An extended version of DBA

Adel Djoudi and Sébastien Bardin

July 2013

Contents

1 Introduction

In order to be able to apply analysis tools on executable code, we need an intermediate representation of the sequence of program instructions.

DBA model. Dynamic Bit-vector Automata (DBA)[1] is a generic and concise formal model for low-level programs. The main design ideas behind DBA are the following: (a) a small set of instructions; (b) a concise and natural modelling for common architectures; (c) self-contained models which do not require a separate description of the memory model or of the architecture; and (d) a sufficiently low-level formalism, so that DBA can serve as a reference semantics of the executable file to analyse.

Extended DBA model. In this report, we enhance the DBA model in the following ways:

- **Basic specification mechanisms:** It is useful in program analysis to be able to insert specifications in the model in order to express properties or to abstract too complex parts of a program. So, we introduce the **assert**, **assume**, **stop** and **nondet** instructions.
- **Region-based memory model:** We propose a partitioned memory model in the vein of that of CompCert [2], allowing more robust analyses and native support of dynamic allocations. We use typed values of the form $(region, val)$, where val is a bit-vector and $region$ can be the Cst region (addresses and constant values), the $Stack$ region (the stack) or a $malloc(id, size)$ region (memory regions created by **malloc** instruction) (cf. section 4). The memory regions are considered separated (no overlap).
- **Access permissions:** We extend the DBA model to handle memory access permissions. A memory access can be a write to /read from memory or an execution of an instruction at a certain address of memory. The set of memory addresses of a region is partitioned into several disjoint subsets sharing the same access permissions. The partition is given by a set of exclusive predicates. Permission semantic is given in section 4.2.

Outline. The rest of the document is structured as follows. Section 2 presents an overview of DBA models and introduces some basic notations. Section 3 describes the syntax of DBA. The semantics of DBA is defined in

Section 4. Section 5 shows the structure of a DBA file through an example. Finally, Section 6 describes an implementation of DBA in OCaml including a simulator.

2 DBA overview

DBA are low-level programs built over unstructured control mechanisms (dynamic jumps) and low-level data (bit-vectors). A DBA manipulates a finite number of variables and an unbounded memory, partitioned into non-overlapping regions. Let $\mathcal{R} = \{Cst, Stack, Malloc(id, size) \mid i, size \in \mathbb{N}\}$ be the set of all possible disjoint regions and $\mathbb{B}v$ be the set of bit-vectors. The size of a bit-vector is given by $size : \mathbb{B}v \rightarrow \mathbb{N}$.

DBA operators. DBA expressions and conditions are built upon a small set of standard fixed-width bit-vector operators, including (signed/unsigned) arithmetic operators, deified (signed/unsigned) arithmetic relational operators, logical bitwise operators, size extensions, shifts, concatenation and restriction[1].

Values. We use typed values lying in the set $\mathbb{L} = \{(r, bv) \mid r \in \mathcal{R}; bv \in \mathbb{B}v\}$. Conceptually, r is the base (start address of a region) and bv is the offset. However, while the base of Cst acts as zero, the bases of other regions are left uninterpreted. To express the value resulting of applying a restriction operator on some $(r, bv) \in \mathbb{L}$ with r different of Cst, we need to introduce a new kind of symbolic values belonging to the set $\mathbb{L}r = \big\{Restrict\big((r, bv), i, j\big) \mid r \in \mathcal{R}_0; bv \in \mathbb{B}v; i, j \in \mathbb{N}\big\}$. Only concatenation and restriction operations can be performed precisely on such values. Finally, a $\perp_V$ value is needed to express undefined values and an $ERROR$ value is used to express the result of bad operations. So, to evaluate DBA expressions, we consider the set of extended values $\mathbb{V} \triangleq \mathbb{L} \uplus \mathbb{L}r \uplus \{\perp_V, Error\}$.

Evaluation environment. Each memory region can be considered as an array of bytes (bit-vectors of size 8). The set of available regions changes dynamically according to the malloc and free instructions. That's why we need an updatable set of regions $\mathcal{R}^* \subset \mathcal{R}$ containing only existing regions. An environment ρ maps each variable to its corresponding value and each region to its corresponding array of bytes if it exists. If no array is associated to a region $r \in \mathcal{R}$ then $\rho(r)$ is an undefined array.

$$\rho : \begin{cases} p \in \mathbb{V}ar & \mapsto v \in \mathbb{V} \\ r \in \mathcal{R} & \mapsto \begin{cases} a : \mathbb{B}v \to \mathbb{V} & if\ r \in \mathcal{R}^* \\ (\lambda bv.\bot_V) & if\ r \in (\mathcal{R} \setminus \mathcal{R}^*) \end{cases} \end{cases}$$

Control mechanism. In order to create a DBA model from a low level program, each instruction of the program is translated into one or more DBA instructions (called hereafter a bloc of instructions). Each DBA instruction have an address of size $size(\backslash addr)$ expressed by a pair (bv, id), where $id \in \mathbb{N}$ is an address identifier and bv is a bit-vector of size $size(\backslash addr)$. DBA instructions belonging to the same bloc have the same bv but different identifiers. The address identifier of the first instruction of a bloc is always zero and the target of a jump instruction leaving a bloc can only be the first instruction of another bloc.

