Formal model of program execution, symbolic, dynamic symbolic execution (concolic execution), and taint tracking

Sergey Vartanov

me@enzet.ru, svartanov@ispras.ru

March 22, 2019

Abstract

The primary objective of this document is to propose a formal definition of some program analysis terms, such as execution (concrete execution), (pure) symbolic execution, dynamic symbolic execution (concolic execution), feasibility, and taint analysis.

Concrete program execution

Program definition

We will start with a program definition. In order to separate program execution from symbolic and dynamic symbolic execution, which will be discussed later, we will use terms *concrete program* and *concrete program execution* since it processes concrete data.

We want to start from the classical definition of a *computational method* from Donald Knuth's *The Art of Computer Programming* [1]. Computational method is $\langle Q, I, \Omega, f \rangle$, where $f : Q \to Q$, Q is states[1], Ω is terminal states, I is input states. In this model execution of a computational method $\forall x \in I$ is $x_0, x_1, \ldots, x_k, \ldots$, where $x_0 = x, x_{k+1} = f(x_k)$. For convenience, we will use slightly different notation because it will interfere with our further considerations (superscript will be used to distinguish different program definitions):

$$\mathcal{P}^{\mathrm{K}} = \langle \mathcal{F}, D, D_I, D_T \rangle, \tag{1}$$

where $\mathcal{F} : D \to D$ is an *execution function*, D is states, D_I is initial states, D_T is terminal states. This definition seems to be concise and comprehensive. It has no any assumptions about the nature of states D and its limitations. However, since we want to highlight some particular aspects of modern programs and computer systems, we will add some constraints and expand this definition with more complex constructions.

Split execution function

Suppose that execution function $\mathcal{F}$ is a piecewise-defined function and it is defined for sets $s_0, s_1, \ldots, s_m, \ldots$ and $D = s_0 \cup s_1 \cup \cdots \cup s_m \cup \ldots, \forall i, j \; i \neq j \Rightarrow s_i \cap s_j = \varnothing$.

$$\mathcal{F}(d) = \begin{cases} f_0(d), & \text{if } d \in s_0, \\ f_1(d), & \text{if } d \in s_1, \\ \ldots & \\ f_m(d), & \text{if } d \in s_m, \\ \ldots \end{cases} \tag{2}$$

[1] Note, that there is no requrement, that all states should be reachable. We can have state $q \in Q$ that is not initial and for all initial states q is not in it's execution.

It is important that sets are disjoint because we have to know current program state at each point in time. We will call these functions *execution function elements* or for short *function elements*. Also, we can treat sets s_i as state. And here we have to clarify what is meant by a term *state*. We have to turn to the definition of Turing's machine [2]. There are two different meanings of a term *state*.

1. First meaning is the one used in Knuth's definition. It is a whole program execution state in some particular moment of time. In [2] it also called state of progress, state of the system, or state formula. For that meaning we will use term ***process state***. For the process state we will use small letter d, for the set of process states—capital letter D.
2. The second one is the one we used. In [2] it also called record of m-configuration, value of state register, or designator of instruction. For that meaning we will use term ***program state***. For the program state we will use small letter s[2], for the set of program states—capital letter S.

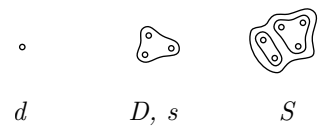

Figure 1: Process state and program state specifiers.

It is importaint to note that function elements and states were defined in such a way that two statements "next function element to be executed is f_i" and "current state is s_i" mean the same. For convenience, we will require that $s_0 = D_I$ so we can treat s_0 as an initial program state and that there is a superset $\{s_j\}$ that form D_T so we can treat them as terminal program states: $S_T = D_T = \bigcup_j s_j$. Now we can propose another definition of the program:

$$\mathcal{P}^1 = \langle F, D, S, s_0, S_T \rangle, \tag{3}$$

where $F = \{f_i\}, f_i : D \to D$ is a set of function elements, D is a set of all possible process states, $S = \{s_j\}$ is a set of program states, s_0 is a initial program state, $S_T = \{s_k\}$ is a set of terminal program states. Note that current program state defines the next function element to be used to proceed program execution. Therefore we can define initial function element instead of initial program state, and terminal function elements instead of terminal program states:

$$\mathcal{P}^2 = \langle F, D, S, f_0, F_T \rangle \tag{4}$$

where f_0 is a initial function element, $F_T = \{f_k\}$ is a set of terminal function elements.

Here F can be treated as a set of program *statements* but not as *instruction set*, because they may share semantics but work for different states.

Program counter

Assume as well that every process state d_i can be divided into two parts: $d_i = (d_i^D, d_i^F)$. So now there are two sets $D^D = \{d_i^D\}$ and $D^F = \{d_i^F\}$, where the second part only defines its state. I.e. $\exists g : g(d_i^F) = j \Leftrightarrow d_i \in s_j$. In that case we can say that d_i^F is a *program state designator*. It can be treated as *instruction pointer* or *program counter*.

Split data flow and control flow

We can a bit redefine function elements and make them return new process state **and** next function element: $\mathcal{P} = \langle F, F_I, F_T, D \rangle$, where function elements and data are defined as follows:

- $F = \{f_j\} = \{(f_j^F, f_j^D)\}$ is a *function element set*. $f_j : s_j \to F \times D$ is a *function element*. $f_j = (f_j^F, f_j^D)$:

[2] D and s are both sets of d and formally the same. But we will use them in different situations.

- $f_j^F : s_j \to F$ is *control flow function element*.
- $f_j^D : s_j \to D$ is *data flow function element*.

Function element could be considered as program instruction, that computes next program instruction $\{f_j^F\}$ and modifies data $\{f_j^D\}$.

- F_I is a set of *initial function elements*.
- F_T is a set of *terminal function elements*.
- $D = \{d_j\}$ is a set of possible *data* values.

Additionally, we will assume the following.

