Instructor: Matt Fredrikson TA: Tianyu Li

Due date: March 9 11:59pm
Total points: 100

Lab 1: Pinning Down Safety

1 Introduction

Unfortunately, it looks like your first attempt at no-so-tiny server had some serious security problems. This is especially frustrating after you spent much time making sure you code is free of memory safety issues. To make matters worse, your users have been complaining about the server's rather limited functionality—they want to do more involved computations on the server's data. Some of your colleagues have also noted that it is not a sound business move to continue providing this service for free, and have implemented a logging component that keeps track of the connections made to the server so that you can later send them a bill.

Rather than extending functionality by adding new binaries to `cgi-bin`, you will allow users to submit jobs for the server in a limited interpreted language. It has facilities for reading and writing variables, branching on conditions, looping, and displaying results. All variables are backed up by the same key-value store implemented in the last lab, so that scripts' state can be saved to disk and later retrieved to use by other scripts. Better yet, all of these programs will be interpreted by a single program that you will carefully implement to make sure there aren't any security vulnerabilities.

By confining users to this scripting language, you achieve a substantial improvement in the security of the server over the previous approach of running `cgi-bin` binaries. The language is very limited in functionality, so the potential damage is also limited. The server no longer needs to call `execve` on untrusted input, and the language's syntax places natural restrictions on the data that can end up in the server's `.db` files. Moreover, programs in the language are much easier to reason about than binary or even C programs, so there may be opportunities to develop additional safeguards using principled techniques when additional security demands arise in the future.

However, you should not be totally comfortable with this solution just yet, as the security of your server now rests on your trust in the interpreter's implementation. Especially concerning is the code responsible for interpreting untrusted scripts. You will also implement a sandbox for the interpreter to make sure that the impact of any vulnerabilities is contained. You will do this using a tool called *Pin* that instruments binary programs dynamically.

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

- Gain experience translating formal semantics to an implementation by writing an interpreter for a small domain-specific language.
- Develop a deeper understanding of safety policy enforcement techniques by implementing a sandbox that enforces security automata and (optionally) SFI.
- Gain experience with a practical code instrumentation tool (*Pin*) that is widely used in industry and research.
- Practice writing secure imperative code by extending the functionality of your server.

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 test cases will *(i)* evaluate the correctness of your interpreter absent any security considerations, and *(ii)* evaluate the extent to which your sandbox achieves the goals described in these instructions by attempting to exploit likely vulnerabilities that violate the safety policy. The point breakdown is as follows.

Correct interpreter (40 points, 5 possible extra credit). The interpreter must implement the language semantics as given in this document, producing the exact output described in these instructions. Partial credit will be given based on how many test cases passed, and in situations where very few tests yield the correct output, manual inspection.

Correct sandbox implementation (55 points, 15 possible extra credit). The sandbox implementation is mostly contained in a Pintool that instruments the interpreter's binary code, but parts of the interpreter must also be written in a way that interacts correctly with the Pintool. When running in your sandbox, the interpreter should still pass all of the functionality test cases, and terminate the interpreter in cases where the policy would be violated.

Discussion (5 points) A brief discussion of your solution, including the assumptions and other security considerations that you made when implementing it.

What to hand in. When you have completed the lab, you should hand in a `.zip` archive of the same directory tree contained in `template`, but with your completed solution filled into the appropriate files. As discussed in Section 2, most of the necessary changes are in `sandbox/em.cpp`, `src/common/extendible_hash.c`, `src/common/ubarray.c`, and `src/tinyscript/interp.c`. Make sure that the archive you hand in has your implementation in these files! **Do not hand in the entire Pin kit!** Rather, take your completed `em.cpp` from the Pin kit directory tree, and replace the template `em.cpp` with it. **Do not hand in binaries!** It is recommended to build everything in a `build` subdirectory (see Section 2.1), 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. Don't forget to hand in any test cases for the interpreter that you develop for possible extra credit (Section 3), in `src/tinyscript/testscripts`.

