Alias Analysis for Assembly

David Brumley and James Newsome

December 15, 2006
CMU-CS-06-180R

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

We propose using Datalog for alias analysis of binary programs. Our alias analysis reasons about whether two memory references may access the same memory cell.

Keywords: assembly alias, x86 alias, datalog

1 Introduction

In order to reason about a program accurately, it is important to be able to reason about memory operations. While many tools exist for accurately analyzing programs written in high level languages, almost none exist for analyzing low-level assembly. However, most users only have access to the program assembly code (via the program binary). Some common scenarios in which a user would like to accurately analyze assembly include:

- Generating vulnerability signatures. A vulnerability signature matches all exploits of a given vulnerability, even polymorphic variants. Two exploits are polymorphic variants if they are syntactically different yet exploit the same underlying vulnerability. Previously, we have shown how to generate vulnerability signatures based upon the program binary [3]. Our techniques allow users to generate accurate signatures without involving the software vendor. However, in order to generate an accurate signature, we must reason about memory reads and writes. Previously, we have relied upon a general-purpose theorem prover to resolve potential alias relationships. A general purpose alias analysis would allow our techniques to scale to larger programs and generate signatures more efficiently.

- Application replay. The ability to accurately replay application dialogs is useful in many applications, such as replaying an exploit for forensic analysis or demonstrating an exploit to a third party. A central challenge in application dialog replay is that the dialog intended for the original host will likely not be accepted by another without modification. For example, the dialog may include or rely on state specific to the original host such as its host-name, a known cookie, etc. In such cases, a straight-forward byte-by-byte replay to a different host with a different state (e.g., different host-name) than the original observed dialog participant will likely fail. These state-dependent protocol fields must be updated to reflect the different state of the different host for replay to succeed. We have previously shown one method for sound application dialog replay can be accomplished via binary program analysis [12]. As in signature generation, the efficiency and scalability of our approach could greatly benefit from accurate alias analysis.

- Detecting trigger-based behavior. Software logic and time bombs, as well as Trojan horses, are often triggered based upon specified inputs such as the current time of the day or given password. We have developed techniques which automatically determine such triggers, e.g., discovering malware will launch a denial-of-service attack on a particular date. However, our techniques are not sound, which means when our analysis discovers a potential trigger, we must then verify it with other means such as resetting the system date and looking for the malicious behavior. Accurate alias analysis would not only improve the efficiency and scalability of our approach, but it would also help towards proving soundness of our techniques.

- Disassembly. Disassembly involves reading the bits from a program binary and interpreting them as assembly instructions that the CPU would execute. In modern architectures, a central problem for correct disassembly is dealing with indirect jumps. An indirect jump is of the form `jmp *eax`, where `eax` is a register that holds a computed value. Indirect jump operands are often derived via memory operations, e.g., a return instruction first loads an address from the stack into a register, then jumps to the target. Thus, we must know the target of the indirect jump in order to know which bits to interpret as assembly instructions.

Without alias analysis, in each of the above scenario when a memory read or write is encountered we must consider it a possible alias with all other memory operations. This prevents efficient whole-program analysis.

1.1 Source code analysis techniques are insufficient

Naively, one may think that techniques for analyzing source code are sufficient for analyzing an executable. However, analyzing binary machine code presents many challenges, including:

- Assembly lacks expressive types. At best, types are n-bit integers where n is the length of the register holding the value during computation. Traditional analysis often uses types to prune out irrelevant statements, i.e., if an operation is of a different type than our subject of interest, we need not consider it. The lack of types in assembly means we often must consider every statement.

- Lack of function abstractions. Assembly control flow is simple: unconditional and conditional jumps to locations. Higher-order abstractions such as functions do not necessarily exist, even when the machine code is produced from higher level languages with such abstractions!

- Memory is treated as one contiguous chunk. This introduces a number of challenges. First, it is often difficult to tell where one object ends and one begins. Second, allocation and deallocation sites may be implicit, e.g., after a function returns it is still possible to refer to variables in "deallocated" frame. Third, memory is byte-accessible though code-pointers are word size. Thus, in IA32 we may store 2 code pointers on the stack, and forge a new pointer by reading 2 bytes from the first stored pointer and two bytes from the second stored pointer.

- Memory addresses are reused for different purposes. For example, memory cells on the stack are reused for different purposes in different functions. Even within a single function, stack slots may be used as generic locations for register spills.

- Many pointers require expression evaluation. Many source-based alias analysis work best when address arithmetic is limited or need not be considered. At the assembly level, *almost all* memory dereferences involve arithmetic. For example, arguments passed on the stack are known via a positive offset in the caller, and via a negative offset in the callee.

- Control flow analysis is hampered by the widespread use of indirect jumps. In the worst case, jump targets can be code created on the fly or to the middle of previously dissassembled instructions, making control-flow analysis nearly impossible. Even if we assume a normal compiler produces the code in the first place, many typical optimizations can result in hard-to-predict code flow.

Thus, although there are many important applications of binary analysis, the above challenges makes such analysis quite difficult.

1.2 Related work

Our work proposes alias analysis for assembly. The most closely related work is that of Balakrishnan and Reps [1, 13, 2], propose value set analysis (VSA). The information VSA discovers allows two memory references at a given point P to be interpreted with respect to the set of possible states that can arise at P, which makes it possible to determine (i) whether they definitely cannot access the same memory cell, or (ii) whether they might access the same memory cell (and thus provides a way to answer may-alias queries). At a high level, our work takes a logic-based approach where we use Datalog to derive all possible alias relationships, and use the Datalog rules as a foundation for a formal correctness proof. Debray et al [15] and Cifuentes and Fraboulet [6] both present methods that reason about values a register can hold, but unlike our approach, do not reason across memory operations.

There is a long history of projects which disassemble and analyze executables. Notable recent work which disassembles executables and translates to an intermediate representation for further analysis include that of Microsoft Phoenix [11], Vulcan [8], and Boomerang [9]. Our alias analysis could be implemented on top of these platforms.

Our approach using Datalog is motivated by the success of Whaley et al. using a BDD-based Datalog database to scale Java points-to analysis [18].

1.3 Paper Overview

At a high level, we develop techniques for alias analysis of assembly code. We disassemble and translate a program into a unambiguous IR. The alias analysis is expressed in Datalog over the IR. The resulting saturated database contains all alias relationships. Subsequent program analysis queries the database.

In this paper, we first describe our RISC-like assembly language, and give its operation semantics (Section 2). We also describe some important assumptions about the semantics of the program (Section 2.3). We then provide a correctness proof sketch (Section 4). We then describe our implementation and initial results (Section 5 and Section 6). We finish future directions and conclusions.

2 Alias Analysis Overview

In this section, we first present the assembly language for which we perform alias analysis. We then present our approach for alias analysis using Datalog.

