

Instructor: Matt Fredrikson

TA: Milijana Surbatovich

Due date: November 23 11:59pm

Total points: 100

Lab 2: Better Control with Information Flow Policies

1 Introduction

Now that your server contains a safe interpreter for working with persistent data, users are beginning to grow uneasy with the fact that all of their data and results are stored in plain view of all the other users. They demand some form of protection, so that others cannot see their data and learn from the results of their script computations.

This lab will have you apply what you learned about information flow type systems to the interpreter you implemented in the previous lab. You will enhance the server so that it keeps track of which users execute scripts, and associates the data stored in its persistent database files with the user that created it. The server will associate each user with an information flow label, and statically verify that the scripts that it runs on behalf of users do not result in information flows from other users' data to any sort of observation that the user executing the script might make.

To make all of this happen, you will need to design a custom type system for the scripting language that enforces non-interference with respect to a security lattice that accounts for all users of the system. You will then implement type checking functionality that embodies the formal rules of your type system, and integrate this functionality with your existing interpreter. When all of this is finished, you will have addressed your users' concerns about the confidentiality of their data stored on the server, and learned a great deal about bringing theory to practice in the service of making your server more secure.

Learning goals. As you complete this lab, you will:

- Design a novel information flow type system for the scripting language introduced in the previous lab. Optionally for extra credit, you will also further develop formal proof skills by showing that your type system is sound.
- Gain experience translating formal typing rules into an implementation of a type checker.
- Develop a deeper understanding of the practical security guarantees afforded by information flow controls by integrating your type checker with the script interpreter that you developed in the previous lab.
- Undertake a critical evaluation of the resulting protections by writing a brief report that weighs the pros and cons of using your information flow type system to achieve the desired protections against possible alternatives.

Evaluation. This lab is worth 100 points, and will be graded by a combination of test cases and manual inspection by the course staff. The manual inspection will evaluate the soundness of your information flow type system, i.e., whether it correctly enforces non-interference for the scripting language. The test cases will (i) evaluate the correctness of your type checker, and (ii) evaluate the correctness of your integration of the type checker with your interpreter. The point breakdown is as follows.

Correct typing rules (35 points, 10 possible extra credit). Your typing rules should enforce the non-interference policy outlined in Section 3. Partial credit will be given based on how many rules are correct. For 10 points of extra credit, you can prove that your rules are sound against the semantics given in Section 3. Partial credit for the soundness proof will be awarded based on how close your proof is to being correct, but incomplete proofs that only cover a subset of the rules will be awarded no points.

Correct type checker (30 points). Your type checker should faithfully correspond to the typing rules that you provided in the first task. Partial credit will be awarded based on how many test cases you pass. If your typing rules from the first task are unsound, you will not be double-penalized for failed test cases that are consistent with the typing rules you provided.

Correct integration (25 points). Your type checker must be integrated with your interpreter to achieve a meaningful practical guarantee of security. Test cases will evaluate whether the interpreter allows information flows that it shouldn't based on the lattice policy, and whether it correctly allows well-typed programs to execute normally. The same double-penalty policy from the previous task applies here. Partial credit will be awarded based on how many test cases your implementation passes.

Discussion (10 points) A brief discussion of your solution, including the security considerations and any assumptions you made while developing it, as well as a critical comparison to other possible approaches to the solution.

What to hand in. When you have completed the lab, you should hand in a `.zip` archive of the same directory tree from the previous lab (see Section 2), but with your completed solution filled into the appropriate files. Your type system should be provided in a file called `typesystem.pdf`, and we strongly urge you to typeset it. We will not attempt to second-guess illegible handwriting, and there will not be opportunities to clarify your rules after the due date. You should not hand in your sandbox implementation from the previous lab. As before, it is recommended to build everything in a `build` subdirectory so that you can easily delete it before handing in.

Finally, if you are doing the extra credit parts, provide a `EC_README` file detailing the tasks you have chosen to complete.

2 Getting started

In this lab you will continue to build on your implementation from the previous two labs. You should copy your current implementation to a new directory after removing any compiled binaries and makefiles emitted by CMake. The directory tree shown below is color-coded to help you set up the lab. **Orange** strikethrough files can be removed. **Blue** files are those in which you will spend most of your time implementing the lab. `sectypes.h` and `sectypes.c` are in the handout, so you will need to copy them over to their proper locations. **Purple** files need to be replaced with new versions provided with the current lab handout (see Section 5).