Excepting the **stop** instruction (having no successor instruction), the **ite** instruction (a choice between two successor instructions) and the **goto** instruction (unknown successor before runtime), each DBA instruction contains the address of its successor instruction

$$\begin{aligned} \mathcal{R}_0 &= \{Cst, Stack\} \\ \mathcal{R} &= \{Cst, Stack, Malloc(id, size) \mid i, size \in \mathbb{N}\} \\ \mathbb{L}r &= \big\{Restrict\big((r, bv), i, j\big) \mid r \in \mathcal{R}_0; bv \in \mathbb{B}v; i, j \in \mathbb{N}\big\} \\ \mathbb{L} &= \{(r, bv) \mid r \in \mathcal{R}; bv \in \mathbb{B}v\} \\ \mathbb{V} &\triangleq \mathbb{L} \uplus \mathbb{L}r \uplus \{\bot_V, Error\} \end{aligned}$$

Figure 1: Summary of basic sets

3 Syntax

3.1 DBA

We denote by $\mathbb{E}xpr$ the set of expressions. Each expression has a statically checkable size and evaluates to a value in $\mathbb{V}$. The set of conditional expressions is denoted by $\mathbb{C}ond$. A conditional expression is an expression evaluating to a value of size 1. Figure 2 summarizes all DBA expressions.

We denote by $\mathbb{I}nstr$ the set of all possible instructions. Each instruction contains the address of the next instruction(s), except *stop* (there is no

In the following, let $r \in \mathcal{R}_0$; $bv \in \mathbb{B}v$; $k, i, j \in \mathbb{N}$; $v \in \mathbb{V}ar$

$$\mathbb{E}xpr : \begin{cases} v, \quad (r, bv) \\ @(expr, \overset{\leftrightarrows}{k}) \qquad \text{//memory access} \\ expr\{i..j\}, \;\; ext_{u,s}(expr, n) \;\; \text{//restriction, extension} \\ expr \;\; \{+, -, \times, /_{u,s}, \%_{u,s}\} \;\; expr \\ expr \;\; \{<_{u,s}, \leq_{u,s}, =, \neq, \geq_{u,s}, >_{u,s}\} \;\; expr \\ expr \;\; \{\wedge, \vee, \oplus\} \;\; expr, \quad \neg expr \\ expr \;\; \{>>, <<_{u,s}, ::\} \;\; expr \;\; \text{//:: is concatenation} \\ alternative \;\; \big((expr_1, expr_2, ..., expr_n), cond\big)^1 \end{cases}$$

$\mathbb{C}ond$: Any expression evaluating to a value of size 1

^{1}This constructor has been proposed by Alan Mycroft at Dagstuhl workshop 2012. "Binary-level analysis: benefits and challenges"[3]. The idea is to provide different but equivalent encodings of a given instruction.

Figure 2: DBA expressions

successor instruction) and *goto expr* (the address of the next instruction is known at runtime). Figure 3 presents DBA instructions.

Syntactic sugar

Instruction $(nondetAssume\big((lhs_1, lhs_2, ..., lhs_n), cond\big); \quad goto \;\; addr)$ is equivalent to the sequence of the following instructions:

$lhs_1 := nondet(Cst);$

$lhs_2 := nondet(Cst);$

...

$lhs_n := nondet(Cst);$

$assume(cond); \quad goto \;\; addr$

3.2 Well formed DBA

In the following, let $e, e_1, e_2 \in \mathbb{E}xpr$; $k, i, j \in \mathbb{N}$; $bv \in \mathbb{B}v$; $cond \in \mathbb{C}ond$ and $\mathbb{B}op \triangleq \{+, -, \times, /_{u,s}, \%_{u,s}, <_{u,s}, \leq_{u,s}, =, \neq, \geq_{u,s}, >_{u,s}, \wedge, \vee, \oplus\}$.

We perform the analysis on a well formed model of DBA, complying with the following statically checkable rules:

$$\mathbb{I}nstr : \begin{cases} lhs := rhs; \quad goto\ addr \\ lhs := nondet(region); \quad goto\ addr \\ lhs := undef; \quad goto\ addr \\ lhs := malloc(size); \quad goto\ addr \\ free(expr); \quad goto\ addr \\ goto\ expr \\ goto\ addr \\ ite\,(cond)?\ goto\ addr_1 \quad : \quad goto\ addr_2 \\ assert(cond); \quad goto\ addr \\ assume(cond); \quad goto\ addr \\ stop \end{cases}$$

Figure 3: DBA instructions

- All used variables must be declared
- $lhs := malloc(bv) \Longrightarrow size(lhs) = size(\backslash addr)$
- $lhs := e \Longrightarrow size(lhs) = size(e)$
- $ext_{u,s}(e, k) \Longrightarrow size(e) < k$
- $@(e, \overset{\leftrightarrows}{k}) \Longrightarrow k \geq 0$ and $size(e) = size(\backslash addr)$
- $e_1 \odot e_2 \Longrightarrow size(e_1) = size(e_2)$ with $\odot \in \mathbb{B}op$
- $cond \in \mathbb{C}ond \Longrightarrow size(cond) = 1$
- $e\{i, j\} \Longrightarrow 0 \leq i \leq j < size(e)$
- $goto(e) \Longrightarrow size(e) = size(\backslash addr)$
- $goto(addr) \Longrightarrow size(addr) = size(\backslash addr)$
- $goto(addr) \Longrightarrow (Cst, addr)$ satisfies eXecution permission (cf. section4.2)

