Katana Paper Additions

February 24, 2010

2 Tracking Object Dependencies

Note that this is part is a work in progress and is not implemented

3 Automated Patching

3.1 Code Patching

3.1.1 Symbol Resolution

we are no longer copying symbol values into the PO only the names of the symbols so we can look them up on patch application (unless you disagree Sergey and want to go back to what the original paper says)

We *are* handling relocations for code patches, including support for the PLT

3.1.2 Patch Application

Code patching with trampoline jumps *is* implemented. I intend to explore relocating all calls to the function to avoid the overhead of the trampoline but have not actually made any efforts in this direction yet

3.2 Data Patching

3.2.1 Type Discovery

This is implemented. The only difference from the original paper is that I'm only insert differing types if a variable actually using that type is encountered

3.2.2 Data Traversal

adding of the variable to the PO is implemented (although as mentioned earlier we do not add its address, preferring to rely on symbol information in the executable). Issues of scope and checking activation frames are not yet implemented (although they're next on my todo list)

3.3 Patch Application

The basics of this are implemented. Pointers are not yet supported. Eventually we should support custom initializers for new members in types the following is a new section. It should

probably go after section 3 in the original Katana paper

2.4 Challenges

I would mention that there is nothing about our approach that inherently won't work with multithreading, just that dealing with it through ptrace takes considerable work. Also, I don't know if this is really the best place to talk about determining patching safety. Perhaps there should be another section for that?

3 Patch Object Format

3.1 Reasons and Needs

We have developed a Patch Object format with the following properties

- a valid ELF file
- information about types and functions requiring patches contained as DWARF information
- type transformations specified using a language defined by the Dwarf standard

Through the use of existing standards and well-structured ELF files utilizing a simple expression language for data patching, we aim to create patches which are easily examined (or modified) with existing tools. This easy compatibility with the existing binary tools and standards brings us to a very good point: why should patching not be a part of the ABI and of the standard toolchain? This does not necessarily have to be the precise format we use for Katana, any such format that would become a standard, whether an actual standard or a de-facto standard should be well vetted by the community, but we argue that something like this should be included in the standard object types, along with relocatable objects, executable objects, shared libraries, and core dumps. Consider the situation. Relocatable objects containing new code and data which may be inserted at runtime are nothing new. This is the entire premise of the dynamic library. User-written functions which may have to run upon this code injection (in the case of patching data where the desired actions cannot be determined automatically) already exist as the `.init` and `.fini` sections. Due to this similarity between some of the functionality needed by patching and the functionality offered by dynamic libraries, some previous systems have performed patching by creating patches as dynamic libraries which contain not only the code and data to be patched but the mechanism to perform the patching [4] [3]. We argue that this is an unnecessary mixing of data and logic, and that a patch which contains merely the information necessary to fix a running process and not the code to do so is more desirable. The code to apply the patch should live in one place on any given system, as most other executable content does. We do not embed emacs within our text files, after all. Dynamic libraries and other relocatable, linkable objects do not contain code and data intended to overwrite data in an existing executable or process. Consider, however, that redefining certain symbols is only a slight twist on ordinary linker behaviour. Ordinary linker behaviour for global symbols is to fail if a symbol is defined more than once. Ordinary behaviour for weak symbols is to use a global definition if available or the weak symbol otherwise. When performing dynamic linking, generally the first appropriate symbol encountered in the chain of symbol tables is used. It is not a far difference to define the linkage rule that the symbol definition from the most recent patch takes precedence. Therefore, applying a patch consists of the following steps

1. Injecting appropriate sections of the patch into memory. This includes putting their contents into memory and performing relocations on these sections (but not on the rest of the in-memory process) so that they fit into their environment

2. Copying existing data to the appropriate regions of the newly mapped-in patch

3. Performing relocation on the entire in-memory process such that the symbols defined by the patch take precedence.

These steps are all such fundamental operations that they should become universally supported by the ABI and the toolchain.

On the other hand, note the specification of a general patch format does not completely prescribe the patch application. From a standard patch format, a patcher is still free to make decisions such as when to patch safely and whether to patch functions by inserting trampolines in the old versions of the functions or by relocating all references to the function (we currently do the former in Katana, but may later transition to doing the latter).

