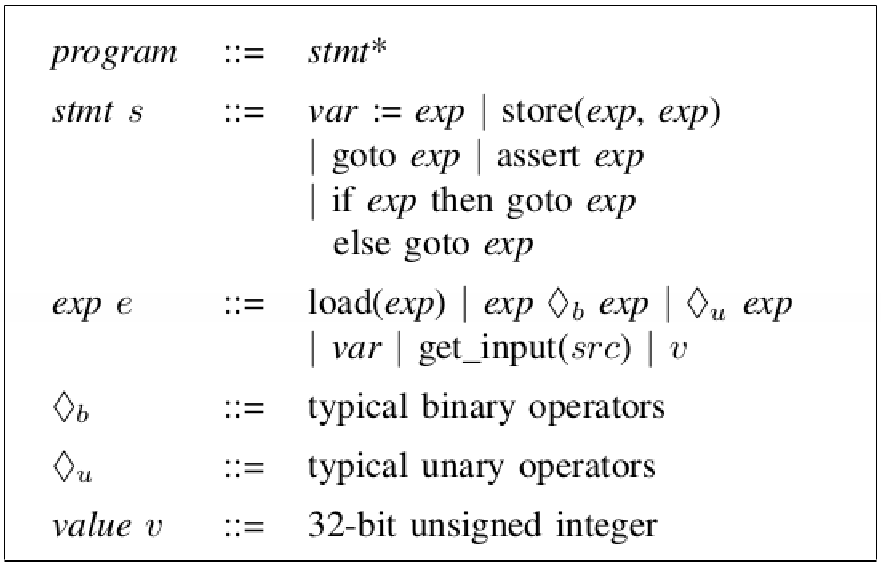
Dynamic Taint Analysis and Forward Symbolic Execution

A General Language

Overview

A simple intermediate language(`SIMPIL`), which is powerful enough to express typical languages such as Java and assembly code.



program	::=	*stmt**
stmt s	::=	*var* := *exp* | store(*exp*, *exp*) | goto *exp* | assert *exp* | if *exp* then goto *exp* else goto *exp*
exp e	::=	load(*exp*) | *exp* $\lozenge_b$ *exp* | $\lozenge_u$ *exp* | *var* | get_input(*src*) | v
$\lozenge_b$	::=	typical binary operators
$\lozenge_u$	::=	typical unary operators
value v	::=	32-bit unsigned integer

Statements in `SIMPIL` consist of assignments, assertions, jumps and conditional jumps.

Execution context

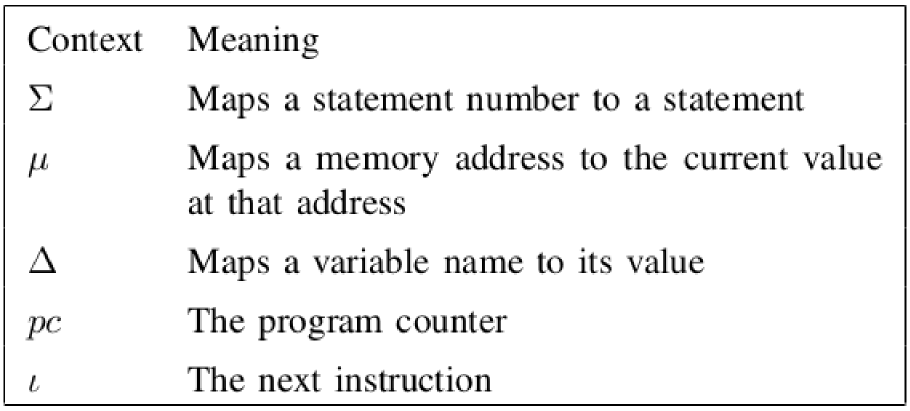
Σ is the list of program statements

μ is the current memory state

Δ is the current value for variables

pc is the program counter

l is the current statement



Context	Meaning
Σ	Maps a statement number to a statement
μ	Maps a memory address to the current value at that address
Δ	Maps a variable name to its value
pc	The program counter
ι	The next instruction

Operational Semantics

Each statement rule is of the form of `propositional logic`:

```
            computation
-----------------------------------  RULE TYPE
current state,stmt -> end state,stmt
```

If the statement matches the `RULE TYPE`, then it does the `computation` and the state will transform.

We use $\mu, \Delta \vdash e \Downarrow v$ to evaluate an expression e to a value v in the current state given by μ and Δ .

