

Inferring Variables From Executables

G. Ramalingam Pushkar Tripathi
Lakshmisubrahmanyam Velaga

November 11, 2007

1 Introduction

In this paper, we address some issues that arise in the context of static analysis of binaries or executables. In several contexts, a program that needs to be analyzed may be available only in binary form: e.g., third party code, potentially malicious code, etc. One example application of binary analysis is in *maintaining application compatibility* when libraries evolve. It is not uncommon for clients of a library to rely on certain aspects of the library's *implementation* that are not guaranteed by the published API (interface). In such cases, changes to the library implementation that do not affect the published API may, nevertheless, break some existing clients. In such cases, analysis of client binaries can help in making decisions relating to implementation changes.

Though we focus on binaries in this paper, the results presented here are applicable to the analysis of code written in languages such as C as well, as we will see.

A number of analyses and tools have been developed for source languages such as C. One way to reuse such analyses and tools is by first translating a given binary program into a C program. This paper addresses the problem referred to by Reps and Balakrishnan as that of *recovering an intermediate representation* from executables. We specifically focus on formalizing a notion of correctness for the problem of *identifying variables* in executables, and using the formalization to derive an abstract interpretation for the problem.

Symbolic information, such as variables and types, available in safe and strongly-typed languages provide certain semantic guarantees about the program execution that are often exploited by static analyses. (Such guarantees are sometimes assumed by analyses even when the language itself does not provide them because analysis designers choose to tradeoff some soundness guarantees for the improvements achieved in terms of precision, efficiency, or implementation complexity.)

Examples of such assumptions include the following. Many field-sensitive analyses assume that different fields of different types can not refer to the same location. An analysis may assume that a local variable of a procedure whose address is not taken can not be modified by an indirect assignment. We want the symbolic information we infer to provide these guarantees.

An Example. Consider the example shown in Fig. 1. The local variables of the example procedure make up a block 4 words long in the stack. These words are located at addresses $bp + i$, $0 \leq i \leq 3$, where bp is a special *base register* that points to the procedure’s activation record. (For the sake of simplicity, we assume here that all instruction operands are 1 word long, and that every location and register stores one word.) The example also makes use of two registers $r1$ and $r2$ as temporaries.

We show three possible solutions to the problem of identifying variables in this example program. The solutions are presented in the form of C code (such that there is a one-to-one correspondence between the C statements and the instructions in the input assembler program).

The first solution models the procedure as having a single variable y (an array of 4 words). Such a solution is correct, and can always be used, but is imprecise in the following sense. Modeling distinct variables as elements of the same array is undesirable because it makes reasoning about the program harder. Many existing analysis (such as pointer analyses) do not make distinctions between different elements of a single array. E.g., with such analyses the assignment statement “ $*(r1+1) = 3$ ” may be seen as potentially updating any of the four words. Hence, a solution with more variables can lead to more precise analysis results downstream than a solution with fewer variables.

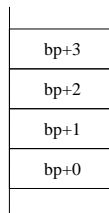
The second solution models the procedure as having four different variables $x0$, $x1$, $x2$, and $x3$ (of size 1 word each). This solution is, however, incorrect. The semantics of C does not guarantee that this program will have the same behavior as the input program. Specifically, this is an ill-behaved program as its semantics depends on compiler decisions (relating to mapping variables to locations). If this program is passed as input to some analyses, they may potentially decide that the assignment statement “ $*(r1+1) = 3$ ” does not modify any of the variables $x0$, $x2$, or $x3$ and, hence, (erroneously) infer that the value of $x2$ used in the subsequent statement must be 1.

The problem in the above case is that a set of locations that should have been modeled as a single variable has been split into two distinct variables $x1$ and $x2$. Such *over-refinement* leads to unsound analysis results. The third program models the procedure as having three variables $v0$ (of size 1 word), $v1$ (an array of 2 words), and $v2$ (of size 1 word). This solution is both correct and precise and represents the kind of solution we would like to recover.

