How to Install and Run an Example with VinE

Original version by Prateek Saxena, with updates from Stephen McCamant

Sep 12th, 2008: SVN trunk r3466 and Ubuntu 8.04

This document is intended to be a quick start guide for setting up and running VinE, the static analysis component of the BitBlaze Binary Analysis Framework. It assumes that you have some familiarity with Linux. The instructions are based on the version of VinE in the SVN trunk as of the date shown in the header, running on a vanilla Ubuntu 8.04 distribution of Linux. The procedure is intermixed with explanations about utilities to give an overview of how things work. The goal in this exercise is to trace from a simple program with symbolic keyboard input, and generate an STP file which models the weakest precondition of the control-flow path the program took. In other words, the conditions on the inputs that cause it to take a execute a certain branch of code. To follow along with the instructions, you'll need to start with a trace file like the one generated in the separate TEMU tutorial.

1 Installation

The following script, which is also found as `docs/install-vine.sh` in the VinE source, shows the steps for building and installing Vine and the other software it depends on:

```
#!/bin/bash
# Instructions for installing VinE on Linux

# In the form of a shell script that runs on a fresh Ubuntu 8.04
# installation. Some adaptation may be needed for other systems.
# Things that require root access are preceded with "sudo".

# See the "INSTALL" file in the VinE source for more discussion on
# what we're doing here.

# Last tested 2008-09-11

# This script will build VinE in a "$HOME/bitblaze" directory
cd ~
mkdir bitblaze
cd bitblaze

# Prerequisite: Valgrind VEX r1749
# VinE uses the VEX library that comes with Valgrind to interpret the
# semantics of instructions. (Note we don't even bother to compile
# Valgrind itself). Because of changes in the VEX interface, earlier
# or later revisions will probably not work without some changes,
```

```
# though we'll probably update to work with a more recent version at
# some point in the future.

# Packages needed to build Valgrind:
sudo apt-get build-dep valgrind
# Extra packages needed to build the SVN version:
sudo apt-get install subversion automake
svn co -r6697 svn://svn.valgrind.org/valgrind/trunk valgrind
(cd valgrind/VEX && svn up -r1749)
(cd valgrind/VEX && make version && make libvex_x86_linux.a && make libvex.a)

# Other prerequisite packages:

# For C++ support:
sudo apt-get install g++

# For OCaml support:
sudo apt-get install ocaml ocaml-findlib libgdome2-ocaml-dev camlidl \
                     libextlib-ocaml-dev

# For the BFD library:
sudo apt-get install binutils-dev

# For the Boost Graph library:
sudo apt-get install libboost-dev libboost-graph-dev

# For the SQLite database:
sudo apt-get install libsqlite3-dev sqlite3 libsqlite3-0 libsqlite3-ocaml-dev

# For building documentation:
sudo apt-get install texlive

# Ocamlgraph >= 0.99c is required. Ocamlgraph is packaged by Debian
# and Ubuntu as libocamlgraph-ocaml-dev, but the latest version in
# Ubuntu is 0.98. The following process for building a package from
# the Debian repository is a bit of a hack.
sudo apt-get install   libocamlgraph-ocaml-dev
sudo apt-get build-dep libocamlgraph-ocaml-dev
sudo apt-get install liblablgtk2-ocaml-dev liblablgtk2-gnome-ocaml-dev \
                     docbook-xsl po4a
sudo apt-get install fakeroot
svn co svn://svn.debian.org/svn/pkg-ocaml-maint/trunk/packages/ocamlgraph \
       -r5983
tar xvzf ocamlgraph/upstream/ocamlgraph_0.99c.orig.tar.gz
mv ocamlgraph/trunk/debian ocamlgraph-0.99c
```

```
perl -pi -e 's[ocaml-nox \(>= 3.10.0-9\)] #\
              [ocaml-nox  (>= 3.10.0-8)]' ocamlgraph-0.99c/debian/control
(cd ocamlgraph-0.99c && dpkg-buildpackage -us -uc -rfakeroot)
sudo dpkg -i libocamlgraph-ocaml-dev_0.99c-2_i386.deb

# VinE itself:
# Trunk:
svn co https://bullseye.cs.berkeley.edu/svn/vine/trunk vine
(cd vine && ./autogen.sh)
(cd vine && ./configure --with-vex=$HOME/bitblaze/valgrind/VEX)
(cd vine && make)
(cd vine/doc && make doc)
```