3.3 Permissions

We can control the access to memory locations by defining suitable permissions on addresses verifying some predicate $\varphi_i \in \mathbb{C}ond$. Permissions are defined once for all the $malloc(id, size)$ regions. We denote by Malloc any $malloc(id, size)$ region. Several predicates can be defined for each of the three main regions Cst, Stack, and Malloc according to the following syntax:

$$\begin{array}{lllll} cst: & (\varphi_1 : \overline{R}\ \ \overline{W}\ \ \overline{X}) & ... & (\varphi_{n_1} : \overline{R}\ \ \overline{W}\ \ \overline{X}) \\ stack: & (\varphi'_1 : \overline{R}\ \ \overline{W}\ \ \overline{X}) & ... & (\varphi'_{n_2} : \overline{R}\ \ \overline{W}\ \ \overline{X}) \\ malloc: & (\varphi''_1 : \overline{R}\ \ \overline{W}\ \ \overline{X}) & ... & (\varphi''_{n_3} : \overline{R}\ \ \overline{W}\ \ \overline{X}) \end{array}$$

Note that $\overline{P}$ means either P or $!P$ and if the symbol ! is put before a Read (R), Write(W) or eXecute (X) permission then the corresponding permission is denied.

3.4 Tags

We introduced annotations to some key points to facilitate certain analyses:

Distinguished jumps. Annotate a jump to specify whether it is a procedure call (`<call addr>` : when the called procedure completes, execution flow resumes at the instruction of address `<addr>`) or a return from procedure (`//ret`) or a simple jump. Useful for partitioning programs into procedures and to perform modular analyses.

Distinguished variables. Variables can be tagged as local/temporary (`<temp>`) or as flags (`<flag>`). This is used for optimization techniques. Many flag updates are useless and then are a prime target for optimization. Local variables can also be considered as useless as soon as we leave the block where they are introduced.

Flag operations. Allow to determine the type of encoded flag when using the *Alternative* expression. If we want to model the carry flag on the 32 bits addition $R := A + B$ for example, the value of the carry flag F can be computed as follows:
$F := alternative\big(R <_u A, (ext_u(A, 33) + ext_u(B, 33))\{32\}\big)$`//carryAdd`

4 Semantics

4.1 Basics

In the region-based memory model used here, the set of available regions changes dynamically according to the malloc and free instructions. That's why we need an updatable set of regions that we denote by $\mathcal{R}^*$. Note that $\mathcal{R}_0 \subseteq \mathcal{R}^* \subset \mathcal{R}$.

An environment ρ maps each variable to its corresponding value and each region to its corresponding array of memory locations if it exists. If no array is associated to a region $r \in \mathcal{R}$ then $\rho(r)$ is the undefined array. The set of environments is written $\mathcal{E}nv$.

$$\rho : \begin{cases} p \in \mathbb{V}ar & \mapsto v \in \mathbb{V} \\ r \in \mathcal{R} & \mapsto \begin{cases} a : \mathbb{B}v \to \mathbb{V} & if\ r \in \mathcal{R}^* \\ (\lambda bv.\bot_V) & if\ r \in (\mathcal{R} \setminus \mathcal{R}^*) \end{cases} \end{cases}$$

The program state can be specified by an environment and the address of the current instruction (ρ, l). However, the program can also reach an error case or achieve the *stop* instruction, so we introduce two other possible sates $Error_{state}$ and End_{state}. Let $\mathbb{P} \triangleq (\mathcal{E}nv \times \mathbb{A}ddress) \uplus \{End_{state}, Error_{state}\}$ be the set of all possible program states.

The concrete semantics of a program is given by the post operator ($post : \mathbb{I}nstr \to \mathbb{P} \to \mathbb{P}$) that performs an instruction and moves the program from one state to another. The program execution is aborted if the program reaches an $Error_{state}$ and if the program reaches an End_{state} then no instruction remains to execute. The remaining behavior of *post* operator is to execute an instruction into an environment and to return a new program state (post: $\mathbb{I}nstr \to \mathcal{E}nv \to \mathbb{P}$).

We use the eval function (eval : $\mathbb{E}xpr \to \mathcal{E}nv \to \mathbb{V}$) to evaluate the expressions of the program(Figure 4). The *post* operator is described in Figure 5. For the sake of brevity, each statement ending with a jump to the address of the next instruction (instructions of the form: *inst*; *goto l*) will be represented by ($[inst]^l$). *goto expr* instruction will be represented by $[goto\ expr]^?$. The instruction (*ite* (*cond*)? *goto addr*$_1$: *goto addr*$_2$) will be represented by $[ite\ (cond)?]^{(addr_1, addr_2)}$ and finally the *stop* instruction will be represented by [*stop*].