- Function element set is not empty: $F \neq \varnothing$.
- $\forall j \ f_j^F$ is a total function.
  - If $f_j \notin F_T$, $\forall d \in D \ f_j^F(d) = f_k, f_k \in F$.
  - If $f_j \in F_T$, $\forall d \in D \ f_j^F(d) = f_j$.
- To simplify considering, we will assume that unless otherwise stated there is one and only one initial function element f_0: $F_I = \{f_0\}$.
- Program has at least one terminal function element: $F_T \neq \varnothing$.

Limitations

We assume **deterministic** program execution using **limited** binary data without **self-modification**. Concrete program execution means normal program execution on given binary data.

- Firstly, we should emphasize that we distinguish data (D) from function elements (F). It implies that self-modification is not possible. To allow self-modification, we should merge F and D into one set.
- Program is deterministic: $f_j^F : D \to F$. For nondeterministic program it should be: $f_j^F : D \to F*$.
- Data is limited and predefined: $|D| < \infty$.

Possible data model definition

Data set can be homogenous or heterogenous. In Knuth's examples data set is heterogenous (different vector sizes). In real-world programs data is set homogenous.

Data set D is a set of all possible values that can be arguments of function elements f_j: $D = \{d_j\}$. Data $d \in D$ is an abstract entity and it can be anything. The simplest data definition is a set of values: $d = \{d^j\}, d \in D$ (we will use superscript for data elements because it is more convenient to use subscript for data instances), where d^j is a Boolean variable, Boolean vector, natural, integer, real number, or anything else.

As we want to model real computer programs we can notice, that simplest definition of data as a Boolean variables is useful enough for majority of computer systems, which operates limited binary memory.

However it is convenient to use more complicated data definitions to simplify modeling.

Binary data:

- $B = (\mathbf{false}, \mathbf{true}), \mathcal{B}_2 = B^8$.

Data model definition with registers and memory:

- $D_1 = \{d_j\} = R \cup M$.

(x, y)	$f^D_\wedge(x, y)$
(**false**, **false**)	(**false**, **true**)
(**false**, **true**)	(**false**, **true**)
(**true**, **false**)	(**false**, **true**)
(**true**, **true**)	(**true**, **true**)

Table 1: Conjunction truth table.

- $R = \{r_j\}$ — *concrete registers*. $r_j \in \mathcal{B}_2$.
- $M = \{m_j\}$ — *concrete memory*. $m_j \in \mathcal{B}_2$.

For the next data model definition we will **avoid limited data condition**. Here Y is a set of sets Y_i. Each set Y_i may contain an arbitrary number of elements. Data model definition with set of inputs, registers, memory, and external knowledge about array allocation:

- $D_2 = \{d_j\} = Y \cup R \cup M \cup L$. $|L| \leq |M|$.
  - $Y = \{Y_i\}$ — *concrete inputs*, $Y_i = \{y_{i,j}\}$ — *concrete input*. $y_{i,j} \in \mathcal{B}_2$.
  - $R = \{r_j\}$ — *concrete registers*. $r_j \in \mathcal{B}_2$.
  - $M = \{m_j\}$ — *concrete memory*. $m_j \in \mathcal{B}_2$.
  - $L = \{l_j\}$ — *concrete array lengths*. $l_j \in \mathcal{B}_2$.

Control flow graph

Set of all possible next function elements for function element f is $f[D] = \{f_j \in F | f_j = f^F_j(d), d \in D\}$. *Control flow graph* is a directed graph where vertices are function elements and there is an edge from vertex f_i to vertex f_j iff $f_j \in f_i[D]$. Function element $f_j = (f^F_j, f^D_j)$ is *branch* iff $|f_j[D]| > 1$. Otherwise (if range of the control flow function element has exactly one element), function element is *operation*, i.e. iff $\exists f_k : f^D_j(d) = f_k, \forall d \in D$.

The set of all branches is $F_B = \{f_j \in F || f_j[D]| > 1\}$. The set of all operations is $F_O = \{f_j \in F || f_j[D]| = 1\}$. Since $|f_j[D]|$ by definition is either equals to 1, or greater than 1, $F = F_B \cup F_O$ and $F_B \cap F_O = \varnothing$.

We will use graphical representation of control flow graph using circles for vertices with function element captions and arrows for edges. Vertices that correspond to terminal function elements will be represented by circles with double stroke. Vertices that correspond to initial function elements will be represented as usual. We assume that either there is a single initial element f_0, or initial elements are represented by vertices that has no incoming edges.

Basic blocks are sets of continue...

Semantics

Concrete semantics is a set of all data function elements f^D_j used in program: F^D. In other words, concrete semantics defines possible ways to transform data in program. It could be considered as an instruction set, or a system of commands.

Semantics cannot be defined without data D definition. For example, let's take a look at logical conjunction. It is a binary logical operator, which means it uses two Boolean operands and returns Boolean result. But to define an operation $f^D_\wedge$, that performs logical conjunction, we should also define data D and how this operation transform data. Let's define its semantics as $f^D_\wedge(x, y) = (x \wedge y, y)$, where $x, y \in B$. It could be represented in the form of assignment $x \leftarrow x \wedge y$. It could also be represented in the form of truth table (see Table).

Execution

We consider that program execution is an iterative process with steps. For further considerations we will use variable $t \in \mathbb{N}$ as *execution step*. At every step program execution has its *program state* $s = (f, d)$: next function element (next instruction) and current *data state*. $f \in F, d \in D$. Set of all possible execution states is S. Program state at the iteration t is $s_t = (f_t, d_t) = \big((f_t^F, f_t^D), d_t\big)$. $(f_0, d_0), f_0 \in F_I$ is *initial state*. Any state (f_t, d_t) such as $f_t \in F_T$ is *terminal state*.

We consider that there is *execution function* $\mathcal{F} : S \to S$ that manages the execution process:

$$\mathcal{F}(s_t) = \mathcal{F}\big((f_t, d_t)\big) = f_t(d_t) = \big(f_t^F(d_t), f_t^D(d_t)\big) = (f_{t+1}, d_{t+1}) = s_{t+1}.$$

Execution function $\mathcal{F}$ defines an *execution rule*:

- $f_{t+1} = f_t^F(d_t)$,
- $d_{t+1} = f_t^D(d_t)$.

