), and data variables (DVAR). A `core` program is a sequence of bindings, each from a module symbol to a sequence of operations, in which each operation is annotated with a control location. An operation may compute a value from values in data variables and store the result in a data variable (Eq. 2, where `OP` represents a set of standard arithmetic operations over integers), may cause control flow to branch based on the value in a data variable (Eq. 3), may read a value from an object to a data variable (Eq. 4), may write a value in a data variable to an object (Eq. 5), may create an object (Eq. 6), or may return control to its environment (Eq. 7). While `add_ssn` and `get_addr` as presented in Fig. 3a, b cannot be expressed directly in the syntax specified in Fig. 5a, the syntactic forms in Fig. 3a, b can be viewed as syntactic sugar for the forms in Fig. 5a.

A `core` state stores values in objects. Let O^* be an infinite universe of objects. A value *store* $\sigma = (D, O, T, \rho)$ is a tuple of (1) a valuation of data variables $D : \text{DVAR} \rightarrow \mathbb{Z}$, (2) a finite set of objects $O \subseteq O^*$ containing an object Mem that models the program memory, (3) a map $T : O \rightarrow \mathcal{P}(O)$ from each object to the set of objects whose information may taint it, (4) a partial map from object symbols to objects $\rho : \text{OSYMS} \hookrightarrow O$, and (5) a map from objects to data $\delta : O \to \mathbb{Z}$. The components of a value store σ are denoted D^σ, O^σ, T^σ, and ρ^σ; the space of `core` stores is denoted V. A `core` state (L, σ) is a control location $L \in \texttt{LOC}$ paired with a value store σ. The space of `core` states is denoted $Q_{\texttt{c}} = \texttt{LOC} \times V$.

A `core` program P defines a transition relation $\rightarrow_P \subseteq Q_{\texttt{c}} \times \text{Op} \times Q_{\texttt{c}}$ using a transition relation $\rightarrow_C \subseteq (V \times \text{Op}) \times V$ over `core` stores. An operation `create(o)` creates a fresh object and binds it to symbol `o`. If a program P executes the operation `ret`, then it transfers control to its *environment*. In the resulting post-state of the program, which we refer to as an *environment state*, the environment may execute any unbounded sequence of `difc` operations, or invoke a program module. A read from object o extends the objects that taint memory with the set of objects that taint o and a write extends the objects that taint o with the objects that taint memory. The `difc` semantics places no restriction on which objects are tainted initially: instead, policies define what propagations of taint define information-flow violations (see Sect. 3.2.2). The semantics of the other `core` operations is straightforward.

`difc` syntax. A `difc` program is a `core` program whose operations are the `core` operations extended with a set of label operations, given in Fig. 5b. The label operations are defined over the space of category variables CVAR and the space of levels LVS = {Low, Mid, High}. A label operation may create a fresh category (Eq. 8) or set the label to be used by the next `create` or `ret` operation to the value of a *label expression* (Eq. 9). A label expression is either an empty map (Eq. 10) or a label updated to bind a particular category to a particular level (Eq. 11). For `difc` programs P and P' containing disjoint sets of module symbols, the *composition* of P and P', denoted by $P \dot{\cup} P'$, is the `difc` program containing the modules of P and P'.

`difc` semantics. A `difc` state is a `core` state paired with a *label store*. Let C^* be a countably infinite set of *categories*. Let a *label* be a total function that maps each category to a level (i.e., the class of labels is $\mathcal{L} = C^* \to \text{LVS}$), and let a *declassification* be a set of categories (i.e., the class of declassifications is $\mathcal{D} = \mathcal{P}(C^*)$, where $\mathcal{P}(S)$ denotes the power-set of a set S). A label store $(C, \lambda, \kappa, L_o)$ is a tuple of (1) a finite set of categories $C \subseteq C^*$, (2) a *store label* $\lambda : O^* \to \mathcal{L}$, (3) a *memory declassification* $\kappa \in \mathcal{D}$, and (4) an *operation label* $L_o \in \mathcal{L}$ to be used by the next operation executed. We denote the space of label stores by $V_{\texttt{d}}$. For label store σ, we denote the categories, store label, memory declassification, and operation label of σ by C^σ, λ^σ, κ^σ, and L_o^σ, respectively. We refer to $\lambda^\sigma(\mathsf{Mem})$ as the *memory label* in σ. If a category c is in κ^σ, then we say that memory *owns* c. The space of `difc` states is denoted by $Q_{\texttt{d}} = Q_{\texttt{c}} \times V_{\texttt{d}}$.

The semantics of many `difc` operations are defined using a flows-to relation over labels, which defines when information may flow from one object to another.

Definition 1 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 for each category $c \in C$, $L_0(c) \leq L_1(c)$.

Typically, we will consider the flows-to relation over the universe of all categories C^*, and thus will write $\sqsubseteq$ in place of $\sqsubseteq_{C^*}$. We also often consider the flows-to relation over the set of categories not declassified by memory. For label store σ, we use $\sqsubseteq_\sigma$ to denote $\sqsubseteq_{C^* \setminus \kappa^\sigma}$.

A `difc` program P defines a transition relation $\rightarrow_P \subseteq Q_{\texttt{d}} \times \mathsf{Op} \times Q_{\texttt{d}}$. $\rightarrow_P$ is defined by the transition relation $\rightarrow_{\texttt{d}} \subseteq (V_{\texttt{d}} \times \mathsf{Op}) \times V_{\texttt{d}}$ that relates `difc` pre-stores, operations, and post-stores. For data operations and control branches, P updates its value store as defined by the semantics of `core`, and does not change its label store. The semantics of reads and

$$\text{read}\ \dfrac{\langle \sigma_V, \mathtt{x} := \mathtt{rd}(\mathtt{o}) \rangle \rightarrow_C \sigma'_V \qquad \lambda^{\sigma_L}(\rho^{\sigma_V}(\mathtt{o})) \sqsubseteq_{\sigma_L} \lambda^{\sigma_L}(\mathsf{Mem})}{\langle (\sigma_V, \sigma_L), \mathtt{x} := \mathtt{rd}(\mathtt{o}) \rangle \rightarrow_{\mathtt{d}} (\sigma'_V, \sigma_L)} \qquad \text{write}\ \dfrac{\langle \sigma_V, \mathtt{wr}(\mathtt{o}, \mathtt{x}) \rangle \rightarrow_C \sigma'_V \qquad \lambda^{\sigma_L}(\mathsf{Mem}) \sqsubseteq_{\sigma_L} \lambda^{\sigma_L}(\rho^{\sigma_V}(\mathtt{o}))}{\langle (\sigma_V, \sigma_L), \mathtt{wr}(\mathtt{o}, \mathtt{x}) \rangle \rightarrow_{\mathtt{d}} (\sigma'_V, \sigma_L)}$$

Fig. 6 Semantic inference rules for `difc`. Conditions on labels are highlighted with a gray background. The rules define a transition relation $\rightarrow_{\mathtt{d}} \subseteq (V_{\mathtt{d}} \times \mathsf{Op}) \times V_{\mathtt{d}}$ from a `difc` pre-store and operation to a post-store

writes are given as inference rules in Fig. 6. For an operation that reads data from an object o, P reads from o only if the label of o flows to the label of memory (Rule read); for an operation that writes data to an object o, P writes to o only if the label of memory flows to the label of o (Rule write). An operation `create(o)` updates the store by creating a fresh object, as described in Sect. 3.1. An operation `ret` returns control to its environment with a memory label equal to the operation label, and an empty declassification.

The label operation `c := create_cat()` creates a fresh category c, binds c to category variable `c`, and adds c to the declassification of memory, and the label operation `set_op_lbl`(LExp) sets the evaluation of LExp in the current label state to be the label used by the next `create` or `envret` operation.

For each `difc` program P, we denote the set of all finite traces of `difc` states generated by executions of P as $\mathcal{T}(P) \subseteq (Q_{\mathtt{d}})^*$.

Definition 2 For `core` program P and `difc` program P', P' is an *instrumentation* of P (denoted by $P \preceq P'$) if P is the `core` program produced by removing each label operation in P'.