We use $\Delta' = \Delta[x \to 10]$ to set x as 10 and get a new context.

$$\frac{v \text{ is input from } src}{\mu, \Delta \vdash \text{get\_input}(src) \Downarrow v}\ \textsc{Input} \qquad \frac{\mu, \Delta \vdash e \Downarrow v_1 \quad v = \mu[v_1]}{\mu, \Delta \vdash \text{load } e \Downarrow v}\ \textsc{Load} \qquad \frac{}{\mu, \Delta \vdash var \Downarrow \Delta[var]}\ \textsc{Var}$$

$$\frac{\mu, \Delta \vdash e \Downarrow v \quad v' = \Diamond_u v}{\mu, \Delta \vdash \Diamond_u e \Downarrow v'}\ \textsc{Unop} \qquad \frac{\mu, \Delta \vdash e_1 \Downarrow v_1 \quad \mu, \Delta \vdash e_2 \Downarrow v_2 \quad v' = v_1 \Diamond_b v_2}{\mu, \Delta \vdash e_1 \Diamond_b e_2 \Downarrow v'}\ \textsc{Binop} \qquad \frac{}{\mu, \Delta \vdash v \Downarrow v}\ \textsc{Const}$$

$$\frac{\mu, \Delta \vdash e \Downarrow v \quad \Delta' = \Delta[var \leftarrow v] \quad \iota = \Sigma[pc+1]}{\Sigma, \mu, \Delta, pc, var := e \rightsquigarrow \Sigma, \mu, \Delta', pc+1, \iota}\ \textsc{Assign} \qquad \frac{\mu, \Delta \vdash e \Downarrow v_1 \quad \iota = \Sigma[v_1]}{\Sigma, \mu, \Delta, pc, \text{goto } e \rightsquigarrow \Sigma, \mu, \Delta, v_1, \iota}\ \textsc{Goto}$$

$$\frac{\mu, \Delta \vdash e \Downarrow 1 \quad \Delta \vdash e_1 \Downarrow v_1 \quad \iota = \Sigma[v_1]}{\Sigma, \mu, \Delta, pc, \text{if } e \text{ then goto } e_1 \text{ else goto } e_2 \rightsquigarrow \Sigma, \mu, \Delta, v_1, \iota}\ \text{TC}\textsc{ond}$$

$$\frac{\mu, \Delta, \vdash e \Downarrow 0 \quad \Delta \vdash e_2 \Downarrow v_2 \quad \iota = \Sigma[v_2]}{\Sigma, \mu, \Delta, pc, \text{if } e \text{ then goto } e_1 \text{ else goto } e_2 \rightsquigarrow \Sigma, \mu, \Delta, v_2, \iota}\ \text{FC}\textsc{ond}$$

$$\frac{\mu, \Delta \vdash e_1 \Downarrow v_1 \quad \mu, \Delta \vdash e_2 \Downarrow v_2 \quad \iota = \Sigma[pc+1] \quad \mu' = \mu[v_1 \leftarrow v_2]}{\Sigma, \mu, \Delta, pc, \text{store}(e_1,\ e_2) \rightsquigarrow \Sigma, \mu', \Delta, pc+1, \iota}\ \textsc{Store}$$

$$\frac{\mu, \Delta \vdash e \Downarrow 1 \quad \iota = \Sigma[pc+1]}{\Sigma, \mu, \Delta, pc, \text{assert}(e) \rightsquigarrow \Sigma, \mu, \Delta, pc+1, \iota}\ \textsc{Assert}$$

For example, consider the statement of

```
x = 2 * get_input();
```

We first use `CONST` rule and `INPUT` rule,

Then we use `BINOP` rule and finally we use `ASSIGN` rule to finish.

$$\cfrac{\cfrac{\cfrac{}{\mu, \Delta \vdash 2 \Downarrow 2}\ \textsc{Const} \quad \cfrac{20 \text{ is input}}{\mu, \Delta \vdash \text{get\_input}(\cdot) \Downarrow 20}\ \textsc{Input} \quad v' = 2 * 20}{\mu, \Delta \vdash 2\text{*get\_input}(\cdot) \Downarrow 40}\ \textsc{Binop} \quad \Delta' = \Delta[x \leftarrow 40] \quad \iota = \Sigma[pc+1]}{\Sigma, \mu, \Delta, pc, x := 2\text{*get\_input}(\cdot) \rightsquigarrow \Sigma, \mu, \Delta', pc+1, \iota}\ \textsc{Assign}$$

High-level semantics

We can deal with high-level semantics in two ways.