Execution of a program $\mathcal{P} = \langle F, F_I, F_T, D\rangle$ on *input data* d_0, where $F_I = \{f_0\}$, is a chain of program states, starting with initial state $s_0 = (f_0, d_0)$, where next state is a function $\mathcal{F}$ of previous state: $s_{t+1} = \mathcal{F}(s_t), s_t \in S$. Therefore execution can be represented as iterative use of execution function: $\mathcal{F}(\mathcal{F}(\dots\mathcal{F}(d_0)\dots))$.

We consider that if program execution approaches state $s_t = (f_t, d_t)$, where $f_t \in F_T$, program execution is terminated. Hence, execution is either infinite chain

$$E(d_0) = s_0 \to s_1 \to \dots \to s_t \to \dots = (f_0, d_0) \to (f_1, d_1) \to \dots \to (f_t, d_t) \to \dots,$$

where $\forall j\ f_j \notin F_T$ (in this case d_t is an *output data* of program execution $E(d_0)$), or finite chain

$$E(d_0) = s_0 \to s_1 \to \dots \to s_t = (f_0, d_0) \to (f_1, d_1) \to \dots \to (f_t, d_t),$$

where $f_t \in F_T \wedge \forall j < t\ f_j \notin F_T$ (in this case execution has no output data). For the first case we will use notation $E(d_0) \to d_t$, for the second: $E(d_0) \to \infty$.

Note, that execution is a sequence of states: $E(d_0) = (s_0, s_1, \dots, s_t, \dots)$. But we will use notation $E(d_0) = s_0 \to s_1 \to \dots \to s_t \to \dots$ to emphasize that it is a process.

Lemma 1. $\forall \mathcal{P} = \langle F, F_I, F_T, D\rangle\ \forall d \in D\ \exists! E(d)$.

Proof. Let's prove it by constructing the execution chain. By the program definition, there is a initial function element f_0. Hence, $\forall d \in D$ there is a initial program state $s_0 = (f_0, d)$. For every program state $s_t = (f_t, d_t)$, either $f_t \notin F_T \Rightarrow f_t(d_t) = (f_{t+1}, d_{t+1})$ and we have next step of the execution, or $f_t \in F_T \Rightarrow$ execution is terminated. All steps of this process are deterministic, hence we will have execution $E(d)$ determined by initial input data d. □

To visualize the process of the program execution we will use circles for program states connected by arrows which depicts function $\mathcal{F}$. If program execution is finite, the last state will be depicted by a circle with double stroke.

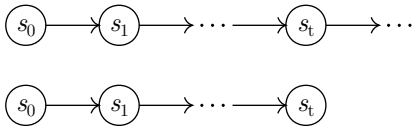

Figure 2: Infinite and finite concrete program execution.

We will also define *execution path* as either any finite sequence $P = (f_0, f_1, \dots, f_t)$, where $\forall j\ f_j \in F \wedge f_{j+1} \in f_j[D]$ and $f_t \in F_T \wedge \forall j < t\ f_j \notin F_T$ or infinite sequence $P = (f_0, f_1, \dots, f_t, \dots)$, where $\forall j\ f_j \in F \wedge f_{j+1} \in f_j[D] \wedge f_j \notin F_T$.

Lemma 2. *There is one and only one execution path P for the execution $E(d_0)$.*

$$\forall E(d_0) = s_0 \to s_1 \to \cdots \to s_t \to \cdots = (f_0, d_0) \to (f_1, d_1) \to \cdots \to (f_t, d_t) \to \cdots$$
$$\exists! P = (f'_0, f'_1, \ldots, f'_t, \ldots) : f'_0 = f_0, f'_1 = f_1, \ldots, f'_t = f_t, \ldots$$

Proof. Let's create a sequence $P = (f_0, f_1, \ldots, f_t, \ldots)$ from execution states. Now we have to proof that P is execution path. Choose arbitrary $f_j = (f_j^F, f_j^D)$ and f_{j+1}. By execution definition, $f_{j+1} = f_j^F(d_j) \Rightarrow f_{j+1} \in f_j[D]$. Hence, P is an execution path. □

We will denote this fact as $E(d_0) \Rightarrow P$.

Let's also define a *branch chain* P_B of a execution path P as a subsequence of P, which contains only $f_j \in F_B$. We can denote this fact as $P \Rightarrow P_B$.

Lemma 3. $\forall P_B \; \exists! P : P \Rightarrow P_B$.

Proof. Add proof. □

Therefore, we can write $P \Leftrightarrow P_B$.

Note, that we define execution path just as a path in control flow graph. It could be infeasible if there is no such input data that implies that path. Execution path P is *feasible in concrete model* iff $\exists d_0 \in D : (f_1, d_1) = f_0(d_0) \wedge \forall j > 0 \; (f_j, d_j) = f_{j-1}(d_{j-1})$.

Some sets of program executions:

- $\mathbb{E}_{\mathcal{P}}$ — set of all possible program executions.
- $\mathbb{P}_{\mathcal{P}}$ — set of all possible execution paths. $\mathbb{P}'_{\mathcal{P}}$ — set of all possible feasible execution paths.

Number of paths

Lemma 4. $|\mathbb{E}_{\mathcal{P}}| = |D||\mathcal{B}|$.

Proof. $|D||\mathcal{B}|$ — number of all possible data states. From lemma 1, $\forall d \in D \; \exists! E(d)$. $\forall d', d'' \in D : d' \neq d'' \; E(d') = (f_0, d') \to \cdots E(d'') = (f_0, d'') \to \cdots .(f_0, d') \neq (f_0, d'') \Rightarrow E(d') \neq E(d'')$. Therefore, number of possible executions is equal to number of possible initial input data. □

Lemma 5. $|\mathbb{P}''_{\mathcal{P}}| \leq |D||\mathcal{B}|$.

Symbolic execution model

Symbolic execution of the program is an execution of a symbolic model of that program. Symbolic model includes a set of symbolic function element (symbolic models of function elements) and symbolic data model. Execution is performed using symbolic variables instead of concrete data.