2.1 The Assembly Language

We consider alias analysis for the RISC-like assembly language shown in Table 1. We are able to convert typical x86 programs into this language

Instructions	i	::=	$*(r_1) := r_2 \mid r_1 := *(r_2) \mid r := v \mid r := r_1 \Box_b r_2$
			$\mid r := \Box_u r_1 \mid$ `label` l_i $\mid$ nop $\mid$halt
			$\mid$ `jmp` ℓ $\mid$ `ijmp` r $\mid$ `if` r `jmp` ℓ_1 `else jmp` ℓ_2
Operations	$\Box_b$	::=	$+, -, *, /, \ll, \gg, \&, \mid, \oplus, =, \neq, <, \leq$ (Binary operations)
	$\Box_u$	::=	$\neg, !$ (unary operations)
Operands	v	::=	n (an integer literal)
			r (a register)
			ℓ (a label)
Instructions	$\mathcal{I}$	::=	$n \mapsto i$ (Maps instruction number to instruction)
Memory	$\mathcal{M}$	::=	$n \mapsto n$ (Maps address to numeric value)
Register	$\mathcal{R}$	::=	$r_i \mapsto n$ (Maps register name to numeric value)
Labels	$\mathcal{L}$	::=	$l_n \mapsto pc$ (Maps label to instruction address pc)
Machine state	S	::=	($\mathcal{I}$,$\mathcal{L}$,$\mathcal{M}$,$\mathcal{R}$,pc,i)

Table 1: Abstract Machine Syntax

(we omit the details for some features such as system calls). Programs are written imperatively as a sequence of instructions. Instruction operands are either integer literals n, registers r, or labels ℓ. There are three basic kinds of instructions: a) memory operations, b) assignments, and c) control flow.

Table 1 also shows the abstract machine syntax. In addition to instructions, we maintain a mapping from instruction numbers to instructions $\mathcal{I}$, a memory $\mathcal{M}$, a register file $\mathcal{R}$, and a map from labels to instruction addresses $\mathcal{L}$. The machine state is given by the tuple $(\mathcal{I}, \mathcal{L}, \mathcal{M}, \mathcal{R}, pc, i)$, where pc is the program counter and i is the current instruction.

Our operational semantics are given in Table 2. A machine step is a transition $(\mathcal{I}, \mathcal{L}, \mathcal{M}, \mathcal{R}, pc, i) \to (\mathcal{I}, \mathcal{L}, \mathcal{M}', \mathcal{R}', pc', i')$. For sequential control flow, executing i will advance the program counter $pc' = pc + 1$ and load the instruction i' at pc'. For jump instructions, we use an auxiliary label map $\mathcal{L}$ which maps the jump target to a new program counter. We use the notation $\mathcal{R}(r) \mapsto n$ to indicate $\mathcal{R}$ maps the register r to the value n (and similarly for $\mathcal{M}$). We use the notation $\mathcal{R}[r = n]$ to mean the mapping $\mathcal{R}$, with key r remapped to value n (and similarly for $\mathcal{M}$). We assume that $\mathcal{I}$ and $\mathcal{L}$ are immutable throughout execution. One important implication is this means our machine cannot create new code and transfer control to it on the fly. We leave such extensions to future work.

Memory operations are either loads or stores; load $r_1 = *r_2$ loads into register r_1 the value in memory addressed by r_2, and store $*r_1 = r_2$ which

$$\frac{\mathcal{L}(\ell) \mapsto pc' \quad \mathcal{I}(pc') \mapsto \text{label } \ell}{(\mathcal{I}, \mathcal{L}, \mathcal{M}, \mathcal{R}, pc, \text{jmp } \ell) \to (\mathcal{I}, \mathcal{L}, \mathcal{M}, \mathcal{R}, pc', \text{label } \ell)} \text{ JMP}$$

$$\frac{\mathcal{R}(r) \mapsto n \quad \hat{L}(n) \mapsto \ell \quad \mathcal{L}(\ell) \mapsto pc' \quad \mathcal{I}(pc') \mapsto \text{label } \ell}{(\mathcal{I}, \mathcal{L}, \mathcal{M}, \mathcal{R}, pc, \text{ijmp } r) \to (\mathcal{I}, \mathcal{L}, \mathcal{M}, \mathcal{R}, pc', \text{label } \ell)} \text{ IJMP}$$

$$\frac{\mathcal{R}(r) \mapsto 0 \quad \mathcal{L}(\ell_1) \mapsto pc' \quad \mathcal{I}(pc') \mapsto \text{label } \ell_1}{(\mathcal{I}, \mathcal{L}, \mathcal{M}, \mathcal{R}, pc, \text{if } r \text{ jmp } \ell_1 \text{ else jmp } \ell_2) \to (\mathcal{I}, \mathcal{L}, \mathcal{M}, \mathcal{R}, pc', \text{label } \ell_1)} \text{ CJMP-T}$$

$$\frac{\mathcal{R}(r) \neq 0 \quad \mathcal{L}(\ell_2) \mapsto pc' \quad \mathcal{I}(pc') \mapsto \text{label } \ell_2}{(\mathcal{I}, \mathcal{L}, \mathcal{M}, \mathcal{R}, pc, \text{if } r \text{ jmp } \ell_1 \text{ else jmp } \ell_2) \to (\mathcal{I}, \mathcal{L}, \mathcal{M}, \mathcal{R}, pc', \text{label } \ell_2)} \text{ CJMP-F}$$

$$\frac{\mathcal{I}(pc+1) \mapsto i}{(\mathcal{I}, \mathcal{L}, \mathcal{M}, \mathcal{R}, pc, \text{label } \ell) \to (\mathcal{I}, \mathcal{L}, \mathcal{M}, \mathcal{R}, pc+1, i)} \text{ LABEL}$$

$$\frac{\mathcal{I}(pc+1) \mapsto i}{(\mathcal{I}, \mathcal{L}, \mathcal{M}, \mathcal{R}, pc, \text{nop}) \to (\mathcal{I}, \mathcal{L}, \mathcal{M}, \mathcal{R}, pc+1, i)} \text{ NOP}$$

$$\frac{\mathcal{R}(r_1) \mapsto n_1 \quad \mathcal{R}(r_2) \mapsto n_2 \quad n_3 = n_1 \square_b n_2 \quad \mathcal{I}(pc+1) \mapsto i}{(\mathcal{I}, \mathcal{L}, \mathcal{M}, \mathcal{R}, pc, r_3 := r_1 \square_b r_2) \to (\mathcal{I}, \mathcal{L}, \mathcal{M}, \mathcal{R}[r_3 = n_3], pc+1, i)} \text{ BINOP}$$

$$\frac{\mathcal{R}(r_1) \mapsto n_1 \quad n_2 = \square_u n_1 \quad \mathcal{I}(pc+1) \mapsto i}{(\mathcal{I}, \mathcal{L}, \mathcal{M}, \mathcal{R}, pc, r_2 := \square_u r_1) \to (\mathcal{I}, \mathcal{L}, \mathcal{M}, \mathcal{R}[r_2 = n_2], pc+1, i)} \text{ UNOP}$$

$$\frac{\mathcal{R}(r_1) \mapsto n \quad \mathcal{I}(pc+1) \mapsto i}{(\mathcal{I}, \mathcal{L}, \mathcal{M}, \mathcal{R}, pc, r_2 := r_1) \to (\mathcal{I}, \mathcal{L}, \mathcal{M}, \mathcal{R}[r_2 = n], pc+1, i)} \text{ MOVE}$$

$$\frac{\mathcal{I}(pc+1) \mapsto i}{(\mathcal{I}, \mathcal{L}, \mathcal{M}, \mathcal{R}, pc, r := n) \to (\mathcal{I}, \mathcal{L}, \mathcal{M}, \mathcal{R}[r = n], pc+1, i)} \text{ MOVEC}$$

$$\frac{\mathcal{R}(r_2) \mapsto n_1 \quad \mathcal{M}(n_1) \mapsto n_2 \quad \mathcal{I}(pc+1) \mapsto i}{(\mathcal{I}, \mathcal{L}, \mathcal{M}, \mathcal{R}, pc, r_1 = *(r_2) \to (\mathcal{I}, \mathcal{L}, \mathcal{M}, \mathcal{R}[r_1 = n_2], pc+1, i)} \text{ LOAD}$$

$$\frac{\mathcal{R}(r_1) \mapsto n_1 \quad \mathcal{R}(r_2) \mapsto n_2 \quad \mathcal{I}(pc+1) \mapsto i}{(\mathcal{I}, \mathcal{L}, \mathcal{M}, \mathcal{R}, pc, *(r_1) = r_2) \to (\mathcal{I}, \mathcal{L}, \mathcal{M}[n_1 = n_2], \mathcal{R}, pc+1, i)} \text{ STORE}$$