├── CMakeLists.txt	Build file, <i>do not modify</i>
├── sandbox	Template sandbox implementation, <i>not needed for this lab</i>
├── em.cpp	Template Pintool
├── makefile	Pin build file
├── makefile.rules	Pin build file
├── src	Template server and interpreter
│ ├── common	Common library for server
│ │ ├── CMakeLists.txt	Build file, <i>do not modify</i>
│ │ ├── csapp.c	Robust IO routines
│ │ ├── extendible_hash.c	Your memory-safe extendible hash
│ │ ├── safemem.c	Sandbox memory manager
│ │ └── ubarray.c	Your memory-safe unbounded array
│ ├── include	Header files
│ │ ├── common	Definitions for common library
│ │ │ ├── csapp.h	Definitions for robust IO
│ │ │ ├── extendible_hash.h	Your extendible hash definitions
│ │ │ ├── safemem.h	Sandbox memory manager definitions
│ │ │ └── ubarray.h	Your unbounded array definitions
│ │ ├── tinyscript	Definitions for interpreter
│ │ │ ├── ast.h	Abstract syntax tree definitions
│ │ │ ├── interp.h	Interpreter definitions
│ │ │ └── sectypes.h	Type checker definitions
│ ├── server	Core server implementation
│ │ ├── CMakeLists.txt	Build file, <i>do not modify</i>
│ │ ├── client.c	Simple client to test server functionality
│ │ └── tiny.c	Server implementation
│ ├── tinyscript	Interpreter
│ │ ├── CMakeLists.txt	Build file, <i>do not modify</i>
│ │ ├── ast.c	Functions to build abstract syntax trees
│ │ ├── interp.c	Core interpreter routines
│ │ ├── interp_main.c	Interpreter shell
│ │ ├── sectypes.c	Type checker implementation
│ │ ├── parser	Parser implementation, <i>do not modify</i>
│ │ │ ├── lexer.l	Rules for scanning strings containing programs
│ │ │ └── parser.y	Grammar for language syntax
│ └── testscripts	Example programs to test interpreter

3 Task 1: Design the type system

The syntax of the scripting language that you will use in this lab is mostly unchanged from Lab 1, with a single exception. Because there are multiple users on the system with varying levels of trust between them, we must now require users to declare who they are and provide credentials in the form of a password to demonstrate that they are authorized to run their script. The `<prog>` production reflects this change by requiring scripts to start with a header containing the user on behalf of whom the script is run, as well as their password. Note that the user names provided in this header correspond with security type labels, and passwords are alphanumeric strings containing no spaces or special symbols.

```

<prog> ::= using table as user with password : <com>
<com>  ::= skip // do nothing
        | x := <aexp> // assignment
        | undef(x) // remove variable
        | output <aexp> // print expression value
        | <com>;<com> // composition
        | if <bexp> then <com> else <com> endif // conditional
        | while <bexp> do <com> done // loop
<aexp> ::= c // integer constant
        | x // variable identifier
        | (<aexp>) // parenthesized expression
        | <aexp> + <aexp> // addition
        | <aexp> - <aexp> // subtraction
        | <aexp> * <aexp> // multiplication
<bexp> ::= true | false // boolean constants
        | hasdef(x) // check that variable is defined
        | !<bexp> // negation
        | (<bexp>) // parenthesized expression
        | <bexp> && <bexp> // conjunction
        | <bexp> || <bexp> // disjunction
        | <aexp> == <aexp> // equality
        | <aexp> <= <aexp> // inequality

```

The semantics are unchanged from Lab 1, and are shown in Figure 1 for your convenience.

Derive typing rules. The first thing that you will do is provide typing rules for the language which ensure that programs satisfy non-interference under a given typing context Γ . You should assume that Γ associates any variable appearing in the script with a label $\ell \in L$ that comes from a security lattice $(L, \sqsubseteq, \perp, \top, \sqcup)$.