In Definition 2, instrumentation is a *syntactic* relationship between programs. Such a relationship is overly restrictive to be satisfactory in practice, because while it allows an instrumentation to perform additional label operations, it does not allow an instrumentation to maintain additional state that it uses to determine what label operations to perform at each program point. However, Definition 2 suffices to explain our approach.

3.2.2 Policy language

A flow automaton defines a language of `difc` state traces that violate a desired policy. Each symbol in the alphabet of the automaton describes a state q by referring only to the control location of q and the set of all objects in q whose values have been tainted by the current execution.

Definition 3 A *flow automaton* is a finite-state automaton in which the alphabet is the space $\Sigma_{Flow} = \mathrm{LOC} \times (\mathrm{OSYMS} \rightarrow \mathcal{P}(\mathrm{OSYMS}))$. The class of flow automata is denoted by $\mathcal{A}_{\mathrm{Flow}}$.

A flow automaton N defines a language of `difc`-state traces in which the taint map in each state of the trace is the taint map in the corresponding symbol of a string accepted by N.

Definition 4 Let $t = \sigma_0, \ldots, \sigma_n$ be a $Q_{\mathtt{d}}$-trace, and let N be a flow automaton.

t violates N if N accepts some trace $t_N = a_0, \ldots, a_n \in \Sigma^*_{Flow}$ such that for each $0 \leq i \leq n$, control location $\mathtt{L}_i$, and map $T_i : \mathrm{OSYMS} \rightarrow \mathcal{P}(\mathrm{OSYMS})$, where $a_i = (\mathtt{L}_i, T_i)$, for each object symbol $s \in \mathrm{OSYMS}$, $T^{\sigma_i}(\rho^{\sigma_i}(s)) = \bigcup_{s' \in T_i(s)} \rho^{\sigma_i}(s')$ holds.

For flow automaton N and `difc` program P, if each trace $t \in \mathcal{T}(P)$ does not violate N, then P *satisfies* N (denoted by $P \models N$).

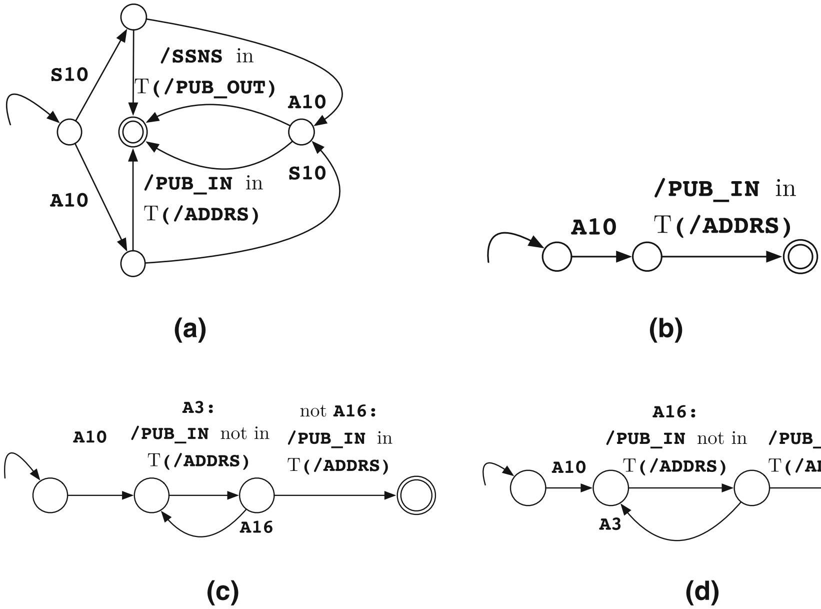


Fig. 7 Flow automata that represent global and local policies of modules in `user_info` For each state q and alphabet symbol a not shown on a transition from q, there is an implicit transition from q to itself on a. **a** Global flow policy for `user_info`. **b** Local flow policy for `add_ssn`. **c** Internal flow assumption for `add_ssn`. **d** External flow policy for `add_ssn`

Practical policies can typically be represented succinctly as upper bounds on acceptable taint maps in particular states. Policies given in such a representation can then be translated to flow automata as defined above.

Example 1 The global flow policy **Global Flow** can be expressed as a flow automaton G, depicted in Fig. 7a. In Fig. 7a, line i of `add_ssn` (Sect. 3.1.2, Fig. 3a) is depicted as Si and line i of `get_addr` (Fig. 3b) is depicted as Ai. Sets of transitions with the same source state q and destination state q' on symbols $\Sigma' \subseteq \Sigma_{Flow}$ are represented as a single arrow from q to q' annotated with a description of Σ'. For example, "`/PUB_IN` in T(`/ADDRS`)" depicts all policy symbols (L, T) $\in \Sigma_{Flow}$ in which `PUB_IN` is in T(`/ADDRS`). All transitions from a source state to itself are omitted.

A violation of G is a run of `user_info` in which either (1) `add_ssn` creates an SSN-dictionary object (bound to `/SSNS` at line `S10`) and the information from the object then flows to `/PUB_OUT` or (2) `get_addr` creates an address dictionary (bound to `/ADDRS` at line `A10`) and information from `/PUB_IN` then flows to the address dictionary.

The DIFC labeling problem is to take a `difc` program P and non-interference policy N, and instrument P to satisfy N.

Definition 5 Let P be a `difc` program and let $N \in \mathcal{A}_{\text{Flow}}$ be a non-interference automaton.

A solution to the *DIFC labeling problem* LABEL(P, N) is a `difc` program P' such that P' is an instrumentation of P (Definition 2) and P' satisfies N (Definition 4).

$$\text{Inst-Dist}\,\frac{P_0 \preceq P'_0 \qquad P_1 \preceq P'_1}{(P_0 \dot\cup P_1) \preceq (P'_0 \dot\cup P'_1)} \qquad \text{Flow-C-E}\,\frac{P_0 \models N_0 \qquad P_1 \models N_1}{P_0 \dot\cup P_1 \models N_0 \times_{\mathsf{Flow}} N_1}$$

$$\text{Flow-G-I}\,\frac{P \models N_0 \qquad P \preceq P' \qquad P' \models N_0 \xrightarrow{Flow} N_1}{P' \models N_1}$$

Fig. 8 Inference rules for compositional instrumentation

3.3 Modular DIFC instrumentation

We have developed a DIFC instrumentation algorithm, MODLIN, that instruments a program to satisfy a global policy by instrumenting each module of the program independently. The soundness of MODLIN is supported by inference rules that relate compositions of modules, the instrumentation relation (Definition 2), and satisfaction of flow policies (Sect. 3.2.2). Such rules are analogous to, and inspired by, inference rules that support the soundness of assume-guarantee reasoning [44] and modular verification [24] by relating compositions of transition-system modules to satisfaction of properties expressed as formulas in temporal logics. In Sect. 3.3.1, we present the inference rules in detail. In Sect. 3.3.2, we describe how MODLIN applies the inference rules to instrument an entire `difc` program by instrumenting the modules of the program independently.

3.3.1 Inference rules

Distribution of instrumentation over composition. The composition of valid instrumentations of two `core` programs P_0 and P_1 is a valid instrumentation of the composition of P_0 and P_1. I.e., for `core` programs P_0 and P_1 and `difc` programs P'_0 and P'_1, if P'_0 is a valid instrumentation of P_0 and P'_1 is a valid instrumentation of P_1, then $P'_0 \dot\cup P'_1$ is a valid instrumentation of $P_0 \dot\cup P_1$ (Fig. 8, Rule Inst-Dist).

The key idea that supports the correctness of Rule Inst-Dist is that each individual module is instrumented in isolation under the assumption that the instrumented versions of other program modules, combined with the environment, may attempt to perform arbitrary sequences of operations. Any instrumentation of the other program modules chosen by an instrumenter trivially satisfies this assumption.