Table 2: Operational Semantics. $\hat{L}$ lifts a value to a label.

stores the value in register r_2 into the memory location addressed by r_1. We have two different move statements, one for register to register moves ($r = r_1$), and one to initialize a register with a constant ($r = n$). Control flow statements are either unconditional jumps (jmp/ijmp) or conditional jumps of the form `if` r `then jmp` ℓ_1 `else` ℓ_2 where if $R[r] = 0$ we jump to label ℓ_1 else ℓ_2. Note an indirect jump `ijmp` r uses a function $\hat{L}$ which converts the numeric value in register r to a label.

The machine is initialized with:

$$(\mathcal{I}, \mathcal{L}, \mathcal{M}, \mathcal{R}, 0, \text{jmp start})$$

We assume the program is well-formed. In our scenario, the program is automatically derived from valid x86 assembly. In Appendix B, we give an example program and its execution trace.

2.2 Assembly Aliases: Our High Level Approach

If $r_1 \equiv r_2 (\text{mod} 2^{32})$, then the expressions $*r_1$ and $*r_2$ refer to the memory cell, and r_1 and r_2 are called *aliases*. Our goal is to statically find all alias relationships. However, since this type of static analysis is well-known to be undecidable, we can only produce approximate results. A *may-alias* algorithm computes all pairs r_1 and r_2 which may be aliases. A *must-alias* computes all pairs r_1 and r_2 which must be aliases.

We focus on performing a may-alias analysis. One goal for our analysis to be conservative; that is, if an alias relationship is possible, our analysis includes it. As mentioned in the introduction, one key problem in computing may-alias relationships is reasoning about the values registers may hold.

Our high level idea is to perform abstract interpretation on the program to determine the possible values each register may hold at each program point. If we can determine a register r can hold values v_1, v_2, v_3, we know any memory operation $*r$ will reference the address v_1 or v_2 or v_3.

2.3 Assumptions

There are several possible assumptions that can affect the efficiency and accuracy of our alias analysis. Throughout the rest of the paper, we make the following assumptions:

1. All memory and registers are initialized prior to any reads. This requirement is normally satisfied since operating systems typically zero

all memory and registers before beginning program execution. Alternatively, we could defined a special syntactic category for uninitialized values.

2. All instructions which may be executed are given in $\mathcal{I}$, and all jumps are to labels in $\mathcal{L}$. This assumption is needed for the abstract machine to be well defined, e.g., that we know the total set of instructions which may execute.

3. All memory cells and register locations are of a single fixed width, and reads and writes do not overlap. This assumption simplifies the correctness proof: allowing several different widths adds a few more otherwise uninteresting cases to consider at each step in the induction.

In our implementation, we meet our assumptions by pre-processing the binary and assembly to not include floating point operations and reducing all integer operations to byte-level operations. In the rest of this subsection, we consider these assumptions and possible alternatives. We came across the need for many of these assumptions while trying to prove correctness of our alias analysis. Thus, correctness proofs are useful not just for showing correctness, but also understanding the space of possible assumptions an analysis can make.

Memory and Register Initialization One important issue we must address is the semantics of reading an uninitialized value, either from memory or from a register. An *uninitialized* value is a value (e.g., a memory cell) that has not been previously written to by the program. We consider several possibilities.

Assumption 1:*All value locations such as registers and memory locations are initialized to χ.*

With this assumption, a load from an address not previously stored to always returns the same value χ. This assumption mimics the fact that many OS's zero-fill memory and registers, i.e., $\chi = 0$. OS's zero-fill memory to prevent information leakage; if they did not initialize memory somehow, then the values from previous allocations (perhaps in another process) may be visible, e.g., process A could read process B's freed password memory.

For example, suppose at some point you have a store instruction $*r = v$ and we know $r = 0, 1$, and $m[0]$ and $m[1]$ have never been written to. With this assumption, the state after the store is $m[0] = \{0, v\}$ and $m[1] = \{0, v\}$. Note we can refine assumption 1 to include the case that different locations are initialized differently. For example, the instruction pointer is set by the operating system, the data memory segment is initialized with the data segment of the executable, etc.

A different assumption we could make is:

Assumption 1′ (alternate): *Reading an uninitialized value returns $\perp$.*

The alternate assumption 1′ is more conservative than assumption 1, and is intended to mimic the operational behavior of the raw x86 platform. We introduce a new value $\perp$ (bottom) which represents an unknown integer. We augment the semantics of integer binary and unary operations to include $\perp$ by assuming any operation with $\perp$ as an operand returns $\perp$, i.e., $\perp = \Box_u \perp$, $\perp = \perp \Box_b v$, $\perp = r \Box_b \perp$, and $\perp = \perp \Box_b \perp$.

If we adopt assumption 1′, our analysis will be very imprecise when a store can be to multiple addresses. For example, suppose at some point you have a store instruction $*r = v$ and we know $r = 0, 1$, and $m[0] = \perp, m[1] = \perp$. With assumption 1′, this store instruction gives us no useful information. Since we cannot say for certain which memory address is written to, after this instruction we must say both memory cells can still have value called $\perp$. Note that $\perp$ is not a single integer, but any possible integer in the domain, so the values at $m[0]$ are $m[1]$ possibility distinct integers.

In general, it seems difficult to tell exactly when a memory cell has been initialized to a computed value. This effect will propagate through the program, e.g., a subsequent read of $m[0]$ or $m[1]$ will result in $\perp$, likely leading to another store at location $\perp$.

Control-Flow Assumptions We already mentioned that we assume code is not created on the fly. More specifically:

Assumption 2: *All instructions which may be executed are given in $\mathcal{I}$, and all jumps are to labels in $\mathcal{L}$.*

In order to understand the significance of this assumption, consider how a program is interpreted by the CPU. The CPU begins reading in a sequence of bytes pointed to by the instruction pointer. The CPU continues to read bytes until either a complete instruction including opcode and operands is decoded, or the CPU can determine the bytes do not represent any instruction. After decoding, the CPU executes the instruction.

Now suppose we decode a move instruction from bytes 0-4. Later on, an indirect jump can jump to address 1 — the middle of a previously decoded instruction! Now the CPU decodes starting at byte 1, and will execute a completely different instruction. We call this instruction packing, since a sequence of bytes can represent many different instruction sequences.

Our assumption means that not only do we assume code isn't created, modified, or deleted on the fly, but also that there is no instruction packing. In practice, if such instructions were encountered we would stop the alias analysis, decode and add the new instructions to our program, and restart the analysis from the beginning.