Outline of Our Approach The approach we take in this paper is as follows: we introduce a simple assembly language (with no symbolic information) and present a semantics for this language; we then introduce a modified version of the language that allows the use of symbolic variables in place of address constants, and present a modified semantics for this language; this modified semantics corresponds to a higher-level language interpretation/view of variables and captures the assumptions commonly made by static analyses about the *independence* of variables; the inference problem can then be formalized as that of producing a symbolic program whose semantics is bisimilar to that of the

store [ebp],1	word y[4]; y[0] = 1;	word x0, x1, x2, x3; x0 = 1;	word v0, v1[2], v3; v0 = 1;
store [ebp+2],1	y[2] = 1;	x2 = 1;	v1[1] = 1;
load r1,[ebp+1]	r1 = &y[1];	r1 = &x1;	r1 = &v1[0];
store [r1+1],3	*(r1+1) = 3;	*(r1+1) = 3;	*(r1+1) = 3;
load r2,[ebp+2]	r2 = y[2];	r2 = x2;	r2 = v1[1];
store [ebp+3],r2	y[3] = r2;	x3 = r2;	v3 = r2;
(a)	(b)	(c)	(d)

Figure 1: (a) An example assembly program. (b) A correct, but imprecise, variable identification. (c) An incorrect variable identification. (d) A correct and precise variable identification.



original program. We use this definition to derive an abstract interpretation that helps identify variables in an assembly program.

2 The Language and Its Semantics

We first introduce a simple assembly language intended to represent binaries without symbolic information. Let *int* denote the set of integers, and let *reg* denote a finite set of registers. Let *bp* denote a special register, *not in reg*, that is used as the *base* address of the current activation record. We make the simplifying assumption that *bp* is modified only during a procedure call and return. (We will initially consider single-procedure programs, where we assume that this register is not modified by the program.) Note that we separate code from data in our language and model. Thus, program execution can not modify program instructions. We want to support the kind of pointer arithmetic usually found in binaries and C programs. We include instructions for *addition* and *subtraction* in our instruction set as they are commonly used for pointer arithmetic. We include an instruction for *multiplication* as a representative of all operations we do not wish to support for pointer values. We also make the simplifying assumption that all operands are of the same size and ignore overflow/underflow that can arise due to the use of limited precision integers (e.g., 32 bit integers).

int	::=	set of integers
reg	::=	a finite set of registers
bp	::=	base register($bp \notin reg$)
$direct-addr$::=	$int \mid [bp] + int$
$indirect-addr$::=	$[reg]$
$addrexp$::=	$direct-addr \mid indirect-addr$
$pc \in iaddr$::=	int
$inst$::=	load $reg, addrexp \mid$ store $addrexp, reg \mid$ lea $reg, addrexp \mid$ loadi $reg, int \mid$ mov $reg, reg \mid$ add $reg, reg, reg \mid$ sub $reg, reg, reg \mid$ mul $reg, reg, reg \mid$ jeq $reg, reg, iaddr \mid$
pgm	::=	$iaddr \rightarrow inst$

We now present a semantics for the language.
Semantic Domains:

$$\begin{aligned}
l \in loc &= int \\
\rho \in \mathcal{R} &= reg \rightarrow int \\
\mu \in \mathcal{M} &= loc \rightarrow int \\
\sigma \in \Sigma &= \mathcal{R} \times \mathcal{M} \times iaddr
\end{aligned}$$

Note that a program state σ is a triple. We will denote the first component of σ (a map from registers to values) by ρ_σ , the second component (a map from locations to values) by μ_σ , and the third component (the program counter) by pc_σ .

Operational semantics:

We first consider *address evaluation*.

$$\begin{array}{c}
\frac{i \in int}{\sigma \vdash i \rightsquigarrow_{lv} i} \\
\frac{\sigma \vdash i \rightsquigarrow_{lv} i}{r \in reg} \\
\frac{\sigma \vdash [r] \rightsquigarrow_{lv} \rho_\sigma(r)}{r \in reg, i \in int} \\
\hline
\sigma \vdash [r] + i \rightsquigarrow_{lv} \rho_\sigma(r) + i
\end{array}$$

$$\begin{array}{c}
\frac{(\rho, \mu, pc) \vdash a \rightsquigarrow_{lv} l, v = \mu[l]}{\text{load } r, a \vdash (\rho, \mu, pc) \rightsquigarrow (\rho[r \mapsto v], \mu, pc + 1)} \\
\frac{(\rho, \mu, pc) \vdash a \rightsquigarrow_{lv} l}{\text{lea } r, a \vdash (\rho, \mu, pc) \rightsquigarrow (\rho[r \mapsto l], \mu, pc + 1)} \\
\frac{(\rho, \mu, pc) \vdash a \rightsquigarrow_{lv} l, v = \rho[r]}{\text{store } a, r \vdash (\rho, \mu, pc) \rightsquigarrow (\rho, \mu[l \mapsto v], pc + 1)} \\
\frac{v_1 = \rho[r_1], v_2 = \rho[r_2]}{\text{add } r_0, r_1, r_2 \vdash (\rho, \mu, pc) \rightsquigarrow (\rho[r_0 \mapsto v_1 + v_2], \mu, pc + 1)} \\
\frac{\rho[r_1] \neq \rho[r_2]}{\text{jeq } r_0, r_1, lbl \vdash (\rho, \mu, pc) \rightsquigarrow (\rho, \mu, pc + 1)} \\
\frac{\rho[r_1] = \rho[r_2]}{\text{jeq } r_0, r_1, lbl \vdash (\rho, \mu, pc) \rightsquigarrow (\rho, \mu, lbl)} \\
\frac{P[pc] \vdash (\rho, \mu, pc) \rightsquigarrow (\rho', \mu', pc')}{(\rho, \mu, pc) \rightsquigarrow_P (\rho', \mu', pc')}
\end{array}$$