- We can compile the missing high-level language constructs down to `SIMPIL`.

  For example, we can compile function calls down to `SIMPIL` by storing the return address and transferring control flow.

```
 1  /* Caller function */
 2  esp := esp + 4
 3  store(esp, 6) /* retaddr is 6 */
 4  goto 9
 5  /* The call will return here */
 6  halt
 7
 8  /* Callee function */
 9  ...
10  goto load(esp)
```

- We can add high-level semantics into `SIMPIL` to enhance analysis.

  For example, the semantics of `CALL`, `RET` and `DYNAMICALLY GENERATED CODE` are shown below:

$$\frac{\mu, \Delta \vdash e_1 \Downarrow v_1 \quad \ldots \quad \mu, \Delta \vdash e_i \Downarrow v_i \quad \Delta' = \Delta[x_1 \leftarrow v_1, \ldots, x_i \leftarrow v_i] \quad pc' = \phi[f] \quad \iota = \Sigma[pc']}{\lambda, \Sigma, \phi, \mu, \Delta, \zeta, pc, \text{call f}(e_1,\ldots,e_i) \rightsquigarrow (pc+1) :: \lambda, \Sigma, \phi, \mu, \Delta', \Delta :: \zeta, pc', \iota}\ \textsc{Call}$$

$$\frac{\iota = \Sigma[pc']}{pc' :: \lambda', \Sigma, \phi, \mu, \Delta, \Delta' :: \zeta', pc, \text{return} \rightsquigarrow \lambda', \Sigma, \phi, \mu, \Delta', \zeta', pc', \iota}\ \textsc{Ret}$$

$$\frac{\mu, \Delta \vdash e \Downarrow v \quad v \notin dom(\Sigma) \quad s = \text{disassemble}(\mu[\text{v}]) \quad \Sigma' = \Sigma[v \leftarrow s] \quad \iota = \Sigma'[v]}{\Sigma, \mu, \Delta, pc, \text{jmp } e \rightsquigarrow \Sigma', \mu, \Delta, v, \iota}\ \text{G}\textsc{en}\text{C}\textsc{ode}$$

Dynamic taint analysis

Semantics

Since dynamic taint analysis is performed on code at runtime, it is natural to add dynamic taint analysis semantics.

To keep track of the taint status, we refines values in `SIMPIL` as $< v, \tau >$, where v is a value in the initial language and τ is the taint status of v.

taint t	::=	**T** | **F**
value	::=	$\langle v, t\rangle$
τ_Δ	::=	Maps variables to taint status
τ_μ	::=	Maps addresses to taint status

So we can add policies to enable dynamic taint analysis in `SIMPIL`.

$$\frac{v \text{ is input from } src}{\tau_\mu, \tau_\Delta, \mu, \Delta \vdash \text{get\_input}(src) \Downarrow \langle v, P_{\mathbf{input}}(\text{src})\rangle}\ \text{T-INPUT} \qquad \frac{}{\tau_\mu, \tau_\Delta, \mu, \Delta \vdash v \Downarrow \langle v, P_{\mathbf{const}}()\rangle}\ \text{T-CONST}$$

$$\frac{}{\tau_\mu, \tau_\Delta, \mu, \Delta \vdash var \Downarrow \langle \Delta[var], \tau_\Delta[var]\rangle}\ \text{T-VAR} \qquad \frac{\tau_\mu, \tau_\Delta, \mu, \Delta \vdash e \Downarrow \langle v, t\rangle}{\tau_\mu, \tau_\Delta, \mu, \Delta \vdash \text{load } e \Downarrow \langle \mu[v], P_{\mathbf{mem}}(t, \tau_\mu[v])\rangle}\ \text{T-LOAD}$$

$$\frac{\tau_\mu, \tau_\Delta, \mu, \Delta \vdash e \Downarrow \langle v, t\rangle}{\tau_\mu, \tau_\Delta, \mu, \Delta \vdash \lozenge_u e \Downarrow \langle \lozenge_u v, P_{\mathbf{unop}}(t)\rangle}\ \text{T-UNOP}$$