Integers are a fixed width. Modern architectures, and in particular the x86 platform we focus on, supports 8, 16, and 32-bit integers, as well as a number of floating point sizes and representations. To simplify matters, in our analysis we assume:

Assumption 3: *All memory cells and register locations are of a single fixed width, and reads and writes do not overlap.*

In real assembly memory pointers are 32-bits for IA32, but registers can be variable width. Memory itself is addressable at the byte addressable, i.e., 2 32-bit integers can be written to the stack at bytes $[m_0, m_7]$, and a single 32-bit integer can be read from bytes $[m_2, m_5]$. In addition, registers may have different sizes and overlap: `%eax` and `%al` overlap on the low 8 bits.

3 Alias Analysis in Datalog

We perform our alias analysis in Datalog. At a high level, the Datalog EDB predicates are derived from program statements. The IDB predicates compute the possible values each register may hold via symbolic execution of the EDB predicates. After saturation, a user can query the database to discover alias relationships. Two register variables r_1 and r_2 are aliases at statement pc if:

$$\text{points\_to}(pc, r_1, \_) \bigcap \text{points\_to}(pc, r_2, \_) \neq \emptyset$$

3.1 EDB Rules

Our EDB predicates are shown in Table 3. They encode the language from Table 1 into an 6-positioned tuple. The first position is the statement number. The second position is the type of operation, and the remaining 4 positions interpretation is determined by the operation type.

If we adopt Assumption 1 (Section 2.3), we can initialize registers and memory locations by prepending a const predicate for each register used,

Predicate	**Statement**
inst(pc, const, r, n, none, none).	pc: r = n
inst(pc, binop,r, bop, r1, r2).	pc: r = r1 bop r2
inst(pc, unop, r, uop, r1, none).	pc: r = uop r1
inst(pc, move, r, r1, none, none).	pc: r = r1
inst(pc, load, r, r1, none, none).	pc: r = *r1
inst(pc, store, r, r1, none,none).	pc: *(r) = r1
inst(pc, jmp, name, l , none, none).	pc: jmp l
inst(pc, ijmp, lval, r , none, none).	pc: ijmp r
inst(pc, cjmp, r, l1, l2,none).	pc: if r then jmp l1 else jmp l2
inst(pc,nop,none,none,none,none).	pc: nop

Table 3: Our initial EDB predicates. The predicate given on the left is generated for statements of the form on the right. pc is always the statement number in the program.

e.g., inst(0, const, r, χ, none, none) initializes r to χ. Memory can be initialized by adding a similar rule for each memory address calculate.

3.2 IDB Rules

Our IDB predicates are given in Table 4. We adopt the more succient notation pc: i for instruction `i` on line `pc` instead of the more cumbersome 6-tuple of the corresponding EDB predicate. The succ(A,B) predicate is true if statement B is a successor of A. We use an auxiliary predicate defined(pc, r) which is not shown, but simply returns true if register r is assigned to by statement pc.

There are two types of predicates: predicate names prefixed with V calculate the value(pc,r,v) predicate is true if at statement pc register r may hold value v. Predicate names prefixed with P calculate the points_to(pc,i,v) which is true if at statement pc the memory at index i may hold value v.

V-CONST initializes a register to a constant. V-MOVE and P-PROP are transitive relationships, propagating values if the register is not redefined or the instruction is not a memory operation. V-BINOP and V-UNOP calculate the values a register may hold. The calculation is done via an external oracle, prefixed with a hash (#). The #binop oracle calculates $V = X \Box_b Y$. Since we assume all registers are initialized before being read, we can always calculate #binop (and similarly #unop).

The P-STORE instruction adds a new points-to relationship. Note we do not model destructive updates of memory: if two statements write to the

$$\frac{\text{succ(P,PC)} \quad \text{P: } R := N}{\text{value(PC,R,N)}}\ \textsc{v-const} \qquad \frac{\text{succ(P,PC)} \quad \text{P: R = R1} \quad \text{value(P,R1,V)}}{\text{value(PC,R,V)}}\ \textsc{v-move}$$

$$\frac{\text{succ(P,PC)} \quad \text{value(P,R,V)} \quad \text{not defined(PC,R)}}{\text{value(PC,R,V)}}\ \textsc{v-prop}$$

$$\frac{\text{succ(P,PC)} \quad \text{P: R = R1 Op R2} \quad \text{value(P,R1,X)} \quad \text{value(P,R2,Y)} \quad \text{\#binop(Op,X,Y,V)}}{\text{value(PC,R,V)}}\ \textsc{v-binop}$$

$$\frac{\text{succ(P,PC)} \quad \text{P: R = Op R1} \quad \text{value(P,R1,X)} \quad \text{\#unop(Op,X,V)}}{\text{value(PC,R,V)}}\ \textsc{v-unop}$$

$$\frac{\text{succ(P, PC)} \quad \text{P: R = *R1} \quad \text{value(P, R1, IV)} \quad \text{points\_to(P,IV, V)}}{\text{value(PC,R,V)}}\ \textsc{v-load}$$

$$\frac{\text{succ(P, PC)} \quad \text{points\_to(P,I,V)}}{\text{points\_to(PC, I, V)}}\ \textsc{p-prop}$$

$$\frac{\text{succ(P,PC)} \quad \text{P: *R = R1} \quad \text{value(P, R, I)} \quad \text{value(P,R1,V)}}{\text{points\_to(PC,I,V)}}\ \textsc{p-store}$$

$$\frac{\text{A: jmp L} \quad \text{B: label L}}{\text{succ(A,B)}}\ \textsc{succ-jmp}$$

$$\frac{\text{A: ijmp R} \quad \text{value(A,R,V)} \quad \text{B: label L} \quad \text{\#lhat(V, L)}}{\text{succ(A,B)}}\ \textsc{succ-ijmp}$$

$$\frac{\text{A: if R jmp L1 else jmp L2} \quad \text{B: label L1}}{\text{succ(A,B)}}\ \textsc{succ-cjmp-t}$$

$$\frac{\text{A: if R jmp L1 else jmp L2} \quad \text{B: label L2}}{\text{succ(A,B)}}\ \textsc{succ-cjmp-f}$$

$$\frac{\text{A: OP} \quad \text{OP} \neq \text{jmp} \quad \text{OP} \neq \text{ijmp} \quad \text{OP} \neq \text{cjmp} \quad \text{B=A+1}}{\text{succ(A,B)}}\ \textsc{succ-inc}$$

Table 4: IDB inference rules.

same cell, then our semantics say the memory cell could have either value after the second write. If we wanted to model destructive updates, then we would have to model when a cell was definitely overwritten vs. possibly overwritten. We leave this extension as future work.

Our rules take care of resolving indirect jumps via the succ-ijmp predicate. We can infer the destination of an indirect jump is label ℓ if the target registers value corresponds to the pc for ℓ. The #lhat predicate takes care of converting the register value to a label. This rule highlights one of the potential applications from Section 1: alias analysis can be used to aid disassembly by providing information about indirect jump targets.

4 Correctness Proof

In this section we prove that for every possible value that a variable or memory location can take on at a particular point in the program, we will have a corresponding fact in our database. This is trivially true before the machine starts executing, because there are not yet any assignments to memory or registers.

What we prove here is that this statement holds for every possible *transition* of the machine state.

