Secure Typed Programming Languages
Communication

Andrew Myers

June 19, 2019

1 Jif

Started off with a Jif code example (`Hello.jif`) which can be found in the Jif distribution.

```
import java.io.PrintStream;
import jif.runtime.Runtime;

// compile with -fail-on-exception
public class Hello
{
    public Hello() {}
    public static void main{}(principal{ p p  } pp,
                              String[]{} args)
        throws (SecurityException, NullPointerException)
    {
        Runtime runtime = Runtime[pp].getRuntime();
        PrintStream out = runtime.out();
        out.println("Hello, world!");
    }
}
```

Principals are first-class values that can be used to express security policies. In the example we see that the `Runtime` class is parameterized by the principal pp. From principals we can build labels to enforce confidentiality (limit the principals that the value can flow to) and integrity (limit the principals that can influence the value).

In the language that was defined in class we handled illegal flows due to control flow using a program counter label, which determines which variables are allowed to be written to depending on the label of the boolean that determines the direction of control flow. Since Java has exceptions, Jif needs to handle control flow due to exceptions too.

Jif supports dynamic checking of labels: if we check that a label L can flow to L2 with a dynamic check, then code is allowed to let values with label L flow to values with label L2 in the body of the if.

```
int{L}x;
int{L2}y;
if (L <= L2) { y = x; }
```

The second example is the game Battleship, a board game in which the players should not be able to see each other's ships. The example demonstrates several classes parameterized by principals and labels. Lists are parameterized by the label that elements of the lists have, and lists also have a label for the structure of the list itself. We can refer to the label of a variable using that variable; a list labeled with this will get the same label as this.

```
import java.io.PrintStream;

/**
 * The class BattleShip is responsible for coordinating gameplay
 * between two Players, P1 and P2.
 */
class BattleShip[principal P1, principal P2] authority (P1, P2) {
    /**
     * The number of covered coordinates that each player is to
     * have on their board. Players can have any number of ships,
     * so long as the ships cover this number of squares.
     */
    public static final int{*<-*} NUM_COVERED_COORDS = 10;

    public void play{P1<-* meet P2<-*}(PrintStream[{}]{P1<-* meet P2<-*} output)
      throws (SecurityException, IllegalArgumentException{P1<-* meet P2<-*})
      where authority (P1, P2) {

        if (output == null) throw new IllegalArgumentException("Null output");


        output.println("Playing battleships, with each player having "
                       + NUM_COVERED_COORDS + " covered coordinates");

        // instantiate the two players
        Player[P1,P2] player1 = new Player[P1,P2]();
        Player[P2,P1] player2 = new Player[P2,P1]();



        // Initialize the two players.
        output.print("  Initializing....");

        // Player 1 first creates a board
        Board[{P1->*;P1<-*}] proposed1 = player1.init(NUM_COVERED_COORDS);

        // Player 2 endorses it (since Player 2 doesn't care where the
        // pieces are located).
        Board[{P1->*;P1<-* meet P2<-*}] accepted1 = player2.endorseBoard(proposed1);

        // Player 1 stores the endorsed board.
        player1.storeBoard(accepted1);

        // Similarly for Player 2: create a board, Player 1 endorses, and board
        // is stored.
```

```
Board[{P2->*;P2<-*}] proposed2 = player2.init(NUM_COVERED_COORDS);
Board[{P2->*;P2<-* meet P1<-*}] accepted2 = player1.endorseBoard(proposed2);
player2.storeBoard(accepted2);

output.println("␣Done.");

// These fields record how many hits each player has scored.
// The game continues until one of the players has
// scored NUM_COVERED_COORDS hits.
int player1Hits = 0;
int player2Hits = 0;

output.println("␣␣Playing␣rounds...");

// loop until a player hits all the covered co-ordinates.
while (player1Hits < NUM_COVERED_COORDS && player2Hits < NUM_COVERED_COORDS) {

    // get player 1's query
    Coordinate[{P1<-*}] play1Query = player1.getNextQuery();

    // Player 1's query is endorsed by Player 2.
    Coordinate[{P1<-* meet P2<-*}] play1QueryEnd = player2.endorseQuery(play1Query);

    output.print("\t"+PrincipalUtil.toString(P1)+":␣" +
                 (play1QueryEnd==null?"null":play1QueryEnd.toString()) +
                 "?␣␣" );

    // Player 2 processes Player 1's query
    boolean result = player2.processQuery(play1QueryEnd);
    player1Hits += result ? 1 : 0;
    output.print((result?"Y":"N"));


    if (player1Hits < NUM_COVERED_COORDS) {
        // player 1 hasn't won, so let player 2 ask a query...
        Coordinate[{P2<-*}] play2Query = player2.getNextQuery();

        //player 1 endorse's the query
        Coordinate[{P1<-* meet P2<-*}] play2QueryEnd = player1.endorseQuery(play2Query);

        output.print("␣␣␣"+PrincipalUtil.toString(P2)+":␣" +
                     (play2QueryEnd==null?"null":play2QueryEnd.toString())
                     + "?␣␣");

        // get player 1 to process player 2's query
```

```
                boolean result2 = player1.processQuery(play2QueryEnd);

                player2Hits += result2 ? 1 : 0;
                output.print((result2?"Y":"N"));

                // print a running total of the scores...
                output.println("␣␣␣Score:␣" + player1Hits + "␣vs.␣" + player2Hits);
            }
        }

        // Let's see who won...
        output.println("\n");
        output.println((player1Hits >= NUM_COVERED_COORDS ? PrincipalUtil.toString(P1)
                                                          : PrincipalUtil.toString(P2))
                       + "␣won!");
    }
}
```

Jif has a limited form of dependent types to handle first class principals. It also supports downgrading; endorse and declassify, to relax the integrity and confidentiality.