Important! The template code provided to get you started with your sandbox implementation makes several platform-specific assumptions about the operating system's binary executable format, as well compiler features present on the system. It has been developed and tested on the Andrew Linux cluster machines (`linux.andrew.cmu.edu`), and will be graded on the same. It may not compile or function correctly on your personal machine, and ***you are encouraged to complete this lab entirely on the Andrew Linux cluster!***

2 Getting started

The `template` directory contains template code to get you started on this lab. The following map lays out what is provided. Red files and directories are essential infrastructure that you should not modify as doing so is likely to break your implementation. Green files are those that you completed as part of the previous lab, and should replace with your code before attempting to compile. Blue files are those in which you will spend most of your time implementing the lab.

```
template
├── CMakeLists.txt ..... Build file, do not modify
├── sandbox ..... Template sandbox implementation
│   ├── em.cpp ..... Template Pintool
│   ├── makefile ..... Pin build file, do not modify
│   └── makefile.rules ..... Pin build file, do not modify
└── src ..... Template server and interpreter
    ├── common ..... Common library for server
    │   ├── CMakeLists.txt ..... Build file, do not modify
    │   ├── 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
    ├── server ..... Core server implementation
    │   ├── CMakeLists.txt ..... Build file, do not modify
    │   ├── client.c ..... Simple client to test server functionality
    │   └── tiny.c ..... Server implementation
    └── tinyscript ..... Interpreter
        ├── CMakeLists.txt ..... Build file, do not modify
        ├── ast.c ..... Functions to build abstract syntax trees
        ├── interp.c ..... Core interpreter routines
        ├── interp_main.c ..... Interpreter shell
        ├── parser ..... Parser implementation, do not modify
        │   ├── lexer.l ..... Rules for scanning strings containing programs
        │   └── parser.y ..... Grammar for language syntax
        └── testscripts ..... Example programs to test interpreter
```

2.1 Build Infrastructure

The template code uses CMake, which generates all of the necessary makefiles after ensuring that the system has the necessary dependencies. One benefit of using CMake in a larger implementation like this is that one can easily place the generated makefiles, object files, and executables in a separate directory heirarchy. After making numerous changes to the implementation, it is straightforward to start with a fresh build by deleting this directory heirarchy.

To use CMake to build this lab, enter the top-level `template` directory, and create a fresh `build` directory. Then generate the makefiles with the command `cd build && cmake ..`, which will result in a number of subdirectories of `interp/build` being created with fresh makefiles. Finally, from the `build` directory, run the `make` command. You should see approximately the following output, and if you do not then please contact the course staff immediately.

```
[user@unix template]$ mkdir build
[user@unix template]$ cd build && cmake ..
-- The C compiler identification is GNU 4.8.5
-- The CXX compiler identification is GNU 4.8.5
-- Check for working C compiler: /usr/lib64/ccache/cc
-- Check for working C compiler: /usr/lib64/ccache/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working CXX compiler: /usr/lib64/ccache/c++
-- Check for working CXX compiler: /usr/lib64/ccache/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Looking for include file pthread.h
-- Looking for include file pthread.h - found
-- Looking for pthread_create
-- Looking for pthread_create - not found
-- Looking for pthread_create in pthreads
-- Looking for pthread_create in pthreads - not found
-- Looking for pthread_create in pthread
-- Looking for pthread_create in pthread - found
-- Found Threads: TRUE
-- Found BISON: /usr/bin/bison (found version "3.0.4")
-- Found FLEX: /usr/bin/flex (found version "2.5.37")
-- Configuring done
-- Generating done
-- Build files have been written to: /afs/andrew.cmu.edu/usr8/usr/template/build
[user@unix template/build]$ make
[ 28%] Built target common
Scanning dependencies of target tinyscript
[ 35%] Building C object src/tinyscript/CMakeFiles/tinyscript.dir/interp_main.c.o
Linking C static library libtinyscript.a
[ 78%] Built target tinyscript
Scanning dependencies of target interp
[ 85%] Building C object src/tinyscript/CMakeFiles/interp.dir/interp_main.c.o
Linking C executable interp
[ 85%] Built target interp
[ 92%] Built target client
Linking C executable server
[100%] Built target server
```

Each time you build your solution, you can simply run `make` from `build` if you do not wish to build from scratch. To clean and rebuild, simply remove `template/build` and repeat this process.

