

VinE Project Documentation

David Brumley

Warning: This document is intended to give an overview of the language. The exact semantics are subject to change.

1 Overview

The VinE project is geared towards analyzing security properties of executables. Traditional software security research has been predicated on the availability of source code. Previous approach has focused on ensuring software security via the software developer, e.g, their choice of safe/unsafe language, implementation methodology, compile-time checks, source-code analysis, etc.

Previous techniques implied that security is completely predicated on choices made by the developer. This is insufficient for many reasons, including:

- Most users of software do not have source code. However, different users may have different security needs. It is difficult for a developer to come up with a balance of security vs. other factors that is acceptable to all users.
- When time is of the essence, it may not be possible to involve a developer to address security measures. If a zero-day exploit is released, it may be impractical to involve the developer in security-critical revisions to the source code, compiler, etc.

Architecture Overview At a high level, the VinE architecture consists of:

- A front-end disassembler. The disassembler is responsible for translating a binary into assembly instructions.
- Intermediate Representation(s) (IR). An IR is an abstraction geared at providing a unambiguous representation of the binary. The IR is semantically equivalent to the program, meaning executing the IR on a properly defined machine should result in the same behavior as executing the binary on a real machine.
- Back-end analysis routines. Analysis is performed on the IR. Example analysis include dead-code elimination, weakest pre-condition computation, etc.
- Applications. Applications use the above components to perform specific (research-oriented) tasks.

Last update Fri Apr 27 11:52:19 2007 David Brumley

2 Administrative Details

There are many different projects using the VinE source code. This section describes how we organize our code, how we write code, and how we create projects around the code. There are a few fundamental rules we all follow:

Fundamental Rules:

- Make API's and do not break others API's.
- Document your source code using ocamldoc-style comments for ocaml, or doxygen style-comments for C/C++.
- Do not check into trunk unless make succeeds.
- Only check source code into trunk.
- Do not check into trunk unless the regression tests pass.
- Make a regression test for any API you care about not breaking. If you don't write a regression test, others may inadvertently break your code without knowing.

If you follow these basic rules, everyone should be happy.

2.1 Subversion

We use subversion to manage our code. Everyone should be familiar with the basics of subversion. A good introductory (and free) book is available at <http://svnbook.red-bean.com/>.

The main subversion repository for VinE is at <https://hayao.ece.cmu.edu/svn/vine>. The repository is currently organized as follows:

- trunk - the activate development source code branch. When you check into trunk, it is expected that the top-level make succeeds.
 - ocaml - the vine library. applications should not go in this directory.
 - libasmir - the asmir library, used for translating from assembly to our IR.
 - *<project>* - a project directory, e.g., an application.
- branches - source code branches. More on branches below.
- tags - tagged versions of the code. Tags correspond to releases or to major checkpoints. We generally create a new tag version after every paper.
- results - this is where you put results for your project, including test cases, source code, or whatever you think is relevant to your research project.
- regression - regression tests for vine.

2.1.1 Code Branches

Everyone should keep their active code development in subversion. This is for your own protection: our subversion machine is backed up regularly, while your local machine may not be.

When you are working on changing code in a way that is incompatible or requires major changes over multiple days to the main trunk, you should create a code branch. A branch is an independent line of development. Branches allow you to use subversion (thus not keeping all your work on your local machine), while still not breaking the fundamental rules.

The subversion book [3] has good documentation on how to make and merge branches appropriately.

2.2 Regression Tests

For each project, there should be a set of regression tests. In general, we have two types of regression types: micro-tests and macro-tests.

Micro-tests are tests in which you can specify the “right” answer. Micro-tests are typically written by hand in the VinE IR. Micro-tests are intended to exercise a specific component in as many ways as you think are appropriate. Micro-tests, since they are small, generally also allow us to more easily debug problems.

Macro-tests are generally real-life tests, e.g., test the scalability of your code. You should again generally know the “right” answer, but it may be harder to debug.

Each test should have a “test.sh” script which when executed returns 0 (zero) for success and something else for failure. We have a top-level test.sh, which is executed over all regression tests.

2.3 Writing Code

When you write code, please try and design a proper API and use others API's. If code doesn't do what you want it to do, ask the author, or design an API yourself. API's should be specified in “.mli” files for ocaml, and in “.h” files for C/C++. Code should be documented. For API's, you should include documentation for all parameters, and what the API returns.

More particularly:

- When writing C code, use GNU 'indent' to indent your code. <https://hayao.ece.cmu.edu/vine/wiki/wiki/indent> has one recommended indent package.
- When writing C/C++ code, use doxygen-style comments.
- When writing ocaml code, using ocaml-doc-style comments.
- When writing ocaml code, I recommend using emacs in tuareg-mode. In ubuntu, this can be installed as 'tuareg-mode'.
- Keep lines under 80 characters long.

2.4 The Wiki and Bugs

We use a wiki to keep track of projects and bugs. The wiki has random relevant information. We primarily use the wiki to keep track of information everyone should know about, and for its bug-tracking system.

You should create a bug report when something is broken and you don't fix it yourself. When you create a bug report, please be as specific as you can. A good bug report also includes a specific example, what is broken about the example, and what the correct output should be.

We also use the bug-reporting system to keep track of new features we want. For example, if during your research project you find that some additional functionality would be great, but don't have time to implement it yourself, add a ticket as a "feature request".

As a general rule, bugs should be actionable items — someone should end up fixing it. Feature requests need not be actionable; they are there as a reminder.

2.5 Creating a New Project

When you want to create a new project, you should:

- Create a directory in the trunk directory to hold your source code.
- Create a directory in the results directory to hold your examples.
- Create a set of regression tests, if appropriate.
- Create a sub-page off the wiki describing your project.

3 x86 Overview

3.1 Calling Conventions

- Chapter 6 of Volume 1 of the IA-32 Architecture Software Developer's manual is devoted to calling and returning [1].

4 Disassembly

Disassembly consists of two steps:

1. Parsing the file format and locating code segments.
2. Disassembling each code segment into a sequence of x86 instructions.