3 Well-Behaved Programs

We restrict our attention to programs where every stored value may be typed as being *address* or *data*. We formalize this restriction below.

A *concrete location typing* for a program is a function $\tau : \Sigma \rightarrow (\text{reg} + \text{loc}) \rightarrow \{\text{data}, \text{address}\}$. Such a typing is said to be correct if

1. The arguments and results of all operations (arithmetic and equality comparison) are well-typed.
2. Any value used to dereference memory is typed *address*.
3. The result produced by a lea instruction is typed *address*.
4. The typing is consistent across state transitions. Thus, if state σ_2 is obtained from state σ_1 by copying the value in x (a register or a memory location) to y (a register or a memory location), then $\tau(\sigma_2, z) = \tau(\sigma_1, z)$ for all $z \neq y$ and $\tau(\sigma_2, y) = \tau(\sigma_1, x)$.

A program is said to be *well-behaved* if it has a correct typing.

4 A Symbolic Assembly Language

We now extend our assembly language with the ability to use symbolic variables (in place of address constants). Let *var* denote a set of variable (identifiers).

The symbolic assembly language is the same as the original language, except that we replace the definition of *addrxp* as follows:

$$\text{addrxp} ::= \text{var} \mid \text{var} + \text{int} \mid [\text{reg}]$$

Variables may be either local or global variables. Variables have an associated (base) address (an absolute address in the case of global variables and an offset to be added to the address of an activation record in the case of local variables) and a size. We capture this (declarative) information about variables in an environment as below. An environment $D \in \mathcal{E}$ is a tuple $(L_D, G_D, \text{base}_D, \text{size}_D)$, where $L_D \subseteq \text{var}$ is the set of local variables, $G_D \subseteq \text{var}$ is the set of global variables, L_D and G_D are disjoint, $\text{base}_D : (L_D \cup G_D) \rightarrow \text{int}$ represents the base address of a variable, and $\text{size}_D : (L_D \cup G_D) \rightarrow \text{int}$ represents the size of a variable.

An environment, as described above, can be used to define a semantics for a symbolic assembly program. A symbolic address can be evaluated to a real address (i.e., an integer) using the current state and an environment. As a first approximation, our goal is to infer from a given program P a symbolic program P_s and an environment D such that the semantics determined by (P_s, D) is *bisimilar* to the semantics determined by P .

However, this is not quite enough. We want to impose some extra conditions.

Illustrate what we want via examples?

In order to formalize the requirements, we introduce a slightly altered semantics that uses a generalized store where addresses are represented as an ordered pair consisting of a variable and an offset (instead of an integer). We say that a pair (v, i) is a *valid* address (with respect to an environment D) iff $0 \leq i < \text{size}_D(v)$. We say that an environment is *valid* if for distinct valid addresses (v, i) and (w, j) , $\text{base}_D(v) + i \neq \text{base}_D(w) + j$. A *storable value* is either an integer or an address.

Semantic Domains:

$$\begin{aligned} \text{loc}' &= \text{var} \times \text{int} \\ \mathcal{V}' &= \text{int} + \text{loc}' \\ \rho \in \mathcal{R}' &= \text{reg} \rightarrow \mathcal{V}' \\ \mu \in \mathcal{M}' &= \text{loc}' \rightarrow \mathcal{V}' \\ \sigma \in \Sigma' &= \mathcal{R}' \times \mathcal{M}' \times \text{iaddr} \end{aligned}$$

We extend the semantics of arithmetic operations to support address arithmetic. Specifically, we allow the standard arithmetic on integer values; the only non-standard arithmetic supported is the addition of an address and an integer, the subtraction of an integer from an address, the subtraction of two addresses with the same variable component. However, we do allow these operations to generate “invalid” addresses (essentially addresses with an offset component that is not in the valid range of offsets associated the corresponding variable). Thus, note that all these operations are *partial*: they are not defined for all possible input values; thus, evaluation may get *stuck* if an operation is applied to operands of an inappropriate type.