2.2 Setting up Pin

Pin is a dynamic binary instrumentation framework[1]) that provides an API for creating dynamic analysis tools. Analysis tools created using Pin are called *Pintools*, and tell Pin's just-in-time compiler how to insert instrumentation into a target executable. Importantly, Pintools can be applied without recompiling the target executable.

Installing the kit. To implement the sandbox for this lab, you will need to download the appropriate Pin kit to your AFS home directory. The following commands will download the kit and move it to a directory called `pin`. It is recommended that you do this outside of the `template` directory, as you should not hand the entire Pin kit in with your final solution.

```
[user@unix ~]$ wget https://goo.gl/yyFBAh; mv yyFBAh pin.tar.gz; tar xzf pin.tar.gz
[user@unix ~]$ mv pin-3.6-97554-g31f0a167d-gcc-linux pin
```

The directory `pin` now contains the following tree, with parts that are particularly relevant highlighted and expanded.

```
pin
├── doc
├── extlicense
├── extras
├── ia32
├── intel64
├── LICENSE
├── pin ..... Pin executable, used to apply Pintools to applications
├── README
├── redist.txt
└── source
    ├── include
    ├── launcher
    └── tools ..... Directory containing Pintools
```

Building example Pintools. Enter the subdirectory `source/tools/SimpleExamples`, and execute `make` to build all of the example Pintools in this directory. Alternatively, set the environment variable PIN_ROOT to your pin distribution before running make, and the pintool will be made in the directory you called make from. If the build does not complete successfully, make sure that you are in fact running on one of the Andrew Linux machines, and if so, **contact the course staff immediately**. A successful build should end with the following output.

```
make[1]: Leaving directory '/afs/.../pin/source/tools/SimpleExamples'
```

If the examples built correctly, then you should see a directory named `obj-intel64` containing a number of `.so`, `.o`, and `.exe` files. The Pintools are in the `.so` files, and despite their Windows-ish extensions, the `.exe` files contain valid executable test applications.

[1]See the lecture notes from 2/13 for background on this technique

Running Pintools. To use a Pintool on an application, run the `pin` binary in the top-level Pin kit directory passing the `.so` of the Pintool with the flag `-t`. The target application is given on the command line after `--`, along with any arguments. Try running the `calltrace` Pintool on any program, e.g., `ls`.

```
[user@unix SimpleExamples]$ ../../../pin -t obj-intel64/calltrace.so -- ls .
```

Running `wc -l calltrace.out` when it finishes should indicate that `ls` made approximately 10,000 function calls during its execution. You can also tell Pin to follow children spawned by `execv` using the flag `-follow_execv`, which you will need to do to ensure that the interpreter spawned by `tiny` is instrumented.

Set up the template sandbox. To compile your sandbox Pintool, it is easiest to move the `template/sandbox` directory into the `pin/source/tools` directory. Then you can use the same commands to build your solution Pintool as we used to compile the tools in `SimpleExamples`, i.e., just run `make`. The tool will then be found in `pin/source/tools/sandbox/obj-intel64/em.so`. ***Note that the build infrastructure for the server and interpreter will not automatically make your Pintool!***

You may find it helpful to define a few environment variables to save keystrokes when working on both the Pintool and server.

`PINROOT` The directory containing the `pin` binary.

`SBOXROOT` The directory containing your sandbox code, i.e. `pin/source/tools/sandbox`.

`SBOXBIN` The directory containing your sandbox Pintool, i.e. `$SBOXROOT/obj-intel64`.

Finally, you probably want to set your `PATH` to point to `PINROOT`.

Useful links

Pin kit for 32 and 64-bit Linux. Tarball containing the Pin kit you should use to complete this lab on Andrew Linux machines. For convenience using `wget` to download it to your home directory over an SSH session, it can be accessed by the following shortened URL: https://goo.gl/yyFBAh.