The lattice that you will use consists of labels ℓ_1, \dots, ℓ_n for each of the users that run scripts on the system (more on this in the next task), as well as two distinguished users `admin` and `pub`. You should assume that `admin` corresponds to the greatest element \top , and `pub` to the least element \perp .

$$\text{pub} \sqsubseteq \ell_i \sqsubseteq \text{admin}, \text{ for all } 1 \leq i \leq n \quad (1)$$

Moreover, none of the remaining ordinary users are allowed to read or write each other's data.

$$\ell_i \not\sqsupseteq \ell_j \text{ and } \ell_j \not\sqsupseteq \ell_i, \text{ for any } i, j \text{ where } 1 \leq i < j \leq n \quad (2)$$

Arithmetic expressions

$$\frac{}{\langle \omega, c \rangle \Downarrow_{\mathbb{Z}} c} \quad \frac{\omega_v(x) \text{ is defined} \quad \omega_v(x) = v}{\langle \omega, x \rangle \Downarrow_{\mathbb{Z}} v} \quad \frac{\langle \omega, e \rangle \Downarrow_{\mathbb{Z}} v}{\langle \omega, (e) \rangle \Downarrow_{\mathbb{Z}} v} \quad \frac{\langle \omega, e \rangle \Downarrow_{\mathbb{Z}} v_1 \quad \langle \omega, \tilde{e} \rangle \Downarrow_{\mathbb{Z}} v_2}{\langle \omega, e \odot \tilde{e} \rangle \Downarrow_{\mathbb{Z}} v_1 \odot v_2}$$

Boolean expressions

$$\frac{}{\langle \omega, \text{true} \rangle \Downarrow_{\mathbb{B}} \top} \quad \frac{}{\langle \omega, \text{false} \rangle \Downarrow_{\mathbb{B}} \perp} \quad \frac{\omega_v(x) \text{ is defined}}{\langle \omega, \text{hasdef}(x) \rangle \Downarrow_{\mathbb{B}} \top} \quad \frac{\omega_v(x) \text{ not defined}}{\langle \omega, \text{hasdef}(x) \rangle \Downarrow_{\mathbb{B}} \perp}$$

$$\frac{\langle \omega, P \rangle \Downarrow_{\mathbb{B}} b}{\langle \omega, !P \rangle \Downarrow_{\mathbb{B}} \neg b} \quad \frac{\langle \omega, P \rangle \Downarrow_{\mathbb{B}} b}{\langle \omega, (P) \rangle \Downarrow_{\mathbb{B}} b} \quad \frac{\langle \omega, P \rangle \Downarrow_{\mathbb{B}} b_1 \quad \langle \omega, Q \rangle \Downarrow_{\mathbb{B}} b_2}{\langle \omega, P \ \&\& \ Q \rangle \Downarrow_{\mathbb{B}} b_1 \wedge b_2} \quad \frac{\langle \omega, P \rangle \Downarrow_{\mathbb{B}} b_1 \quad \langle \omega, Q \rangle \Downarrow_{\mathbb{B}} b_2}{\langle \omega, P \ || \ Q \rangle \Downarrow_{\mathbb{B}} b_1 \vee b_2}$$

$$\frac{\langle \omega, e \rangle \Downarrow_{\mathbb{Z}} v_1 \quad \langle \omega, \tilde{e} \rangle \Downarrow_{\mathbb{Z}} v_2}{\langle \omega, P == Q \rangle \Downarrow_{\mathbb{B}} v_1 = v_2} \quad \frac{\langle \omega, e \rangle \Downarrow_{\mathbb{Z}} v_1 \quad \langle \omega, \tilde{e} \rangle \Downarrow_{\mathbb{Z}} v_2}{\langle \omega, P <= Q \rangle \Downarrow_{\mathbb{B}} v_1 \leq v_2}$$

Commands

$$\frac{}{\langle \omega, \text{skip} \rangle \Downarrow \omega} \quad \frac{\langle \omega, e \rangle \Downarrow_{\mathbb{Z}} v}{\langle \omega, x := e \rangle \Downarrow \omega\{x \mapsto v\}} \quad \frac{\langle \omega, e \rangle \Downarrow_{\mathbb{Z}} v \quad \omega'_v = \omega_v \quad \omega'_o = \omega_o, v}{\langle \omega, \text{output } e \rangle \Downarrow \omega'}$$

$$\frac{\omega_v(x) \text{ is defined} \quad \omega' = \omega \text{ except } x \text{ is undef. in } \omega'_v}{\langle \omega, \text{undef}(x) \rangle \Downarrow \omega'} \quad \frac{\omega_v(x) \text{ is undef.}}{\langle \omega, \text{undef}(x) \rangle \Downarrow \omega'}$$

$$\frac{\langle \omega, \alpha \rangle \Downarrow \omega_1 \quad \langle \omega_1, \beta \rangle \Downarrow \omega_2}{\langle \omega, \alpha; \beta \rangle \Downarrow \omega_2} \quad \frac{\langle \omega, P \rangle \Downarrow_{\mathbb{B}} \top \quad \langle \omega, \alpha \rangle \Downarrow \omega'}{\langle \omega, \text{if}(P) \ \alpha \ \text{else } \beta \rangle \Downarrow \omega'} \quad \frac{\langle \omega, P \rangle \Downarrow_{\mathbb{B}} \perp \quad \langle \omega, \beta \rangle \Downarrow \omega'}{\langle \omega, \text{if}(P) \ \alpha \ \text{else } \beta \rangle \Downarrow \omega'}$$