$$\begin{aligned}
i \oplus j &\rightsquigarrow i + j \\
(v, i) \oplus j &\rightsquigarrow (v, i + j) \\
i \oplus (v, j) &\rightsquigarrow (v, i + j) \\
i \ominus j &\rightsquigarrow i - j \\
(v, i) \ominus j &\rightsquigarrow (v, i - j) \\
i \otimes j &\rightsquigarrow i * j
\end{aligned}$$

We now consider *address evaluation*:

$$\frac{v \text{ is a variable}}{\sigma \vdash v \rightsquigarrow_{lv} (v, 0)}$$

$$\frac{v \text{ is a variable}}{\sigma \vdash v + i \rightsquigarrow_{lv} (v, i)}$$

$$\frac{}{\sigma \vdash [r] \rightsquigarrow_{lv} \rho_{\sigma}(r)}$$

$$\frac{0 \leq i < \text{size}_D(v)}{D \vdash \text{valid}(v, i)}$$

$$\frac{(\rho, \mu, pc) \vdash a \rightsquigarrow_{lv} l, D \vdash \text{valid}(l), v = \mu[l]}{D, \text{load } r, a \vdash (\rho, \mu, pc) \rightsquigarrow (\rho[r \mapsto v], \mu, pc + 1)}$$

$$\frac{(\rho, \mu, pc) \vdash a \rightsquigarrow_{lv} l,}{D, \text{lea } r, a \vdash (\rho, \mu, pc) \rightsquigarrow (\rho[r \mapsto l], \mu, pc + 1)}$$

$$\frac{(\rho, \mu, pc) \vdash a \rightsquigarrow_{lv} l, D \vdash \text{valid}(l), v = \rho[r]}{D, \text{store } a, r \vdash (\rho, \mu, pc) \rightsquigarrow (\rho, \mu[l \mapsto v], pc + 1)}$$

$$\frac{\rho[r_1] \oplus \rho[r_2] \rightsquigarrow v}{D, \text{add } r_0, r_1, r_2 \vdash (\rho, \mu, pc) \rightsquigarrow (\rho[r_0 \mapsto v], \mu, pc + 1)}$$

$$\frac{D, P[pc] \vdash (\rho, \mu, pc) \rightsquigarrow (\rho', \mu', pc')}{(\rho, \mu, pc) \rightsquigarrow_{(P, D)} (\rho', \mu', pc')}$$

Now, we are almost ready to formalize the inference problem. Given a concrete instruction ci and a symbolic instruction si , we say that $ci \simeq si$ if the two instructions are the same except for the replacement of a concrete *address* by a symbolic *address*. Given a program P , we say that a symbolic program \hat{P} is a symbolic transformation of P if $P(i) \simeq \hat{P}(i)$ for every instruction location i .

Given a program P , we want to infer a symbolic transformation \hat{P} and an environment D such that the semantics determined by (\hat{P}, D) is essentially the same as the semantics determined by P .

We define $glocs_D(\sigma)$ to be the set $\{base_D(g) + i \mid g \in G, 0 \leq i < size_D(g)\}$. We define $llocs_D(\sigma)$ to be the set $\{\rho_\sigma[bp] + base_D(l) + i \mid l \in L, 0 \leq i < size_D(l)\}$. We define $locs_D(\rho)$ to be $glocs_D(\sigma) \cup llocs_D(\sigma)$. We say that a state σ is *collision-free* if $glocs_D(\sigma)$ and $llocs_D(\sigma)$ are disjoint.

In order to relate the two semantics, we first define a notion of equivalence between the program state as encoded by the symbolic semantics and the program state as encoded by the original semantics.

Note that, in general, a *value* manipulated by a binary program may be either *address* or *data*. In the standard semantics both types of values are represented using integers. The symbolic semantics uses distinct representations for these two types of values.

We define a function $stoc_D : int \times \mathcal{V}' \rightarrow \mathcal{V}$ that transforms a symbolic value into a standard value as follows:

$$\begin{aligned} stoc_D(b, i) &= i \\ stoc_D(b, (v, i)) &= \begin{cases} base_D(v) + i & v \in G_D \\ b + base_D(v) + i & v \in L_D \end{cases} \end{aligned}$$

We say that a location ℓ corresponds to a variable v if there exists a valid address (v, i) such that $stoc_D(b, (v, i))$ is ℓ .