Elimination of conjunctions of flow automata. The problem of instrumenting a composition of programs to satisfy multiple flow policies can be decomposed across modules and policies. For each pair of flow automata $N_0, N_1 \in \mathcal{A}_{\text{Flow}}$, the *conjunction* of N_0 and N_1, denoted by $N_0 \times_{\mathsf{Flow}} N_1$ is the flow automaton that accepts each trace accepted by N_0 *or* N_1 (thus, program P satisfies $N_0 \times_{\mathsf{Flow}} N_1$ only if P satisfies both N_0 *and* N_1). For `difc` programs $P_0, P_1 \in$ `difc` and flow automata $N_0, N_1 \in \mathcal{A}_{\text{Flow}}$, if P_0 satisfies N_0 and P_1 satisfies N_1, then the composition of P_0 and P_1 satisfies the conjunction of N_0 and N_1 (Fig. 8, Rule Flow-C-E, for **Flow** **C**onjunction **E**limination).

Example 2 MODLIN applies Rule Flow-C-E to decompose the problem of instrumenting `user_info` to satisfy the global policy **Global Flow** into the independent problems of instrumenting `get_addr` to satisfy `get_addr`-Local (depicted in Sect. 3.2.2, Fig. 7b) and instrumenting `add_ssn` to satisfy `add_ssn`-Local (not depicted in Fig. 7). This strategy is beneficial because the conjunction of `add_ssn`-Local and `get_addr`-Local entails **Global Flow**.

Introduction of guarded flow automata Instrumenting a module M to satisfy a flow policy N can be decomposed to (1) proving that M satisfies a flow policy N_0 and (2) instrumenting M so that it performs no violation of N that is not also a violation of N_0. Let N_1 *guarded by* N_0, denoted by $N_0 \xrightarrow{Flow} N_1$, be the flow automaton that accepts each trace not accepted by N_0, or accepted by N_1. For `difc` program P and flow policies $N_0, N_1 \in \mathcal{A}_{\mathrm{Flow}}$, if (1) P satisfies N_0, (2) P' is a valid instrumentation of P, and (3) P' satisfies $N_0 \xrightarrow{Flow} N_1$, then P' satisfies N_1 (Fig. 8, Rule Flow-G-I for **Flow Guard Introduction**).

An internal flow policy for a program module M is conventionally defined by a set of source objects I and sink objects O [42]. An internal flow policy (I, O) can be represented as a flow automaton; the construction is straightforward, and we only illustrate it by example.

Example 3 Modlin can apply Rule Flow-C-E to decompose the problem of instrumenting `get_addr` to satisfy local policy `get_addr`-Local (Example 2) into instrumenting `get_addr` to satisfy the external policy `get_addr`-External (Sect. 3.2.2, Fig. 7d) under the assumption that each execution of the uninstrumented `get_addr` satisfies `get_addr`-Internal (Fig. 7c). Note that `get_addr`-Internal only places a condition on subtraces of `user_info` within `get_addr`—i.e., traces that occur after a state at control location `A0` and up to a state at control location `A16`.

3.3.2 Compositional instrumentation

Modlin uses the above rules to instrument a program to satisfy an input flow policy. Modlin takes as input (1) an uninstrumented program P, (2) a global flow policy G, and for each module $M \in P$, a local flow policy L_M and an internal flow policy I_M. Modlin first checks that the conjunction of policies $\prod_{M \in P} L_M$ entails policy G by performing a standard language-containment check. Modlin then applies a DIFC instrumenter Monolith, developed as an extension of techniques presented in previous work [26] (and described in detail in Sect. 3.4.2), to each module $M \in P$ to find an instrumentation M' of M such that M' satisfies the external policy $I_M \xrightarrow{Flow} L_M$. If Monolith finds an instrumentation for each module, then Modlin returns the composition $\dot{\cup}_{M \in P} M'$ as an instrumentation of P that satisfies G. For each module M, the internal assumption I_M can be discharged with a standard program analysis for an information-flow language [6,23,34,42,60,61].

The key property satisfied by Modlin is that for each program P and flow policy N, if Modlin, given P and N, generates program P', then P' satisfies N. One proof that Modlin satisfies such a property proceeds in two steps. The first step establishes that for each program Q and M, if Monolith, given Q and M generates a program Q', then Q' satisfies M. We provided a proof of this step in previous work [26]. The second step establishes that each of the inference rules given in Sect. 3.3.1 is sound. We provided proofs of soundness in a manuscript not previously published, which describes this work in greater detail [27].

3.4 Evaluation

We performed an empirical evaluation of Modlin in order the answer the following questions. First, can Modlin instrument a program to satisfy a global flow policy by instrumenting each program module to satisfy a local flow policy? Second, can Modlin use internal assumptions to decompose the problem of instrumenting a module to satisfy a local flow policy into a problem of instrumenting the module to satisfy a weaker external flow policy? Third, do programs instrumented by Modlin perform comparably with programs instrumented by hand? To answer the above questions, we implemented MODLIN as a tool, `modstar`, that performs a source-to-source translation on the LLVM intermediate language [38] to instrument programs to be run on the HiStar DIFC system. `modstar` uses a novel monolithic DIFC instrumenter, `monostar` (discussed in Sect. 3.4.2), to instrument each individual module to satisfy its local external policy.

To determine if `modstar` could decompose global flow policies into local flow policies, we wrote a desired global flow policy for a suite of four modules that, combined, implement a mutually-untrusting login service [71]. For each module, we wrote a local flow policy that we believed a correctly-instrumented version of the module should satisfy; we found that the conjunction of all local flow policies served as a natural global flow policy for the entire login service. We also wrote an internal flow assumption that we believed that each uninstrumented module satisfied; We then applied `modstar` to (1) check that conjunction of local policies entailed the global policy and (2) instrument each module to satisfy its local policy, assuming that the uninstrumented version of the program satisfies its internal flow assumption. To evaluate the benefit of `modstar`'s modular approach, we applied `monostar` to attempt to instrument monolithically all modules to satisfy the conjunction of external flow policies.

In short, `modstar` successfully instrumented the program to satisfy its global flow policy by instrumenting each module in isolation, whereas `monostar` exhausted all system resources when attempting to instrument all modules. A more nuanced view, discussed in detail in Sect. 3.4.5, is that our results answer the above experimental questions affirmatively, with some reservations.

We now describe our evaluation in more detail. Section 3.4.1 describes the semantics of the HiStar operating system, an operating system that implements DIFC features that extend the features of `difc` from Sect. 3.1.1. Section 3.4.2 describes how a monolithic instrumenter soundly and accurately instruments programs to use HiStar label primitives. Section 3.4.3 describes in more detail the policies of each module in the `login` service, and the instrumented version of each module that `modstar` generates. Section 3.4.4 discusses the results of the evaluation. Section 3.4.5 draws conclusions from the results.

3.4.1 The HiStar DIFC system

The HiStar DIFC operating system [71] maps each system object, i.e., each process, file, or directory, to a label. Similarly to `difc` (Sect. 3.1), a label is a map from a space of categories to an ordered level, and each process p can read from or write to a system object o depending on the label of p and o.

A process can create a *gate*, which is labeled storage bound to a module that another process can call to execute a fixed operation with temporarily-elevated privilege. Each process p and gate g has *ownership* of a set of categories, O_p and O_g, respectively. A process p can create a gate g with a label L_g and ownership set O_g if the label of p flows to L_g and the ownership set of p contains O_g. A process q can call g with a label L' that may have an arbitrary level at every category owned by q *or* g, and has a level as high as the levels of both L_q *and* L_g at every category not owned by q or g.

HiStar associates each process and gate with a *verification set* of categories. A process p may create a gate g with a verification set that is a subset of the verification set of p. When a process attempts to call a gate g, HiStar only allows the gate call to proceed if the verification set of the calling process contains the verification set of g.

3.4.2 A monolithic instrumenter for HiStar

Our implementation of MODLIN for HiStar (`modstar`) invokes a monolithic instrumenter `monostar` to instrument individual HiStar modules independently (see Sect. 3.3.2). There are several challenges to developing a monolithic instrumenter to instrument even individual modules for a practical DIFC system, such as HiStar, compared to the illustrative language `difc` (Sect. 3.2.1). A primary challenge is that `difc` programs operate on a set of objects that are bound to a fixed set of object symbols, but HiStar programs operate on a potentially-unbounded set of linked labeled objects that reside on a filesystem. Modeling the semantics of such programs is beyond the scope of existing DIFC instrumenters [20,28].