$$\frac{\langle \omega, P \rangle \Downarrow_{\mathbb{B}} \perp}{\langle \omega, \text{while}(P) \ \alpha \rangle \Downarrow \omega} \quad \frac{\langle \omega, P \rangle \Downarrow_{\mathbb{B}} \top \quad \langle \omega, \alpha; \text{while}(P) \ \alpha \rangle \Downarrow \omega'}{\langle \omega, \text{while}(P) \ \alpha \rangle \Downarrow \omega'}$$

Figure 1: Semantics of the scripting language held over from Lab 1.

Scripts are executed by the server on behalf of the user specified in the program header, which we will assume corresponds to label ℓ_u . **Your goal in designing the information flow type system is to make sure that this user can only observe values that are derived from variables that this security lattice allows.** So for example, if the system has users Alice and Bob corresponding to labels ℓ_{Alice} and ℓ_{Bob} , respectively, then when Alice executes a script she should only be able to observe the value of expressions typed $\ell \sqsubseteq \ell_{\text{Alice}}$ but not necessarily those typed $\ell' \sqsubseteq \ell_{\text{Bob}}$. Before defining the typing rules, you are advised to create a list of all of the ways in which the interpreter might allow users to “observe” something about an expression; `output` commands are one way, but error messages returned by the server are also a type of observation that might constitute an implicit flow.

You will provide two forms of typing rules: one for expressions, and one for commands. The rules that you provide for expressions will give judgments of the form shown in Equation 3, associating each expression with a label from the security lattice.

$$\Gamma \vdash e : \ell \quad (3)$$

The rules that you provide for commands will give judgments of the form shown in Equation 4, which state that the given command is well-typed under context Γ for execution on behalf of the user with label ℓ_u .

$$\Gamma \vdash_{\ell_u} \alpha \quad (4)$$

You are encouraged to draw inspiration from the information flow type system that we studied in class. However, notice that there are command and expression forms in our scripting language that were not discussed in lecture. While you do not need to prove soundness of your type system to receive full credit for this part of the lab, it is your job to design rules for these forms that soundly enforce non-interference. You should think carefully about how information could possibly flow into program states as the result of executing each form, and set up typing rules to ensure that these flows will not violate the security lattice described above. Correctly designing rules for new program forms is the key conceptual challenge of this lab, so please work independently to reason through this part, and don't be afraid to check your work by hand on smaller example programs containing any constructs you are uncertain about.

Extra credit (10 points): prove soundness. Are you confident that your type system soundly enforces non-interference? There's only one way to know for sure—prove it! Notice that your type system must work with a more complicated lattice than the simple L/H one that we discussed in class. Come to think of it we did not define non-interference for general security lattices, so before proving soundness you will need to write down a formal definition of non-interference when labels from an arbitrary lattice appear in Γ .

The key to getting this right is to generalize the notion of state equivalence that underpins non-interference, and your formal definition should capture the following high-level intuition: *two states are ℓ -equivalent with respect to context Γ if and only if all variables x where $\ell \sqsubseteq \Gamma(x)$ are the same in both states.* Once you have formalized this more general notion of state equivalence, you should formalize non-interference as the property satisfied by programs that preserve ℓ -equivalence between initial and final states.

4 Task 2: Implement the type checker

Now that you have derived a set of typing rules to enforce non-interference, it's time to use them. You will implement everything needed to verify non-interference using types in two parts.

Given: the lattice. To make implementing the type inference rules simpler, we have provided you with the data structures for defining the lattice and some routines for managing security labels. You may add or change the struct fields if you wish, but you don't have to.

```

1 struct sec_label {
2     char *name; // human-readable name associated with label, e.g., "admin"
3 };
4 typedef struct sec_label sec_label;
5
6 struct sec_lattice {
7     sec_label* user_label;
8     ubarray* uba; // ubarray of sec_label* for pub, users, and admin
9 };
10 typedef struct sec_lattice sec_lattice;