The relevant files for this step are contained in `libasmir/src/disasm`. In particular, `asm_program.cpp` contains most of the high-level interfaces we use, while other files contain lower-level interfaces. Also, note some of the code is taken from `objdump.c` in `binutils`, which is a good starting point for those interested at more depth.

4.1 Locating Code Segments

We use the GNU BFD library [5] for reading in an executable. BFD supports object files, though our code currently does not. I first give a high level description of ELF, then describe BFD sections. Although Windows uses a different file format (PE — Portable Executable), at a high level the same discussion is relevant.

ELF Every ELF begins with an ELF header. The header contains information such as the architecture, whether it is an executable or object file, the programs start address, a *program header structure* and a *section header table*, etc. In addition, it contains the sections that make up the program. Almost everything interesting is within the sections.

The section header table focuses on identifying the various parts of the program are within the ELF file. The program header describes where and how these parts are loaded into memory. The section header table is for use by the compiler and linker, while the program header table is for use by the program loader. The program header is optional for object files. The section header table is optional for an executable [7].

Each section is an `Elf32_Shdr` structure. Important sections include:

- `.bss` Holds uninitialized data of the program. Initialized to all zero's when the process starts up.
- `.data` & `.data1` Hold initialized data.
- `.dynamic` Holds dynamic linking information.
- `.hash` Holds the symbol hash table.
- `.rodata` & `.rodata2` Contain read-only data. GCC puts strings constants and constant floating points here.
- `.symtab` Holds the symbol table, if present.
- `.text` The executable instructions of the program.
- `.init` Initialization functions. executed upon load.
- `.fini` Like `.init`.

BFD BFD provides an abstract interface to executables. BFD is also used by the linux programs `readelf` and `objdump`. These programs are very useful when trying to untangle an executable, though often assume an executable was produced by a compiler. An executable consists of header data, optional symbol table, and zero or more sections.

Each section is marked with a type, and an optional name. The type may be used by the OS when loading the file, e.g., a `SEC_DATA` (data segment) may be marked read-only. Example section types include:

- `SEC_CODE` The code segment, i.e., a segment marked as containing executable code. The `.text` section is an example of a `SEC_CODE`.
- `SEC_RELOC` Relocable code. We do not process relocable code.
- `SEC_DATA` A section containing read-only data.
- `SEC_NOFLAGS` No information is available.

4.2 Disassembling Each Code Segment

Our disassembler is based upon Kruegel *et. al.*'s disassembler [6] operating in linear sweep mode. The linear sweep algorithm is given in the following psuedo-code:

```
offset = 0;
while(offset < section_length)
    inst = blob + offset;
    offset = inst->length;
```

This loop is intended to simulate the instruction decode-execute loop that the processor uses when executing the program (see wikipedia's entry on "Von Neumann Architecture" for more information on this sort of loop). However, we are performing this statically, while the processor does it dynamically and thus has more information. For most executables, this loop works fine, though it is worth noting other algorithms such as the recursive traversal algorithm can produce more accurate results [6].

At a high level, the input to disassembly is a section with symbol table, and the output is a `asm_program_t`, which consists of an `asm_function_t` for each function disassembled. The instruction, along with its disassembly, can be found in `asm_function_t`.

Symbol Tables Although you could disassemble arbitrary byte sequences, we expect to only be disassembling functions. We currently identify functions via the symbol table. The symbol table, as produced by a compiler, contains information about the executable. For example, the start address of functions (though not the end address) is given in the symbol table.

Each symbol is marked with a type by BFD, hinting at what the symbol is used for, such as:

- `BSF_FUNCTION` indicates a function entry point. `BSF_FUNCTION` is used by ELF, and maybe others.
- `BSF_GLOBAL` a global symbol.
- `BSF_LOCAL` a local symbol such as static in C.

We only disassemble `BSF_FUNCTION`'s. Sometimes functions are not marked with `BSF_FUNCTION`, e.g., hand-generated assembly often omits this information. If you wish to create hand-generated assembly, you must mark each function with a ".type" declaration. For example:

```
> cat hello.c
int main()
{
    return 42;
}
> gcc -S hello.c /* The -S flag generates assembly and stop */
> cat hello.s
...
.globl main
        .type    main, @function
main:
...
```

4.3 Disassembly to IR

The output of the disassembly phase is an `asm_program_t`, which consists primarily of a list of `asm_function_t`'s for each function in the executable. `asm_function_t`'s are sequentially processed and converted into the IR, as discussed in the next section.

4.4 Future Directions

Our current disassembler has been well-tested for code generated by gcc. Our future direction is to allow the user to plug in other disassemblers. The idea is:

- A disassembler disassembles the file and produces a set of instruction addresses and function entry/exit points.
- Optionally, self-decrypting binaries would require passing in the actual byte array to our infrastructure. See `libasmir/ir/ir_program.cpp:asm_insn_to_ir()` for an example.
- The instruction addresses and entry/exit points are given to libasmir, along with the executable. The output is again `asm_function_t`'s.

The output `asm_function_t`'s are then used by the rest of the infrastructure, exactly as with our current infrastructure.

Note: There is currently (10/13/2006) a master's student working on this interface with IDA pro.

4.5 Notes

- We currently have a hack to disassemble executables without a symbol table by doing a linear disassembly of the entire section and returning a single `asm_function_t` for all instructions in the section.
- Separating code from data has been shown to be reducible to the halting problem. Thus, disassembly itself is reducible to the halting problem. Therefore, there will always be limitations to any disassembler.
- It is quite common for a 1 line C program to be disassembly to be tens of thousands of lines long. The central reason is standard libraries for loading, dealing with possible errors, and dealing with global definitions are quite large, and included in every executable. For example:

```
void main() return 42;
```

results in an executable about 4000 bytes long. See [A Whirlwind Tutorial on Creating Really Teensy ELF Executables for Linux](#) [7] for more information on how to make such a program much smaller.

- We provide some interfaces for disassembling and converting to the IR instruction(s) named by an instruction address (or address list) in `ir_program.h`. However, we do not provide an interface for just disassembling instructions as named by an address list. A starting point would be to look at `ir_program.cpp:asm_insn_to_ir()`.