3.2 Our Patch Object Format

Our Patch Object (PO) format is an ELF-based format. Figure 1 shows the sections contained in a simple patch. `.text.new` and `.rodata.new` are of course the new code and supporting constants to inject. `.rela.text.new` allows `.text.new` to be properly relocated after it is adjusted. While System V based systems use only relocation sections of type `SHT_REL`, we chose to use `SHT_RELA` in our patch objects because they make addends much easier to keep track of as we relocate from patched binary to patch objectp to patched process in memory. This is all really nothing new, storing ELF sections to be injected in-memory has been done before in other systems. What is new in our patch object is the inclusion of DWARF sections. The `.debug_info` section in an ordinary executable program contains a tree of DIEs (Debugging Information Entities) with information about every type, variable, and procedure in each compilation unit in the program. In a patch object, we store information only about the procedures and variables which have

```
Section Headers:
  [Nr] Name              Type
  [ 0]                   NULL
  [ 1] .strtab           STRTAB
  [ 2] .symtab           SYMTAB
  [ 3] .text.new         PROGBITS
  [ 4] .rodata.new       PROGBITS
  [ 5] .rela.text.new    RELA
  [ 6] .debug_info       PROGBITS
  [ 7] .debug_abbrev     PROGBITS
  [ 8] .debug_frame      PROGBITS
  [ 9] .rel.debug_info   REL
  [10] .rel.debug_frame  REL
```

Figure 1: Headers for the PO

changed. This of course includes storing the type information for changed variables. An example of the DWARF DIE information contained in a patch can be seen in Figure 2.

Note that we store considerably less information about each entity than is typically contained. This is because we read most of the information from the DWARF and symbol table information of the executing process. This allows the patch to be more flexible as it does not require that all variables and procedures be located at exactly the addresses they were expected to be at when the patch was generated. This flexibility allows a single patch between versions *va* and *vb* to patch both an executable which was originally compiled to *va* and an executable which was patched from earlier versions to be equivalent to *va*. Ksplice, one of the few other patchers which operates solely at the binary level, does not have this capability [2]. Note that patch versioning is currently a work in progress and not fully implemented.

Most of the information in the DIE tree is concerned only with names or how to locate code within the patch object (high and low pc). Of special interest, however, is the `fde` attribute of the `DW_TAG_structure_type`. This attribute specifies an offset in the `.debug_frame` section of an FDE (Frame Description Entity). Dwarf FDEs are designed for use in transforming one call frame into the previous call frame, and thereby walking up a call stack for either debugging purposes or exception handling purposes (using the `.eh_frame` section). Transforming one call frame to another, however, is not such a different operation from transforming one structure to another version of the same structure. We have aided this use with an implementation of the DWARF virtual machine which defines several special register types (exploiting the fact that for the purposes of generality, DWARF registers are specified as LEB128 numbers, giving an unlimited number of registers). The DWARF register instructions contained in the FDE referenced in Figure 2 for copying `field1`,`field2`, and `field3` from the original version of a structure `_Foo` to a new version of `_Foo` which has gained the extra member `field_extra` in the middle of the existing fields would be represented as in Figure 3.

```
.debug_info

COMPILE_UNIT<header overall offset = 0>:
<0><   11>DW_TAG_compile_unit
    DW_AT_name                    main.c

LOCAL_SYMBOLS:
<1><   19>DW_TAG_subprogram
    DW_AT_name                    printThings
    DW_AT_low_pc                  0x0
    DW_AT_high_pc                 0x70
<1><   40>DW_TAG_structure_type
    DW_AT_name                    _Foo
    DW_AT_byte_size               16
    DW_AT_MIPS_fde                16
    DW_AT_sibling                 <93>
<2><   55>DW_TAG_member
    DW_AT_name                    field1
<2><   63>DW_TAG_member
    DW_AT_name                    field_extra
<2><   76>DW_TAG_member
    DW_AT_name                    field2
<2><   84>DW_TAG_member
    DW_AT_name                    field3
<1><   93>DW_TAG_base_type
    DW_AT_name                    int
    DW_AT_byte_size               4
<1><   99>DW_TAG_variable
    DW_AT_name                    bar
    DW_AT_type                    <40>
```

