

Operational Semantics for BoolInt* Language

Yuri Tatishchev
San José State University
iurii.tatishchev@sjsu.edu

Abstract. In this paper, we will provide a review of the big-step operational semantics for the BoolInt* language we discussed in class. We extend the language with a `secret` construct and security labels (H for private, L for public) to track information flow and prevent secret leakage.

BoolInt* is a very minimal language that allows us to experiment with operational semantics.

First, we define the valid expressions in our language. These expressions dictate the possibilities of what expressions we may have in our source programs. (Note that other language might also have *statements*. Statements might not evaluate to a value; expressions always will.)

$e ::=$		<i>Expressions</i>
	<code>true</code>	true value
	<code>false</code>	false value
	i	integers
	<code>if e then e else e</code>	conditional expressions
	<code>succ e</code>	successor
	<code>pred e</code>	predecessor
	<code>secret e</code>	secret
$v ::=$		<i>Values</i>
	<code>true</code>	true value
	<code>false</code>	false value
	i	integers
$\ell ::=$		<i>Security Labels</i>
	L	low / public
	H	high / private

Figure 1: The BoolInt* language with security labels

Figure 1 shows the list of expressions, values, and security labels for the BoolInt* language. Expressions can be the value `true`, the value `false`, integers i , the conditional expression `if e then e else e` , arithmetic operations `succ e` and `pred e` , and the security construct `secret e` . Note that conditional expressions have a recursive structure, with 3 sub-expressions.

After evaluating a program, we produce a *labeled value* (v, ℓ) , where v is the value and ℓ is a security label indicating whether the result is public (L) or private (H). (In other languages, we might hit a bad situation and need to crash instead; that won't happen in this language.) The valid values for BoolInt* are `true`, `false`, and i .

We define a *join* operation \sqcup on security labels as follows:

$$L \sqcup L = L \quad L \sqcup H = H \sqcup L = H \sqcup H = H$$

Intuitively, any interaction with a private value taints the result as private.

With our expressions and values defined, we can now specify the semantics for our language. To do so, we will use the following big-step evaluation relation:

$$e \Downarrow (v, \ell)$$

The above line should be read as “the expression e evaluates to the value v with security label ℓ ”.

Evaluation Rules: $e \Downarrow (v, \ell)$

$$[\text{B-VALUE}] \quad \frac{\emptyset}{v \Downarrow (v, L)}$$

$$[\text{B-SECRET}] \quad \frac{e \Downarrow (v, \ell)}{\text{secret } e \Downarrow (v, H)}$$

$$[\text{B-IFTRUE}] \quad \frac{e_1 \Downarrow (\text{true}, \ell_1) \quad e_2 \Downarrow (v, \ell_2)}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \Downarrow (v, \ell_1 \sqcup \ell_2)}$$

$$[\text{B-IFFALSE}] \quad \frac{e_1 \Downarrow (\text{false}, \ell_1) \quad e_3 \Downarrow (v, \ell_3)}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \Downarrow (v, \ell_1 \sqcup \ell_3)}$$

$$[\text{B-SUCC}] \quad \frac{e \Downarrow (i, \ell)}{\text{succ } e \Downarrow (i + 1, \ell)}$$

$$[\text{B-PRED}] \quad \frac{e \Downarrow (i, \ell)}{\text{pred } e \Downarrow (i - 1, \ell)}$$

Figure 2: Big-step semantics for BoolInt* with information flow

Figure 2 shows the big-step evaluation rules for the BoolInt* language with information flow tracking.

The [B-VALUE] rule applies when the expression (to the left of “ \Downarrow ”) is also a value, as defined in Figure 1. There are no premises for this rule (above the line), meaning that it is an *axiom*. This rule states that a bare value evaluates to itself with a public label L, so that `true` evaluates to `(true, L)` and `3` evaluates to `(3, L)`.

The [B-SECRET] rule handles the `secret` construct. It evaluates the sub-expression e and forces the security label to H, regardless of the original label. For example, `secret 3` evaluates to `(3, H)`.

Two different rules are needed for handling conditional expressions. Which rule applies depends on the premises. Critically, both rules *join* the condition’s label ℓ_1 with the branch result’s label.

For the [B-IFTRUE] rule, the premise states that e_1 evaluates to `(true, ℓ_1)` and e_2 evaluates to `(v, ℓ_2)`. The result carries $\ell_1 \sqcup \ell_2$ as its label. The structure of [B-IFFALSE] is similar.

This join is essential for preventing *implicit flows*. Consider the expression `if secret true then 1 else 0`. Without the join, this would produce `(1, L)` — leaking which branch was taken and thus revealing the secret condition. With the join, the condition evaluates to `(true, H)`, so the result becomes `(1, H \sqcup L) = (1, H)`, correctly tainting the output as private.

We also have rules for handling the integer operations. The [B-Succ] rule states that if the operand e evaluates to (i, ℓ) , then the expression `succ e` evaluates to $(i + 1, \ell)$. This propagates the security label faithfully. For example, `succ secret 3` evaluates to $(4, H)$, whereas `succ 3` evaluates to $(4, L)$. Similarly, [B-PRED] evaluates `pred e` to $(i - 1, \ell)$ if e evaluates to (i, ℓ) .

In summary, under these semantics, there is no expression an attacker can write that produces an L-labeled value whose content depends on any H-labeled (secret) input. The join in the conditional rules ensures that even *implicit* information flows through branch selection are tracked and labeled private.