Theorem 4.1 *Given a machine state transition:*

$$(\mathcal{I}, \mathcal{L}, \mathcal{M}, \mathcal{R}, pc, i) \rightarrow (\mathcal{I}, \mathcal{L}, \mathcal{M}', \mathcal{R}', pc', i')$$

If we have the facts corresponding to the register and memory assignments of the starting state:

$$\mathbf{A_1} \;\; \forall r \in dom(\mathcal{R}) s.t. \mathcal{R}(r) \mapsto v_2 : value(pc, r, v_2)$$
$$\mathbf{A_2} \;\; \forall n \in dom(\mathcal{M}) s.t. \mathcal{M}(n) \mapsto v_1 : points\_to(pc, n, v_1)$$

and the facts corresponding to the beginning and ending instruction:

$$\mathbf{A_3} \;\; pc : i$$
$$\mathbf{A_4} \;\; pc' : i'$$

Then we can derive the facts corresponding to the register and memory assignments of the next state:

$$\forall r \in dom(\mathcal{R}') s.t. \mathcal{R}'(r) \mapsto v_2 : value(pc', r, v_2)$$
$$\forall n \in dom(\mathcal{M}') s.t. \mathcal{M}'(n) \mapsto v_1 : points\_to(pc', n, v_1)$$

Proof: We perform inversion over the transition. Hence, we must consider each possible transition rule from Table 2.

Case: $$\frac{\mathcal{I}(pc+1) = i}{(\mathcal{I}, \mathcal{L}, \mathcal{M}, \mathcal{R}, pc, \text{nop}) \to (\mathcal{I}, \mathcal{L}, \mathcal{M}, \mathcal{R}, pc+1, i)} \textsc{Nop}$$
with $\mathcal{M}' = \mathcal{M}$, $\mathcal{R}' = \mathcal{R}$, pc' = $pc + 1$

pc: nop	assumption $\mathbf{A_3}$
A:OP	with A=pc and OP=nop
OP $\neq$ jmp	OP=nop
OP $\neq$ ijmp	OP=nop
OP $\neq$ cjmp	OP=nop
B=A+1	with A=pc, B=pc'
succ(pc,pc')	SUCC-INC
value(pc,R,V) $\forall$R $\in$ dom($\mathcal{R}$)s.t.$\mathcal{R}$(R) $\mapsto V$	assumption $\mathbf{A_1}$
$\forall$R $\in$ dom($\mathcal{R}$) not defined(pc,R)	construction of edb
value(pc',R,V) $\forall$R $\in$ dom($\mathcal{R}'$)s.t.$\mathcal{R}'$(R) $\mapsto V$	V-PROP
points_to(pc,I,V) $\forall$I $\in$ dom($\mathcal{M}$)s.t.$\mathcal{M}$(I) $\mapsto V$	assumption $\mathbf{A_2}$
points_to(pc',I,V) $\forall$I $\in$ dom($\mathcal{M}'$)s.t.$\mathcal{M}'$(I) $\mapsto V$	P-PROP

Case: $$\frac{\mathcal{I}(pc+1) = i}{(\mathcal{I}, \mathcal{L}, \mathcal{M}, \mathcal{R}, pc, \text{label } \ell) \to (\mathcal{I}, \mathcal{L}, \mathcal{M}, \mathcal{R}, pc+1, i)} \textsc{Label}$$
with $\mathcal{M}' = \mathcal{M}$, $\mathcal{R}' = \mathcal{R}$, pc' = $pc + 1$

value(pc',R,V) $\forall$R $\in$ dom($\mathcal{R}'$)s.t.$\mathcal{R}'$(R) $\mapsto V$	symmetric to NOP
points_to(pc',I,V) $\forall$I $\in$ dom($\mathcal{M}'$)s.t.$\mathcal{M}'$(I) $\mapsto V$	symmetric to NOP

Case: $$\frac{\mathcal{R}(r_1) \mapsto n_1 \quad \mathcal{R}(r_2) \mapsto n_2 \quad n_3 = n_1 \Box_b n_2 \quad \mathcal{I}(pc+1) = i}{(\mathcal{I}, \mathcal{L}, \mathcal{M}, \mathcal{R}, pc, r_3 := r_1 \Box_b r_2) \to (\mathcal{I}, \mathcal{L}, \mathcal{M}, \mathcal{R}[r_3 = n_3], pc+1, i)} \textsc{BinOp}$$
with $\mathcal{M}' = \mathcal{M}$, $\mathcal{R}' = \mathcal{R}[r_3 = n_3]$, pc' = $pc + 1$

$pc : r_3 = r_1 \Box_b r_2$	assumption $\mathbf{A_3}$
$\mathcal{R}(r_1) \mapsto n_1$	BINOP inversion
value(pc,r_1, n_1)	assumption $\mathbf{A_1}$
$\mathcal{R}(r_2) \mapsto n_2$	BINOP inversion
value(pc,r_2, n_2)	assumption $\mathbf{A_1}$
$n_3 = n_1 \Box_b n_2$	BINOP inversion

#vbinop($\square_b$, n_1, n_2, n_3) by def.
succ(pc,pc') SUCC-INC (symmetric to NOP)
value(pc', r_3, n_3) V-BINOP
value(pc,R,V) $\forall$R $\in$ dom($\mathcal{R}$)s.t.$\mathcal{R}$(R) $\mapsto V$ assumption $\mathbf{A_1}$
$\forall$R $\in$ dom($\mathcal{R}$), s.t.R $\neq r_3$ not defined(pc,R) construction of edb
value(pc',R,V) $\forall$R $\in$ dom($\mathcal{R}'$), s.t.R $\neq r_3$and$\mathcal{R}'$(R) $\mapsto V$ V-PROP
points_to(pc,I,V) $\forall$I $\in$ dom($\mathcal{M}$)s.t.$\mathcal{M}$(I) $\mapsto V$ assumption $\mathbf{A_2}$
points_to(pc',I,V) $\forall$I $\in$ dom($\mathcal{M}'$)s.t.$\mathcal{M}'$(I) $\mapsto V$ P-PROP

Case: $$\frac{\mathcal{R}(r_1) \mapsto n_1 \quad n_2 = \square_u n_1 \quad \mathcal{I}(pc+1) \mapsto i}{(\mathcal{I}, \mathcal{L}, \mathcal{M}, \mathcal{R}, pc, r_2 := \square_u r_1) \rightarrow (\mathcal{I}, \mathcal{L}, \mathcal{M}, \mathcal{R}[r_2 = n_2], pc+1, i)} \text{ UNOP}$$
with $\mathcal{M}' = \mathcal{M}$, $\mathcal{R}' = \mathcal{R}[r_2 = n_2]$, pc' = $pc + 1$

value(pc', r_2, n_2) symmetric to BINOP
value(pc',R,V) $\forall$R $\in$ dom($\mathcal{R}'$), s.t.R $\neq r_2$and$\mathcal{R}'$(R) $\mapsto V$ Symmetric to BINOP
points_to(pc',I,V) $\forall$I $\in$ dom($\mathcal{M}'$)s.t.$\mathcal{M}'$(I) $\mapsto V$ symmetric to BINOP

Case: $$\frac{\mathcal{R}(r_1) \mapsto n \quad \mathcal{I}(pc+1) \mapsto i}{(\mathcal{I}, \mathcal{L}, \mathcal{M}, \mathcal{R}, pc, r_2 := r_1) \rightarrow (\mathcal{I}, \mathcal{L}, \mathcal{M}, \mathcal{R}[r_2 = n], pc+1, i)} \text{ MOVE}$$
with $\mathcal{M}' = \mathcal{M}$, $\mathcal{R}' = \mathcal{R}[r_2 = n]$, pc' = $pc + 1$

value(pc', r_2, n) symmetric to BINOP
value(pc',R,V) $\forall$R $\in$ dom($\mathcal{R}'$), s.t.R $\neq r_2$and$\mathcal{R}'$(R) $\mapsto V$ Symmetric to BINOP
points_to(pc',I,V) $\forall$I $\in$ dom($\mathcal{M}'$)s.t.$\mathcal{M}'$(I) $\mapsto V$ symmetric to BINOP