$$eval[\![e_1 \odot e_2]\!]\rho \triangleq$$
$$\text{let } eval[\![e_1]\!]\rho = (r_1, v_1) \text{ and } eval[\![e_2]\!]\rho = (r_2, v_2) \text{ in}$$
$$\begin{cases} (Cst, v_1 \odot v_2) & if\ \ r_1 = r_2 = Cst \\ (Cst, v_1 \odot v_2) & if\ \ r_1 = r_2 \text{ and } \odot \in \{=, \neq, \leqslant_{u,s}, \geqslant_{u,s}, ...\} \\ (r_1, v_1 + v_2) & if\ \ \odot \text{ is } + \text{ and } r_2 = Cst \\ (r_2, v_1 + v_2) & if\ \ \odot \text{ is } + \text{ and } r_1 = Cst \\ (r_1, v_1 - v_2) & if\ \ \odot \text{ is } - \text{ and } r_2 = Cst \\ (Cst, v_1 - v_2) & if\ \ \odot \text{ is } - \text{ and } r_1 = r_2 \\ (Cst, 1) & if\ \ \odot \text{ is } \neq \text{ and } r_1 \neq r_2 \\ (Cst, 0) & if\ \ \odot \text{ is } = \text{ and } r_1 \neq r_2 \\ Error & \text{Otherwise} \end{cases}$$
$$eval[\![e_1 :: e_2]\!]\rho \triangleq$$
$$\begin{cases} (r_1, bv_1 :: bv_2) & if\ \ eval[\![e1[\![\rho = (r_1, bv_1) \text{ and } eval[\![e_2]\!]\rho = (r_2, bv_2) \\ & \quad \text{and } r_1 = r_2 \\ Restrict((r, bv), i_1, j_2) & if\ \ eval[\![e_1]\!]\rho = Restrict((r, bv), i_1, j_1) \text{ and} \\ & \quad eval[\![e_2]\!]\rho = Restrict((r, bv), i_2, j_2) \text{ and} \\ & \quad j_1 = i_2 - 1 \\ Error & \text{Otherwise} \end{cases}$$
$$eval[\![e\{i, j\}]\!]\rho \triangleq$$
$$\begin{cases} (Cst, v\{i, j\}) & if\ \ eval[\![e]\!]\rho = (r, v) \text{ and } \ r = Cst \\ Restrict((r, v), i, j) & if\ \ eval[\![e]\!]\rho = (r, v) \text{ and } \ r \neq Cst \\ Restrict((r, v), i, j) & if\ \ eval[\![e]\!]\rho = Restrict((r, v), i', j') \\ Error & \text{Otherwise} \end{cases}$$
$$eval[\![@[e, \overrightarrow{k}]]\!]\rho \triangleq \big(\rho(r)\big)(i) :: \big(\rho(r)\big)(i+1) :: ... :: \big(\rho(r)\big)(i+k-1)$$
$$\text{s.t. } eval[\![e]\!]\rho = (r, i)$$

Figure 4: Evaluation of expressions

$$post[\![[v := e]^l]\!]\ \rho \triangleq \big(\rho[v \mapsto eval[\![e]\!]\ \rho], l\big)$$

$$post[\![[@(e_1, \overrightarrow{k}) := e_2]^l]\!]\ \rho \triangleq$$
$$\big(\rho\big[\big(\rho(r)\big)(j) \mapsto eval[\![e_2\{8(j-i); 8(j-i+1)\}]\!]\ \rho\big], l\big)$$
$$\forall j : i \leqslant j \leqslant i + k - 1,\ eval[\![e_1]\!]\ \rho = \ (r, i)$$

$$post[\![[lhs := malloc(size)]^l]\!]\ \rho \triangleq$$
$$\mathcal{R}^* := \mathcal{R}^* \cup \{Malloc(id, size)\};$$
$$post[\![[lhs := (Malloc(id, size), 0)]^l]\!]\ \rho$$
$$\text{s.t. } Malloc(id, size) \in \mathcal{R} \text{ and } \rho(malloc(id, size)) = \bot_V$$

$$post[\![[free(e)]^l]\!]\ \rho \triangleq$$
$$\begin{cases} \mathcal{R}^* := \mathcal{R}^* \setminus \{r\};\ \ (\rho, l) & \text{if } r = malloc(id, size) \text{ and } bv = 0 \\ Error_{state} & \text{Otherwise} \end{cases}$$
$$\text{s.t. } eval[\![e]\!]\rho = (r, bv)$$

$$post[\![[goto(addr)]^{addr}]\!]\ \rho \triangleq (\rho, addr)$$

$$post[\![[goto(e)]^?]\!]\ \rho \triangleq (\rho, bv) \text{ s.t. } eval[\![e]\!]\ \rho = (r, bv)$$

$$post[\![[\text{ ite } (b)?]^{(bv_1, bv_2)})]\!]\ \rho \triangleq$$
$$\begin{cases} (\rho, bv_1) & \text{if } eval[\![b]\!]\ \rho = (Cst, 1) \\ (\rho, bv_2) & \text{if } eval[\![b]\!]\ \rho = (Cst, 0) \\ Error_{state} & \text{Otherwise} \end{cases}$$

$$post[\![[stop]]\!]\ \rho \triangleq End_{state}$$

$$post[\![[assert(b)]^l]\!]\ \rho \triangleq$$
$$\begin{cases} (\rho, l) & \text{if } eval[\![b]\!]\ \rho = (Cst, 1) \\ Error_{state} & \text{Otherwise} \end{cases}$$

$$post[\![[assume(b)]^l]\!]\ \rho \triangleq$$

We restrict the behaviour to continue the execution only if $eval[\![b]\!]\rho = (Cst, 1)$.

$$post[\![[lhs := nondet(r)]^l]\!]\ \rho \triangleq$$
$$post[\![[lhs := (r, bv)]^l]\!]\ \rho;$$
choose $bv \in \mathbb{B}v$, s.t. size(bv) = size(lhs)

Figure 5: Post operator

4.2 Permissions

We use three applications denoted $\mathcal{P}_R$, $\mathcal{P}_W$ and $\mathcal{P}_X$ mapping each region to it corresponding predicate to check the read, write and execution permissions respectively.