ASPLOS 2008 Pin tutorial. Good tutorial to understand the basics of Pin. ***It is strongly recommended that you read the first 11 pages of this tutorial, as well as the section titled "Build Your Own Pin Tool – `strace` Using Pin"*** prior to starting the sandbox portion of this lab.

Pin User's Manual. The user's manual has detailed information on all of Pin's features and API routines. If you encounter problems using Pin to develop your sandbox, consult this reference before asking the course staff for assistance.

Pin main website. This website is the main trustworthy source of information about Pin. Additional tutorials and technical documentation are available here.

3 Task 1: Implement the Interpreter

Your first task after getting set up and familiarizing yourself with Pin is to implement an interpreter for a simple imperative language. The context-free grammar below is a complete syntactic description of the language you are to implement. Each rule consists of a non-terminal identifier, given in bold-face font and surrounded by angular brackets (e.g., ⟨**com**⟩), followed by one or more production rules. Elements in typewriter font (e.g., `exit`) are terminals, and correspond to keywords and punctuation required by the language.

⟨**prog**⟩	::=	`using` *table* `:` ⟨**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*

Programs begin by declaring the `.db` file containing the state on which they will execute, with the syntax `using` *table* `:` ..., following by a command containing the operations to perform on the state. Note that the filename is not in quotation marks, and is immediately preceded by a colon. Whitespace is ignored in programs. **You should also make sure that your program file always ends with a newline in the end.**

Semantics. The primary component of program state is a mapping from variables to values, much like in the languages we have discussed in lecture. There are two key differences from the states that we have discussed in lecture. First, the variable mapping is partial, and may be undefined on some variables. Second, programs in the server's scripting language can output results by writing to `stdout`. So a program state ω consists of two components, the partial variable mapping ω_v and output stream ω_s. We model the output stream as a (possibly empty) sequence of integers, i.e. $\omega_s \in \mathbb{Z}^*$.

The semantics of the language are given in Figure 1. They are given in the big-step transition style, described in lectures 10 and 11. There are three relatiions $\Downarrow_{\mathbb{Z}}, \Downarrow_{\mathbb{B}}$, and $\Downarrow$ for arithmetic

Arithmetic expressions

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

Boolean expressions

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

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

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

Commands

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

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

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

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

Figure 1: Semantics of the scripting language.

expressions, Boolean expressions, and commands, respectively. The judgement $\langle \omega, e\rangle \Downarrow_{\mathbb{Z}} v$ denotes that the arithmetic expression e evaluates to value v in state ω; $\langle \omega, P\rangle \Downarrow_{\mathbb{B}} b$ denotes that Boolean expression P evaluates to value b (i.e., either $\top$ or $\bot$) in state ω; and $\langle \omega, \alpha\rangle \Downarrow \omega'$ denotes that command α terminates in state ω when started in state ω'.

Any valid execution of a command must have a corresponding derivation from the rules in Figure 1. If no such derivation exists, then the program is "stuck" and cannot complete a valid execution. For example, if we were to attempt to evaluate the program $x := x + 1$ in a state ω in which x is undefined, then there would be no way to evaluate the right-hand side of the assignment using the rules from Figure 1; there is no rule to evaluate variables when they are undefined. **These cases amount to aborted executions.**

Reading and parsing programs. Programs are sent to the server over a network connection as strings. A client is implemented for you in the server subdirectory to handle this. When the main server loop in `tiny.c` receives a connection, it will do the following:

1. Log the connection's time and client IP to a specified log file.
2. Read the string sent and save it in a temporary file.
3. Spin up a new interpreter via `fork` and `exec` of the `interp` binary.