We thus developed a novel monolithic DIFC instrumenter [26], `monostar`, that applies a *logic-analysis engine* [41] to model the set of system objects, links between objects, categories, and the level of each object at each category as a *relational structure*, and soundly abstracts the potentially-unbounded set of states reached by a program as a *three-valued logical structure* [47]. Rather than use sets of object symbols, as described in the definition of policy automata in Sect. 3.2.2, `monostar` describes potentially-unbounded sets of objects as formulas in first-order logic with transitive closure (FOLTC).

3.4.3 A mutually-untrusting login service

The HiStar mutually-untrusting login service [71] allows a client to supply a username *u* and password-attempt value to request ownership of the private category for *u* (upriv), while controlling to which objects on the system the client's password-attempt may flow. The login service is implemented as four distinct modules: a logging service `auth_log`, a map `auth_map` from each user to their authentication gate, user-authenticator `auth`, and an authentication client `clnt`. Each login session is performed by calling multiple gates, which read from, write to, and create other objects, gates, and directories. The key goals of the login service are that (1) `auth` should not grant ownership of its user's private category until the client that it interacts with provides the user's password, (2) even when `clnt` interacts with an untrustworthy authenticator, the authenticator should not be able to leak the client's password-attempt value, but (3) `clnt` should still be able to use `auth_log` to log when a client requests ownership of a client's category.

A session of the login service in which the client obtains ownership of the user's private category proceeds as follows. The initial system state contains gates for `auth_log` and `auth_map`. Before a login session occurs, `auth_log` creates a log file `log` and creates a gate `logger` that its environment can invoke to append a message to the log [Fig. 9, arc (1)]. `auth_map` creates a gate `map` [arc (2)], which takes a user name *u* as input and returns the `auth` gate registered for *u*.

`clnt` initiates a login session by calling `map` to obtain the `auth` gate for a desired user [arc (3)]. `clnt` then calls the `auth` gate [arc (4)], and provides a *session directory* `seshdir` writable by `auth`. Because `seshdir` is only a directory, not a gate, it is not depicted in Fig. 9. In response, `auth` creates as children of `seshdir` a gate bound to a password-checking module `chk_pw` [arc (5)] and a gate bound to a permission-granting module `grant` [arc (6)] as children of `seshdir`. `clnt` then calls the `chk_pw` gate with the client's password-attempt value [arc (7)]. `chk_pw` checks whether the password-attempt value supplied by `clnt` is actually correct, but `clnt` ensures that `chk_pw` executes with only the capabilities to (a) receive the password-attempt value from `clnt`, and (b) after a successful match of the hashed password-attempt value with the stored hash of the actual password, returns an authorization token to `clnt`. `clnt` then provides the authentication

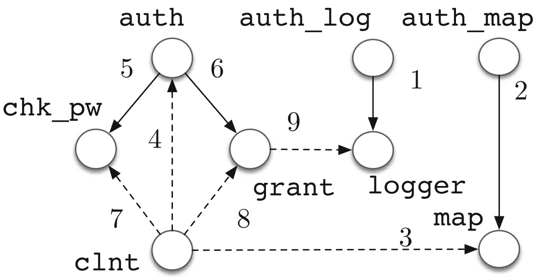


Fig. 9 Gates created during an authentication session of the mutually-untrusting login service. Each node denotes a gate; a solid edge $g \rightarrow h$ denotes that in a state in which the program executes gate g, the program creates gate h; a dashed edge $g \rightarrow h$ denotes that in a state in which the program executes gate g, the program calls gate h

token, *but not the password-attempt*, to grant [arc (8)], which grants clnt ownership of upriv, and uses logger to log the event to a public file [arc (9)], and clnt returns control to the program that provided to it a password-attempt value, owning upriv.

We now describe, for each module M of login, the local flow policy and internal assumption of M, and the instrumented version of M generated by MODLIN that satisfies M's policies. We also describe informally the *access-rights* policy of each module M, which are the conditions under which M guarantees that its environment can read to or write from a given object, or own a given user's permission. I.e., access-right policies describe the assumptions and guarantees that modules provide for a program to perform desired *functionality*.

auth_log policy. The local flow policy of auth_log specifies that information should only flow to log from the value of logger's input at entry to logger. The internal assumption of auth_log assumes that the uninstrumented auth_log only allows information to flow from its input message to log. The access-right policy of auth_log specifies that when auth_log and logger return control to their environment, the environment should be able to read from log. When logger executes, it should be able to write to log.

auth_log instrumentation. The instrumented version of auth_log, generated by modstar creates a category c, creates log with a level that is low at c, creates the logger gate with an ownership set that contains c, and returns control to the environment with a label with level Mid at c, and with an ownership that does not contain c; thus, the environment can read from but not write to log. The instrumented logger executes with ownership of c, but returns control to its environment with level Mid at c, and without ownership of c. Thus logger can write to log, but the environment of logger cannot.

auth_map policy. The local flow policy, internal assumption, and access-right policy for auth_map are similar to the policy and assumption for auth_log. The local flow policy for auth_map specifies that information should only flow to the map file map from the value of map's input at entry to map. The internal assumption is that auth_map only allows its input request to affect the value in map. The access-right policy for auth_map specifies that when auth_map and map return control to their environment, the environment should be able to read from map.

auth_map instrumentation. The instrumented version of auth_map generated by modstar uses label operations similar to the label operations used by the instrumented version of auth_log generated by modstar; we omit a full description.

auth policy. auth's local flow policy specifies that information should only flow from the user's password file to a system object during an execution of chk_pw (which leaks a single bit of the password file when checking its value). The internal assumption of auth is that no module other than chk_pw allows information in the user's password file to flow to any other object.

auth's access-rights policy specifies what access rights auth should grant to the environment for the environment to own the user's category exactly when the environment provides the password and allows auth to log the attempt. I.e., (1) initially, the environment must be able to call the auth gate, (2) when auth returns, the environment must be able to call the chk_pw gate, (3) if chk_pw validates the given password, then the environment must be able to call the grant gate, and (4) if the environment calls the grant gate, then when grant returns, the environment should own the user u's private category upriv. The environment may only own upriv if the above sequence of events occurs.

auth instrumentation. The instrumented version of auth generated by modstar maintains the following key invariant on labels: the environment can only own upriv after calling grant, but the environment cannot call grant until it owns an *authentication-token category* tok. The environment can only own tok 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 tok. (2) auth creates the chk_pw gate so that the gate owns tok, and so that the gate's verification set does not contain tok. (3) auth creates the grant gate so that the gate's verification set does contain tok; thus the environment will not be able to call the grant gate unless it is in a state in which it owns chk_pw. (4) If the chk_pw gate is called with a client password that matches the password of user u, then chk_pw exits in a state that owns tok. Otherwise, chk_pw exits in a state that does not own tok. (5) If the grant gate is called, then grant exits with a store that owns the category upriv.

clnt policy. The local flow policy of clnt specifies that over all executions of clnt and its environment, clnt's password-attempt value should not flow to any object of the system that is not a descendent of seshdir. The internal assumption of clnt specifies that over all executions of clnt, the client's password-attempt value flows only to a process calling a gate that is the child of seshdir. The access-right policy of clnt specifies that its environment should be able to write to seshdir.

clnt instrumentation. The instrumented version of clnt generated by modstar executes the following label operations during each execution: (1) After clnt calls the auth gate to create the session's chk_pw and grant gates, clnt creates a category pw. (2) clnt calls the chk_pw gate to execute with memory that is high at pw. (3) If the chk_pw gate determines that the client provided a password-attempt value that matched the actual password of user u, then clnt calls the grant gate with memory in whose label pw is bound to Mid.