$\mathcal{P}_R$ **:** $\mathcal{R}^* \to \mathbb{C}ond$

$\mathcal{P}_W$ **:** $\mathcal{R}^* \to \mathbb{C}ond$

$\mathcal{P}_X$ **:** $\mathcal{R}^* \to \mathbb{C}ond$

Example 1
Assume that
cst : (φ_1 : !R !W X) (φ_2 : R !W !X) ... (φ_n : !R W !X).
Then $\mathcal{P}_R$, $\mathcal{P}_W$, $\mathcal{P}_X$ are defined by
$\mathcal{P}_R[Cst \mapsto \neg\varphi_1 \wedge \neg\varphi_n]$; $\mathcal{P}_W[Cst \mapsto \neg\varphi_1 \wedge \neg\varphi_2]$; $\mathcal{P}_X[Cst \mapsto \neg\varphi_2 \wedge \neg\varphi_n]$

We refine the semantics of the eval function and post operator. In fact, concerning the Read and Write permissions, the semantics changes for the expressions and instructions involving memory access (Figures 6 and 7). The eXecute permission must also be checked at each dynamic jump instruction(Figure 7).

$$eval[\![@(e, \overrightarrow{k})]\!]\ \rho \triangleq$$
$$\begin{cases} \big(\rho(r)\big)(i) :: \big(\rho(r)\big)(i+1) :: ... :: \big(\rho(r)\big)(i+k-1) & \text{if } eval[\![\mathcal{P}_R(r)]\!]\rho = (Cst, 1) \\ \bot_V & \text{Otherwise} \end{cases}$$
$$\text{s.t. } eval[\![e]\!]\rho = (r, i)$$

Figure 6: Evaluation of expression with read permissions

4.3 Summary of ambiguous cases

We recall here the undefined behaviours and error cases.

Errors. Error cases are:

- Performing operations on some region elements that prevent us to compute the actual resulting value over a given region, ex:
  $(Stack, 5) + (Stack, 8) = Error$,
  $(Constant, 5) + Restrict\big((Stack, 78), 8, 15\big) = Error$.

$$
\begin{array}{l}
post[\![[@[e_1, \overrightarrow{k}] := e_2]^l]\!]\ \rho \triangleq \\
\quad \text{let } val_j = eval[\![e_2\{8(j-i); 8(j-i+1)\}]\!]\ \rho] \\
\quad \text{ and } eval[\![e_1]\!]\ \rho = \ (r, i) \text{ in} \\
\quad \begin{cases} \big(\rho[\big(\rho(r)\big)(j) \mapsto val_j, l\big); & \text{if } eval[\![\mathcal{P}_W(r)]\!]\rho = (Cst, 1) \\ Error_{state} & \text{Otherwise} \end{cases} \\
\quad \text{for } j = i, i+1, ..., i+k-1
\end{array}
$$

$$
\begin{array}{l}
post[\![[goto(addr)]^{addr}]\!]\ \rho \triangleq \\
\quad \begin{cases} (\rho, addr) & \text{if } eval[\![\mathcal{P}_X(r)]\!]\rho = (Cst, 1) \\ Error_{state} & \text{Otherwise} \end{cases}
\end{array}
$$

$$
\begin{array}{l}
post[\![[goto(e)]^{?}]\!]\ \rho \triangleq \\
\quad \text{let } eval[\![e]\!]\ \rho = (r, bv) \text{ in} \\
\quad \begin{cases} (\rho, bv) & \text{if } r = Cst \text{ and } eval[\![\mathcal{P}_X(r)]\!]\rho = (Cst, 1) \\ Error_{state} & \text{Otherwise} \end{cases}
\end{array}
$$

$$
\begin{array}{l}
post[\![\text{ ite } (b)?]^{(bv_1, bv_2)}]\!]\ \rho \triangleq \\
\begin{cases} (\rho, bv_1) & \text{if } eval[\![b]\!]\ \rho = (Cst, 1) \text{ and } eval[\![\mathcal{P}_X(r)]\!]\rho = (Cst, 1) \\ (\rho, bv_2) & \text{if } eval[\![b]\!]\ \rho = (Cst, 0) \text{ and } eval[\![\mathcal{P}_X(r)]\!]\rho = (Cst, 1) \\ Error_{state} & \text{Otherwize} \end{cases}
\end{array}
$$

Figure 7: post operator with Write and eXecute permissions

- Use of uninitialized variables or region elements.
- Reading (resp. writing, executing) at an address (r, bv) when R (resp. W, X) permission is denied on (r, bv).
- $\text{eval}[\![e_1 /_{u,s} e_2]\!]\rho = Error$ if $eval[\![e_2]\!]\rho = (Constant, 0)$
- $\text{post}[\![[assert(cond)]^l]\!]\rho = Error_{state}$ if $eval[\![cond]\!]\rho = (Constant, 0)$
- $$\text{post } [\![goto(e)]\!]\rho = \begin{cases} Error_{state} & \text{if } eval[\![e]\!]\rho = \bot_V \\ Error_{state} & \text{if } eval[\![e]\!]\rho \in \mathbb{L}_r \\ Error_{state} & \text{if } eval[\![e]\!]\rho = (r, bv) \text{ and r} \neq \text{Constant}^{1} \\ Error_{state} & \text{if } eval[\![e]\!]\rho = (r, bv) \text{ and} \\ & \quad eval[\![\mathcal{P}_X(r)]\!]\rho = (Cst, 0) \end{cases}$$
- $$eval[\![@[e, k]]\!]\rho = \begin{cases} Error & \text{if } eval[\![e]\!]\rho \in \mathbb{L} \text{ and } r \text{ is deallocated} \\ & \qquad \text{by a } free \text{ instruction} \\ Error & \text{if } eval[\![e]\!]\rho \notin \mathbb{L} \end{cases}$$
- $post[\![[lhs := nondet(malloc)]^l\ ]\!]\rho = Error_{state}$ if no non freed malloc region is available.
- $eval[\![e]\!]\rho = Error$ if $eval[\![e]\!]\rho = (Malloc(id, size), bv)$ and $bv \geq size$
- $post[\![free(e)]\!]\rho =$
$$\begin{cases} Error_{state} & \text{if } eval[\![e]\!]\rho = \bot_V \\ Error_{state} & \text{if } eval[\![e]\!]\rho = (r, bv) \text{ and } r \neq Malloc(id, size) \\ Error_{state} & \text{if } eval[\![e]\!]\rho = (r, bv) \text{ and } bv \neq 0 \\ Error_{state} & \text{if } eval[\![e]\!]\rho = (r, bv) \text{ and } r \text{ is deallocated by} \\ & \qquad \text{a } free \text{ instruction} \\ Error_{state} & \text{if } eval[\![e]\!]\rho \in \mathbb{L}_r \end{cases}$$