Case: $$\frac{\mathcal{I}(pc+1) \mapsto i}{(\mathcal{I}, \mathcal{L}, \mathcal{M}, \mathcal{R}, pc, r := n) \rightarrow (\mathcal{I}, \mathcal{L}, \mathcal{M}, \mathcal{R}[r = n], pc+1, i)} \text{ MOVEC}$$
with $\mathcal{M}' = \mathcal{M}$, $\mathcal{R}' = \mathcal{R}[r = n]$, pc' = $pc + 1$

value(pc', r, n) symmetric to BINOP
value(pc',R,V) $\forall$R $\in$ dom($\mathcal{R}'$), s.t.R $\neq r$and$\mathcal{R}'$(R) $\mapsto V$ Symmetric to BINOP
points_to(pc',I,V) $\forall$I $\in$ dom($\mathcal{M}'$)s.t.$\mathcal{M}'$(I) $\mapsto V$ symmetric to BINOP

Case: $$\frac{\mathcal{R}(r_2) \mapsto n_1 \quad \mathcal{M}(n_1) \mapsto n_2 \quad \mathcal{I}(pc+1) \mapsto i}{(\mathcal{I}, \mathcal{L}, \mathcal{M}, \mathcal{R}, pc, r_1 = *(r_2) \to (\mathcal{I}, \mathcal{L}, \mathcal{M}, \mathcal{R}[r_1 = n_2], pc+1, i)} \text{ LOAD}$$

with $\mathcal{M}' = \mathcal{M}$, $\mathcal{R}' = \mathcal{R}[r_1 = n_2]$, pc' = $pc + 1$

$pc : r_1 = *r_2$	assumption $\mathbf{A_3}$
$\mathcal{R}(r_2) \mapsto n_1$	LOAD inversion
value(pc, r_2, n_1)	assumption $\mathbf{A_1}$
$\mathcal{M}(n_1) \mapsto n_2$	LOAD inversion
points_to(pc, n_1, n_2)	assumption $\mathbf{A_2}$
succ(pc, pc')	symmetric to NOP
value(pc', r_1, n_2)	V-LOAD
value(pc',R,V) $\forall$R $\in$ dom($\mathcal{R}'$), s.t.R $\neq r_1$and$\mathcal{R}'$(R) $\mapsto V$	Symmetric to BINOP
points_to(pc',I,V) $\forall$I $\in$ dom($\mathcal{M}'$)s.t.$\mathcal{M}'$(I) $\mapsto V$	symmetric to BINOP

Case: $$\frac{\mathcal{R}(r_1) \mapsto n_1 \quad \mathcal{R}(r_2) \mapsto n_2 \quad \mathcal{I}(pc+1) \mapsto i}{(\mathcal{I}, \mathcal{L}, \mathcal{M}, \mathcal{R}, pc, *(r_1) = r_2) \to (\mathcal{I}, \mathcal{L}, \mathcal{M}[n_1 = n_2], \mathcal{R}, pc+1, i)} \text{ STORE}$$

with $\mathcal{M}' = \mathcal{M}[n_1 = n_2]$, $\mathcal{R}' = \mathcal{R}$, pc' = $pc + 1$

$pc : *r_1 = r_2$	assumption $\mathbf{A_3}$
$\mathcal{R}(r_1) \mapsto n_1$	STORE inversion
value(pc, r_1, n_1)	assumption $\mathbf{A_1}$
$\mathcal{R}(r_2) \mapsto n_2$	STORE inversion
value(pc, r_2, n_2)	assumption $\mathbf{A_1}$
succ(pc, pc')	symmetric to NOP
points_to(pc', n_1, n_2)	V-STORE
value(pc',R,V) $\forall$R $\in$ dom($\mathcal{R}'$), s.t.$\mathcal{R}'$(R) $\mapsto V$	symmetric to NOP
points_to(pc',I,V) $\forall$I $\in$ dom($\mathcal{M}'$)s.t.$\mathcal{M}'$(I) $\mapsto V$	symmetric to NOP

Case: $$\frac{\mathcal{L}(\ell) = pc' \quad \mathcal{I}(pc') = i}{(\mathcal{I}, \mathcal{L}, \mathcal{M}, \mathcal{R}, pc, \text{jmp } \ell) \to (\mathcal{I}, \mathcal{L}, \mathcal{M}, \mathcal{R}, pc', i)} \text{ JMP}$$

with $\mathcal{M}' = \mathcal{M}$, $\mathcal{R}' = \mathcal{R}$

pc: jmp ℓ	assumption $\mathbf{A_3}$
pc': label ℓ	assumption $\mathbf{A_4}$
succ(pc, pc')	SUCC-JMP
value(pc',R,V) $\forall$R $\in$ dom($\mathcal{R}'$), s.t.$\mathcal{R}'$(R) $\mapsto V$	symmetric to NOP
points_to(pc',I,V) $\forall$I $\in$ dom($\mathcal{M}'$)s.t.$\mathcal{M}'$(I) $\mapsto V$	symmetric to NOP

Case: $$\frac{\mathcal{R}(r) = n \quad \hat{L}(n) \mapsto \ell \quad \mathcal{L}(\ell) \mapsto pc' \quad \mathcal{I}(pc') = i}{(\mathcal{I}, \mathcal{L}, \mathcal{M}, \mathcal{R}, pc, \text{ijmp} r) \rightarrow (\mathcal{I}, \mathcal{L}, \mathcal{M}, \mathcal{R}, pc', i)} \text{ IJMP}$$
with $\mathcal{M}' = \mathcal{M}$, $\mathcal{R}' = \mathcal{R}$

pc: ijmp r	assumption $\mathbf{A_3}$
$\mathcal{R}(r) \mapsto n$	IJMP inversion
value(pc,r,n)	assumption $\mathbf{A_1}$
$\hat{L}(n) \mapsto \ell$	IJMP inversion
#lhat(n, ℓ)	definition of #lhat
$\mathcal{L}(\ell) \mapsto pc'$	IJMP inversion
pc': label ℓ	assumption $\mathbf{A_4}$
succ(pc, pc')	SUCC-IJMP
value(pc',R,V) $\forall$R $\in$ dom($\mathcal{R}'$), s.t.$\mathcal{R}'$(R) $\mapsto V$	symmetric to NOP
points_to(pc',I,V) $\forall$I $\in$ dom($\mathcal{M}'$)s.t.$\mathcal{M}'$(I) $\mapsto V$	symmetric to NOP

Case: $$\frac{\mathcal{R}(r) \mapsto n \quad n = 0 \quad \mathcal{L}(\ell_1) \mapsto pc' \quad \mathcal{I}(pc') \mapsto i}{(\mathcal{I}, \mathcal{L}, \mathcal{M}, \mathcal{R}, pc, \text{if } r \text{ jmp } \ell_1 \text{ else jmp } \ell_2) \rightarrow (\mathcal{I}, \mathcal{L}, \mathcal{M}, \mathcal{R}, pc', i)} \text{ CJMP-T}$$
with $\mathcal{M}' = \mathcal{M}$, $\mathcal{R}' = \mathcal{R}$

value(pc',R,V) $\forall$R $\in$ dom($\mathcal{R}'$), s.t.$\mathcal{R}'$(R) $\mapsto V$	symmetric to JMP
points_to(pc',I,V) $\forall$I $\in$ dom($\mathcal{M}'$)s.t.$\mathcal{M}'$(I) $\mapsto V$	symmetric to JMP