5 VinE Formalism

5.1 VinE Abstract Syntax

The VinE abstract syntax is shown in Table 1. Items in bold are keywords in the VinE grammar. This grammar is designed to make translation from assembly as easy as possible.

There are a couple of other things noteworthy about this grammar:

- The declaration `var x:t1[t2]` declares a map which takes in a `t2` and returns a `t1`. Maps are used to model memory.
- We allow casting when the translation `is_strict` variable is false. In this case, maps can be “cast” as a different type by annotating it with the desired type. This is used to allow otherwise broken code, such as:

```
var mem:reg8_t[reg32_t];
var x:reg32_t;
mem[2:reg8_t] = x;
```

This code will not typecheck by default. However, when `is_strict` is false, we will translate code based upon a user annotation such as:

```
mem[2:reg8_t]:reg32_t[reg32_t] = x;
```

to

```
mem[2:reg8_t] = byte0(x);
mem[3:reg8_t] = byte1(x);
mem[4:reg8_t] = byte2(x);
mem[5:reg8_t] = byte3(x);
```

The annotation is only valid if it specifies a smaller value type (e.g., index types cannot be cast) of the array.

- The `addr_t` type denotes the memory address type on the host machine. Variables of type `addr_t` are used to implement calls to external functions.
- We have a couple of derived forms, i.e., they are syntactic sugar for more complicated expressions.

e_1	e_2
true	1:reg1_t
false	0:reg1_t
NULL	0:addr_t

- We currently only support basic integer operations. In particular, we do not support floating point.
- All functions, even those “built-in”, must be explicitly declared. An example built-in function is `addr_t alloc(reg32_t x)`, which allocates `x` contiguous bytes of memory on the host machine.

As normal, some statements accepted by the abstract syntax are not well-formed. The job of typechecking in our context is to check, as much as possible, that statements are well formed, i.e., the types make sense and that execution will not obviously “get stuck”.

program	::=	[stmt]*
stmt	::=	jmp (exp); cjmp (exp, exp, exp); special (string); label id : simplestmt ; fundecl ; vardecl ; fundefn block ;
simplestmt	::=	lval = exp exp return exp return lval = id (args)
fundefn	::=	typ id (formals) block
fundecl	::=	[extern]? typ id (formals);
formals	::=	ϵ id:typ [, id:typ]*
vardecl	::=	var id [, id]* : typ
block	::=	{ [stmt]* }
lval	::=	exp [exp] (lval) id lval : typ
exp	::=	(exp) exp binop exp unop exp const unknown string name (id) lval let lval = exp in exp cast (exp) cast_typ : typ
args	::=	ϵ exp [, exp]*
typ	::=	basetyt attrs typ attrs [typ attrs] typ attrs *
basetyt	::=	reg1_t reg8_t reg16_t reg32_t reg64_t void string_t addr_t
cast_typ	::=	U Unsigned S Signed H High L Low
attrs	::=	ϵ _attr_ (id[,id]*)
binop	::=	arith_bops rel_bops
arith_bops	::=	PLUS (+) MINUS (-) TIMES (*) DIVIDE (/) SDIVIDE (/ \$) MOD (%) LSHIFT (<<) RSHIFT (>>) ARSHIFT (@ >>) AND (&) OR () XOR (^)
rel_bops	::=	EQ (==) NEQ (<>) LT (<) SLE (<= \$) SLT (< \$)
unop	::=	NEG (-) NOT (!)
const	::=	[0-9][0-9]*: typ string true false NULL
string	::=	“ [any char except quote and newline]* “

Table 1: VinE abstract syntax.

GCC Pre-processing We use gcc to pre-process VinE input programs written in the abstract syntax. This allows users to use `#include` and `#define`.

5.2 VinE Internal Representation (IR)

Our IR is shown in Table 2. Our IR consists of statements (`stmt`) and expressions (`exp`). Expressions are pure, i.e., side-effect free. One thing to note is that both jumps (`Jmp`) and conditional jumps (`CJmp`) have targets that are expressions, not necessarily labels. That is because the jump target may be indirect, i.e., calculated via an expression.¹

Statements in our language are:

- `Jmp(exp)` - `Jmp` is an unconditional jump. `exp` is the jump target. If `exp` is of type `Name`, then the jump is to a known location, i.e., a direct jump. If `exp` is not of type `Name`, then the jump is indirect.
- `CJmp(exp, true_exp_target, false_exp_target)` - `CJmp` is a conditional jump. `exp` is evaluated, and if true, control is transferred to `true_exp_target`, else control is transferred to `false_exp_target`. When constructing a `CJmp`, `exp` should evaluate to type `bool`, though we do not explicitly check. `true_exp_target` and `false_exp_target` are similar to the expression in `Jmp`: if they are of type `Name`, then the jump target is known, else the jump target is indirect. Note that in binaries, usually `false_exp_target` will be a fall-through address, thus a known location, thus a `Name`.
- `Move(lhs_exp, rhs_exp)` - `Move` is our assignment statement `lhs_exp := rhs_exp`. `Move` is used for both load and store, i.e, if `lhs_exp` is a `Mem`, then the `Move` is a store to memory, and if `lhs_exp` is a `Temp`, then this is a load.
- `Call(lvalue_option, var, exp_list)` - calls the function named by `id` with arguments `exp_list`. The call can optionally return a value. Functions with no return have `None` as the `lvalue_option`, otherwise will have `Some(x)` where the statement can be read as `x = id(arg1, arg2, ..., argn)`.
- `Function(var, typ, decl_list, bool, stmt_option)` - if `bool` is false, this is an external function. If `stmt_option = None`, then this is a declaration. If `stmt_option = Some`, then this is a function definition. The function name is `var`, with return type `typ`, and formal arguments named by `decl_list`. We require formals to have names and types, unlike C (which just requires the types).
- `Special(string)` - `Special` are for instructions that change the processor state, such as halt, interrupts, etc. The `string` is the x86 instruction name.
- `Label(exp)` - A label is an abstract location in the program, e.g., the beginning of a basic block. Labels serve as targets for jumps (both `Jmp` and `CJmp`). Direct jumps will have `exp` be a `NAME`, while indirect jumps will have some arithmetic expression. Labels have no effect on execution.
- `ExpStmt(exp)` - An `ExpStmt` is a statement which executes an expression, then throws away the result. `ExpStmt` is useful for analysis: you will not see a direct translation of an x86 instruction to `ExpStmt`.
- `Comment(string)` - A `Comment` is a user-written comment in the code. Comments may be inserted by analysis, during translation, etc, and are used to make the code more readable. Comments have no effect on execution.