Absorbing values.

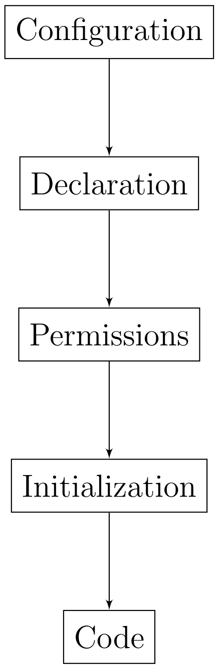
- Error is an absorbing element
- $\bot_V$ is an absorbing element with respect to DBA operations.

In the implemented version of the simulator of DBA models, an exception is raised for each of the error cases stated above.

[1]Implicitly, only the Cst region is executable

5 DBA file structure

A DBA file is organised into five parts as following:



5.1 Example of a DBA file

The DBA file given below aims to summarize the most of instructions offered by DBA models, regardless of the functionality of the resulting program.

```
# configuration
\addr : 32
\endianess : big
\entry_point : (0x00000002, 0)

# declaration
var x : 32 <flag>
var y : 8
var z : 32
var c1 : 32
var c2 : 32
var c3 : 8
var c4 : 32
var c5 : 24
var c6 : 32
var c7 : 16
var c8 : 34
var c9 : 32
var c10 : 32 <temp>
var v : 32

# permissions
begin permissions
stack : (true : R W !X)
malloc : (true : R W !X)
cst : (\addr <=u 18<32> : R !W X)
      (\addr >u 18<32> : R W !X)
end permissions

# initialisation
```

```
x := 8
y := 8
z := nondet(stack)
c1 := (cst,8)
c2 := 1
c3 := 16<8>
c4 := 0x00000028
c5 := 11184810
c6 := 67
@[8<32>, 7] := 789865765654
@[(stack,8<32>), ->, 7] := \undef
c7 := 3456
c10 := malloc(12)


# code
(0x00000000,0) x := 0x00000008 ; goto (0x00000001,0)
(0x00000001,0) x := x + c2; goto (0x00000002,0)
(0x00000002,0) if (x{1,1}) goto (0x00000003,0) else goto (0x00000004,0)
(0x00000003,0) goto x  // call (0x00000004,0)
(0x00000004,0) y := c3 ; goto (0x00000005,0)
(0x00000005,0) @[c4, ->, 3] := c5; goto (0x00000006,0)
(0x00000006,0) @[z, ->, 2] := c7; goto (0x00000007,0)
(0x00000007,0) print "printing values at runtime :\n" >>
                "@[z, ->, 2] = " >> @[z, ->, 2] >>
                ", x = " >> x{6,6} >> ", y = " >>  y >> "\n"; goto (0x00000008,0)
(0x00000008,0) nondet_assume ({@[c6, 2], y}, (y = 254<8>)) ; goto (0x00000009,0)
(0x00000009,0) assert (x <=u 1999990<32>) ; goto (0x0000000A,0)
(0x0000000A,0) x := alternative (x + c1, c1 + x); goto (0x0000000B,0)
(0x0000000B,0) print "x = " >> x >> ", c1 = " >> c1 >> "\n"; goto (0x0000000C,0)
(0x0000000C,0) c8 := (extu x 34) + 100<34>; goto (0x0000000D,0)
(0x0000000D,0) c9 := malloc (16); goto (0x0000000E,0)
(0x0000000E,0) @[c9, ->, 6] := 1234<48> ; goto (0x0000000F,0)
(0x0000000F,0) @[c4, ->, 4] := malloc (16); goto (0x00000010,0)
(0x00000010,0) @[@[c4, ->, 4], <-, 3] := 1234<24> ; goto (0x00000011,0)
(0x00000011,0) print "c4 = " >> c4 >> "\n"; goto (0x00000012,0)
(0x00000012,0) v := malloc(16); goto (0x00000013,0)
(0x00000013,0) free (@[(cst, 40<32>), ->, 4]); goto (0x00000014,0)
(0x00000014,0) stop OK
```

5.2 Configuration

The size of the addresses ($size(\backslash addr)$) must be defined at the beginning of a DBA file. the address size is checked statically at each **goto** instruction. A type of endianess can be specified as a default value in the rest of the program, so that we no longer have to specify it at each memory access. The first address of the program must be defined in this section.

```
# configuration
\addr : 32
\endianess : big
\entry_point : (0x00000002, 0)
```