We define corresponding functions $stoc_D : int \times \mathcal{R}' \rightarrow \mathcal{R}$, $stoc_D : int \times \mathcal{M}' \rightarrow \mathcal{M}$, and $stoc_D : \Sigma' \rightarrow \Sigma$:

1. $stoc_D(b, \rho')(r) = stoc_D(b, \rho'(r))$
2. $stoc_D(b, \mu)(j) = \begin{cases} stoc_D(b, \mu'(v, i)) & \exists \text{ valid } (v, i) \text{ such that } base_D(v) + i = j \\ 0 & \text{otherwise} \end{cases}$
3. $stoc_D(\rho, \mu, pc) = (stoc_D(b, \rho), stoc_D(b, \mu), pc)$

Converting a concrete value or state into a symbolic value or state, however, is not straightforward. Specifically, we need extra *type* information to decide whether an integer value should be treated as *data* or *address*; even if we know that a concrete value represents an *address*, we can't be sure what *symbolic* address it represents (though it can represent at most one *valid* symbolic address). (In other words, the function $stoc_D : \mathcal{V}' \rightarrow \mathcal{V}$ is many-to-one.) So, instead, we just define what it means for a transformation function to be a correct encoding.

We say that a function $ctos : \Sigma \rightarrow \Sigma'$ is a *decoding function* if for every concrete state σ , $stoc_D(ctos(\sigma)) \cong \sigma$, where $(\rho_1, \mu_1, pc_1) \cong (\rho_2, \mu_2, pc_2)$ iff $\rho_1 = \rho_2$, $\mu_1(l) = \mu_2(l)$ for every $l \in locs_D(\rho_1)$, $pc_1 = pc_2$, (where \cong is equivalence modulo our set of variables).

Note: A decoding function is essentially a *program-point-dependent* typing function.

Definition 1. Let P be a program, \widehat{P} a symbolic program, and D an environment. We say that (\widehat{P}, D) identifies the variables in P correctly if there exists a decoding function $ctos_D$ such that for any collision-free initial concrete state σ_1 ,

$$P \vdash \sigma_1 \xrightarrow{\pi} \sigma_2 \Leftrightarrow (\widehat{P}, D) \vdash ctos(\sigma_1) \xrightarrow{\pi} ctos(\sigma_2)$$

5 A Characterization of the Solution

We now describe an algorithm for correctly identifying variables for a well-behaved program.

The key to determining variables is to identify the *origin* of any location dereferenced during program execution. Specifically, an address value in a program state may be generated in one of the following ways: an address value may be generated by the evaluation of a *direct-addr* expression, in which case the *direct-addr* expression is defined to be the origin of the address value; or an address value may be in the initial program state (i.e., it may be a parameter to the procedure), in which case the initial location that contains this address value is defined to be the origin of the address value; or an address value u may be computed by adding or subtracting some offset to another address value v , in which case the origin of u is defined to be the same as the origin of v . (We ignore other ways of generating address values, a point we will discuss later.)

We now introduce the notion of a *source*. A source is either an occurrence of a *direct-addr* in a program instruction or the value stored in a location or register in the initial program state. We represent the source corresponding to the occurrence of a *direct-addr* in the instruction at program location x by cs_x and the source corresponding to a memory location or register y by ds_y .

We present an instrumented version of the standard semantics in which an address value is represented as a pair consisting of a location and a source. The instrumented semantics permits only well-typed operations (both arithmetic as well as comparison operations). Any attempt to perform an ill-typed operation causes program execution to get stuck. We will subsequently use an abstract interpretation over the instrumented semantics to compute an environment.

Note that all of the following definitions are with respect to the execution of the program (under the instrumented semantics) on an initial state σ , and are, hence, implicitly parameterized by σ .

Let s_1 and s_2 be two sources. We say that $s_1 \approx s_2$ if the program execution leads to a comparison or subtraction of two address values with sources s_1 and s_2 . We define a binary relation \rightsquigarrow between sources and locations as follows: if a source s is used to generate a dereferenced location l , we say $s \rightsquigarrow l$. We say that two locations l_1 and l_2 have a common source, denoted $l_1 \approx l_2$, if there exist sources s_1 and s_2 such that $s_1 \approx s_2$, $s_1 \rightsquigarrow l_1$, and $s_2 \rightsquigarrow l_2$.

Theorem 1. *Let P be a program that does not get stuck for any input state σ . An environment D identifies variables correctly for a P if, for any input state σ , any two locations that have a common source correspond to the same variable.*

Note: A program that does not get stuck is well-behaved.

Thus, our abstract interpretation algorithm consists of two components. First, it conservatively determines if the program will get stuck (is well-behaved). Second, it determines a conservative approximation of the \approx relation (over all possible input states). **restate in terms of local offsets and global locations?**