Model

$\overline{\mathcal{P}} = \langle \overline{F}, \overline{F}_I, \overline{F}_T, \overline{D} \rangle$ is a *symbolic model* of program $\mathcal{P} = \langle F, F_I, F_T, D \rangle$. There are a lot of ways to create a symbolic model of a program, therefore we will use some particular set of rules R_{SE}. We can write it as $\mathcal{P} \xrightarrow{R_{\text{SE}}} \overline{\mathcal{P}}$. It denotes that for all $\mathcal{P}$ exists only one symbolic model $\overline{\mathcal{P}}$ constructed using the set of rules R_{SE}.

Function elements and data is defines as follows.

- $\overline{F} = \{\overline{f}_j\} = \{(\overline{f}_j^F, \overline{f}_j^D, \overline{f}_j^C)\}$ — *symbolic function element models* that correspond to function elements of the program $\mathcal{P}$.

- $\overline{F}_I$ is a set of *initial symbolic function element models*, that correspond to initial function elements of the program $\mathcal{P}$.
- $\overline{F}_T$ is a set of *terminal symbolic function element models*, that correspond to terminal function elements of the program $\mathcal{P}$.
- $\overline{f}_j : \overline{D} \times C \to (\overline{F} \times \overline{D} \times C)*$ — *symbolic function element model.*

Data model

Symbolic data model $\overline{D}$ is a symbolic model of program data D: $D \xrightarrow{R_{\text{SE}}} \overline{D}$. To build a symbolic data model $\overline{D}$ of a program, we should choose, which items of the original data D to be symbolized. Let's assume out original data D is a set of elements ($D = \{d_j\}$) and we want to symbolize all of them in our model.

Firstly, we have to define a set of *symbolic variables* as a set of variables $X = \{x_j\}$. And let symbolic data model $\overline{D}$ be a

Data model with memory, registers, and array lengths.

- $\overline{D} = \{\overline{d}_j\} = \overline{R} \cup \overline{M} \cup \overline{L}$ — *symbolic data model.*
  - $\overline{R} = \{\overline{r}_j\}, \overline{r}_j = x_r, x_r \in X_R$ — *symbolic registers model.*
  - $\overline{M} = \{\overline{m}_j\}, \overline{m}_j = x_m, x_m \in X_M$ — *symbolic memory model.*
  - $\overline{L} = \{\overline{l}_j\}, \overline{l}_j = x_l, x_l \in X_L$ — *symbolic length model.*
- $X = X_M \cup X_L \cup X_R$.
  - $X_M = \{x_m\}$ — *memory symbolic variables.* $x_m \in \mathcal{B}$.
  - $X_L = \{x_l\}$ — *length symbolic variables.* $x_l \in \mathcal{B}$.
  - $X_R = \{x_r\}$ — *register symbolic variables.* $x_r \in \mathcal{B}$.

We don't consider data element indices as entities. That's why they cannot be represented by symbolic variables (or can they?). There should be such a data model.

Path condition

Set C is called *path condition* or *path constraint*. It is a logical expression over Boolean variables $X = \{x_k\}, x_k \in \mathcal{B}$ called *symbolic variables*.

Symbolic function element models

Symbolic function element models are constructed from function elements of the original program: $F \xrightarrow{R_{\text{SE}}} \overline{F}$, and $\forall j \; f_j \xrightarrow{R_{\text{SE}}} \overline{f}_j$.

$$\overline{f}_t(d_t, c_t) = \begin{cases} (\overline{f}, \overline{d}, c), \\ (\overline{f}, \overline{d}, c), \\ \dots \\ (\overline{f}, \overline{d}, c). \end{cases} \tag{5}$$

Symbolic function element models are multivalued functions. Number of values for each $\overline{f}_j$ is predefined and doesn't depend on its arguments.

Execution

Symbolic execution unlike concrete execution is not an iterative process. However, it has execution states $\overline{s} = (\overline{f}, \overline{d}, c)$ — *symbolic state*, $\overline{f} \in \overline{F}, \overline{d} \in \overline{D}, c \in C$. Set of all possible symbolic states is $\overline{S}$. $(\overline{f}_0, \overline{d}_0, \textbf{true})$ is an *initial symbolic state*. Any state $(\overline{f}_t, \overline{d}_t, c_t)$, such as $\overline{f}_t \in \overline{F}_T$ is a *symbolic terminal state*. Note, that t is not an execution step here, but a state identifier.

We consider that there is a *symbolic execution function* $\overline{\mathcal{F}} : \overline{S} \to \overline{S}$ that manages the symbolic execution process:

$$\overline{\mathcal{F}}(s_{k,\dots,l}) = \overline{\mathcal{F}}\big((f_{k,\dots,l}, d_{k,\dots,l}, c_{k,\dots,l})\big) = \tag{6}$$

$$= \big((f_{k,\dots,l,m}, d_{k,\dots,l,m}, c_{k,\dots,l,m}), \dots, (f_{k,\dots,l,n}, d_{k,\dots,l,n}, c_{k,\dots,l,n})\big) = \tag{7}$$

$$= (s_{k,\dots,l,m}, \dots, s_{k,\dots,l,n}) \tag{8}$$

Symbolic execution $\overline{E} = \overline{s}_0 \to \cdots$ visualization is presented on Figure .

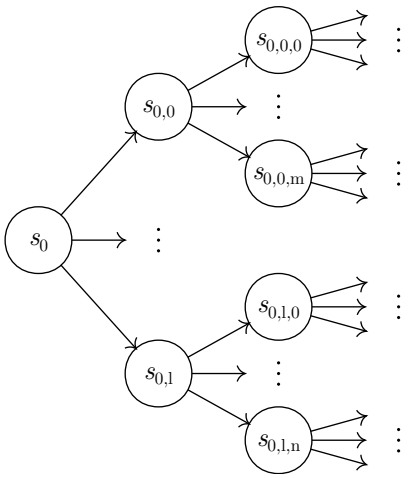

Figure 3: Symbolic execution process.