$$\frac{\tau_\mu, \tau_\Delta, \mu, \Delta \vdash e_1 \Downarrow \langle v_1, t_1\rangle \quad \tau_\mu, \tau_\Delta, \mu, \Delta \vdash e_2 \Downarrow \langle v_2, t_2\rangle \quad P_{\mathbf{bincheck}}(t_1, t_2, v_1, v_2, \lozenge_b) = \mathbf{T}}{\tau_\mu, \tau_\Delta, \mu, \Delta \vdash e_1 \lozenge_b e_2 \Downarrow \langle v_1 \lozenge_b v_2, P_{\mathbf{binop}}(t_1, t_2)\rangle}\ \text{T-BINOP}$$

$$\frac{\tau_\mu, \tau_\Delta, \mu, \Delta \vdash e \Downarrow \langle v, t\rangle \quad \Delta' = \Delta[var \leftarrow v] \quad \tau'_\Delta = \tau_\Delta[var \leftarrow P_{\mathbf{assign}}(t)] \quad \iota = \Sigma[pc+1]}{\tau_\mu, \tau_\Delta, \Sigma, \mu, \Delta, pc, var := e \rightsquigarrow \tau_\mu, \tau'_\Delta, \Sigma, \mu, \Delta', pc+1, \iota}\ \text{T-ASSIGN}$$

$$\frac{\begin{array}{c}\iota = \Sigma[pc+1] \quad P_{\mathbf{memcheck}}(t_1, t_2) = \mathbf{T} \\ \tau_\mu, \tau_\Delta, \mu, \Delta \vdash e_1 \Downarrow \langle v_1, t_1\rangle \quad \tau_\mu, \tau_\Delta, \mu, \Delta \vdash e_2 \Downarrow \langle v_2, t_2\rangle \quad \mu' = \mu[v_1 \leftarrow v_2] \quad \tau'_\mu = \tau_\mu[v_1 \leftarrow P_{\mathbf{mem}}(t_1, t_2)]\end{array}}{\tau_\mu, \tau_\Delta, \Sigma, \mu, \Delta, pc, \text{store}(e_1,\ e_2) \rightsquigarrow \tau'_\mu, \tau_\Delta, \Sigma, \mu', \Delta, pc+1, \iota}\ \text{T-STORE}$$

$$\frac{\tau_\mu, \tau_\Delta, \mu, \Delta \vdash e \Downarrow \langle 1, t\rangle \quad \iota = \Sigma[pc+1]}{\tau_\mu, \tau_\Delta, \Sigma, \mu, \Delta, pc, \text{assert}(e) \rightsquigarrow \tau_\mu, \tau_\Delta, \Sigma, \mu, \Delta, pc+1, \iota}\ \text{T-ASSERT}$$

$$\frac{\tau_\mu, \tau_\Delta, \mu, \Delta \vdash e \Downarrow \langle 1, t_1\rangle \quad \tau_\mu, \tau_\Delta, \mu, \Delta \vdash e_1 \Downarrow \langle v_1, t_2\rangle \quad P_{\mathbf{condcheck}}(t_1, t_2) = \mathbf{T} \quad \iota = \Sigma[v_1]}{\tau_\mu, \tau_\Delta, \Sigma, \mu, \Delta, pc, \text{if } e \text{ then goto } e_1 \text{ else goto } e_2 \rightsquigarrow \tau_\mu, \tau_\Delta, \Sigma, \mu, \Delta, v_1, \iota}\ \text{T-TCOND}$$

$$\frac{\tau_\mu, \tau_\Delta, \mu, \Delta \vdash e \Downarrow \langle 0, t_1\rangle \quad \tau_\mu, \tau_\Delta, \mu, \Delta \vdash e_2 \Downarrow \langle v_2, t_2\rangle \quad P_{\mathbf{condcheck}}(t_1, t_2) = \mathbf{T} \quad \iota = \Sigma[v_2]}{\tau_\mu, \tau_\Delta, \Sigma, \mu, \Delta, pc, \text{if } e \text{ then goto } e_1 \text{ else goto } e_2 \rightsquigarrow \tau_\mu, \tau_\Delta, \Sigma, \mu, \Delta, v_2, \iota}\ \text{T-FCOND}$$

$$\frac{\tau_\mu, \tau_\Delta, \mu, \Delta \vdash e \Downarrow \langle v_1, t\rangle \quad P_{\mathbf{gotocheck}}(t) = \mathbf{T} \quad \iota = \Sigma[v_1]}{\tau_\mu, \tau_\Delta, \Sigma, \mu, \Delta, pc, \text{goto } e \rightsquigarrow \tau_\mu, \tau_\Delta, \Sigma, \mu, \Delta, v_1, \iota}\ \text{T-GOTO}$$

Policy

Taint Introduction

It specifies how taint is introduced into a system.

