A Note on Issues Encountered with VinE

May 27, 2008

Too Many False Semantic Differences

I noticed that most semantic differences reported by Debin's semantic difference finder were not reported as syntactic differences by DarunGrim. This is opposite the result we want! Upon further investigation, this appears to be caused by the Vine module not handling pointer variables as I would hope for many common cases.

For example, suppose two basic block variables are pointers p_1 and p_2, with $\&p_1 \neq \&p_2$ and $p_1 = p_2$. In many instances, Vine will use the pointer address (the location of the pointer on the stack), instead of the pointer value (the address the pointer is pointing to), thus reporting semantic differences that do not actually exist. The syntactic difference finder, on the other hand, handles the pointers as expected (by using the address the pointer is pointing to).

Here is a specific example from aspnet_filter.dll. The first basic block of the HttpFilterProc function was reported by Debin's semantic difference finder as semantically different. However, DarunGrim reported the blocks as syntactically equivalent! Let's investigate.

First, the VinE disassembly:

```
VinE disassembly, without patch:
6006184a:   push ebp
6006184b:   lea  fffffc70(%esp,1),%ebp
60061852:   sub $0x410,%esp
60061858:   mov 6006305c,%eax
6006185d:   xor %ebp, %eax
6006185f:   mov %eax, 38c(%ebp)
60061865:   push %ebx
60061866:   push %esi
```

```
60061867:   mov 398(%ebp),%esi
6006186d:   mov $0x1000,%ebx
60061872:   cmp %ebx, 39c(%ebp)
60061878:   lea  ffffff8c(%ebp),%eax
6006187b:   push %edi
6006187c:   mov  3a0(%ebp), %edi
60061882:   movl  $0x8000002, ffffff88(%ebp)
60061889:   mov %eax, ffffff84(%ebp)
600618a3:   je 0x60061932

VinE disassembly, with patch:
60061861:   push ebp
60061862:   lea  fffffc70(%esp,1),%ebp
60061869:   sub $0x410,%esp
6006186f:   mov 6006305c,%eax
60061874:   xor %ebp, %eax
60061876:   mov %eax, 38c(%ebp)
6006187c:   push %ebx
6006187d:   push %esi
6006187e:   mov 398(%ebp),%esi
60061884:   mov $0x1000,%ebx
60061889:   cmp %ebx, 39c(%ebp)
6006188f:   lea  ffffff8c(%ebp),%eax
60061892:   push %edi
60061893:   mov  3a0(%ebp), %edi
60061899:   movl  $0x8000002, ffffff88(%ebp)
600618a0:   mov %eax, ffffff80(%ebp)
600618a3:   je 0x6006194c
```

Notice the semantic difference is caused by the second to last instruction in the basic block: the pre-patched version has an ebp offset of `ffffff84` and the post-patched version has an ebp offset of `ffffff80`. However, this VinE disassembly is incorrect, as the following syntactic disassembly shows:

```
DarumGrim disassembly, pre- and post-patch:
push ebp
lea  ebp [esp - 390h]
sub esp 410h
mov eax dword_6006305C
xor eax ebp
```

```
mov [ebp + 390h + var_4] eax
push ebx
push esi
mov esi [ebp + 390h + pfc]
mov ebx 1000h
cmp [ebp + 390h + NotificationType] ebx
lea eax [ebp + 390h + var_404]
push edi
mov edi [ebp + 390h + pvNotification]
mov [ebp + 390h + var_408] 8000002h
mov [ebp + 390h + lpString] eax
jz jump_location
```

Instead of reporting a difference in the next-to-last instruction, DarunGrim claims that ebp is offset by `390h` plus the value pointed to by stack variable `lpString`. This disassembly is also confirmed by IDAPro. Looking more closely at disassembly information in the IDAPro database (note that IDAPro was used at point in the disassembly for both VinE and DarunGrim), we have the following stack frames:

```
Pre-patch stack frame:
-00000410:   var_410
-0000040C:   lpString
-00000408:   var_408
-00000404:   var_404
...
-00000004:   var_4
...
+00000008:   pfc
+0000000C:   NotificationType
+00000010:   pvNotification

Post-patch stack frame:
-00000410:   lpString
-0000040C:   var_40C
-00000408:   var_408
-00000404:   var_404
...
-00000004:   var_4
...
```

```
+00000008:   pfc
+0000000C:   NotificationType
+00000010:   pvNotification
```

All the stack frame variables listed above are dword pointers. `lpString` is the same pointer (points to the same location in memory) in both versions of the file even though it has different locations on the stack frame. The difference in `lpString` stack locations between file versions is 4 bytes, precisely the difference between the ebp offsets `ffffff80` and `ffffff84` reported by VinE. Thus, it appears that VinE is incorrectly using the pointer address instead of the location the pointer is pointing to.

Unfortunately, this scenario was a very common case in the files I analyzed. Therefore, in all test cases, the reported semantic differences falsely outnumbered the reported syntactic differences.

Translating into VinE IR

One of the biggest problems I encountered was how to properly translate binaries into the VinE IR. I had two methods to choose from:

1. Translate using the `toir` shell script (in `vine/utils`).
2. Translate using the `from_elf` and `from_ida` VinE high-level functions (accessible directly from OCaml).

I had success with method #2 only. Although method #1 did translate the binary, no matter what combinations of options I chose, the resulting IR broke the semantic difference finder.

Additionally, there are too many poorly-documented translation options. I understand that VinE is still research quality code, but when attempting to write a tool on top of VinE, it would be very helpful if all the options were documented clearly and completely. Examples of such options include:

- deendization - Describe in detail the method used to deendize the code.
- descoping - What does this do, precisely? No clear definition given in documentation.
- thunks - What are the consequences of enabling or not enabling eflags thunks? No clear definition given in documentation.
- gvn - What is global value numbering? No clear definition given in documentation.

- dc - What is the definition of dead code?
- nosaveglobals - What are the consequences of not collecting and restoring globals in functions? How are globals defined?
- default - What are the default optimizations in its entirety? The documentation says it *may* include deadcode, gvn, etc, but does not provide a comprehensive nor definite list.

In the end, there were too many unanswered questions for me to feel completely comfortable about the soundness of the resulting IR. Did I optimize too many things away? My recommendation is to create a consistent translation interface, with all translations options clearly and completely documented.