5.3 Declaration

All variables used in the program must be declared in this section by specifying their sizes. `<flag>` and `<temp>` tags can be added.

```
# declaration
var x : 32 <flag>
var y : 8
var z : 32
var c1 : 32
var c2 : 32
var c3 : 8
var c4 : 32
var c5 : 24
var c6 : 32
var c7 : 16
var c8 : 34
var c9 : 32
var c10 : 32 <temp>
var v : 32
```

5.4 Permissions

This section is optional. It allows to specify the Read, Write or eXecute permissions on specific regions. It is possible to specify permissions on parts of regions satisfying some predicates. Predicates are conditions where only variable *addr* is allowed. At runtime, and just before any memory access, the variable *addr* will take the value of the memory address targeted by any memory access operation. Note that the size of the *addr* variable is determined by the address size defined at the configuration section.

In the example of section 5.1, the execution of instructions is denied on both Stack and Malloc regions. In the Cst region, addresses from 0 to 20 are reserved for the program instruction, this is why writing is denied on this range of addresses. Otherwise, beyond the address 18, execution permission is denied.

```
# permissions
begin permissions
stack : (true : R W !X)
malloc : (true : R W !X)
cst : (\addr <=u 20<32> : R !W X)
      (\addr >u 20<32> : R W !X)
end permissions
```

5.5 Initialization

It is possible to give an initial value to each declared variable or memory locations. Constant values can be introduced in several ways:

- **Explicit size and implicit region:** in decimal representation of numbers, the size is specified between < and > just after the decimal value, ex: $16 < 8 >$. Whereas, in a hexadecimal representation of numbers the size is deduced from the number of symbols used to express the value, ex: 0x00000028 is on 32 bits but 0x28 is on 8 bits. The region is set to Cst by default.

- **Implicit size and region:** This kind of values representation can only be introduced in the initialization section and the value must be the right hand side of an assignment. The size of the value is deduced from the size of the left hand side of the assignment. The region is Cst by default, ex: $c2 := 1$.

- **Explicit region:** The value is expressed as a couple (r, bv), where r is either a Cst region or a Stack region (no use of malloc regions here) and bv can be expressed as in the previous cases, ex: $(cst, 8)$.

```
# initialisation
x := 8
y := 8
z := nondet(stack)
c1 := (cst ,8)
c2 := 1
c3 := 16<8>
c4 := 0x00000028
c5 := 11184810
c6 := 67
@[8<32>, 7] := 789865765654
@[(stack,8<32>), ->, 7] := \undef
c7 := 3456
c10 := malloc(12)
```

5.6 Code

Each address maps to an instruction pointing to the address of the next instruction.

```
# code
(0x00000000,0) x := 0x00000008 ; goto (0x00000001,0)
(0x00000001,0) x := x + c2; goto (0x00000002,0)
(0x00000002,0) if (x{1,1}) goto (0x00000003,0) else goto (0x00000004,0)
(0x00000003,0) goto x  // call (0x00000004,0)
(0x00000004,0) y := c3 ; goto (0x00000005,0)
(0x00000005,0) @[c4, ->, 3] := c5; goto (0x00000006,0)
(0x00000006,0) @[z, ->, 2] := c7; goto (0x00000007,0)
(0x00000007,0) print "printing values at runtime :\n" >>
                "@[z, ->, 2] = " >> @[z, ->, 2] >>
                ", x = " >> x{6,6} >> ", y = " >>   y >> "\n"; goto (0x00000008,0)
(0x00000008,0) nondet_assume ({@[c6, 2], y}, (y = 254<8>)) ; goto (0x00000009,0)
(0x00000009,0) assert (x <=u 1999990<32>) ; goto (0x0000000A,0)
(0x0000000A,0) x := alternative (x + c1, c1 + x); goto (0x0000000B,0)
(0x0000000B,0) print "x = " >> x >> ", c1 = " >> c1 >> "\n"; goto (0x0000000C,0)
(0x0000000C,0) c8 := (extu x 34) + 100<34>; goto (0x0000000D,0)
(0x0000000D,0) c9 := malloc (16); goto (0x0000000E,0)
(0x0000000E,0) @[c9, ->, 6] := 1234<48> ; goto (0x0000000F,0)
(0x0000000F,0) @[c4, ->, 4] := malloc (16); goto (0x00000010,0)
(0x00000010,0) @[@[c4, ->, 4], <-, 3] := 1234<24> ; goto (0x00000011,0)
(0x00000011,0) print "c4 = " >> c4 >> "\n"; goto (0x00000012,0)
(0x00000012,0) v := malloc(16); goto (0x00000013,0)
(0x00000013,0) free (@[(cst, 40<32>), ->, 4]); goto (0x00000014,0)
(0x00000014,0) stop OK
```

6 Implementation

6.1 Code organization

Our simulator of DBA models is implemented in OCaml language. The code is organized in several files as follows:

lexer.mll, parser.mly: recovery of the syntax tree from the textual description of a DBA model

dba.ml, dba.mli: description of the basic types of DBA

bitvector.ml, bitvector.mli: specification and implementation of the bit vector operations.

mmregion.ml: implementation of the memory model with several regions by redefining all possible operations on bit vectors.

eval.ml: evaluation of DBA expressions. This file contains also the read and write functions that control the memory accesses according to the specified permissions

simulate.ml: execution of a DBA instruction and returning the updated memory and the next instruction address

test.ml: launch of the simulation starting from a given initial address

utils.ml: Definition of all needed maps and data structures

options.ml: Parsing of arguments and definition of possible execution options, such as the number simulations introduced by "-fuzz" option

6.2 Example

To compile the source code to native code, run the **make nc** command from the source directory.