3.4.4 Results

Table 1 contains the results of our experiments. The measurements in Table 1 are divided into (1) features of each program module, (2) features of the policies that we wrote for each module, and (3) features of the performance of modstar. Each feature is described in the caption of Table 1. Instrumenting each module takes no longer than approximately 30 min,

Table 1 Experimental results

Program			Policy			modstar			
Name	LoC	Label sites	Loc.	Int.	Acc.	Ext. time	Instr. time	Total time	Mem. (GB)
auth_log	54	5	4	3	3	0 min 29 s	0 min 04 s	0 min 33 s	4.5
auth_map	157	6	4	3	3	0 min 32 s	0 min 46 s	1 min 18 s	7.0
auth	281	19	3	3	8	0 min 24 s	29 min 33 s	29 min 57 s	16.0
clnt	254	15	3	3	3	0 min 41 s	8 min 41 s	9 min 22 s	8.0

Under the "Program" header, "Name" gives the name of the instrumented program, "LoC" shows the number of lines of code of the program, measured with the cloc utility (which does not count white space or comments); "Label Sites" indicates 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.," "Int," and "Acc." show the number of states in the module's local flow, internal assumption, and access-right policies; Under the "modstar" header, "Ext. Time" shows the amount of time required to construct an external policy from the module's given local and internal assumption; "Instr. Time" shows the amount of time required to rewrite the module; "Total Time" shows the sum of times in "Ext. Time" and "Instr. Time"; "Mem" shows the peak memory used, expressed in gigabytes

and the sum of times taken to instrument all modules is less than 45 min (see column "Total Time"). modstar takes almost as much or more time to construct the external policies for auth_log and auth_map as it does to instrument each module to satisfy its policy. This is likely due to the fact that modstar constructs the external policy by invoking the GOAL automaton library [63] to execute in separate processes; each time modstar provides the automata to GOAL with an unoptimized, explicit representation of their transition relations, and requires GOAL to determinize and minimize the final result. We believe that this component of the implementation of modstar would benefit considerably from standard optimizations.

When we applied monostar to attempt to monolithically instrument the entire login service, monostar exhausted all memory allocated (up to 32 GB). Each module instrumented by modstar executes with execution time of multiple of at most 1.1 over a version of the module instrumented by hand by the HiStar developers.

3.4.5 Conclusions

Our experience using modstar to instrument login gives positive, if cautious, support that MODLIN can be applied to instrument programs for DIFC systems. The key positive result supporting the utility of modstar is that given sufficient local flow and access-right policies and internal assumptions, modstar can efficiently instrument large, modular programs to satisfy practical information-flow policies.

Our belief, based on manually auditing the login modules, is that existing program analyses for type-safe and memory-safe languages [6,23,34,42,60,61], could feasibly be adapted to a manageable subset of C or LLVM to discharge most of the internal flow policies in login: each module operates on sensitive information is small regions of code, using simple operations. However, modstar can make only weak guarantees about sensitive information that is declassified by program modules. In particular, modstar was applied to instrument auth to enforce that information about the user's password is released during executions of chk_pw (and not, e.g., by auth's environment). However, modstar cannot be applied to instrument a version of auth to release an acceptable amount of information about the user's password;

describing and enforcing acceptable declassification is itself a difficult problem [39,46,65] that is outside the scope of our work.

We suspect that the effort required to use MODLIN could be significantly lessened by inferring many local policies directly for uninstrumented modules. For example, multiple access-right policies simply assert that if a module is entered with sufficient rights to perform required operations, then the module does not fail to perform the operations by unnecessarily relinquishing access rights. However, many access-right policies, such as the access-right policy for `auth` and, all local flow policies appear to require a high-level understanding of the requirements of the instrumented module that not apparent from the uninstrumented code.

An ideal DIFC instrumenter would generate a secure implementation of `auth` from a completely naïve implementation of `auth` that merely checks a client's password-attempt value against a user's password, logs the attempted login, and optionally grant ownership of the user's category. I.e., the naïve implementation not only would not contain any label operations, but would not create separate gates to check the password-attempt, log the attempt, and grant ownership of the user's category if the client's password-attempt value matched. Such a DIFC instrumenter would have to (1) partition the naïve implementation into multiple modules $\mathcal{M}$, (2) synthesize new code that creates and links objects O and gates G bound to modules in $\mathcal{M}$, and (3) synthesize label operations that ensure that O and the process executing G have correct labels. Problems related to subproblem (1) are addressed by previous work on partitioning programs to ensure security and performance guarantees [8,11,12]. Subproblem (3) is addressed by MODLIN. Problems related to subproblem (2) have been addressed by recent techniques that synthesize programs that operate on relational and recursive data structures [1,31], which can model the structure of objects and links between them. Extending and composing techniques for solving all three problems is an interesting direction of future work, but is beyond the scope of this paper.

In general, the problem of generating an optimal instrumentation in contexts where the cost of a suboptimal instrumentation is significant is a critical problem. In practice, for DIFC systems, the overhead of MODLIN's instrumentation was negligible (Sect. 3.4.4). We suspect that the overhead induced by instrumentation is negligible at least partly due to the fact that label operations are implemented as efficient operations on maps and integers, whose costs are quickly overshadowed by other common program operations, such as I/O.

The primary goal of our evaluation was to evaluate MODLIN's effectiveness when attempting to instrument a suite of mutually-untrusting programs that must cooperate to perform desired functionality, using a subtle protocol of DIFC primitives. We did not attempt to evaluate MODLIN's effectiveness in instrumenting programs at scale. We believe that in principle, MODLIN could scale well to larger programs, similarly to `capweave`, which scaled to instrument the `php` interpreter. In general, program instrumenters for interactive-security systems only need to model the state relevant to system state, namely the state of file descriptors and other objects shared by processes. Even large applications that implement complex functionality have relatively small modules of code that perform operations on such objects. Evaluating the ability of MODLIN to scale in practice is a promising direction for future work.

4 Future work: synthesizing programs that use secure isolated regions

Modern computer users can perform resource-intensive computations on their sensitive data on networked cloud servers that provide computation as a service. In particular, a cloud server may run vulnerable systems infrastructure that can be compromised by a user who runs an application co-located with a user's computation. A malicious user may then exploit the vulnerability to learn information about another user's private data, or tamper with the result of a user's critical computation.

To address this problem, novel architectures have been developed that implement special instructions that an application invokes to contain segments of code and data in hardware-level *Secure Isolated Regions* (SIRs) (also called *enclaves*).[1] In particular, the SGX enclave feature supported by recent Intel processors [35] and the TrustZone feature supported by recent ARM processors [5] support an extended instruction-set architecture (ISA) that a user-level application can use to ensure confidentiality and integrity. Academic alternatives such as the MIT Sanctum platform [14] provide similar features to user-level applications using a combination of hardware and software. Code that executes in a SIR cannot be tampered with by other agents on a system. Data in a SIR is encrypted when written to non-SIR memory or persistent storage, ensuring that even an adversary with complete control over hardware connected to the core processor cannot learn information about the application's data or tamper with the data without being detected.

Such hardware features are powerful [49], but present application programmers with new, significant challenges. Figure 10a contains pseudocode for a simple *reduce* procedure, named `reduce`, that could potentially be run on a cloud server as a component in a map-reduce framework. The core functionality of `reduce` is to read as input a sequence of numeric values as input (lines 3–9), compute their sum (lines 11–15), and output the sum. While the core functionality is relatively simple, the implementation of `reduce` is complicated by the fact that for it to trust the integrity of its inputs, it must establish an encrypted input channel and read values from the input channel (lines 3–8). Similarly, for `reduce` to establish trustworthiness of its output, it must establish an encrypted output channel and write values to the output channel (lines 18–19). Both channels are established using several low-level operations on memory that, if incorrect, could compromise the correctness of `reduce`. Moreover, `reduce` must not inadvertently store a value with information about its sensitive inputs to an address outside of its SIR.