We initialize all variables, memory cells as untainted.

In `SIMPIL`, we only have a single source of user input: `get_input()`

A taint policy will also distinguish between different input sources.

Taint Propagation

It specifies the taint status for data derived from tainted or untainted operands.

Taint Checking

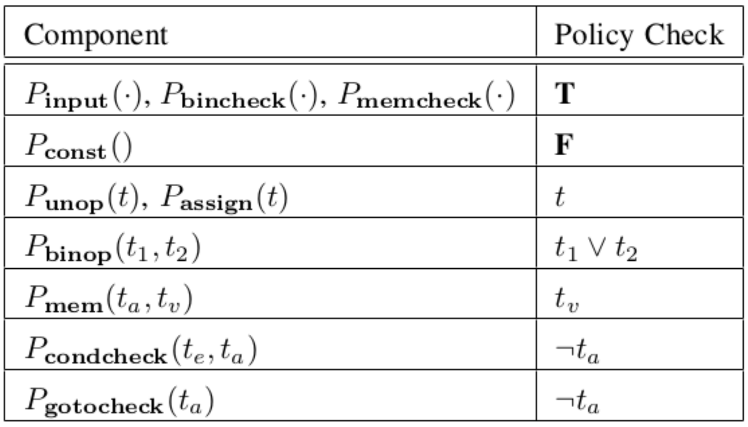
In `SIMPIL`,we perform checking by adding the policy to the premise of the operational semantics.

For example, `T-GOTO` rule uses $P_{gotocheck}(t)$ policy, which returns T if it is safe to perform a jump operation when the target is safe to perform a jump operation when the target address has taint value t.

A Typical Taint Policy

`Tainted Jump Policy`, which aims to protect a potentially vulnerable program from control flow hijacking attacks.

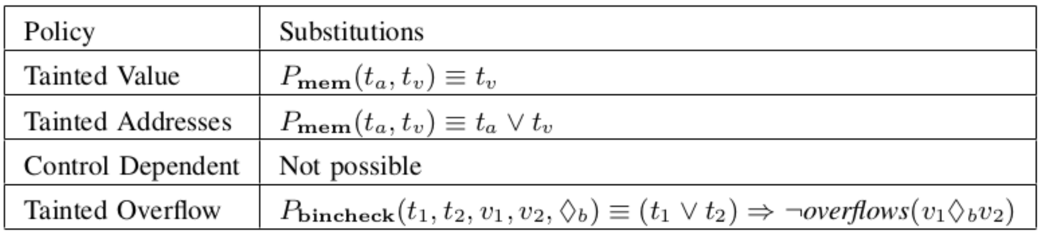
The main idea is that an input-derived value will never overwrite a control-flow value such as a return address or function pointer.



Component	Policy Check
$P_{\mathbf{input}}(\cdot)$, $P_{\mathbf{bincheck}}(\cdot)$, $P_{\mathbf{memcheck}}(\cdot)$	**T**
$P_{\mathbf{const}}()$	**F**
$P_{\mathbf{unop}}(t)$, $P_{\mathbf{assign}}(t)$	t
$P_{\mathbf{binop}}(t_1, t_2)$	$t_1 \vee t_2$
$P_{\mathbf{mem}}(t_a, t_v)$	t_v
$P_{\mathbf{condcheck}}(t_e, t_a)$	$\neg t_a$
$P_{\mathbf{gotocheck}}(t_a)$	$\neg t_a$

This policy introduces taint into the system by marking all values returned by `get_input()` as tainted.

To fix the problem of undertainting address injection, we can use `Tainted Address Policy`, where a memory cell is tainted if either the value or address is tainted.



Policy	Substitutions
Tainted Value	$P_{\mathbf{mem}}(t_a, t_v) \equiv t_v$
Tainted Addresses	$P_{\mathbf{mem}}(t_a, t_v) \equiv t_a \vee t_v$
Control Dependent	Not possible
Tainted Overflow	$P_{\mathbf{bincheck}}(t_1, t_2, v_1, v_2, \lozenge_b) \equiv (t_1 \vee t_2) \Rightarrow \neg \mathit{overflows}(v_1 \lozenge_b v_2)$

Control-flow Taint

Unfortunately, pure dynamic taint analysis cannot compute control dependencies, because reasoning about control dependencies requires reasoning about multiple paths, and dynamic analysis executes on a single path at a time.