To use the simulator, simply run the following command always from the source directory : **./bincoa "example.dba" [-fuzz i]**

There are some examples of DBA files provided in the tests directory, ex: the command **./bincoa tests/test1.dba -fuzz 3** performs three simulations of the DBA model described in the "tests/test1.dba" file and gives the following results:

```
$ ./bincoa tests/test1.dba -fuzz 3
@SIMULATION(1):
printing values at runtime :
@[z, ->, 2] = Cst +32781, x = Cst +0, y = Cst +16
x = Cst +16, c1 = Cst +8
c4 = Cst +40
@MEMORY STATE AFTER SIMULATION:
\addr = Cst +2
c1 = Cst +8
c10 = Malloc1 +0
c2 = Cst +1
c3 = Cst +16
c4 = Cst +40
c5 = Cst +11184810
c6 = Cst +67
c7 = Cst +3456
c8 = Cst +116
c9 = Malloc2 +0
v = Malloc4 +0
x = Cst +16
y = Cst +254
z = Stack +403593985

Cst[40]        = (Malloc3 +0){0, 7}
Cst[41]        = (Malloc3 +0){8, 15}
Cst[42]        = (Malloc3 +0){16, 23}
Cst[43]        = (Malloc3 +0){24, 31}
Cst[67]        = Cst +98
Cst[68]        = Cst +70

Stack[403593985]       = Cst +128
Stack[403593986]       = Cst +13

Malloc2[0] = Cst +210
Malloc2[1] = Cst +4
Malloc2[2] = Cst +0
Malloc2[3] = Cst +0
Malloc2[4] = Cst +0
Malloc2[5] = Cst +0

Malloc3[0] = Cst +0
Malloc3[1] = Cst +4
Malloc3[2] = Cst +210


@SIMULATION(2):
printing values at runtime :
@[z, ->, 2] = Cst +32781, x = Cst +0, y = Cst +16
x = Cst +16, c1 = Cst +8
c4 = Cst +40
@MEMORY STATE AFTER SIMULATION:
\addr = Cst +2
c1 = Cst +8
c10 = Malloc1 +0
c2 = Cst +1
c3 = Cst +16
c4 = Cst +40
```

```
c5 = Cst +11184810
c6 = Cst +67
c7 = Cst +3456
c8 = Cst +116
c9 = Malloc2 +0
v = Malloc4 +0
x = Cst +16
y = Cst +254
z = Stack +989720480

Cst[40]      = (Malloc3 +0){0, 7}
Cst[41]      = (Malloc3 +0){8, 15}
Cst[42]      = (Malloc3 +0){16, 23}
Cst[43]      = (Malloc3 +0){24, 31}
Cst[67]      = Cst +108
Cst[68]      = Cst +223

Stack[989720480]      = Cst +128
Stack[989720481]      = Cst +13

Malloc2[0] = Cst +210
Malloc2[1] = Cst +4
Malloc2[2] = Cst +0
Malloc2[3] = Cst +0
Malloc2[4] = Cst +0
Malloc2[5] = Cst +0

Malloc3[0] = Cst +0
Malloc3[1] = Cst +4
Malloc3[2] = Cst +210


@SIMULATION(3):
printing values at runtime :
@[z, ->, 2] = Cst +32781, x = Cst +0, y = Cst +16
x = Cst +16, c1 = Cst +8
c4 = Cst +40
@MEMORY STATE AFTER SIMULATION:
\addr = Cst +2
c1 = Cst +8
c10 = Malloc1 +0
c2 = Cst +1
c3 = Cst +16
c4 = Cst +40
c5 = Cst +11184810
c6 = Cst +67
c7 = Cst +3456
c8 = Cst +116
c9 = Malloc2 +0
v = Malloc4 +0
x = Cst +16
y = Cst +254
z = Stack +976751075

Cst[40]      = (Malloc3 +0){0, 7}
Cst[41]      = (Malloc3 +0){8, 15}
Cst[42]      = (Malloc3 +0){16, 23}
Cst[43]      = (Malloc3 +0){24, 31}
Cst[67]      = Cst +22
Cst[68]      = Cst +94

Stack[976751075]      = Cst +128
Stack[976751076]      = Cst +13

Malloc2[0] = Cst +210
Malloc2[1] = Cst +4
Malloc2[2] = Cst +0
Malloc2[3] = Cst +0
Malloc2[4] = Cst +0
Malloc2[5] = Cst +0

Malloc3[0] = Cst +0
Malloc3[1] = Cst +4
Malloc3[2] = Cst +210
```

References

[1] Sébastien Bardin, Philippe Herrmann, Jérôme Leroux, Olivier Ly, Renaud Tabary, and Aymeric Vincent. The bincoa framework for binary code analysis. In *CAV*, pages 165–170, 2011.

[2] Sandrine Blazy and Xavier Leroy. Mechanized semantics for the clight subset of the c language. *J. Autom. Reasoning*, 43(3):263–288, 2009.

[3] Andy King, Alan Mycroft, Thomas W. Reps, and Axel Simon. Analysis of executables: Benefits and challenges (dagstuhl seminar 12051). *Dagstuhl Reports*, 2(1):100–116, 2012.