Case: $$\frac{\mathcal{R}(r) \mapsto n \quad n \neq 0 \quad \mathcal{L}(\ell_2) \mapsto pc' \quad \mathcal{I}(pc') \mapsto i}{(\mathcal{I}, \mathcal{L}, \mathcal{M}, \mathcal{R}, pc, \text{if } r \text{ jmp } \ell_1 \text{ else jmp } \ell_2) \rightarrow (\mathcal{I}, \mathcal{L}, \mathcal{M}, \mathcal{R}, pc', i)} \text{ CJMP-F}$$
with $\mathcal{M}' = \mathcal{M}$, $\mathcal{R}' = \mathcal{R}$

value(pc',R,V) $\forall$R $\in$ dom($\mathcal{R}'$), s.t.$\mathcal{R}'$(R) $\mapsto V$	symmetric to JMP
points_to(pc',I,V) $\forall$I $\in$ dom($\mathcal{M}'$)s.t.$\mathcal{M}'$(I) $\mapsto V$	symmetric to JMP

□

5 Implementation

We have implemented a tool which performs alias analysis on x86 executables. Our tool:

- Disassembles the executable to a sequence of assembly statements.
- Translates the assembly code into an intermediate representation (IR).
- Translates the IR into the EDB predicates in Section 3.1.
- Runs the Datalog implementation with the IDB predicates from Section 3.2.

For this project, we wrote about 1100 lines of OCaml and about 250 lines of C++. All in all, our tool consists of about 4500 lines of OCaml and 20000 lines of C++. There are 41 lines of Datalog.

VINE: Disassembly and Converting to the IR. We have developed an infrastructure, called VINE, which disassembles a program and converts each assembly statement to a series of IR instructions. Our IR syntax is given in Appendix A.

Translating an x86 instruction into a sequence of IR statements is actually quite tricky. Since x86 is CISC, a single instruction may perform a complex operation, e.g., the `rep` family of instructions are single-instruction loops which continue to execute until a stop condition is met. For example, `rep stosd addr` stores the word in the `%eax` register in `%ecx` words starting at address `addr`. Note that in this example, a) the single assembly instruction translates into a sequence of IR statements, b) the translated IR contain a loop, even though it is for a single instruction, and c) the assembly does not mention `%ecx` and `%eax` explicitly. Thus, the translation engine produces a series of IR statements $\vec{s} =< s_0, s_1, s_2, ..., s_n >$ for each assembly statement.

Translating the IR into EDB Predicates. Each IR statement is translated into an EDB predicate. The translation essentially simplifies expressions such that each statement conforms to the grammar in Table 1, then writes out the corresponding EDB predicate.

Analysis Implementation. A key problem that we needed to address in this project was which Datalog implementation to use to perform the analysis. Our main requirement was a Datalog implementation which supported binary operations over integers in $[0, 2^{32}]$, i.e., the size of IA32 registers. We surveyed several Datalog implementations, including:

- bddbddb [16, 17], a Datalog implementation which uses BDD's.

- DES [14], a Datalog implementation written in Prolog.

- DLV [10, 7] (Disjunctive Datalog), a research Datalog system for disjunctive Datalog with constraints and "true" negation.

Of these systems, only DLV supported native arithmetic. However, DLV arithmetic is constrained to the domain $[0, 999999999]$. DLV also only supports addition and multiplication, and not other operations such as division or xor.

After emailing the authors, we discovered a variant of DLV called DLV-Ex [5, 4]. DLV-Ex extends DLV with external predicates. The external predicates are implemented in C++. External predicates are referenced in rule bodies using a hash (#). For example, the rule:

```
sum(X,Y,Z) :- integer(X), integer(Y), integer(Z),
              #binop(plus, X,Y,Z)
```

uses an external predicate named `binop`. The `binop` predicate has several different implementations: one which returns true if `X + Y = Z` when all are ground, one which instantiates `Z` when `X` and `Y` are ground but `Z` is free, etc.

We implemented the binary and unary evaluation using the external predicate mechanism. The resulting queries are still safe since arithmetic is performed over a finite domain. One small issue is that DLV-Ex is also limited to integers less than 999999999. In order to get around this issue, we encode each integer as a string atom. Our external predicate converts the string to an integer, performs the requested operation, and converts the resulting integer back to a string atom.

6 Evaluation

We have begun evaluation of our prototype. We have created several small examples and performed points-to analysis on the result.

We have also begun testing our points-to analysis on real programs. We have tested our implementation on several variants of the following C test program:

```
int g(int x)
 return x;
void f(int x)
 g(x);
void main()
  f(0x2);
```

We compile the program using gcc, then use our tool to disassemble the source, convert to the VINE IR, convert the vine IR to EDB predicates, and saturate the Datalog database. For this program, the EDB contains 356 predicates. Saturation takes 1.8 seconds inside a VM on a Pentium 1.7 GhZ machine. Our example generates 2 indirect jumps: one for each return from the calling function. We resolve the 2 indirect jump returns correctly. There are 7 different points-to relationships from the example corresponding to placing return addresses on the stack and passing arguments.

7 Future Work and Conclusion

Our immediate future work is to re-implement our system using bddbddb. There are two reasons for doing this. First, we can more efficiently initialize all memory prior to analysis. Our current implementation assumes all memory is initialized by the program itself before any reads. However, a program may load a value it has never written. To accurately describe these semantics, we need to initialize all memory locations as part of the EDB. BDD's seem a good approach. Second, we may want to efficiently clone our database as allowed by BDD implementations. Whaley et al [18] has previously shown that cloning is important to scaling inter-procedural alias analysis. In our setting, we initially do not have procedures. However, our alias analysis allows us infer when a chunk of assembly will act like a procedure. Thus, we too may be able to do a type of inter-procedural (or inter-chunk) analysis where cloning would be useful.

We also need to perform more tests on real programs. We expect we will need more intelligent support for loops. Our current analysis will iterate over loops until the database is saturated, which may take a very long time given the domain for each input variable is $|2^{32}|$. We are currently investigating how best to introduce widening into our semantics.

Our primary goal in this project was to develop a firm foundation for research on assembly alias analysis. We have developed an initial alias analysis for assembly, and built a prototype system to test some of our ideas for alias analysis. We have also shown our approach produces a conservative approximation of possible alias relationships. We believe these are important first steps towards accurate and efficient alias analysis of assembly programs.

Acknowledgments

We would like to thank Frank Pfenning for his helpful comments and thoughts while performing this research, as well as Gogul Balakrishnan and Thomas Reps for their comments on a previous version of this paper.

8 References

[1] G Balakrishnan and T Reps. Analyzing memory accesses in x86 executables. In *Proceedings of the International Conference on Compiler Construction*, pages 5–23. Springer-Verilag, 2004.

[2] G Balakrishnan and T Reps. DIVINE: DIscovering Variables IN eXecutables. In *Proceedings for the Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, 2007.

[3] David Brumley, James Newsome, Dawn Song, Hao Wang, and Somesh Jha. Towards automatic generation of vulnerability-based signatures. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, pages 2–16, 2006.