¹One may wonder why we have `Label` and `Name`. Both name a location in the program. The reason we need `Name` as an expression is because jump targets should be expressions, i.e., they may be calculated, or they may be a known location given by `Name`. However, we want program locations to be “higher level” than expressions, so we also have `Label`. Thus, we have `Name` of `Label`, i.e., an expression containing a statement. This may seem weird, but since `Label` has no side effects, it still is at least consistent in some respect.

stmt	::=	Jmp of exp CJmp of exp * exp * exp Move of lvalue * exp Special of string Label of label Block of decl list * stmt list ExpStmt of exp Comment of string Function of var * typ * decl list * bool * stmt option Call of lvalue option * var * exp list Attr of stmt * attribute
exp	::=	BinOp of binop_type * exp * exp UnOp of unop_type * exp Constant of typ * value Name of label Cast of cast_type * reg_type * exp Unknown of string Lval of lvalue Let of lvalue * exp * exp
lvalue	::=	Temp of var * typ Mem of var * typ * exp
binop_type	::=	arith_bops rel_bops
arith_bops	::=	PLUS (+) MINUS (-) TIMES (*) DIVIDE (/) SDIVIDE (/ \$) MOD (%) LSHIFT (<<) RSHIFT (>>) ARSHIFT (@ >>) AND (&) OR () XOR (^)
rel_bops	::=	EQ (==) NEQ (<>) LT (<) SLE (<= \$) SLT (< \$)
unop_type	::=	NEG (-) NOT (!)
typ	::=	reg_t Array of typ * typ TAttr of attributes * typ
reg_t	::=	REG_64 REG_32 REG_16 REG_8 REG_1
cast_type	::=	CAST_UNSIGNED CAST_SIGNED CAST_HIGH CAST_LOW CAST_FLOAT CAST_INTEGER CAST_RFLOAT CAST_RINTEGER
decl	::=	var * typ
var	::=	string
label	::=	string

Table 2: Our IR Constructors

5.2.1 VinE Typechecking

All VinE IR statements are typechecked. Our typing contexts and types are as follows:

<i>Attributes</i>	$\alpha ::= \text{const} \mid \text{string}$
<i>Types</i>	$\tau ::= \text{reg1_t} \mid \text{reg8_t} \mid \text{reg16_t} \mid \text{reg32_t} \mid \text{reg64_t}$ $\mid \text{string_t} \mid \text{addr_t} \mid \text{void} \mid \tau_1 [\tau_2] \mid \tau^\alpha$
<i>Functions</i>	$\Sigma ::= \cdot \mid \Sigma, (x:\tau; [p_1:\tau_1, \dots, p_n:\tau_n])$
<i>Variables</i>	$\Gamma ::= \cdot \mid \Gamma, x:\tau$
<i>Labels</i>	$\mathcal{L} ::= \cdot \mid \mathcal{L}, x:d$

We use the following conventions:

Symbol	Meaning
Γ	Our variable typing context.
Σ	Our function typing context.
Ω	Our current return type.
\mathcal{L}	Our current label context.
$\bar{\Gamma}(x)$	x is free in the variable context Γ .
$\bar{\Sigma}(x)$	x is free in the function context Σ .
$\bar{\mathcal{L}}(x)$	x is free in the label context \mathcal{L} .
$\vdash \Gamma \text{ VCon}$	Γ is a valid variable context.
$\vdash \Sigma \text{ SCon}$	Σ is a valid function context.
$\vdash \mathcal{L} \text{ LCon}$	\mathcal{L} is a valid label context.

Our function context Σ binds a function name to its type. Our variable context Γ binds a variable name to its declared type. The purpose of the function and variable context is to ensure consistency among declarations, e.g., variables should be declared before use and variables should not be redeclared with a new type.

The purpose of the label context \mathcal{L} is to ensure that 1) all direct jumps are to well-defined labels 2) jumps targets in the global space are only to global labels, e.g., no jumping to the middle of a function, and 3) that jumps targets inside a function are to labels within the function, e.g., no jumps outside the function. The label context \mathcal{L} binds a label named to one of three states: defined but not referenced d, referenced but not defined state r, or a referenced and defined state dr. While the

The purpose of the return type context Ω is to keep track of the current return type for a function.

A typing rule is of the form:

$$\frac{\text{premise 1} \quad \text{premise 2} \quad \dots \quad \text{premise } n}{\text{Context} \vdash \text{Foo} : \text{Bar}}$$

and is read from bottom to top in the following way: *Context* tells us *Foo* is of type *Bar* **when** *premise 1* **and** *premise 2* **and** ... *premise n* are true. If any of the premises is not true, then we cannot conclude anything, and typechecking fails.

Auxiliary Predicates: We first define auxiliary predicates for type compatibility ($T_{\text{compat}}(a, b)$), integer types ($T_{\text{int}}(\tau)$), and valid types ($T_{\text{valid}}(\tau)$).

$$\frac{\tau \in \{\text{reg1_t}, \text{reg8_t}, \text{reg16_t}, \text{reg32_t}, \text{reg64_t}, \text{addr_t}\}}{\vdash T_{\text{int}}(\tau)} \quad \frac{\vdash T_{\text{compat}}(\tau_1, \tau_2)}{\vdash T_{\text{compat}}(\tau_1^{\alpha_1}, \tau_2^{\alpha_2})}$$

$$\frac{}{\vdash T_{\text{compat}}(\text{addr_t}, \text{reg32_t})} \quad \frac{}{\vdash T_{\text{compat}}(\tau, \tau)} \quad \frac{\tau \text{ is a type}}{\vdash T_{\text{valid}}(\tau)}$$

We also define an auxiliary predicate that all labels referenced are also defined:

$$\frac{\forall \ell \in \mathcal{L} : \mathcal{L} \not\ni \ell : r}{\vdash L_{\text{valid}}(\mathcal{L})}$$