4. Pass the temporary filename containing the network data to the newly-created interpreter.

The entry point of the `interp` is in `interp_main.c`. It simply reads the given filename from disk into the fixed buffer `pbuf`, and calls `parse_buf` to obtain an abstract syntax tree (AST) of the target program. Note that we are making an assumption about the length of the script we are executing here. This is a simplifying assumption and it is okay to assume that the client never sends a program longer than the size of the buffer. The AST is a data structure defined in `include/tinyscript/ast.h`, and is a linked (tree) structure that contains all of the information needed to interpret a program. **The necessary functionality to parse a string containing a syntactically-correct program into an AST has been provided as part of the template, and you do not need to do anything more with this aspect of the implementation.**

Interpreting programs. Once the program is in an AST stored in variable `program`, `interp_main.c` does the following:

1. Uses the `init_state` function implemented in `interp.c` to initialize a new program state from the specified database file (`program->table`).
2. Interprets the program's command (`program->command`) using the `interp_com` function implemented in `interp.c`.
3. If the program cannot be interpreted to its natural termination, prints an error message and exits. Otherwise, writes the program state after evaluation to the same file as before (`program->table`), using `store_state` implemented in `interp.c`.

The template provides initial implementations for `init_state` and `store_state`, and a skeleton implementation of `interp_com`. **Your task is to finish the implementation of `interp_com` so that the `interp` binary correctly interprets well-formed programs that are given to it, starting in the state represented in the database file, and saves the final state back to the given database file.**

You should use the semantics in Figure 1 as your exclusive reference when implementing `interp_com` and any functions that it depends on. The big-step transition rules translate into code naturally, by implementing each syntactic construct as a set of cases corresponding to the antecedents of each rule for that construct, returning values or updating state according to the consequent. To make sure that you don't miss any cases, you may find it helpful to cross off the corresponding rule in this handout as you complete the corresponding implementation.

As you implement the interpreter, use the following guidelines to make sure that the server's behavior reflects changes to the script state.

- You are free to implement the partial variable mapping component of the state in any way that you like, but you may find it easiest to rely on your previous implementation of `extendible_hash` and `ubarray`. It is up to you to make sure these are bug-free, but you are welcome to come and see us if you think there are lingering bugs in your code preventing you from completing this lab. The template code has been set up accordingly, so if you wish to use a different approach then you should remember to update `init_state` and `store_state` accordingly so that mappings can be read from and written to disk before and after scripts execute.

- You do not need to save the output stream component of state to disk. Rather, you should ensure that changes to ω_o are reflected on a new line of `stdout` as they occur. That is, each time a new integer v is appended to ω_o, print it to `stdout` any way you wish (e.g., `printf`) with a newline at the end. The provided server and client code ensures that whatever you print out to command line in the interpreter is sent back to the client.
- You will need to account for executions that cannot proceed according to the big-step transition rules, i.e. those that abort. For example, attempts to read a variable that is not defined in the current state. **For any such situation in which there is no valid way to proceed with execution, the interpreter should stop immediately, and `interp_com` should return a negative code to signify aborted execution.**
- If memory resources are exhausted while executing a command, then the interpreter should stop its executing and `interp_com` should return a positive (non-zero) code to signify early termination for reasons other than aborted execution.

Preventing DoS. You don't want a user-provided script to exhaust resources on the server indefinitely, thus mounting a denial-of-service attack on all the other users. You should take steps to ensure that the interpreter stops executing divergent scripts rather than looping forever. When the interpreter terminates execution for this reason, `interp_com` should return a positive (non-zero) status code. **You can implement this protection however you like, but your implementation should allow scripts to execute either until they terminate naturally, and otherwise for at least 10,000 commands.**

Test it. As you implement the interpreter, don't forget to write test cases to make sure you've done it correctly. Whenever your implementation fails a test, it is likely that someone else's implementation will as well, but please do not share your test cases with other students. Instead, hand your test cases in with your final solution, naming them `andrewid_test1.tiny`, `andrewid_test2.tiny`, ..., and placing them in `template/src/tinyscript/testscripts`. These should be well-formed programs that can be run directly as arguments to the interpreter. Handin an additional Readme file if you think a test case is tricky in anyway and needs further explanation. **For each of your tests that surfaces a bug in another student's implementation, you will receive one point of extra credit on this lab (up to 5 points).**

Scope of the task. At first glance, this might seem like a daunting amount of work. However, it need not be, and the key to a good solution lies in not making the solution more complicated than necessary. If your code in `interp.c` exceeds 500-600 lines (including the 350 lines with comments provided in the template), then please get in touch with the course staff, as we may be able to help simplify your solution.