Execution rule:

- $\{\overline{f}_{0,t_1,\dots,t_n,0}, \dots, \overline{f}_{0,t_1,\dots,t_n,m}\} = \overline{f}^F_{0,t_1,\dots,t_n}(\overline{d}_{0,t_1,\dots,t_n})$,
- $\{\overline{d}_{0,t_1,\dots,t_n,0}, \dots, \overline{d}_{0,t_1,\dots,t_n,m}\} = \overline{f}^D_{0,t_1,\dots,t_n}(\overline{d}_{0,t_1,\dots,t_n})$.

Execution path $P = \{\overline{f}_0, \overline{f}_1, \dots, \overline{f}_t\}$ is *feasible in symbolic model* iff $c_{0,1,\dots,t}$ is true.

Difference between forward and backward symbolic execution? What is exactly backward symbolic execution?

Symbolic semantics

The major part of symbolic program model is a symbolic semantics. We should define symbolic semantics for each function element from the original program.

Symbolic execution and concrete execution

Feasibility in concrete model vs feasibility in symbolic model.

Static symbolic execution

Some thoughts goes here.

Taint tracking

Taint tracking is a technique used to track data flow in program during its execution. To use taint tracking we should define a set that will be used to mark data elements, initial taint marks distribution (or taint sources), and taint policy—rules that describes taint propagation.

Model

The set of a possible taint marks will be denote as $\mathcal{T}$. Simplest useful taint set is $B = \{\textbf{false}, \textbf{true}\}$, that divides data elements into two disjoint sets: what is tainted and what is not.

Taint data model is $D^T = \{d_j^T\}, d_j^T \in \mathcal{T}$.

Taint semantics is called *taint policy*.

Possible data model definitions

If original data model consists of memory and registers $D = (M, R)$, taint data model could be defined as $D^T = (M^T, R^T)$:

- $M^T = \{m_j^T\}, m_j^T \in \mathcal{T}$ — *taint memory model*,
- $R^T = \{r_j^T\}, r_j^T \in \mathcal{T}$ — *taint registers model*.

Definitions of undertainting and overtainting.

Dynamic symbolic (concolic) execution model

Dynamic symbolic execution or, in other words, concolic execution, is an execution of a program along with execution of its symbolic model (that means construction of a path condition) but for the particular execution path that is defined by initial input data d_0.

Model

Model for dynamic symbolic execution consists of program model $\mathcal{P} = \langle F, F_I, F_T, D\rangle$ and its symbolic model $\overline{\mathcal{P}} = \langle \overline{F}, \overline{F}_I, \overline{F}_T, \overline{D}\rangle$. Dynamic symbolic execution program model could we presented as

$$\widetilde{\mathcal{P}} = \langle F, F_I, F_T, D, \overline{F}, \overline{F}_I, \overline{F}_T, \overline{D}\rangle.$$

- $Y = \{Y_i\}$ — *taint sources*. $Y_i = (\{y_{i_j}\}, y_{i_L})$, $y_{i_j} \in \mathcal{B}$ — *taint source elements*, $y_{i_L} \in \mathcal{B}$ — *taint source size*.

Execution

As a concrete execution, it is iterative process with steps. We will also use variable $t \in \mathbb{N}$ as execution step. At every step, there are program execution state $s_t = (f_t, d_t)$ and symbolic execution state $\overline{s}_t = (\overline{f}_t, \overline{d}_t, c_t)$, which we combine into *concolic state* $\widetilde{s}_t = (f_t, d_t, \overline{f}_t, \overline{d}_t, c_t)$.

We consider that there is dynamic symbolic execution function $\widetilde{\mathcal{F}} : \widetilde{S} \to \widetilde{S}$ that manages the execution process:

$$\widetilde{\mathcal{F}}(\widetilde{s}_t) = \widetilde{\mathcal{F}}\big((f_t, d_t, \overline{f}_t, \overline{d}_t, c_t)\big) = \tag{9}$$

$$= (f_t^F(d_t), f_t^D(d_t), \overline{f}_t^F(\overline{d_t}), \overline{f}_t^D(\overline{d_t}), c_{t+1}) = \tag{10}$$

$$= (f_{t+1}, d_{t+1}, \overline{f}_{t+1}, \overline{d}_{t+1}, c_{t+1}) = \tag{11}$$

$$= \widetilde{s}_{t+1}. \tag{12}$$

$c_{t+1} = c_t \wedge e_t$.

We will define e_t as follows.

If f_t is an operation ($f_t \in F_O \Rightarrow f_{t+1} = \text{const}$):

$$\forall d', d'' \in D \tag{13}$$

$$f_t^D(d') = d'' \Rightarrow e_t\Big|_{\overline{d}_t = d', \overline{d}_{t+1} = d''} = \textbf{true} \tag{14}$$

$$f_t^D(d') \neq d'' \Rightarrow e_t\Big|_{\overline{d}_t = d', \overline{d}_{t+1} = d''} = \textbf{false} \tag{15}$$

If f_t is a branch ($df_t \in F_B \Rightarrow_{t+1} = d_t, f_{t+1} = f' or f''$):

$$\forall d', \in D \tag{16}$$

$$f_t^D(d') = d'' \Rightarrow e_t\Big|_{\overline{d}_t = d', \overline{d}_{t+1} = d'} = \textbf{true} \tag{17}$$

$$f_t^D(d') \neq d'' \Rightarrow e_t\Big|_{\overline{d}_t = d', \overline{d}_{t+1} = d'} = \textbf{false} \tag{18}$$

Execution function $\widetilde{\mathcal{F}}$ defines an *execution rule*:

- $f_{t+1} = f_t^F(d_t)$,
- $d_{t+1} = f_t^D(d_t)$,
- $\overline{f}_{t+1} = ?$,
- $\overline{d}_{t+1} = ?$,
- $c_{t+1} = ?$.