Expression Types: In this step we typecheck expressions. For Name's, we return void and extend the label context. As we will see, when typechecking a function, we create an empty label context, typecheck the function body, then make sure all labels referenced are also defined. Similarly, we use a unique label context for typechecking the global space.

$$\frac{T_{\text{int}}(\tau)}{\vdash \text{Constant}(\tau, \text{Int}(x)) : \tau} \quad \frac{}{\vdash \text{Constant}(\text{addr_t}, \text{Int}(x)) : \text{addr_t}}$$

$$\frac{}{\mathcal{L}; \Gamma \vdash \text{Constant}(\text{string_t}, \text{String}(x)) : \text{string_t}} \quad \frac{\Gamma \vdash x : \tau}{\mathcal{L}; \Gamma \vdash \text{Lval}(x) : \tau}$$

$$\frac{\mathcal{L}; \Gamma \vdash t_1 : \tau \quad T_{\text{int}}(\tau) \quad \circ \in \text{unop_type}}{\mathcal{L}; \Gamma \vdash \text{UnOp}(\circ, t_1) : \tau} \quad \text{No rule for Unknown}$$

$$\frac{\mathcal{L}; \Gamma \vdash t_1 : \tau_1 \quad \mathcal{L}; \Gamma \vdash t_2 : \tau_2 \quad T_{\text{int}}(\tau_1) \quad T_{\text{compat}}(\tau_1, \tau_2) \quad \circ \in \text{arith_bops}}{\mathcal{L}; \Gamma \vdash \text{BinOp}(\circ, t_1, t_2) : \tau}$$

$$\frac{\mathcal{L}; \Gamma \vdash t_1 : \tau_1 \quad \mathcal{L}; \Gamma \vdash t_2 : \tau_2 \quad T_{\text{int}}(\tau_1) \quad T_{\text{compat}}(\tau_1, \tau_2) \quad \circ \in \text{rel_bops}}{\mathcal{L}; \Gamma \vdash \text{BinOp}(\circ, t_1, t_2) : \text{reg1_t}}$$

$$\frac{\Gamma \vdash lv : \tau_1 \quad \mathcal{L}; \Gamma \vdash e_1 : \tau_2 \quad T_{\text{compat}}(\tau_1, \tau_2) \quad \Gamma \vdash \Gamma, lv : \tau_1 \quad VCons \quad \mathcal{L}; \Gamma, lv : \tau_1 \vdash e_2 : \tau}{\mathcal{L}; \Gamma \vdash \text{Let}(lv, e_1, e_2) : \tau}$$

$$\frac{\mathcal{L}; \Gamma \vdash e : \tau_1}{\mathcal{L}; \Gamma \vdash \text{Cast}(ct, \tau, e) : \tau} \quad \frac{\mathcal{L} \vdash \ell : d}{\mathcal{L}; \Gamma \vdash \text{Name}(\ell) : \text{addr_t}}$$

L-Values:

$$\frac{\Gamma \vdash x : \tau}{\mathcal{L}; \Gamma \vdash \text{Temp}(x, \tau) : \tau} \quad \frac{\mathcal{L}; \Gamma \vdash e : \tau_2 \quad \Gamma \vdash x : \tau_1[\tau_2]}{\mathcal{L}; \Gamma \vdash \text{Mem}(x, \tau_1[\tau_2], e) : \tau_1}$$

Extensions to Contexts: Statements may extend our context. Each extension must be checked for consistency, e.g., a variable is not redeclared with a new type in the same scope. We do variables to be redeclared in the same context (i.e., shadowed) as long as the type is the same. Functions cannot be redefined, but may be redeclared any number of times. We define valid extensions to our context as follows:

$$\frac{}{\Gamma \vdash \cdot VCon} \quad \frac{\Gamma(x) \vdash \tau}{\Gamma \vdash \Gamma, x : \tau VCon} \quad \frac{\Gamma(x) \vdash \bar{\Gamma}(x)}{\Gamma \vdash \Gamma, x : \tau VCon}$$

$$\frac{}{\Sigma \vdash \cdot SCon} \quad \frac{\Sigma \vdash \bar{\Sigma}(f) \quad T_{\text{valid}}(\tau_r) \quad \forall i : \Gamma \vdash p_i : \tau_i}{\Sigma \vdash \Sigma, (f; \tau_r; \text{reg1_t}; p_1 : \tau_1, p_2 : \tau_2, \dots, p_n : \tau_n) SCon}$$

$$\frac{}{\mathcal{L} \vdash \cdot LCon} \quad \frac{\mathcal{L} \vdash \bar{\mathcal{L}}(\ell)}{\mathcal{L} \vdash \mathcal{L}, \ell : d LCon} \quad \frac{\mathcal{L} \vdash \ell : d}{\mathcal{L} \vdash \mathcal{L}, \ell : d LCon}$$

$$\frac{\mathcal{L} \vdash \bar{\mathcal{L}}(\ell)}{\mathcal{L} \vdash \mathcal{L}, \ell : r LCon} \quad \frac{}{\mathcal{L}, \ell : d \vdash \mathcal{L}, \ell : rd LCon}$$

Statements: Statements are valid if they have type void. In the following rules, we use “decls::stmts” to describe a list of declarations followed by a list of statements, with list elements separated by a semi-colon “;”, and “.” to denote the empty list.