4 Task 2: Sandbox the Interpreter

Having implemented the interpreter, it's time to make sure that it doesn't let bad things happen on the server. You will use Pin to implement a sandbox that enforces two safety policies: *(i)* a system call monitor, and (optionally) *(ii)* software fault isolation. **If you have not already followed the instructions in Section 2.2 and followed the tutorial linked at the end of that section, then do so now before starting on this task.**

4.1 System call monitor

Modern operating systems that run on conventional hardware, such as Linux and Windows, mediate programs' access to disk, network, display, and most other devices. There are numerous good reasons to architect systems in this way, but one of the practical upshots of doing so is that the impact of malicious programs that are under the influence of an attacker can be more easily contained. If the operating system were to block access to all devices, then the only impact an attack in a user's program could have would be to change memory and registers, and these are discarded when the program terminates anyway.

System calls are the mechanism by which programs interact with the operating system. Any attempt to read or write files or sockets, manage memory, or use other devices must be sent to the operating system by way of a system call. If you implemented it correctly, your interpreter will make use of the following system calls.

File operations: Obviously the interpreter needs to `open`, `read`, `write`, and `close` files, and printing to `stdout` is also implemented as writing to a file (with descriptor `0x1`). These are all system calls. Although you may not have used them explicitly in your code, the standard library will also use `stat`, `fstat`, `lstat`, and `poll`, which return information about a file descriptor or wait for an event on one. Finally, the interpreter may also use `lseek` to change the position of the read/write pointer associated with a file descriptor.

Memory management: The operating system manages the virtual address space for processes. One of the most common system calls made is `brk`, which changes the size of the data segment to give the program more heap space. Programs also use `mmap` to map the contents of a file to memory, and `munmap` to release such memory. Finally, `mprotect` changes the protection flags on virtual memory. The interpreter may make use of all these system calls as it executes a script.

Before the interpreter enters `main`, however, the loader will execute other syscalls on behalf of the program to populate memory with the executable's code and that of any libraries it uses. This will involve the `exec` and `access` system calls.

Armed with the knowledge of how the interpreter should use system calls as it goes about its business, your task is to implement a Pintool that makes sure the interpreter never strays from its intended correct syscall behavior. You should extend the template in `template/sandbox/em.cpp` to complete this part of the lab.

Part 1: Whitelist system calls. Given the above discussion, there is no good reason for the interpreter to ever make a system call that is not in the following list after it has entered `main`:

$$\texttt{open}, \texttt{read}, \texttt{write}, \texttt{close}, \texttt{stat}, \texttt{fstat}, \texttt{lstat}, \texttt{poll}, \texttt{brk}, \texttt{mmap}, \texttt{munmap}, \texttt{mprotect} \quad (1)$$

Your system call monitor should implement the following safety policy: *after entering the `main` function in the `interp` image, the program does not execute system calls other than those in (1).* To accomplish this, the tool will need to keep track of whether the `main` in `interp` has been entered, as well as each system call that the program makes as it executes.

In order to implement the first part which tracks entry to `main`, your tool should scan all of the routines in each image that is loaded, and use the `RTN_Name` and `IMG_Name` functions to determine when to instrument the appropriate routine in the correct image. On finding the right routine, you can instrument the entry and exit point of the function body using Pin's `RTN_InsertCall` API, which places instrumentation that invokes a callback function inside your Pintool each time the routine is entered or exited. This callback can maintain the necessary state to enforce the policy. You should consult the documentation and tutorials listed in Section 2.2 to learn more about setting up these callback functions in your Pintool.