Sanitization

`Taint Sanitization Problem` identifies when taint can be removed from a value.

For example, we want to know when the program computes constant functions(e.g. if $b = a \oplus a$, then $b = 0$ regardless of a)

Forward Symbolic Execution

We can reduce the reasoning about the behavior of the program to the domain of logic.

Applications and Advantages

Multiple inputs

It can be used to reason about more than one input class at once.

Semantics of Forward Symbolic Execution

When `get_input()` is evaluated symbolically, it returns a `symbol` instead of a `concrete value`.

When a new `symbol` returns, there is no constrain on it.

So `SIMPIL` should be modified as

value v	::=	32-bit unsigned integer | *exp*
Π	::=	Contains the current constraints on symbolic variables due to path choices

Branches constrain the values of symbolic variables to the set of values that would execute the path, and the updated semantics is shown below:

$$\frac{v \text{ is a fresh symbol}}{\mu, \Delta \vdash \text{get\_input}(\cdot) \Downarrow v}\ \textsc{S-Input}$$

$$\frac{\mu, \Delta \vdash e \Downarrow e' \quad \Pi' = \Pi \wedge e' \quad \iota = \Sigma[pc+1]}{\Pi, \Sigma, \mu, \Delta, pc, \text{assert}(e) \rightsquigarrow \Pi', \Sigma, \mu, \Delta, pc+1, \iota}\ \textsc{S-Assert}$$

$$\frac{\mu, \Delta \vdash e \Downarrow e' \quad \Delta \vdash e_1 \Downarrow v_1 \quad \Pi' = \Pi \wedge (e' = 1) \quad \iota = \Sigma[v_1]}{\Pi, \Sigma, \mu, \Delta, pc, \text{if } e \text{ then goto } e_1 \text{ else goto } e_2 \rightsquigarrow \Pi', \Sigma, \mu, \Delta, v_1, \iota}\ \textsc{S-TCond}$$

$$\frac{\mu, \Delta, \vdash e \Downarrow e' \quad \Delta \vdash e_2 \Downarrow v_2 \quad \Pi' = \Pi \wedge (e' = 0) \quad \iota = \Sigma[v_2]}{\Pi, \Sigma, \mu, \Delta, pc, \text{if } e \text{ then goto } e_1 \text{ else goto } e_2 \rightsquigarrow \Pi', \Sigma, \mu, \Delta, v_2, \iota}\ \textsc{S-FCond}$$

Symbolic Memory Addresses

In `LOAD` and `STORE` rules, it uses a integer value to reference a particular memory cell.

The `Symbolic Memory Address` problem arises whenever an address referenced in a `LOAD` or a `STORE` operation is an expression instead of a concrete value.

- Load: a sound strategy is to consider it a load from any possible satisfying assignment for the expression.

- Store: a store could overwrite any value for a satisfying assignment to the expression.

Also, symbolic memory addresses can lead to `aliasing` issues.

There are several ways to deal with symbolic reference:

- Make unsound assumptions: transform the address cell as variables.
- Let subsequent analysis deal with them.(We may use `SMT` solver to reason about the aliasing problem)
- perform alias analysis

Path Selection

When Forward Symbolic Execution encounters a branch, it must decide which branch to follow first.

It can be viewed as a tree-search problem. The analysis begins with only a root node in the tree. Then it add the forked states as children to the current node.

Forward Symbolic execution needs a strategy for choosing which state to explore next(e.g. `DFS`, `Concolic Testing`, `Random Paths` and `Heuristics`).

Symbolics Jumps

The jump target may be an expression instead of a concrete location.

Handling System and Library Calls

Some system-level calls introduce fresh symbolic variables and also have additional side effects.

- We can create summaries to describe the side effect
- Concolic execution to use values returned from system calls on previous concrete execution

Performance

A straightforward implementation of forward symbolic execution has

- $T = O(2^b)$, where T is the running time and b is the number of program branches
- $f = O(2^n)$, where f is the number of formulas and n is the number of program line
- an exponentially-sized formula per branch

So we can mitigate the problems in three ways:

- Exploiting parallelism
- Giving each variable assignment a unique name
- Eliminating the redundancies between formulas

Mixed Execution

Allowing some inputs to be concrete and others symbolic is called `mixed execution`.