Previous work on verification. We have performed previous work on verifying that an application uses SIRs correctly [51,52]. In particular, our work addresses the problem of taking (1) an application manually written to use SIRs, and (2) a policy denoting the sensitive information operated on by the application and verifying that the application does not leak sensitive information to an adversarial system. The key result of our work is a semi-automated design methodology that a programmer follows to write an application that uses SIRs to satisfy confidentiality properties. In particular, the programmer writes the program to use a communication library—one artifact resulting from the work—to marshal the communication of all sensitive information to its environment. We manually verified that procedures in the library reveal information given to them only to target storage.

The programmer also runs an automatic verifier—a second artifact resulting from the work—on code assigned to run in a SIR to ensure that it only releases sensitive information through the verified communication library. Verifying the property amounts to verifying that code designated to execute in a SIR only (1) runs code designated to execute in a SIR (as opposed to, e.g., incorrectly branching to untrusted code) and (2) only writes sensitive information to memory addresses within a SIR. When successful, the verifier automatically generates a formal proof of security.

[1] We adopt the nomenclature of SIR from [51].

```
1  void reduce(BYTE *keyEnc, BYTE *valuesEnc,
2              BYTE *outputEnc) {
3    KeyAesGcm *aesKey = ProvisionKey();
4    char key[KEY_SIZE];
5    aesKey->Decrypt(keyEnc, key, KEY_SIZE);
6    char valuesBuf[VALUES_SIZE];
7    aesKey->Decrypt(valuesEnc, valuesBuf,
8                    VALUES_SIZE);
9    StringList *values =
10     (StringList *) valuesBuf;
11   long long usage = 0;
12   for (char *value = values->begin();
13        value != values->end();
14        value = values->next()) {
15     long lvalue =
16       mystrtol(value, NULL, 10);
17     usage += lvalue; }
18   char cleartext[BUF_SIZE];
19   sprintf(cleartext, "%s %lld",
20           key, usage);
21   aesKey->Encrypt(cleartext, outputEnc,
22                   BUF_SIZE); }
```

(a)

```
1  void reduce_sec(Channel<char*>& channel) {
2    char key[KEY_SIZE];
3    channel.recv(key, KEY_SIZE);
4    char valuesBuf[VALUES_SIZE];
5    channel.recv(valuesBuf, VALUES_SIZE);
6    StringList *values =
7      (StringList *) valuesBuf;
8    long long usage = 0;
9    for (char *value = values->begin();
10        value != values->end();
11        value = values->next()) {
12     long lvalue = mystrtol(value, NULL, 10);
13     usage += lvalue; }
14   char cleartext[BUF_SIZE];
15   sprintf(cleartext, "%s %lld", key, usage);
16   channel.send(cleartext); }
```

(b)

Fig. 10 Pseudocode for insecure and secure versions of a `reduce` function used in a map-reduce framework. **a** `reduce`: performs a reduction, in the context of a map-reduce framework. **b** `reduce_sec`: a verified version of `reduce`, adapted to use secure channels

Figure 10b contains `reduce` revised to use a secure channel developed in our work. The implementation of core functionality in `reduce` (Fig. 10a, lines 9–15) is identical to `reduce_sec` (Fig. 10b, lines 6–12). However, all code in `reduce` that manually establishes secure input (Fig. 10a, lines 3–8) and output channels (lines 18–19) are replaced with usage of the verified `Channel` library (Fig. 10b, lines 3 and 15).

Proposed work on synthesis. Our previous work enables a programmer to verify that a program that has already been *manually* written to use SIRs satisfies a desired confidentiality policy. However, it does not address the problem of writing or rewriting an application to correctly use SIRs. Writing an application to correctly use SIRs typically requires a developer to write the application to execute low-level instructions so that multiple goals are satisfied simultaneously. In particular, (1) the code segments designated to execute in a SIR must be sufficiently *expansive* that they can perform desired computations on sensitive inputs. (2) The code segments designated to execute in a SIR must be sufficiently *compact* that they do not leak information about the sensitive information that they operate on, and that such a property can be verified automatically. (3) Data that must be operated on both by code that executes in a SIR and code that does not execute in a SIR must be *shared efficiently*. (4) Code that executes in a SIR but receives data not stored in a SIR must run sufficiently strong *validation* checks on such data (which can be tampered with by the adversary in arbitrary ways) to ensure that it can use the data while preserving the desired *integrity* of the computation that it performs. (5) Such validation checks must be sufficiently *permissive* that if code that executes out of an SIR is not corrupted by an adversary, then the entire application correctly performs desired *functionality*.

Applications are verified by verifying that their components each satisfy key properties. We provide a summary of the verification technique presented in detail in previous work [51, Sec. 5]. Let U be the user application and L a small-time runtime library that provides core primitives, such as memory management and encrypted channels. The application U can only communicate with the untrusted platform through the narrow interface provided by L, which enables compositional verification. Verification is performed in two steps. (1) The library L is manually verified to correctly implement a secure encrypted channel and memory management. Such verification, though not automatic, can be performed with reasonable design effort because L is small. Furthermore, the verification task needs to be performed only once across all applications that use L.

(2) U is verified to be correct, assuming that L implements a secure channel and memory management. This amounts to verifying that U satisfies a relaxed definition of control-flow integrity (CFI); i.e., the target of each jump executed by U is the address of an instruction in L of the entry point of a implemented in L, and U does not access state used by cryptographic operations implemented in L. U is verified by analyzing the machine code and generating verification conditions in an SMT theory, which are discharged using an SMT solver.

Our previous work on the verification of applications that use SIRs is based on the assumption that a programmer has already written an application with the above conditions in mind. Given such an application, our verifier validates that the program uses SIRs to satisfy conditions (1) and (2). However, it does not attempt to verify that the program shares data efficiently (condition (3)) or that it satisfies desired integrity guarantees (condition (4)). In fact, integrity properties of practical programs are typically difficult even to *express* because they are typically highly specific to its individual application. Finally, because the verifier operates only on code that has been adapted to use SIRs, but not the original code as a reference, the verifier cannot be used to verify that the use of SIRs preserves critical functionality (condition (5)). Specifications of core functionality that must be preserved, similar to desired integrity properties, are typically each also highly specific to individual applications.

As future work, we propose to develop a synthesizer that takes an application that uses no SIR primitives and instruments it to use SIR primitives so that it satisfies each of conditions (1)–(5). We expect that developing such a synthesizer will be feasible based on key observations that connect the problem with our previous work on developing synthesis frameworks. In particular, a key result of our previous work developing the SyGuS synthesis framework is that a program synthesizer can often be structured as a procedure that iteratively synthesizes a candidate program, runs a verifier to determine if the program satisfies key properties, and uses the result of the verifier to generate the next candidate program—a so-called *counterexample-guided inductive synthesis* (CEGIS) approach [2,58]. We believe that it is highly feasible that we can develop a synthesizer that synthesizes programs that satisfy properties (1) and (2) by iteratively running the verifier for SIR programs that we have developed in previous work and inspecting its results to synthesize new candidate uses of SIR primitives. Rich integrity properties (condition (4)) will be expressed as hyperproperties [13]. Equivalence to an original program will be established by synthesizing relational invariants that accompany the synthesized program.

5 Related work

Capability systems. Capabilities were introduced in the MULTICS system [48], and were developed further in the capability systems PSOS [43] and EROS [50]. 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 [68] 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 Sect. 2 describes the design and evaluation of a program weaver that automatically instruments programs to use capabilities on Capsicum. We suspect that such instrumentation techniques 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. In this paper, we have primarily discussed the problem of instrumenting program modules. Partitioning a program consists of choosing which modules to bind to RPC services and which capabilities to associate with each service, and is discussed in detail in our previous work [30]. Previous work [8,12] automatically partitions programs so that high and low confidentiality data are processed by separate processes, or on separate hosts. The SOAP project [25] 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 [53] 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 operating systems. DIFC operating systems, such as Asbestos [20], HiStar [71], and Flume [37], explicitly track the flow of information between system objects, such as processes and files. The Laminar runtime system [45] explicitly tracks the flow of information through both system and program memory objects. Such systems are designed so that a program, in principle, can satisfy strong information-flow properties when interacting with potentially-malicious programs. 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. MODLIN complements information-flow operating systems: it takes from a programmer explicit non-interference and access-rights policies, and rewrites a program to use label operations so that it satisfies the policies.