2 Secure Program Partitioning

Distributed systems are difficult to write. Secure program partitioning in Swift [1] and Jif/split [2] attempts to alleviate this by allowing client-server applications to be written as a single program rather than as a separate client and server. The program can automatically be partitioned to run as a distributed system. The Swift system for web applications compiles the server part of the program to Java and the client UI part of the program to Javascript. The compiler synthesizes a protocol with which the server and the client communicate with each other. The Jif compiler checks the integrity and confidentiality of the program.

The splitter ensures that secret data is kept on the server and not sent to the client. Trusted computation is done on the server, and optionally by the client for performance reasons, but that computation must be verified on the server. An example is web form validation: the client side Javascript can verify that an email address field contains a syntactically valid email address, but since the server does not trust the client, the server must redo that check. It is still worthwhile to do the check on the client to give feedback to the user with lower latency.

We can also describe a trust configuration, which the splitter can use to generate different code. Therefore Swift can improve productivity as well as security; we do not need to manually rewrite the program to take trust configuration changes into account.

3 Showing Enforcement of Noninterference

Now, we seek to prove that a type system enforces termination insensitive noninterference. This was originally defined in Volpano's '96 paper [3], which shows that certifying a program for secure information flow can be done with proofs of noninterference. That is, all well-typed programs enforce noninterference.

We recall that for a given state s_1 and s_2, noninterference means that $s_1 \sim_L s_2 \Rightarrow [\![s_1]\!] \approx_L [\![s_2]\!]$. Furthermore, we define a state $s = (c, \gamma)$ where $\gamma : \text{var} \to \mathbb{Z}$ is a store that contains the values of all variables.
Let L be an arbitrary lattice element, then

$$\gamma_1 \sim_L \gamma_2 \Leftrightarrow \forall x \in Var.\Gamma(x) \sqsubseteq L \Rightarrow \gamma_1(x) = \gamma_2(x)$$

Intuitively, this says that two stores are low-related if the low variables in the stores have the same values. then we can write, with c_1 being our command and γ_1 being our state:

$$(c_1, \gamma_1) \sim_L (c_2, \gamma_2) \overset{\triangle}{\Leftrightarrow} \gamma_1 \sim_L \gamma_2$$

Now, if we assume big-step semantics, we can take a state $(c, \gamma) \Downarrow \gamma'$, and only get a new state if our command is convergent.

We now define our program behavior as

$$[\![s]\!] = [\![(c, \gamma)]\!] = \begin{cases} \gamma' \ \texttt{if} \ (c, \gamma) \Downarrow \gamma' \\ \bot \ \texttt{otherwise} \end{cases}$$

Thus, we note that

$$[\![s_1]\!] \approx_L [\![s_2]\!] \begin{cases} \texttt{if} \ [\![s_1]\!] = \bot \wedge [\![s_2]\!] = \bot \\ \texttt{or} \ [\![s_1]\!] = \gamma_1 \wedge [\![s_2]\!] = \gamma_2 \wedge \gamma_1 \sim_L \gamma_2 \end{cases}$$

Then, we can write our termination insensitive noninterference statement as follows

$$\gamma_1 \sim_L \gamma_2 \wedge pc \vdash c \wedge ((c_1, \gamma_1) \Downarrow \gamma_1' \wedge (c_2, \gamma_2) \Downarrow \gamma_2' \Rightarrow \gamma_1' \sim_L \gamma_2')$$

We can prove our noninterference statement on derivation of $c, \gamma_1 \Downarrow \gamma_1'$ via a series of cases.

Our first case is the trivial case of *skip*

$$\gamma_1' = \gamma_1 \ \texttt{and} \ \gamma_2' = \gamma$$

This trivially follows to be

$$\gamma_1' \sim_L \gamma_2'$$

Our next case is $c = (x := a)$ and we define as follows.

$$\Gamma(x) \not\sqsubseteq L$$

$$\gamma_1' = \gamma_1[x \Rightarrow v_1] \text{ and } \gamma_2' = \gamma_2[x \Rightarrow v_2]$$

Therefore, $\gamma_1' \sim_L \gamma_2'$, which follows because of the low equivalences.
Our next case is $x := a$ and we define it as

$$\Gamma(x) \sqsubseteq L, pc \sqsubseteq L, \vdash a : l, l \sqsubseteq L$$

To prove this, we need a typing rule to ensure a only uses low variables which are the same in both expressions. This lemma is written below.