Figure 2: Dwarf DIEs in the PO

`CURR_TARG_NEW` and `CURR_TARG_OLD` are special symbolic values defined by the virtual machine. If we are patching the variable 'bar', then the `CURR_TARG_OLD` will be the old address of bar (it's value in the symbol table) and `CURR_TARG_NEW` will be the new address bar is being relocated to. Our registers take advantage of the LEB128 encoding to hold a considerable amount of information in the register identifier. In the case seen above, the first byte identifies the class of the register (CURR_TARG_NEW or CURR_TARG_OLD in this example), the following word specifies the size of the storage addressed (this is included so that register assignments may copy an arbitrary number of bytes), and the final word specifies an offset from the address referred to by CURR_TARG_(NEW—OLD).

the following is an addition to the related work section

4 Related Work

There are several hot-patching systems preceding Katana. One of the most well known is probably Ginseng [4]. Ginseng (and systems drawing inspiration from it such as Polus [3]) have successfully demonstrated patching of such important software as apache and sshd. These systems perform analysis of the differences between the original and patched versions at the source code level. This introduces considerable (and we argue unnecessary) complexity and inability to deal well with some optimizations such as inlining and hand-written assembly. The complexity of analyzing the source code ties these systems to generally a single language (C in the case of both Ginseng and Polus). By contrast, Katana is language agnostic as it works at the level of the binary ABI, and although we have not yet demonstrated it doing so, it should be able to patch binaries compiled from any language, providing the necessary symbol and relocation information is supplied. Ginseng also requires significant programmer interaction in annotating the code [1] and requires compiling the code to use type-wrappers, allowing the patching of data types but at the cost of indirect access to them. The more programmer effort involved in generating a patch, the more likely the patch is to be incomplete or incorrect.

Motivated by many of the points in the above paragraph, the successful Ksplice system [2] patches at the binary level, as we do. We claim the following differences from and improvements over Ksplice.

- Ksplice operates on the kernel. As their paper states, most of their technique is not specific to the kernel, but there is no evidence that it has been implemented to function on userland programs. Katana operates on userland.
- Ksplice makes no attempt to patch data, relying entirely on programmer-written transformation functions when data types do change
- Ksplice patches are created as kernel modules. Ksplice does not provide a mechanism to perform operations, such as composition, on these patches.

To the best of our knowledge, Katana is the first system to utilize DWARF type information in patching.

```
DW_CFA_register {CURR_TARG_NEW,0x4 bytes,0x0 off} {CURR_TARG_OLD,0x4 bytes,0x0 off}
DW_CFA_register {CURR_TARG_NEW,0x4 bytes,0x8 off} {CURR_TARG_OLD,0x4 bytes,0x4 off}
DW_CFA_register {CURR_TARG_NEW,0x4 bytes,0xc off} {CURR_TARG_OLD,0x4 bytes,0x8 off}
```

Figure 3: FDE instructions for data patching

References need to be merged with the original paper, at least the Ksplice one is already cited by the original paper

References

[1] Ginseng user's guide. `http://www.cs.umd.edu/projects/PL/dsu/software.shtml`. Contained in source distribution from the web page.

[2] Arnold, J., and Kaashoek, M. F. Ksplice: automatic rebootless kernel updates. In *EuroSys '09: Proceedings of the 4th ACM European conference on Computer systems* (New York, NY, USA, 2009), ACM, pp. 187–198.

[3] Chen, H., Yu, J., Chen, R., Zang, B., and Yew, P.-C. Polus: A powerful live updating system. In *ICSE '07: Proceedings of the 29th international conference on Software Engineering* (Washington, DC, USA, 2007), IEEE Computer Society, pp. 271–281.

[4] Neamtiu, I., Hicks, M., and Stoyle, G. Practical dynamic software updating for c. In *Proceedings of the ACM Conference on Programming Languages Design and Implementation* (2006), pp. 72–83.