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 automatically generates code that initializes the labels of each thread [19], or chooses labels that the program should hold at each of its control locations to hold sufficient access rights, but not allow insecure information flows [28]. However, both approaches must be applied to instrument all modules in a program simultaneously. Both of the approaches can be viewed as implementations of the monolithic DIFC instrumenter MONOLITH.

In previous work [29], we proposed a technique that takes abstract models of the semantics of a program, a policy, and an operating-system as visibly-pushdown automata [4], and instruments the program by finding a modular strategy to a visibly-pushdown game [3]. Further work demonstrated that the reduction to visibly-pushdown games can be applied to instrument programs to correctly operate on capabilities [30]. We have applied these techniques to create MONOLITH, which is used as a component of `modstar`. While MONOLITH can instrument programs to satisfy richer policies than policies supported by instrumenters proposed in previous work [19,28], MONOLITH itself does not scale to instrument programs as complex as the HiStar mutually-untrusting login service, unless the programs and policies have been decomposed using MODLIN.

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, either reasoning statically about all program executions [6,42,60,61], or dynamically monitoring a single program execution [23,34]. MODLIN instruments programs to interact with an operating system to manage how sensitive resources are accessed by an uncontrolled environment. MODLIN uses internal assumptions about a given uninstrumented program, which could in principle be discharged by the analyses and type-checkers provided by information-flow languages.

An *Inline Reference Monitor (IRM)* [21] uses code placed in an untrusted program by an IRM rewriter to monitor the program's behavior at runtime and halts the program immediately before the program would violate the security policy. An IRM mediates only the operations of the program in which it is instrumented. DIFC instrumenters, including MODLIN, address a different problem: to rewrite a program so that the program can ensure the security of its resources as the program is granted control by, and returns controls to, uninstrumented programs in its environment.

Compositional verification. Assume-guarantee reasoning [44] and compositional verification [24] are techniques for efficiently model-checking transition systems that are composed of multiple modules. Both techniques decompose the problem of checking that a large system composed of multiple modules satisfies a global property G to smaller independent problems of checking that each module M in G satisfies a guarded internal property of M. MODLIN is directly inspired by compositional verification. However, the goal of MODLIN is to instrument a modular program to satisfy a desired non-interference policy, not to check if the program satisfies a temporal property.

Partial synthesis. Several techniques have been proposed for extending incomplete programs (i.e., *partial* program models) to satisfy a desired property. Game-solving has been applied to "repair" programs to satisfy temporal specifications [36]. Program sketching, as a form of syntax-guided synthesis [2], takes a program with "holes" for program expressions and statements and a specification, and synthesizes a complete program by choosing an expression or statement for each hole [55–58]. MODLIN can be thought of as a modular partial synthesizer for a specific class of DIFC properties. For a generic partial program synthesizer S to be applied to the problem that MODLIN addresses, S would have to be able to instrument a program implemented in a language that can model DIFC labels to satisfy properties that can encode non-interference automata (see Sect. 3.2.2). However, S used in this fashion would mimic MONOLITH, and would not support a modular approach, as MODLIN does.

Language support for security. The general topic of this paper is synthesis of secure programs, which is related to a paper by Holzer et al. [33], which describes a tool that achieves Secure Two-Party Computation [32,70] for ANSI C. Their work is based on a combination of model-checking techniques and two-party computation based on garbled circuits—a primitive introduced by Yao in 1982. The key insight is a nonstandard use of the bit-precise model checker CBMC which enables them to translate C programs and "synthesize" equivalent Boolean circuits. To achieve their goal, they modified the standard CBMC translation from programs into Boolean formulas whose variables correspond to the memory bits manipulated by the program. Because CBMC attempts to minimize the size of the formulas, the circuits obtained by their tool chain are also succinct. To further improve the efficiency of the garbled circuit evaluation, their tool performs optimizations on the circuits produced by CBMC. The paper also has an extensive experimental evaluation. Several researchers are following up on this line of work. As usual, Helmut Veith was a one of pioneers in this general research trend. Helmut Veith was an incredible researchers and his insight and creativity will be missed by the entire community.

6 Conclusion

Interactive security systems provided a small set of powerful primitives that enable an application programmer to potentially write an application that satisfies a strong security guarantee. However, such primitives do not enable programmers to specify the security requirements of their program directly, much less verify that their application satisfies such a policy. Furthermore, such systems typically require a programmer to provide guarantees in multiple contexts: in particular, each module must not enable a malicious environment (e.g., injected code in a Capsicum process or a malicious program on a HiStar system) to perform insecure operations, but must allow a cooperative module (i.e., another module in the HiStar login service) to perform desired functionality.

The main contribution of our previous work has been policy languages in which a developer can express desired security policies and desired functionality directly, and program instrumenters that take a program and a policy and instrument the program to satisfy the policy by invoking primitives of an interactive-security system. To enable the scalable development of programs consisting of multiple modules that wish to cooperate in the presence of a malicious environment, we have developed a framework for decomposing global information-flow and functionality policies into per-module policies that can be discharged using previously developed techniques.

As computer end-users aim to perform operations on increasing amounts of their sensitive data on cloud systems running untrusted software platforms, the importance of architectures that enable an application to construct Secure Isolated Regions will continue to increase. Our previous experience demonstrates that verifying that an application satisfies desired security properties is challenging, but tractable. A promising direction for future work will be to extend existing techniques for verifying that programs correctly construct SIRs to satisfy high-level security properties to techniques for synthesizing such programs automatically.

Acknowledgements The authors wish to thank the many researchers and collaborators who contributed to the work described in this paper, including Jonathan Anderson, Manuel Costa, Akash Lal, Nuno Lopes, Roman Manevich, Sriram Rajamani, Mooly Sagiv, Rohit Sinha, Kapil Vaswani, Robert Watson, and Nickolai Zeldovich. The work described in this paper was supported, in part, by a gift from Rajiv and Ritu Batra; by DARPA under Cooperative Agreement HR0011-12-2-0012; by NSF under Grants CCF-0904371, CNS-1228620, CNS-1228782, and SATC-1526211; by the NSF STARSS Grant CNS-1528108; by SRC contracts 2460.001 and 2638.001; by a gift from Microsoft Research; by AFRL under DARPA CRASH Award FA8650-10-C-7088, DARPA MUSE Award FA8750-14-2-0270, DARPA STAC Award FA8750-15-C-0082, and DARPA XD3 Award HR0011-16-C-0059; by USAF and DARPA under Contract No. FA8650-15-C-7562; and by the UW-Madison Office of the Vice Chancellor for Research and Graduate Education with funding from the Wisconsin Alumni Research Foundation. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors, and do not necessarily reflect the views of the sponsoring agencies.