$$\vdash a : l \wedge l \sqsubseteq L \wedge \gamma_1 \sim_L \gamma_2 \Rightarrow a \text{ has the same values in } \gamma_1 \text{ and } \gamma_2$$

This lemma means that

$$\gamma_1' = \gamma_1[x \Rightarrow v]$$
$$\gamma_2' = \gamma_2[x \Rightarrow v]$$
$$\gamma_1' \sim_L \gamma_2'$$

Another case is that of sequential composition, where $c = c_1; c_2$ If c evaluates to a final state, then c_1 and c_2 can be evaluated to a final state. The technique of big step evaluation says that

$$(c_1, \gamma_1) \Downarrow \gamma_1'' \text{ and } (c_2, \gamma_1'') \Downarrow \gamma_1'$$

and

$$(c_1, \gamma_2) \Downarrow \gamma_2'' \text{ and } (c_2, \gamma_2'') \Downarrow \gamma_2'$$

then by induction both $\gamma_1'' \sim_L \gamma_2''$ and $\gamma_1' \sim_L \gamma_2'$ hold.

Another example is the case of while. $c =$ `while` b `do` c' We can define this as a series of cases.
Our first case is where case where b evaluates to false and is low.

$$b, \gamma \Downarrow \texttt{false} \text{ and } \vdash b : l \text{ and } l \sqsubseteq L$$

This is trivial and we note that $\gamma_1' = \gamma_1$, $\gamma_2' = \gamma_2$.

The next case is where b evaluates to true and is low, e.g, $(b, \gamma) \Downarrow$ `true` . We start in state γ and after evaluating the body once, we end up in γ'', and after the loop finishes we end up in γ' so we can represent this state transition as follows.

$$(c', \gamma) \Downarrow \gamma''$$
$$\texttt{while } b \texttt{ do } c', \gamma'' \Downarrow \gamma'$$

By induction, we get the following: $\gamma_1'' \sim_L \gamma_2''$ and $\gamma_1' \sim_L \gamma_2'$.

We leave the case of where b and while are not low as an exercise to the reader. The key to proving these is an additional *high-pc* lemma, which says that execution of a command that type-checks in a high pc cannot cause low-observable effects to the store:

$$pc \vdash c \ \wedge \ pc \not\sqsubseteq L \ \wedge \ c, \gamma \Downarrow \gamma' \ \Rightarrow \ \gamma \sim_L \gamma'$$

This lemma is easily proved using induction on the evaluation of commands.
We also leave the case of `if` commands to the reader.

4 Modeling Nondeterminism

To handle nondeterminism (e.g., arising from concurrency), we need a small-step semantics rather than a big-step semantics. Small-step semantics make it difficult to deal with the program counter going back down at the end of high-security contexts introduced by `if` and `while` commands. In general, the two executions of the program that we're comparing may end up executing copmletely different commands. For example, a command `if h then c1 else c2` , so we need to define what it means for two states with different commands to be related: $(c_1, \gamma_1) \sim_L (c_2, \gamma_2)$. To do this we need to add label tracking to the small step semantics, and prove that the new semantics faithfully simulates the original semantics.

In the proof by induction on the small steps we have two kinds of cases: steps in which the program counter is low and steps in which the program counter is high. For the low cases we have to prove that if we start in two related states $(c_1, \gamma_1) \sim_L (c_2, \gamma_2)$, then the states $(c_1', \gamma_1') \sim_L (c_2', \gamma_2')$ that they step to are also related. For the high cases we know that

the commands preserve indistinguishability because commands with a high program counter cannot affect the low variables: if $(c_1, \gamma_1) \sim_L (c_2, \gamma_2)$ and (c_1, γ_1) steps to (c_1', γ_1'), then we prove that $(c_2, \gamma_2) \sim_L (c_1', \gamma_1')$.

Small step proofs are more work than big step proofs. Pottier and Simonet [4] invented a different method to deal with small step proofs called "brackets". They define a new semantics that simultaneously executes the two programs, and marks the places in which the programs differ with brackets. States look like $c = \_\_\_ < c_1, c_2 > \_\_\_$, where the lines denote parts of the state that are the same in both (the parts with a low pc), and the brackets indicate parts of the state that are different in both (the parts with a high pc). If we show that the new semantics simulates the execution of two separate programs, then type soundness for the new semantics gives us noninterference. This proof strategy recovers some of the ease of the proof with the big step semantics.

References

[1] Stephen Chong, Jed Liu, Andrew C Myers, Xin Qi, Krishnaprasad Vikram, Lantian Zheng, and Xin Zheng. Secure web applications via automatic partitioning. *ACM SIGOPS Operating Systems Review*, 41(6):31–44, 2007.

[2] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C Myers. Untrusted hosts and confidentiality: Secure program partitioning. *ACM SIGOPS Operating Systems Review*, 35(5):1–14, 2001.

[3] Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. A sound type system for secure flow analysis. *Journal of computer security*, 4(2-3):167–187, 1996.

[4] François Pottier and Vincent Simonet. Information flow inference for ml. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 25(1):117–158, 2003.