```

Next we define some utility functions for working with labels and lattices. Your typing rules should have made use of the partial order \sqsubseteq , so you will need the function `sec_lessthan` in `sectypes.c` that encapsulates its meaning. `sec_lessthan` takes two labels `l1` and `l2`, as well as a security lattice structure `L`, and returns `true` whenever $l1 \sqsubseteq l2$ according to `L`, or `false` otherwise. Again, you can change the internal implementation of these functions if you wish, but we ask that you don't change function names or parameters.

```

1 bool sec_lessthan(sec_lattice *L, sec_label *l1, sec_label *l2) {
2     // should implement the partial order relation to return true
3     // if and only if l1 is less than (i.e., can "flow to") l2
4     // in the lattice defined by L, and false otherwise
5
6     // l1 = pub or l2 = admin
7     if((strcmp(l1->name, "pub") == 0) || (strcmp(l2->name, "admin") == 0)) return true;
8
9     // l1 = admin and l2 = pub
10    if(strcmp(l1->name, "admin") == 0 && strcmp(l2->name, "pub") == 0) return false;
11
12    // otherwise, neither l1 nor l2 is admin/pub
13    return (strcmp(l1->name, l2->name) == 0);
14 }

```

The next operation that you should expect to make use of is the least-upper-bound \sqcup defined by a given security lattice. Again located in `sectypes.c`, the function `sec_lub` takes in two labels — `l1` and `l2` — and a security lattice `L`, and returns the element $l1 \sqcup l2$.

```

1 sec_label *sec_lub(sec_lattice *L, sec_label *l1, sec_label *l2) {
2     // should implement the least-upper-bound operation to return
3     // the smallest element of L that is at least as large as both
4     // l1 and l2
5
6     // if both are pub, pub is the least upper bound
7     if (strcmp(l1->name, "pub") == 0 && strcmp(l2->name, "pub") == 0) return l1;
8

```

```

9 // if either is admin, then admin will be the least upper bound
10 if (strcmp(l1->name, "admin") == 0) return l1;
11 if (strcmp(l2->name, "admin") == 0) return l2;
12
13 // if one is pub, then the other is a user, so return the user
14 if (strcmp(l1->name, "pub") == 0) return l2;
15 if (strcmp(l2->name, "pub") == 0) return l1;
16
17 // if the names aren't equal (same user), return admin
18 if (strcmp(l1->name, l2->name) == 0) {
19     return l1;
20 } else {
21     return (sec_label*)(*ubarray_elem(L->uba, 1));
22 }
23 }

```

You may find it useful to define additional helper functions, such as ones that create fresh unpopulated lattices, fresh labels, and add labels associated with users to existing lattices. Do so at your own discretion.

Implement type inference for expressions. Since you now have everything you need to work with security labels and lattices, it is time to put them to use on expressions. For this, you will assume that you are given an already-constructed type context, and you will write a function that returns a label denoting the highest label class of information that could be carried by an expression. There is no part of this that “checks” the type against any criterion, so rather this is a form of type inference where the type of an entity whose label is not explicitly provided is derived according to your rules and the labels given in the context.

But first, you will need to construct a data type that holds the context. In the next task, you will populate this data type with label information stored in the server’s state, but for now, you can focus on writing a suitable definition and using it to implement inference and checking. In `sectypes.h`, we have provided a possible definition for the `sec_ctxt` structure.

```

1 struct sec_ctxt {
2     sec_label* pc; // the pc
3     hash_table_t* ht; // maps variable ids (char *) to indices in the lattice uba
4 };
5 typedef struct sec_ctxt sec_ctxt;

```

Now, you will need to implement the functions `type_aexp` and `type_bexp` in `sectypes.c` using your rules for expressions from the previous task, where you defined the typing system. If $\Gamma \vdash e : \ell$ under your rules, then these functions should return the `sec_label` corresponding to ℓ .

```

1 sec_label *type_aexp(sec_ctxt *G, sec_lattice *L, aexp *a) {
2     // should implement type inference using your rules for arithmetic
3     // expressions to return a label reflecting the greatest label of
4     // information that the given expression could carry under context G
5 }
6 sec_label *type_bexp(sec_ctxt *G, sec_lattice *L, bexp *b) {
7     // should implement type inference using your rules for boolean
8     // expressions to return a label reflecting the greatest label of
9     // information that the given expression could carry under context G
10 }

```

We recommend that you confirm these functions work in isolation before moving on to the next part.