Dynamic symbolic execution (or *concolic execution*) of a program $\mathcal{P} = \langle F, F_I, F_T, D\rangle$ and a symbolic model $\overline{\mathcal{P}} = \langle \overline{F}, \overline{F_I}, \overline{F_T}, \overline{D}\rangle$ is a chain of concolic states, starting with initial state $\widetilde{s}_0 = (f_0, d_0, \overline{f}_0, \overline{d}_0, \textbf{true})$, where the next state is a function $\widetilde{\mathcal{F}}$ of previous state:

$$s_{t+1} = (f_{t+1}, d_{t+1}) = (f_t^F(d_t), f_t^D(d_t)) = \widetilde{\mathcal{F}}(s_t), s_t \in S$$

$\widetilde{E}(d_0) = \widetilde{s}_0 \to \widetilde{s}_1 \to \ldots \to \widetilde{s}_t \to \ldots.$

Path alternation

*Path alternation*is a technique to construct new execution paths using dynamic symbolic execution. Assume, that we have dynamic symbolic execution path $\widetilde{E}(d_0) = \widetilde{s}_0 \to \widetilde{s}_1 \to \ldots \to \widetilde{s}_t \to \ldots.$ There are states that we will call *branch states*: $\widetilde{S}_B = \{\widetilde{s}_t = (f_t, d_t, \overline{f}_t, \overline{d}_t, c_t) : f_t \in F_B\}$. For each branch state we can try to perform path alternation.

Consider we have chose a branch state $\widetilde{s}_t = (f_t, d_t, \overline{f}_t, \overline{d}_t, c_t) \in \widetilde{S}_B$. As it is a branch state, $f_t \in F_B$. Hence, $|f_t[D]| > 1$. Suppose, $f_t(d_t) = f_u \Rightarrow \exists f_v \neq f_u : f_v \in f_t[D]$.

$(c_0, c_1, \ldots, c_t), c_0 = \mathbf{true}$.

$\forall j = \overline{1,t}\ c_j = e_0 \wedge e_1 \wedge \cdots \wedge e_j$.

$c_t = e_0 \wedge e_1 \wedge \cdots \wedge e_t$.

$c*_t = e_0 \wedge e_1 \wedge \cdots \wedge \overline{e_t}$.

Lemma 6. *If* $\exists d' : c *_t (d')$, $\widetilde{E}(d') = (s'_0, s'_1, \ldots, s'_t, \ldots), s'_t = (f_t, d'_t, \overline{f}'_t, \overline{d}'_t, c'_t), f_t(d'_t) = f_v \neq f_u$.

Divergence?

Selective symbolic execution.

Examples

In that section we will consider some primitive programs to research their properties.

Division by two

Simple data-driven program, that shows how we can split program execution into different functions without any explicit program counter. Program divides a natural number by two until it is possible: $\mathcal{P} = \langle \mathcal{F}, D, D_I, D_T \rangle, D = \mathbb{N}, D_I = \mathbb{N}, D_F = \{2n - 1 | n \in \mathbb{N}\}$,

$$\mathcal{F}(d) = \begin{cases} \dfrac{d}{2}, & \text{if } d = 2n, \\ d, & \text{otherwise.} \end{cases} \tag{19}$$

There are two explicit functions: $f_0(d) = \dfrac{d}{2}$ and $f_1(d) = d$. So. we can slightly change program definition: $\mathcal{P} = \langle F, D, S, s_0, S_T \rangle, F = \{f_0, f_1\}, s_0 = \{2n | n \in \mathbb{N}\}, s_1 = \{2n - 1 | n \in \mathbb{N}\}$. Why s_0 is initial. It should be whole D.

Simplest program

Definition

Simplest program that performs some logical binary operation $\diamond$ of two Boolean variables: $x \leftarrow x \diamond y$ looks like:

- $\mathcal{P}_\diamond = \langle F, F_I, F_T, D \rangle = \langle \{f_0, f_1\}, \{f_0\}, \{f_1\}, \{x, y\} \rangle$,
- $x, y \in \mathcal{B} = \{\mathbf{false}, \mathbf{true}\}$,
- $f_0 = (f_0^F, f_0^D), f_0^F(\{x, y\}) = f_1, f_0^D(\{x, y\}) = \{x \diamond y, y\}$.
- $f_1 = (f_1^F, f_1^D), f_1^F(\{x, y\}) = f_1, f_1^D(\{x, y\}) = \{x, y\}$.

Properties

Since the program has no branches, symbolic execution tree is the same as control flow graph and there is only one execution path. Since its data contains only two Boolean variables, there are 4 possible executions. For example, if operation $\diamond$ is logical and ($\wedge$):

- $E(\mathbf{false}, \mathbf{false}) \rightarrow (\mathbf{false}, \mathbf{false})$,

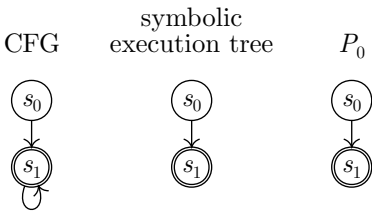

Figure 4: Control flow graph, execution paths, and symbolic execution tree of a simple program.

- $E(\mathbf{false}, \mathbf{true}) \to (\mathbf{false}, \mathbf{true})$,
- $E(\mathbf{true}, \mathbf{false}) \to (\mathbf{false}, \mathbf{false})$,
- $E(\mathbf{true}, \mathbf{true}) \to (\mathbf{true}, \mathbf{true})$.

We can write a static symbolic execution formula of this program:

$$(x, y) \to (x \diamond y, y)$$

Simplest branch program

Definition

Simplest program that contains branch:

- $\mathcal{P}_{\text{branch}} = \langle F, F_I, F_T, D \rangle = \langle \{f_0, f_1, f_2\}, \{f_0\}, \{f_1, f_2\}, D \rangle$,
- $f_0^F[D] = \{f_1, f_2\}$, $f_1^F[D] = \{f_1\}$, $f_2^F[D] = \{f_2\}$.

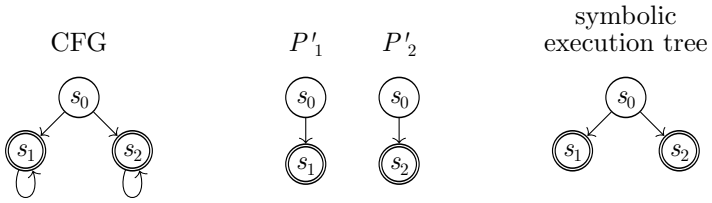

Figure 5: Control flow graph, execution paths, and symbolic execution tree of a program with one branch.