References

1. Albarghouthi A, Gulwani S, Kincaid Z (2013) Recursive program synthesis. In: CAV
2. Alur R, Bodík R, Juniwal G, Martin M M K, Raghothaman M, Seshia S A, Singh R, Solar-Lezama A, Torlak E, Udupa A (2013) Syntax-guided synthesis. In: FMCAD
3. Alur R, La Torre S, Madhusudan P (2006) Modular strategies for recursive game graphs. Theor Comput Sci 354(2):230–249
4. Alur R, Madhusudan P (2004) Visibly pushdown languages. In: STOC
5. ARM (2016) Products. https://www.arm.com/products/security-on-arm/trustzone. Accessed 9 Sept 2016
6. Barthe G, Fournet C, Grégoire B, Strub P-Y, Swamy N, Béguelin SZ (2014) Probabilistic relational verification for cryptographic implementations. In: POPL
7. Bittau A, Marchenko P, Handley M, Karp B (2008) Wedge: splitting applications into reduced-privilege compartments. In: NSDI
8. Brumley D, Song D X (2004) Privtrans: automatically partitioning programs for privilege separation. In: USENIX security symposium
9. C. E. Board. CVE-2007-4476. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2007-4476, Aug 2007
10. C. E. Board. GNU Tar and GNU Cpio rmt_read__() function buffer overflow. http://xforce.iss.net/xforce/xfdb/56803, Mar 2010
11. Cheung A, Arden O, Madden S, Myers AC (2012) Automatic partitioning of database applications. PVLDB 5(11):1471–1482
12. Chong S, Liu J, Myers A C, Qi X, Vikram K, Zheng L, Zheng X (2007) Secure web application via automatic partitioning. In: SOSP
13. Clarkson MR, Schneider FB (2010) Hyperproperties. J Comput Secur 18(6):1157–1210
14. Costan V, Lebedev I, Devadas S (2015) Sanctum: minimal hardware extensions for strong software isolation. Cryptology ePrint Archive, Report 2015/564. http://eprint.iacr.org/
15. CVE-2004-1488. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CAN-2004-1488, Feb 2005
16. CVE-2007-3798. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2007-3798, July 2007
17. CVE-2010-0405. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-0405, Apr 2010
18. Denning DE (1976) A lattice model of secure information flow. Commun ACM 19(5):236–243
19. Efstathopoulos P, Kohler E (2008) Manageable fine-grained information flow. In: EuroSys
20. Efstathopoulos P, Krohn M N, Vandebogart S, Frey C, Ziegler D, Kohler E, Mazières D, Kaashoek MF, Morris R (2005) Labels and event processes in the Asbestos operating system. In: SOSP
21. Erlingsson Ú, Schneider FB (2000) IRM enforcement of Java stack inspection. In: SSP
22. FreeBSD 9.0-RELEASE announcement. http://www.freebsd.org/releases/9.0R/announce.html, Jan 2012
23. Giffin DB, Levy A, Stefan D, Terei D, Mazières D, Mitchell JC, Russo A (2012) Hails: protecting data privacy in untrusted web applications. In: OSDI
24. Grumberg O, Long DE (1994) Model checking and modular verification. ACM Trans Program Lang Syst 16(3):843–871
25. Gudka K, Watson RNM, Hand S, Laurie B, Madhavapeddy A (2012) Exploring compartmentalization hypothesis with SOAPP. In: AHANS 2012
26. Harris W (2014) Secure programming via game-based synthesis. PhD thesis, University of Wisconsin—Madison
27. Harris W, Zeldovich N, Jha S, Reps T, Manevich R, Sagiv M (2014) Modular synthesis of DIFC programs. Technical report, Georgia Insitute of Technology
28. Harris WR, Jha S, Reps T (2010) DIFC programs by automatic instrumentation. In: CCS
29. Harris WR, Jha S, Reps T (2012) Secure programming via visibly pushdown safety games. In: CAV
30. Harris WR, Jha S, Reps T, Anderson J, Watson RNM (2013) Declarative, temporal, and practical programming with capabilities. In: SSP
31. Hawkins P, Aiken A, Fisher K, Rinard MC, Sagiv M (2011) Data representation synthesis. In: PLDI
32. Hazay C, Lindell Y (2010) Efficient secure two-party protocols: techniques and constructions. Springer, Berlin

33. Holzer A, Franz M, Katzenbeisser S, Veith H (2012) Secure two-party computations in ANSI C. In: CCS
34. Hriţcu C, Greenberg M, Karel B, Pierce BC, Morrisett G (2013) All your IFCException are belong to us. In: SSP
35. Intel Software (2016) Intel SGX homepage. https://software.intel.com/en-us/sgx. Accessed 9 Sept 2016
36. Jobstmann B, Griesmayer A, Bloem R (2005) Program repair as a game. In: CAV
37. Krohn MN, Yip A, Brodsky MZ, Cliffer N, Kaashoek MF, Kohler E, Morris R (2007) Information flow control for standard OS abstractions. In: SOSP
38. Lattner C (2011) http://llvm.org/, Nov 2011
39. Livshits B, Chong S (2013) Towards fully automatic placement of security sanitizers and declassifiers. In: POPL
40. Livshits VB, Nori AV, Rajamani SK, Banerjee A (2009) Merlin: specification inference for explicit information flow problems. In: PLDI
41. Manevich R (2011) http://www.cs.tau.ac.il/tvla, June 2011
42. Myers AC (1999) Jflow: practical mostly-static information flow control. In: POPL
43. Neumann PG, Boyer RS, Robinson L, Levitt KN, Boyer RS, Saxena AR (1980) A provably secure operating system. Technical report CSL-116, Stanford Research Institute
44. Pnueli A (1985) Logics and models of concurrent systems. In: Apt KR (ed) In transition from global to modular temporal reasoning about programs. Springer, New York
45. Roy I, Porter DE, Bond MD, McKinley KS, Witchel E (2009) Laminar: practical fine-grained decentralized information flow control. In: PLDI
46. Sabelfeld A, Sands D (2005) Dimensions and principles of declassification. In: CSFW-18
47. Sagiv S, Reps T, Wilhelm R (2002) Parametric shape analysis via 3-valued logic. ACM Trans Program Lang Syst 24(3):217–298
48. Saltzer JH, Schroeder MD (1975) The protection of information in computer systems. Proc IEEE 63(9):1278–1308
49. Schuster F, Costa M, Fournet C, Gkantsidis C, Peinado M, Mainar-Ruiz G, Russinovich M (2015) VC3: trustworthy data analytics in the cloud using SGX. In: SP
50. Shapiro JS, Smith JM, Farber DJ (1999) EROS: a fast capability system. In: SOSP
51. Sinha R, Costa M, Lal A, Lopes NP, Rajamani SK, Seshia SA, Vaswani K (2016) A design and verification methodology for secure isolated regions. In: PLDI
52. Sinha R, Rajamani SK, Seshia SA, Vaswani K (2015) Moat: verifying confidentiality of enclave programs. In: CCS
53. Skalka C, Smith SF (2000) Static enforcement of security with types. In: ICFP, pp 34–45
54. Sohail S, Somenzi F (2009) Safety first: a two-stage algorithm for LTL games. In: FMCAD
55. Solar-Lezama A, Arnold G, Tancau L, Bodík R, Saraswat VA, Seshia SA (2007) Sketching stencils. In: PLDI
56. Solar-Lezama A, Jones CG, Bodík R (2008) Sketching concurrent data structures. In: PLDI
57. Solar-Lezama A, Rabbah RM, Bodík R, Ebcioglu K (2005) Programming by sketching for bit-streaming programs. In: PLDI
58. Solar-Lezama A, Tancau L, Bodík R, Seshia SA, Saraswat VA (2006) Combinatorial sketching for finite programs. In: ASPLOS
59. Swamy N, Chen J, Fournet C, Strub P-Y, Bhargavan K, Yang J (2011) Secure distributed programming with value-dependent types. In: ICFP
60. Swamy N, Corcoran BJ, Hicks M (2008) Fable: a language for enforcing user-defined security policies. In: SSP
61. Swamy N, Hicks M (2008) Verified enforcement of stateful information release policies. SIGPLAN Not 43(12):21–31
62. T. M. Corporation (2011) Cwe—2011 cwe/sans top 25 most dangerous software errors
63. Tsai M-H, Tsay Y-K, Hwang Y-S (2013) GOAL for games, omega-automata, and logics. In: CAV
64. U.S.D. of Defense. Trusted computer system evaluation criteria. DoD Standard 5200.28-STD, Dec 1985
65. Vaughan JA, Chong S (2011) Inference of expressive declassification policies. In: SSP
66. Vulnerability note VU#520827. http://www.kb.cert.org/vuls/id/520827, May 2012
67. Vulnerability note VU#381508. http://www.kb.cert.org/vuls/id/381508, July 2011
68. Watson RNM, Anderson J, Laurie B, Kennaway K (2010) Capsicum: practical capabilities for UNIX. In: USENIX security symposium
69. Wright C, Cowan C, Smalley S, Morris J, Kroah-Hartman G (2002) Linux security modules: general security support for the Linux kernel. In: USENIX security symposium
70. Yao A (1982) Protocols for secure computations. In: FOCS
71. Zeldovich N, Boyd-Wickizer S, Kohler E, Mazières D (2006) Making information flow explicit in HiStar. In: OSDI