Implement type checking for commands. Finally, you have everything that you need to verify that a given command satisfies non-interference with respect to a type context G and security lattice L to execute on behalf of user with security label lu . Implement a function `typecheck_com` in `sectypes.c` so that it returns `true` iff $\Gamma \vdash_{\ell_u} \alpha$ for command α according to your rules, and `false` otherwise.

```
1 bool *typecheck_com(sec_ctxt *G, sec_lattice *L, sec_label *lu, com *c) {
2     // should implement type checking for commands to verify that
3     // the script will not leak information to label lu in violation
4     // of the lattice policy L in type context G.
5 }
```

Again, we cannot stress enough that you should test this function on its own using examples for which you know the correct outcome prior to integrating the type checker with your server in the next task. The type inference and checking functions in this task are the most intricate part of the lab, and it will be much easier to identify and isolate mistakes if you set up the type context and lattice by hand, and check that the result is what you expect.

5 Task 3: Integrate the type system

Update the parser. The first thing that you will need to do to integrate information flow type checking into the server is update the parser. The lab handout contains updated versions of `lexer.l`, `parser.y`, `ast.h`, and `ast.c`. Replace these files in your current server implementation with those provided as part of the lab handout.

```
lexer.l  →  src/tinyscript/parser/lexer.l
parser.y →  src/tinyscript/parser/parser.y
ast.h    →  src/include/tinyscript/ast.h
ast.c    →  src/tinyscript/ast.c
```

Verify that your server still compiles by cleaning out your `build` subdirectory holding the makefiles emitted by `CMake`, and completing a fresh build.

Check login credentials. Now that different users hold their own data and must authenticate to access and modify it, the server will need to check provided login credentials. You should associate each username with a password, and store this information in a `passwd.db` file (you can choose where to put this). The format of the `passwd.db` file should match the format of the interpreter's database files, with space-separated username/password pairs separated by newlines `\n`. How you choose to manage this file is up to you, but one potential option is to use the `extendible_hash` data structure and its deserialization routine. When testing your server, you are responsible for creating example `passwd.db` files.

Your server should check provided username/password pairs against the contents of `passwd.db` prior to loading the specified program table, performing typechecking, or interpreting a script. If the check passes, then the server should proceed with the above three tasks. Otherwise, the interpreter should exit after producing the output:

```
Fatal Error: unauthorized access, invalid credentials
```

Populate the type context. Before you can run the type checker, you will need to populate the type context with information that associates every variable appearing in the program with a `sec_label`. In addition to the server storing values of variables in named tables, it will now also store label databases corresponding to every table. For any table `table.db`, the server should maintain a file `table.labels` that associates a security label with every variable stored in `table.db`.

The labels appearing in this file should correspond to the human-readable names contained in the `sec_label.name` field. So for example, if `table.db` contains variables `x`, `y`, and `foo`, then `table.labels` might look as follows:

```
1 x alice
2 y bob
3 foo admin
```

Prior to performing type checking, the interpreter should start by populating a fresh type context using the label associations stored in the `.labels` file corresponding to the `.db` file specified by the script. However, it may be that the script references variables that are not mentioned in the `.db` or `.labels` files, and **in these cases the context should associate the label of the user executing the script with each such variable.**

Verify before interpreting. Once you have correctly populated the type context, the server should be in a position to verify that the provided script satisfies the noninterference policy implemented in your type system. Use the `typecheck_com` function that you implemented in the previous task to ensure that the script will not leak policy-violating information to the currently logged-in user. If type checking completes successfully (i.e., returns `true`), then the server should proceed to execute the script and save the resulting state in the `.db` file given in the program header. If type checking fails, then the server should exit immediately after producing the output:

```
Fatal Error: unauthorized access, policy violation
```

Persist security labels. If the server successfully authenticates the user with the provided credentials, typechecks the given script, and interprets it to its natural termination, then it can exit successfully without producing further output. However, before doing so it must update the `.labels` database associated with the state database of the program it just interpreted. Because the script may have created new variables that are saved in the state `.db` file, the label database must be updated to reflect the type context under which the current script was checked and executed.

Test it. As you implement the interpreter, don't forget to write test cases to make sure you've done it correctly.

6 Task 4: Discussion

Write a brief report about your information flow type system and its implementations. Explain your rationale for any of the rules that you think might be counter-intuitive, or that differ non-trivially from the rules we discussed in class. If you found it necessary to make assumptions about the threat model, your implementation, or the platform that it runs on to achieve the desired security goals, briefly discuss those as well.

Finally, evaluate the protections offered by your information flow type system critically. Is this the best way to achieve the broad, high-level security goals described in the introduction and first task? Would a simple access control mechanism work as well, if not better?