To implement the second part you should make use of Pin's system call API, which has been simplified and improved since the writing of the ASPLOS 2008 tutorial referenced in Section 2.2. You can use the `PIN_AddSyscallEntryFunction` and `PIN_AddSyscallExitFunction` to insert a callback to your tool whenever a system call begins and ends executing, respectively. The signature of the entry and exit callbacks is:

```
VOID Callback(THREADID thrIndex, CONTEXT *ctxt, SYSCALL_STANDARD std, VOID *v)
```

Where the arguments mean the following:

`thrIndex`: The Pin thread ID of the thread that executes the system call. You do not need to worry about this, as the interpreter is single-threaded.

`ctxt`: The target application's register state immediately before execution of the system call.

`std`: The calling standard of the system call instruction.

`v`: A value defined by the tool to pass to the callback. You can probably ignore this, unless you think of something useful to put in this value.

Inside the callback, your monitor will need to examine the system call number, which identifies the function being invoked, to determine its membership in (1). You can obtain this with `PIN_GetSyscallNumber`. A mapping from system call numbers to their corresponding names can be found on `github.com/torvalds`, and is also defined in `<sys/syscall.h>`. Generally, including this file will let you refer to system call numbers as `SYS_name`, so for example `SYS_open` is defined to the number for `open` on the platform the tool is compiled for.

Finally, if the policy is violated, the monitor should terminate its target by calling `exit`.

Part 2: Stateful monitoring. Implementing the whitelist safety policy from Part 1 will prevent someone from exploiting an unknown vulnerability, and then for example calling `execve` to load a totally different executable from somewhere on the system, or opening new network connections to

wait for further instructions or transmit confidential information. But certain behaviors are still permitted, such as opening and overwriting files on the server. We need to refine our safety policy further to prevent a potential attack through these channels.

Make your syscall monitor more restrictive by implementing a safety policy that restricts the file operations performed by the interpreter. In more detail, implement a policy that gives the interpreter only what it needs in terms of file operations:

- Before executing any other file operations, the interpreter is allowed to open the file created by the server containing the script to execute. It must only do so in mode `O_RDONLY`.
- While the script file is open, the only file operations permitted are `fstat`, `lseek`, `read`, and `close`. The file descriptor used in these syscalls must be the same one returned by the `open` on the script file.
- After closing the script file, it can open a `.db` file in the interpreter's directory, and it must use mode `O_RDONLY|O_CREAT`.
- While the database file is open, it may not write to any files, and it can only read from the descriptor given from calling `open` on the `.db` file.
- After closing the `.db` file, the only file operations permitted are `write` and `open`. Moreover, all `write` calls must be to file descriptor `0x1`, and the only allowed `open` call is to the same `.db` file as before in mode `O_WRONLY|O_TRUNC`.
- After calling `open` on the `.db` file for the second time, the only file operations permitted are `fstat`, `lseek`, `write`, and `close`. The file descriptor used in these syscalls must be the same one returned by the most recent `open` on the `.db` file.

You may find it helpful to write this policy out as a security automaton before implementing it in code, both as a sanity check that you have understood the policy and as a guide for when you implement.

In order to implement this policy, you will need to inspect the arguments passed to system calls. You can do so with the `PIN_GetSyscallArgument` API, which takes the current register context, the system call convention, and the argument number. You can find more information on which arguments to inspect by consulting the relevant man pages. In general, file operation syscalls take file descriptors in argument 0, except for `open` which takes a C string denoting the filename in that argument.

For arguments that contain pointers, your Pintool can access memory at the corresponding location by casting an appropriate pointer variable. For example, to inspect the buffer in argument 0 of `open`, the following code will suffice.

```
VOID Callback(THREADID thrIndex, CONTEXT *ctxt, SYSCALL_STANDARD std, VOID *v) {
  if(PIN_GetSyscallNumber(ctxt, std) == SYS_open) {
    char *fname = (char *)PIN_GetSyscallArgument(ctxt, std, 0);
    if(!strcmp(fname, "/etc/passwd")) { ... }
  }
}
```

Note that you may not have needed to use `PIN_AddSyscallExitFunction` to implement Part 1, but you will need to use it here because system call return values are not available on entry, only

on exit. You will need to examine the return value of certain syscalls to keep track of returned file descriptors, and can do so with `PIN_GetSyscallReturn`.