[4] F. Calimeri and G. Ianni. External sources of computation for answer set solvers. In *Proceedings of the 8th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, pages 105–118, 2005.

[5] Francesco Calimeri, Giovambattista Ianni, and Susanna Cozza. Dlv-ex. Software available at: `http://www.mat.unical.it/ianni/wiki/dlvex`.

[6] C Cifuentes, D Simona, and A Fraboulet. Intraprocedural static slicing of binary executables. In *International Conference on Software Maintenence*, pages 188–195, 1997.

[7] Tina Dell'Armi, Wolfgang Faber, Giuseppe Ielpa, Nicola Leone, and Gerald Pfeifer. System description: DLV with aggregates. In Vladimir Lifschitz and Ilkka Niemela, editors, *Proceedings of the 7th International Conference on Logic Programming and Non-Monotonic Reasoning (LPNMR)*, volume 2923, pages 326–330.

[8] Andrew Edwards, Amitabh Srivastava, and Hoi Vo. Vulcan: Binary transformation in a distributed environment. Technical Report MSR-TR-2001-50, Microsoft Research, 2001.

[9] M. Van Emmerik and T Waddington. Using a decompiler for real-world source recovery. In *Proceedings of 11th Working Conference on Reverse Engineering*, pages 27–36, 2004.

[10] Nicola Leone, Gerald Pfeifer, and Wolfgang Faber. The dlv project. Software available at: `http://www.dbai.tuwien.ac.at/proj/dlv/`.

[11] Microsoft. Phoenix framework. `http://research.microsoft.com/phoenix/`.

[12] James Newsome, David Brumley, Jason Franklin, and Dawn Song. Replayer: Automatic protocol replay by binary analysis. In Rebecca Write, Sabrina De Capitani di Vimercati, and Vitaly Shmatikov, editors, *In the Proceedings of the* 13^{th} *ACM Conference on Computer and and Communications Security (CCS)*, pages 311–321, 2006.

[13] T Reps, G Balakrishnan, and J Lim. Intermediate-representation recovery from low-level code. In *Proceedings of the Workshop on Partial Evaluation and Program Manipulation (PEPM)*, 2006.

[14] Fernando Saenz. Datalog educational system (DES). Software available at: `http://www.fdi.ucm.es/professor/fernan/DES`.

[15] S.K.Debray, R Muth, and M Weippert. Alias analysis of executable code. In *Proceedings of the 1988 Principles of Programming Languages Conference (POPL)*, pages 12–24, 1988.

[16] John Whaley. bddbddb. Software available at: `bddbddb.sf.net`.

[17] John Whaley, Dzintars Avots, Michael Carbin, and Monica S. Lam. Using datalog with binary decision diagrams for program analysis. In Kwangkeun Yi, editor, *Proceedings of Programming Languages and Systems: Third Asian Symposium (APLAS)*, volume 3780, pages 97–118, nov 2005.

[18] John Whaley and Monica Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *In Proceedings of Program Language Design and Implementation (PLDI)*, pages 131–144, 2004.

<table>
<tr><td>stmt</td><td>::=</td><td>`Jmp` of exp | `CJmp` of exp * exp * exp
| `Move` of lvalue * exp | `Label` of label | `Stop`</td></tr>
<tr><td>exp</td><td>::=</td><td>`BinOp` of binop_type * exp * exp | `UnOp` of unop_type * exp
| `Name` of label |`Constant` of reg_t * value
| `Lval` of lvalue | `Let` of lvalue * exp * exp</td></tr>
<tr><td>lvalue</td><td>::=</td><td>`Temp` of var * typ | `Mem` of var * typ * exp</td></tr>
<tr><td>binop_type</td><td>::=</td><td>$+ \mid - \mid * \mid / \mid \% \mid \ll \mid \gg \mid \wedge \mid \vee \mid \oplus \mid == \mid <> \mid < \mid \leq$</td></tr>
<tr><td>unop_type</td><td>::=</td><td>$\neg \mid !$</td></tr>
<tr><td>typ</td><td>::=</td><td>reg_t | `Array` of typ * typ</td></tr>
<tr><td>reg_t</td><td>::=</td><td>REG_64 | REG_32 | REG_16 | REG_8 | REG_1</td></tr>
</table>

Table 5: Our internal representation (IR) for assembly language.

A The VINE IR

Table 5 shows the IR produced by the VINE infrastructure. Note that a since x86 instruction will likely correspond to a series of IR instructions.

B Example Program Execution in our Language.

An example program which computes the factorial of register `r0` is:

```
label start
 r1 := 0
 r2 := 1
 r3 := r0 - r1
 if r3 then jmp done else jmp fact
 label fact:
   r1 := r1 + 1
   r2 := r2 * r1
   r3 := r0 - r1
   if r3 then jmp done else jmp fact
 label done:
   halt
```

If we assume `r0 = 2`, then the execution is:

$(I, L, M, R[r0 = 2], 0, \text{jmp start}) \rightarrow$
$(I, L, M, R[r0 = 2], 1, \text{label start}) \rightarrow$
$(I, L, M, R[r0 = 2], 2, \text{r1 := 0}) \rightarrow$
$(I, L, M, R[r0 = 2, r1 = 0], 3, \text{r2 := 1}) \rightarrow$
$(I, L, M, R[r0 = 2, r1 = 0, r2 = 1], 4, \text{r3 := r0 - r1}) \rightarrow$
$(I, L, M, R[r0 = 2, r1 = 0, r2 = 1, r3 = 2], 5, \text{if r3 then jmp done else jmp fact}) \rightarrow$
$(I, L, M, R[r0 = 2, r1 = 0, r2 = 1, r3 = 2], 6, \text{label fact}) \rightarrow$
$(I, L, M, R[r0 = 2, r1 = 0, r2 = 1, r3 = 2], 7, \text{r1 := r1 + 1}) \rightarrow$
$(I, L, M, R[r0 = 2, r1 = 1, r2 = 1, r3 = 2], 8, \text{r2 := r2 * r1}) \rightarrow$
$(I, L, M, R[r0 = 2, r1 = 1, r2 = 1, r3 = 2], 9, \text{r3 := r0 - r1}) \rightarrow$
$(I, L, M, R[r0 = 2, r1 = 1, r2 = 1, r3 = 1], 10, \text{r3 := r0 - r1}) \rightarrow$
$(I, L, M, R[r0 = 2, r1 = 1, r2 = 1, r3 = 1], 11, \text{if r3 then jmp done else jmp fact}) \rightarrow$
$(I, L, M, R[r0 = 2, r1 = 1, r2 = 1, r3 = 1], 6, \text{label fact}) \rightarrow$
$(I, L, M, R[r0 = 2, r1 = 1, r2 = 1, r3 = 1], 7, \text{r1 := r1 + 1}) \rightarrow$
$(I, L, M, R[r0 = 2, r1 = 2, r2 = 1, r3 = 1], 8, \text{r2 := r2 * r1}) \rightarrow$
$(I, L, M, R[r0 = 2, r1 = 2, r2 = 2, r3 = 1], 9, \text{r3 := r0 - r1}) \rightarrow$
$(I, L, M, R[r0 = 2, r1 = 2, r2 = 2, r3 = 0], 10, \text{if r3 then jmp done else jmp fact}) \rightarrow$
$(I, L, M, R[r0 = 2, r1 = 2, r2 = 2, r3 = 0], 11, \text{label done})$
$(I, L, M, R[r0 = 2, r1 = 2, r2 = 2, r3 = 0], 12, \text{halt})$