2 Generating the IR and the STP formula

We start with a trace (generated, for instance, by TEMU) that records the instructions executed on a program run, the data values they operated on, and which data values were derived from a distinguished set of ("tainted") input values. We're going to do operations where we consider that input to be a symbolic variable, but the first step is to interpret the trace. The x86 instructions in the trace are a pretty obscure representation of what is actually happening in the program, so we'll translate them into a cleaner intermediate representation (IR) language.

First, let us check if we have got a meaningful trace. One way to do so is to print the trace, and see that at least the expected instructions are marked as tainted. For this, you may use the `trace_reader` command utility in Vine. As shown below, in the output you should be able to see the compare (or similar) instruction that comapares the input to the immediate value 5. The presence of tainted operands in any instruction are indicated by the record containing "T1".

```
% cd bitblaze/vine
% ./utils/trace_reader -trace /tmp/foo.trace | grep T1
...
...
804845a:       cmpl    $0x5,-0x4(%ebp)   I@0x00000000[0x00000005] \
       T0      M@0xbffffac4[0x00000005]        T1 {15 (1001, 0) (1001, 0) \
         (1001, 0) (1001, 0) }
```

Of course, the real output of that command contains lots of instructions, but we've picked out a key one: an instruction from the main program (you can tell because the address is in the `0x08000000` range) in which a value from the stack (`-0x4(%ebp)`) is compared (a `cmpl` instruction) with a constant integer 5 (`$0x5`).The later fields on the line represent the instruction operands and their tainting.

We can then use a single program, the `appreplay` utility, to both convert the trace into IR for and then to generate an STP formula given the constraints on the symbolic input. The invocation looks like:

```
% ./utils/appreplay -trace /tmp/foo.trace -use-thunks false -use-post-var true \
  -stp-out foo.stp -ir-out foo.ir -wp-out foo.wp
...
Time to create sym constraint from TM: 0.480301
```

This command line produces the final stp file as `foo.stp`, and the intermediate files `foo.ir` and `foo.wp` for aiding explanation and understanding of the internals. Remember that VinE uses its own IR to model the exact semantics of instructions in a simpler RISC-like form. The IR and WP output files are in this IR language. If you aren't interested in these files, you can omit the `-ir-out` and `-wp-out` options.

Intuitively, `appreplay` models the logic of the executed instructions, generating a path constraint needed to force the execution down the path taken in the trace. A variable `post` is introduced, which is the conjunction of the conditions seen along the path. In the file `foo.ir`, you can see this variable is assigned at each conditional branch points as $post = post \wedge condition$, where a condition is a variable modeling the compare operation's result that must be true to force execution to continue along the path taken. (Because the language is explicitly typed and `appreplay` is careful to generate unique names, the full name of the `post` variable is likely something like `post_1034:reg1_t`, where the part after the colon tells you it's a one-bit (boolean) variable.)

This weakest precondition formula is then converted to the format of the STP solver's input.

This Vine utility has several options which are detailed later in Section 4. The ones important here are:

- `-trace` ⟨*file*⟩. Specifies the trace input file.
- `-use-thunks` ⟨*bool*⟩. For many analyses in BitBlaze, the implicit effects of the instruction are irrelevant. It is sometimes not useful to generate complicated side-effect VINE IR inline in the IR statements of instructions. Instead, we can ask VinE to generate calls to these flag computation IR "thunk" functions that model the flag operations. In our case, we do need the flags information to be present inline for the STP formula generation to interpret implicit dependences between compare and branch instructions. Hence, we have disabled the thunk generation.
- `-use-post-var` ⟨*bool*⟩. If this is set to true, then 'assert' statements will be rewritten to update a variable 'post', such that at the end of the trace 'post' will have value 1 iff all assertions would have passed.