Properties

$|\mathbb{P}_{\mathcal{P}}| = 2$. $1 \leq |\mathbb{P}''_{\mathcal{P}}| \leq 2$.

Simplest cycle program

Definition

Simplest program that contains a cycle:

- $\mathcal{P}_{\circlearrowleft} = \langle F, F_I, F_T, D \rangle = \langle \{f_0, f_1, f_2\}, \{f_0\}, \{f_1\}, D \rangle$,
- $f_0^F[D] = \{f_0, f_1\}$. $f_1^F[D] = \{f_1\}$.

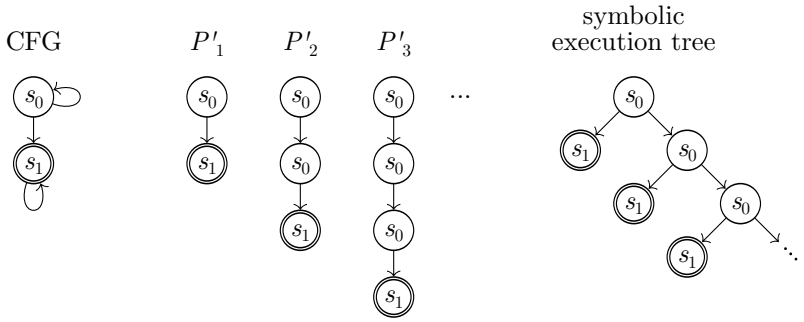

Figure 6: Control flow graph, execution paths, and symbolic execution tree of a program with one cycle.

Properties

$|\mathbb{P}_{\mathcal{P}}| = \infty$. $1 \leq |\mathbb{P}''_{\mathcal{P}}|$.

Change data under tainted condition

Code snippet

Listing 1: Listing

```
int x = read();
int y;

if (x == 1) {
    y = 1;
} else {
    y = 2;
}
y += 1;
```

Definition

We can formalize it as follows.

- $\mathcal{P}_{\text{IF}} = \langle F, F_I, F_T, D \rangle = \langle \{f_0, f_1, f_2, f_3, f_4, f_5\}, \{f_0\}, \{f_5\}, \{x, y\} \rangle$,
- $x, y \in \mathcal{B} = \{\textbf{false}, \textbf{true}\}$,

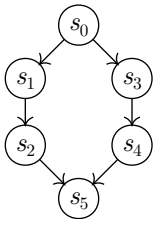

Figure 7: Implicit flow program control flow graph.

Table values

Code snippet

For this one we should have memory indexing. How can we reach that?

Listing 2: Listing

```
char* a = read_array();
char* b = init_array();
char* map = {"a...zA...Z"};

for (int i = 0; i < a.length(); i++) {
    if (a[i] > 26) {
        b[i] = map[a[i] - 'a' - 26]
    } else {
        b[i] = a[i];
    }
}
```

It is just write by tainted index.

Sequence comparison

Code example, in which program compare input string with some predefined string constant, could be considered as a classic example. Variations of this example can be found in papers describes dynamic symbolic execution tools.

Example from SAGE paper [3]: compare input string of 4 bytes length with string constant `"bad!"`, if it so, trigger an error state.

Listing 3: Listing

```
void top(char input[4]) {
    int cnt=0;
    if (input[0] == 'b') cnt++;
    if (input[1] == 'a') cnt++;
    if (input[2] == 'd') cnt++;
    if (input[3] == '!') cnt++;
    if (cnt >= 3) abort(); // error
}
```

Example from Veritesting paper [4]: check input string of 100 bytes length, if it contains exactly 75 symbols `'a'`, trigger an error state.

Listing 4: Listing

```
for (int i = 0; i < 100; i++) {
    if (input[i] == 'a') counter++;
}
if (counter == 75) abort();
```

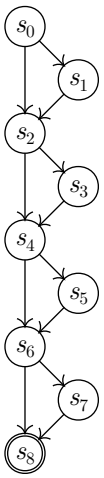

Figure 8: Sequence comparison program control flow graph

As summary, we propose simpler example. The program compares 3 bytes (x, y, z) with $(1, 1, 1)$ using variable c as a counter. $\mathcal{P}_{\text{SC}} = \langle F, F_I, F_T, D\rangle = \langle\{f_0, f_1, \ldots, f_8\}, \{x, y, z, c\}\rangle$, $x, y, z, c \in \overline{0, 255}$.

Function elements will be defined as follows.

$$f_0^F(\{x, y, z, c\}) = \begin{cases} f_1, & \text{if } x = 1, \\ f_2, & \text{if } x \neq 1. \end{cases}$$

$$f_2^F(\{x, y, z, c\}) = \begin{cases} f_3, & \text{if } y = 1, \\ f_4, & \text{if } y \neq 1. \end{cases}$$

$$f_4^F(\{x, y, z, c\}) = \begin{cases} f_5, & \text{if } z = 1, \\ f_6, & \text{if } z \neq 1. \end{cases}$$

$$f_6^F(\{x, y, z, c\}) = \begin{cases} f_7, & \text{if } c = 3, \\ f_8, & \text{if } c \neq 3. \end{cases}$$

And alternative definition of element f_6:

$$f_6'^F(\{x, y, z, c\}) = \begin{cases} f_7, & \text{if } c = 2, \\ f_8, & \text{if } c \neq 2. \end{cases}$$