$$\frac{\Sigma, \Omega, \mathcal{L}; \Gamma \vdash \text{decls} :: \text{stmts} : \text{void}}{\Sigma; \Omega; \mathcal{L}; \Gamma \vdash \text{Block}(\text{decls} :: \text{stmts}) : \text{void}} \quad \frac{\Gamma \vdash \Gamma, x : \tau VCon \quad \Sigma, \Omega, \mathcal{L}; \Gamma, x : \tau \vdash \text{decls} :: \text{stmts} : \text{void}}{\Sigma; \Omega; \mathcal{L}; \Gamma \vdash x : \tau; \text{decls} :: \text{stmts} : \text{void}}$$

$$\frac{\Sigma; \Omega; \mathcal{L}; \Gamma \vdash s : \tau_1 \quad \Sigma, \Omega, \Gamma \vdash \text{stmts} : \text{void}}{\Sigma; \Omega; \mathcal{L}; \Gamma \vdash \cdot :: s; \text{stmts} : \text{void}}$$

$$\frac{\mathcal{L}; \Gamma \vdash e : \text{addr_t}}{\Sigma; \Omega; \mathcal{L}; \Gamma \vdash \text{Jump}(e) : \text{void}}$$

$$\frac{\Gamma \vdash e : \text{reg1_t} \quad \mathcal{L}; \Gamma \vdash e_t : \text{addr_t} \quad \mathcal{L}; \Gamma \vdash e_f : \text{addr_t}}{\Sigma; \Omega; \mathcal{L}; \Gamma \vdash \text{CJump}(e, e_t, e_f) : \text{void}}$$

$$\frac{\Gamma \vdash l : \tau_1 \quad \mathcal{L}; \Gamma \vdash e : \tau_2 \quad T_{\text{compat}}(\tau_1, \tau_2)}{\Sigma; \Omega; \mathcal{L}; \Gamma \vdash \text{Move}(l, e) : \text{void}}$$

$$\frac{\mathcal{L} \vdash \bar{\mathcal{L}}(\ell)}{\Sigma; \Omega; \mathcal{L}; \Gamma \vdash \text{Label}(\ell) : \text{void}, \mathcal{L}, \ell : d LCon} \quad \frac{\mathcal{L} \vdash \ell : r}{\Sigma; \Omega; \mathcal{L}; \Gamma \vdash \text{Label}(\ell) : \text{void}, \mathcal{L}, \ell : rd LCon}$$

$$\frac{}{\Sigma; \Omega; \mathcal{L}; \Gamma \vdash \text{Special}(x) : \text{void}} \quad \frac{\mathcal{L}; \Gamma \vdash x : \tau}{\Sigma; \Omega; \mathcal{L}; \Gamma \vdash \text{ExpStmt}(x) : \text{void}}$$

$$\frac{\Sigma; \Omega; \mathcal{L}; \Gamma \vdash x : \tau}{\Sigma; \Omega; \mathcal{L}; \Gamma \vdash \text{Attr}(a, x) : \text{void}} \quad \frac{\Sigma \vdash x : (x : \text{void}, \tau_1, \tau_2, \dots, \tau_n) \quad \forall n : \mathcal{L}; \Gamma \vdash e_n : \tau_n}{\Sigma; \Omega; \mathcal{L}; \Gamma \vdash \text{Call}(\text{None}, x, e_1, e_2, \dots, e_n) \vdash \text{void}}$$

$$\frac{\Gamma \vdash l : \tau \quad \Sigma \vdash x : (x : \tau_r, \tau_1, \tau_2, \dots, \tau_n) \quad T_{\text{compat}}(\tau, \tau_r) \quad \forall n : \mathcal{L}; \Gamma \vdash e_n : \tau_n}{\Sigma; \Omega; \mathcal{L}; \Gamma \vdash \text{Call}(\text{Some}(l), x, e_1, e_2, \dots, e_n) \vdash \text{void}}$$

$$\frac{\mathcal{L}; \Gamma \vdash e : \tau \quad \Omega \vdash \tau}{\Sigma; \Omega; \mathcal{L}; \Gamma \vdash \text{Return}(\text{Some}(e))} \quad \frac{\Omega \vdash \text{void}}{\Sigma; \Omega; \mathcal{L}; \Gamma \vdash \text{Return}(\text{None})}$$

Functions: Although functions are a type of statement, we only allow functions in the global scope. For ease of writing, we have eliminated the scoping context from rules, treating this as a side condition.

Typechecking function definitions may look complicated, but hopefully each step makes sense. We

1. As a side condition, verify we are in the global scope.
2. Extend Σ with the function type (i.e., the definition is also a declaration).

6 WEAKEST PRE-CONDITION

3. Extend Γ with the formal parameters since they are local variables.
4. Create an empty label context $\mathcal{L}' := \cdot$.
5. Create an empty return context $\Omega := \cdot$.
6. Typecheck the function body.
7. Check if the return type is the declared return type.
8. Check if all labels referenced are also defined $L_{\text{valid}}(\mathcal{L})$.

$$\frac{\text{scope} = 0; \Sigma \vdash \Sigma, (x : \tau_r; \text{false}; p_1 : \tau_1, \dots, p_2 : \tau_n) \text{SCon} \quad \forall n : \Gamma \vdash \Gamma, p_1 : \tau_n \text{VCon}; \mathcal{L}' := \text{Labels} \in s; \Omega := \tau_r; \Omega, \mathcal{L}', \Gamma \vdash \text{Block}(s) : \text{void}; L_{\text{valid}}(\mathcal{L}')}{\Sigma; \Omega; \mathcal{L}; \Gamma \vdash \text{Function}(x, \tau_r, p_1 : \tau_1, p_2 : \tau_2, \dots, p_n : \tau_n, \text{false}, \text{Block}(s)) : \text{void}}$$

Function declarations are much simpler: we just need to verify the function type is well-formed. We do this for both external (when the second-to-last parameter is true), and for local functions (when the second-to-last parameter is false).

$$\frac{\Sigma \vdash \Sigma, (x : \tau_r; \text{true}; p_1 : \tau_1, \dots, p_2 : \tau_n) \text{SCon} \quad \forall n : \Gamma \vdash \Gamma, p_1 : \tau_n \text{VCon} \quad \text{scope} = 0}{\Sigma; \Omega; \mathcal{L}; \Gamma \vdash \text{Function}(x, \tau_r, p_1 : \tau_1, p_2 : \tau_2, \dots, p_n : \tau_n, \text{true}, \text{None}) : \text{void}}$$

$$\frac{\Sigma \vdash \Sigma, (x : \tau_r; \text{false}; p_1 : \tau_1, \dots, p_2 : \tau_n) \text{SCon} \quad \forall n : \Gamma \vdash \Gamma, p_1 : \tau_n \text{VCon} \quad \text{scope} = 0}{\Sigma; \Omega; \mathcal{L}; \Gamma \vdash \text{Function}(x, \tau_r, p_1 : \tau_1, p_2 : \tau_2, \dots, p_n : \tau_n, \text{false}, \text{None}) : \text{void}}$$

Typechecking a program: To typecheck a program = stmt list, we

1. Initialize Σ , Γ , Ω , and \mathcal{L} to the empty set
2. Build a label context \mathcal{L} for all defined labels in each scope. These rules are not shown, but essentially each function will have its own label context, as well as the global scope. The label contexts for the global and each function scope are disjoint.
3. Iterate over all statements and typecheck using the above rules.