When your monitor detects a policy violation, it should terminate the application as in Part 1. Your final solution should enforce both policies from Parts 1 and 2.

4.2 Software Fault Isolation (Optional, Extra Credit: 15 points)

Pin is a powerful tool for analyzing and modifying the runtime behavior of binary applications. It would be a shame to stop improving the security of the interpreter with the system call monitor, as there are more possible attacks that this policy does not address. For example, the interpreter does not need indiscriminate access to memory once it has been given a program, so perhaps we want to make sure that while the interpreter is executing a script, its memory accesses all stay inside a pre-defined sandbox region.

The build script provided in the template sets up a memory sandbox at addresses `0x18000000` – `0x18ffffff`, which is managed by the memory allocation routines in `safemem.c`. Additionally, the parser, scanner, and setup code in `interp_main.c` have been designed to make use of this sandbox, so that all memory allocations made in those files will be confined to the sandbox region. All that remains is to make small modifications to the functions involved with managing the variable mapping, to ensure that they also use `malloc`, `calloc`, and `free` as implemented in `safemem.c`, and to extend the Pintool to confine memory accesses appropriately.

For extra credit, complete this functionality in your sandbox execution monitor, `em.cpp`. In particular, make sure that when the primary interpreter function `interp_com` executes, none of the memory reads or writes that it makes, with the few exceptions described below, access addresses outside the range `0x18000000` – `0x18ffffff`. Use the instrumentation techniques discussed in class for SFI to rewrite all memory operands so that this policy is never violated.

- On Intel64 architectures, compilers generally manage function-local variables on the stack pointed to by `esp`. Because your interpreter most likely uses function arguments and local variables, your execution monitor will need to make an exception for accesses to stack memory.
- Some high-level language constructs, such as C's `switch` statement, make use of read-only memory outside of the main code section (`.text` on most Unix systems) to store the targets of indirect jumps. This will result in legitimate memory reads outside of the sandbox, so you may need to make exceptions for these cases.
- If your implementation of `interp_com` makes use of libraries, then it may invoke code that accesses memory outside the sandbox. In practice, depending on the degree to which such libraries are trusted, this may warrant an exception to the SFI policy. For this assignment, you can assume that such libraries are trusted, and make an exception when library code accesses memory outside the specified range.

Because this part of the lab is extra credit, the course staff will not answer all questions about this task. We will answer questions about the specification of this particular SFI policy, but we will not point you to the Pin API functions needed to implement it or help you debug this portion of the assignment. If you choose to implement this policy, then you should consult the API documentation, and also look for example Pintools that provide functionality similar to the components needed to enforce this policy.

5 Task 3: Discussion

Write a brief report on your experience using PIN and discuss what you have achieved. Please address the following points.

- What threat model do you need to assume in order for the sandbox to provide meaningful protection? That is, what restrictions must you place on the attacker's capabilities in terms of making network connections, permissions on the server, filesystem rights, and anything else that could possibly invalidate your sandbox?
- What specific assumptions must be made about the interpreter implementation to provide reasonable security for the server? Given that your sandbox will run alongside the interpreter, does it matter if the interpreter was written by a sleep-deprived developer who knows nothing about security?
- What components, both within the server implementation as well as the operating system running it, do you need to trust for your security guarantee to be warranted?
- Can you think of other security issues that might arise that are not covered by the safety policy you implemented in `em.cpp`?

You do not need to write more than three or four paragraphs for this part of the lab. Additionally, if one of your test cases from Section 3 mounts an attack on the server that is not prevented by the security policy, please document it here with a summary of the steps leading to attack for another point of extra credit (for up to **two** points on that test case!).

Your report does not need to consist of more than 3-5 paragraphs, and should be handed in in `template/report.txt` (or `template/report.pdf` if you choose to typeset it).