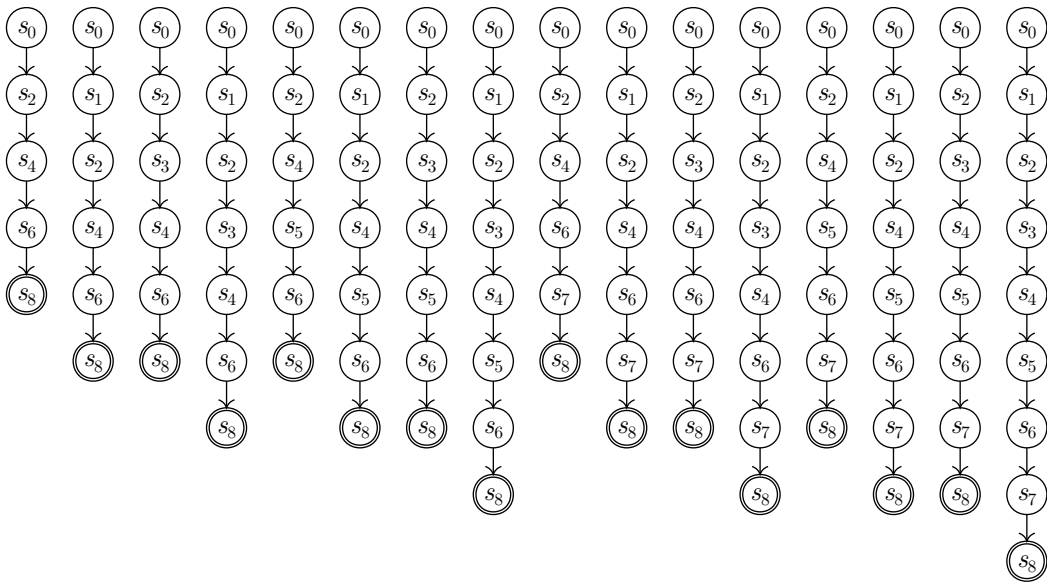

Figure 9: Sequence comparison program possible execution paths.

We can propose another variant of sequence comparison, as in `strcmp`.

Listing 5: Listing

```
1 void top(char input[4]) {
2     if (input[0] == 'b')
3         if (input[1] == 'a')
4             if (input[2] == 'd')
5                 if (input[3] == '!')
6                     abort(); // error
7 }
```

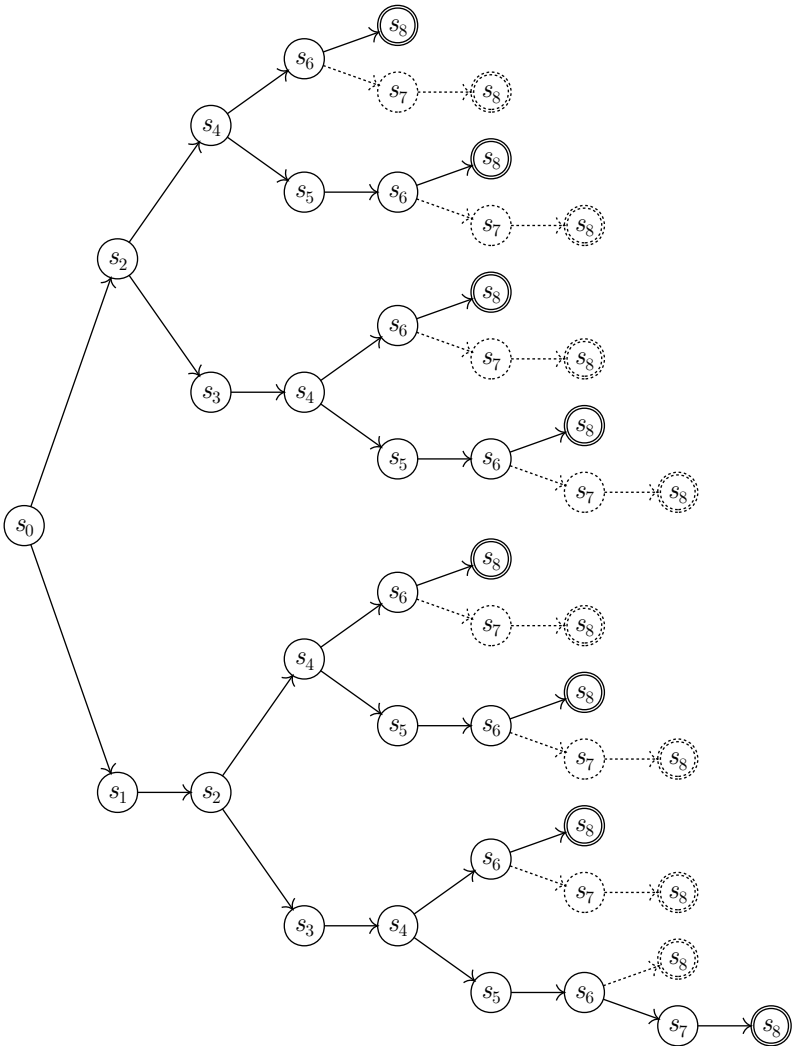

Figure 10: Sequence comparison program symbolic execution tree.

Other models

We compare our model with other concrete, symbolic, and dynamic symbolic program execution models. Add models from:

- A Theory and Practice of Program Development. By Derek J. Andrews
- Distributed Programming: Theory and Practice. By A. Udaya Shankar.

Schwartz

Program model described in [5] is based on statements.

Splat

Splat [6] program model is based on labeled statements: $\ell : s$. Statement is:

1. **halt**, **abort**,
2. **input**(l, k), l — buffer address, k — size,

3. $l := e$, l — address, e — expression,
4. $\mathbf{if}(e)\,\mathbf{goto}\,\ell$, e — expression, ℓ — label,
5. $l := \mathbf{malloc}(k)$, k — size,
6. $\mathbf{free}(e)$, e — address.

Contents

Index

References

[1] Donald E. Knuth. *The Art of Computer Programming, Volume 1 (3rd Ed.): Fundamental Algorithms*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1997.

[2] Alan Mathison Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London mathematical society*, 2(1):230–265, 1937.

[3] Patrice Godefroid, Michael Y. Levin, and David Molnar. Automated whitebox fuzz testing. 2008.

[4] Thanassis Avgerinos, Alexandre Rebert, Sang Kil Cha, and David Brumley. Enhancing symbolic execution with veritesting. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 1083–1094, New York, NY, USA, 2014. ACM.

[5] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, pages 317–331, 2010.

[6] Ru-Gang Xu, Patrice Godefroid, and Rupak Majumdar. Testing for buffer overflows with length abstraction. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA '08, pages 27–38, New York, NY, USA, 2008. ACM.