In practice, the last two steps can be combined in the typechecking implementation.

6 Weakest Pre-Condition

6.1 Background

Let \mathcal{P} be a program, and $\text{Var}(\mathcal{P}) = \langle v_1, v_2, \dots, v_k \rangle$ be the set of program variables. In assembly, we consider all registers including the instruction counter and memory locations program variables. The *state space* of the program is the cross product of all variables: $v_1 \times v_2 \times \dots \times v_k$. A predicate on the state space is a function on state-space variables which returns either true (true) or false (false).

Let Q be a predicate on the state space of \mathcal{P} . There are three things that can happen when we run \mathcal{P} :

1. \mathcal{P} can terminate in a state satisfying Q
2. \mathcal{P} can terminate in a state satisfying $\neg Q$.
3. \mathcal{P} does not terminate.

The *weakest pre-condition* $wp(\mathcal{P}, Q)$ characterizes the minimum requirements on the pre-states for which running \mathcal{P} will terminate in a state satisfying Q . In other words, activating \mathcal{P} in a state satisfying $wp(\mathcal{P}, Q)$ is *gaurenteed* to terminate in a state satisfying Q . $wp(\mathcal{P}, Q)$ is called the weakest pre-condition

because there may be stronger pre-conditions Q_s which also result in the program terminating in a state satisfying Q , i.e., $P_s \rightarrow wp(\mathcal{P}, Q)$. Thus, the set of states characterized by $P_s \subseteq wp(\mathcal{P}, Q)$.

For example, to calculate whether \mathcal{P} will always correctly terminate, we calculate $wp(\mathcal{P}, true)$ because:

$$wp(\mathcal{P}, Q) \wedge wp(\mathcal{P}, \neg Q) \equiv wp(\mathcal{P}, Q \wedge \neg Q) \equiv wp(\mathcal{P}, true)$$

The *weakest liberal pre-condition* $wlp(\mathcal{P}, Q)$ characterizes the minimum requirements on the pre-states for which running \mathcal{P} will terminate in a state satisfying Q if it terminates at all. wlp is more liberal because it only guarantees \mathcal{P} won't terminate in a wrong state: it does not guarantee the program terminates.

We can therefore define 7 possible outcomes [4]:

1. Activation of \mathcal{P} will establish Q :
 $wp(\mathcal{P}, Q) = wlp(\mathcal{P}, Q) \wedge wp(\mathcal{P}, true)$
2. Activation of \mathcal{P} will establish $\neg Q$:
 $wp(\mathcal{P}, \neg Q) = wlp(\mathcal{P}, \neg Q) \wedge wp(\mathcal{P}, true)$
3. Activation will fail to lead to a terminating state:
 $wlp(\mathcal{P}, false) = wlp(\mathcal{P}, Q) \wedge wlp(\mathcal{P}, \neg Q)$
4. Activation will lead to a terminating state, but the initial state is insufficient to determine if Q is satisfied:
 $wp(\mathcal{P}, true) \wedge \neg(wlp(\mathcal{P}, Q) \wedge \neg wlp(\mathcal{P}, \neg Q))$
5. If activation leads to a final state, then it will satisfy Q , but the initial state does not determine whether the activity will terminate:
 $wlp(\mathcal{P}, Q) \wedge \neg wp(\mathcal{P}, true)$
6. If activation leads to a final state, then that state will not satisfy Q . However, the initial state does not tell us whether we will terminate:
 $wlp(\mathcal{P}, \neg Q) \wedge \neg wp(\mathcal{P}, true)$
7. The initial state does not determine whether or not we will satisfy Q or terminate:
 $\neg(wlp(\mathcal{P}, Q) \vee wlp(\mathcal{P}, \neg Q) \vee wp(\mathcal{P}, true))$

The last four possibilities only exist for non-deterministic machines.

Two important final notes. First, weakest pre-conditions are monotonic: if $Q_1 \rightarrow Q_2$ then $wp(\mathcal{P}, Q_1) \rightarrow wp(\mathcal{P}, Q_2)$. Second, $wp(\mathcal{P}, false) = false$ always (this is sometimes referred to as the principle of excluded miracle).

6.2 The Guarded Command Language (GCL)

The weakest pre-condition is calculated in a syntax-directed manner from the *guarded command language* (GCL), which is shown in Table 3. Later we will describe how we translate our assembly into GCL.

An **assert** statement asserts that an expression is true. If the expression is false, the assert blocks and the computation has *gone wrong* along that path. If the asserted expression is true, then the assert block is equivalent to a nop. For example, **assert**(false) can be used to indicate an infeasible path.

An **assume** statement adds an assumption about the expression. For example, **assume**($x > 5$) would restrict x to the values greater than 5 in all subsequent computation.

A, B, S ∈ GCL stmt	::= assert (exp)
	assume (exp)
	lv := exp (lv is a valid l-value)
	A; B
	A □ B
	skip

Table 3: The guarded command language (left)

$\frac{wp(A, Q) Q_1 \quad wp(B, Q) Q_2}{wp(A \square B, Q) Q_1 \wedge Q_2}$	WP-BAR	$\frac{wp(B, Q) Q_1}{wp(A; B, Q) wp(A, Q_1)}$	WP-SEQ
$\frac{}{wp(lv := e) Q(lv/e)}$	WP-ASG	$\frac{}{wp(\mathbf{skip}, Q) Q}$	WP-SKIP
$\frac{}{wp(\mathbf{assume} \ e, Q) e \Rightarrow Q}$	WP-ASSUME	$\frac{}{wp(\mathbf{assert} \ e, Q) e \wedge Q}$	WP-ASSERT

Table 4: Syntax-directed method for calculating the weakest pre-condition.