3 Querying STP

Now, in the last step we wish to ask the question "what input values force the execution down the path taken in the execution?". In the formula we've built, this is equivalent to asking for a set of assignments that make the variable `post` true. We use STP to solve this for us. Notice that the STP file has the symbolic `INPUT` variable marked free in the final formula.

A symbolic formula F is *valid* iff it is true in all interpretations. In other words, F is valid iff all assignments to the free (symbolic) variables make F true. Given a formula STP decides whether it is valid or not. If it is invalid, then there exists at least one set of inputs that make the

formula false, then STP gives such an assignment (a *counterexample*). We use this trick to get the assignment to the free `INPUT` variable in the formula that makes the execution follow the traced path.

To do this, we add the following 2 lines at the end of the STP file and run STP on it:

```
% cat >>foo.stp
QUERY(FALSE);
COUNTEREXAMPLE;
% ./stp/stp foo.stp
Invalid.
ASSERT( INPUT_1001_0_41  = 0hex35  );
```

STP's reply of `Invalid.` indicates it has determined that the query formula `FALSE` is not valid: there is an assignment to the program inputs that satisfies the other assertions in the file (i.e., would lead the program to execute the same path that was observed), but still leaves `FALSE` false. As a counterexample it gives one such input (in this case, the only possible one), in which the input has the hex value `0x35` (ASCII for `5`).

4 Documentation of various utilities in VinE

Here is a slightly more detailed explanation of the VinE utilities used in this exercise.

4.1 Appreplay

- `-trace` : specifies the TEMU execution trace file to process
- `-state` and `-state-range` are used to initialize ranges of memory locations from a TEMU state snapshot.
- `-conc-mem-idx` is an optimization to do some constant propagation, which appears to help STP quite a bit. This will likely become deprecated once some of the STP optimization issues are resolved.
- `-prop-consts` is another optimization that propagates all constant values using VinE's evaluator.
- `-use-thunks` if set to true, the generated IR will have calls to functions to update the processor's condition codes (`EFLAGS` for the x86). If false, this code will be inlined instead. For most analysis purposes this should be disabled. It may be useful for generating a smaller IR with the intent of giving it to the evaluator rather than to STP.
- `-use-post-var` if this is set to true, then `assert` statements will be rewritten to update a variable 'post', such that at the end of the trace `post` will have value true if and only if all assertions would have passed. This is mostly for backwards compatibility for before we introduced the `assert` statement.

- `-deend` performs "deendianization", i.e. rewrites all memory expressions to equivalent array expressions. This should usually be enabled.
- `-concrete` initializes all the 'input' symbols to the values they had in the trace.
- `-verify-expected` is mostly for regression/sanity tests, in conjunction with `-concrete`. `-verify-expected` adds assertions to verify the all operands subsequently computed from those symbols have the same value as they did in the trace, as they should in this case.
- `-include-all` translates and includes *all* instructions, rather than only those that (may) operate on tainted data. Generally not desirable, but sometimes useful for debugging.
- `-ir-out` specify the output ir file.
- `-wp-out` and `-stp-out` tell appreplay to compute the weakest precondition (WP) over the variable `post` (described above), and convert the resulting IR to an STP formula. the formula holds for inputs that would follow the same execution path as in the trace.

5 Reporting Bugs

Please report bugs to the bugzilla at: `https://bullseye.cs.berkeley.edu/bugzilla/`.

VinE contains a `VERSION` file in its source base directory. Please report the version number from that file when filing bugs. And please also report if you notice something wrong or out of date in this document.