GCL also offers assignments, sequences ($A; B$), **skip** which is equivalent to a nop, and the logical constants.

The GCL statement $A \square B$, pronounced *A bar B*, is a choice statement between either A or B . For example, we can write an if-then-else statement if e then A else B as $(\mathbf{assume}(e); A;) \square (\neg \mathbf{assume}(e); B)$.

Note that at this time we do not have constructs for loops. The reason is that calculating the weakest pre-condition for a loop requires additional information about the behavior of the loop that cannot be determined syntactically (or necessarily statically). Therefore, for now we focus on loopless programs.

6.3 Calculating the weakest pre-condition

The weakest pre-condition is calculated in a syntax-directed manner from the guarded command language (GCL), as shown in Table 4. Each rule is of the form $S|Q$, which should be read as given S , we output pre-condition Q . For example, $wp(s_1; s_2; s_3, Q)$ is computed as $wp(s_1, wp(s_2, wp(s_3, Q)))$.

$\frac{A(lv/e) A_1 \quad B(lv/e) B_2}{A \text{ bop } B \ (lv/e) A_1 \text{ bop } B_1}$	SUB-BOP	$\frac{A(lv/e) A_1}{uop \ A(lv/e) uop \ A_1}$	SUB-UOP
$\frac{t \in \mathbf{TEMP} \quad lv = t}{t(lv/e) e}$	SUB-TEMP	$\frac{A(lv/e) A_1 \quad \mathbf{alias}(mem[A_1], lv) = true}{MEM[A](lv/e) e}$	SUB-TALIAS
$\frac{A(lv/e) A_1 \quad \mathbf{alias}(MEM[A_1], lv) = false}{MEM[A](lv/e) MEM[A_1]}$	SUB-FALIAS	$\frac{A(lv/e) A_1 \quad \mathbf{alias}(MEM[A_1], lv) = \mathbf{M}}{MEM[A](lv/e) \mathbf{if} \ A_1 = lv \ \mathbf{then} \ e \ \mathbf{else} \ MEM[A_1]}$	SUB-MALIAS

Table 5: Semantics for substitution on post-condition Q .

When we encounter an assignment statement $wp(lv := e, Q)$, we substitute all occurrences of lv in Q for e . Substitution is written $Q(t/x)$ which means substitute all occurrences of x for t in Q . Substitution is formally defined in Table 5. This captures the semantics of assignment in a logical form. Replacing “all occurrences” requires some thought, however, when faced with memory references. The central problem with memory references is that two memory references may be aliased. For example, we may have $MEM[x] = MEM[y]$ when $x = y$. However, we cannot determine statically when $x = y$, e.g., if either x or y is symbolic there may be some values where $x = y$ and some where $x \neq y$.

Therefore, during substitution we rely on an auxiliary *alias* function which for $alias(x, y)$ returns one of three values: true if x and y *must* be aliased, false if x and y are *definitely not* aliased, and **M** if x and y *may* be aliased (but are not definitely aliased, e.g., may and must do not overlap). Note returning **M** is always sound, and can be used in lieu of real alias analysis.

Using the weakest pre-condition The weakest pre-condition $wp(\mathcal{P}, Q)$ is also a binary predicate over the state space of the program. Thus, if the state space is n -dimensions, the weakest pre-condition will define some n -dimensional sub-space.

In our project, we often want to find a binding of values to variables which appear in $wp(\mathcal{P}, Q)$, which we call *solving* the weakest pre-condition. It is important to keep in mind that we are using the term “solving” loosely: we are simply finding a single point in the solution space. We discuss how we find a solution later.

6.4 Efficient weakest pre-condition calculation

The weakest pre-condition computation algorithm above is an adaption of traditional methods, dating back to Dijkstra in 1976 [4]. The weakest pre-condition calculation as presented may result in a pre-condition that is exponential in the program size. For example, consider:

$$\begin{aligned} x_1 &= x_0 + x_0 \\ x_2 &= x_1 + x_1 \\ x_3 &= x_2 + x_2 \\ &\dots \\ x_n &= x_{n-1} + x_{n-1} \end{aligned}$$

Post-conditions involving x_n will result in an exponentially sized pre-condition. For example, if $Q = x_3 < 5$, then the resulting weakest pre-condition is:

$$((x_0 + x_0) + (x_0 + x_0)) + ((x_0 + x_0) + (x_0 + x_0)) < 5$$

To address the size issue, we can simply use `let`-bindings for all assignments. The resulting pre-condition will be linear in the size of the program. Again, if $Q = x_3 < 5$, the weakest pre-condition is:

```
let  $x_1 = x_0 + x_0$  in
  let  $x_2 = x_1 + x_1$  in
    let  $x_3 = x_2 + x_2$  in
       $x_3 < 5$ 
```

Note by using `let` bindings we have simply reduced the space for representing the weakest pre-condition: we have not necessarily reduced the complexity of solving the formula (i.e., instantiating values for variables such that the weakest pre-condition is true).

There will be more added here later.

6.5 Other Notes

The standard references for weakest-preconditions are [2,4].

References

- [1] IA-32 Intel Architecture Software Developer's Manual: Volume 1, Basic Architecture, 2004.
- [2] Edward Cohen. Programming in the 1990's. Springer-Verlag, 1990.
- [3] Ben Collins-Sussman, Brian W. Fitzpatrick, and C. Michael Pilato. Version Control with Subversion. O'Reilly, 2007.
- [4] E.W. Dijkstra. A Discipline of Programming. Prentice Hall, Englewood Cliffs, NJ, 1976.
- [5] GNU Software Foundation. LIB BFD, the binary file descriptor library. <http://www.gnu.org/software/binutils/manual/bfd-2.9.1/bfd.html>.
- [6] Christopher Kruegel, William Robertson, Fredrik Valeur, and Giovanni Vigna. Static disassembly of obfuscated binaries. In Proceedings of the 13th USENIX Security Symposium, 2004.
- [7] Brian Raiter. A whirlwind tutorial on creating really teensy elf executables for linux. <http://www.muppetlabs.com/~breadbox/software/tiny/teensy.html>.