Recompiling the metamorphism

VX Heaven

herm1t
Valhalla #2
March 2012

The Shadow of the Past

Unlike my previous articles, this one has no single line of code. I would like to briefly discuss why neither metamorphism, nor other code mutation techniques doesn't lead to the essential changes in the "game". Antiviruses has been learnt to detect the RPME, Mistafall, Lexotan and a few others experimental viruses rather quickly. I want to share a few thoughts on how to fix the situation.

I have tried to write this article for the several times and may be this time it would turn out something good. I began to dimly understand how the virus of my dreams will look like many years ago, but I felt the obstacles on the way to this "super-engine" as almost insurmountable. I have discussed the metamorphism with some people since then, and some of them already tried to implement something of the kind, while others had a purely theoretical interest in the topic. By the way, I am glad to catch an opportunity and want to thank wK and SPTH for their thoughts and comments. I am not ready to implement it as well, but it's not ruled out that this notes might be of some use and may be even urge one to an action.

Before going to any details, I wish to say some words on words. The discontentedly misleading ones. As with any new knowledge (the virology is still showing its infancy), our brave pioneers were lavishly poured words and gave names without thinking about how good those words describing the subject. I have nothing against it, but if you would slightly change the verbiage, you could look on the subject from the different angle.

Real Programmers Don't Use Asm

In his classical article The Mental Driller said: "If you have realized, the pseudoassembler is quite general, ..." [1], But, no, that wan't do! Excuse me, but why the hell this should be assembler? In the programming the assembly considered the bad taste, because it's hard to

support the assembly code and give the questionable advantages. Until one would not exhaust all other possibilities, the assembly is a kind of preliminary opimization in its pure form, not to speak about the complications it could cause to the author and his engine.

Metamorphic virus consists of the several parts: it gains control, then it tries to disassemble its own code, "shrink" (or *optimize*) it, then the usual doubts about searching for a victim and changing it, "expand" (or *obfuscate*) the code and finally writing it to a file. Not too bad, but the virus performs both of the most significant phases of re-compilation in such a way, like it was trying to build the ship in the bottle, or in the compiler's language: through a *peep hole* (modifying a few instructions at a time), leaving out of account both the structure and semantic of the code.

It's the inherent defect of the low-level programming. The "fact" that "real coderz use asm" unintentionally compels a developer's attention on machine instructions, registers, code size, and other irrelevant matters. Fancy that! One could carefully write an assembly code, count bytes, just to see how the metamorphic engine will turn it into the pile of unreadable thrash. Makes a perfect sence.

It would be better to use high level language instead of assembly. By "high level" I mean a possibility to (automatically) extract additional information from the code (*not what it do, but what it mean*) rather than builtin support for the features like hashes, iterators or objects. The special feature of a smart software is its ability to maintain interconnections and meaning of the code and data rather than shuffling it like a deck.

The Giant's Shoulders

Like many other remarkable ideas, the concept of *polymorphic compiler* emerged long ago. I have found one of the first occurences in the Risks digest:

> I think that the polymorphic compiler approach is still stronger than that of the existing polymorphic engines. It also has the advantage that it will work fine in operating systems and environents that don't like self-modifying code. The compiler could even re-compile itself, so that there wouldn't be one speck of invariant code. Although many of the simple approaches could waste a greal deal of space by adding randomized jumps, etc, an efficient polymorphic compiler could probably work about as well as a poorly-optimized conventional compiler, if not better. Register allocations could be randomized, code could be spaghettied and one could store different implementations for primitive operations. [2]

Bontchev (who took part in the discussion) didn't recognized the possible consequences of Paul Houle's idea, but such compiler actually appeared at the very end of the MS-DOS era. That compiler is Amazing Code Generator [3] (you could read the decription in the Virus Bulletin [4]). The similar thoughts was expressed by Z0mbie in the early zeroes. Recent addition to this topic is Kaze's KPASM.

If it's so good why it's not widespread? John Aycock proposed the criteria which would help to distinguish the ground-breaking technology [5]. To be a game changer the technology has to change the defensive measures essentially and it must shift the motivation and business model of an adversary. I am sure that poly- and metamorphic compilers will met these conditions. Among other reasons, the black market could provide a stimulus too. Today's malware is heavily dependent on cryptors. To keep the cryptor in actual form is expensive and time consuming process, based on the large amount of the routine work. The compiler which is able to produce large amount of complex and variable code could supplant the cryptors and cause a lot of head ache to anti-virus vendors. That's good, because they should replace their signature-based idiocy with something clever. It's time to scramble out of the cozy hole both for VXers and AVers.

The first encrypted viruses written by Mark Washburn were detected, but the latter polymorphic viruses wreck a havoc amongst AVers, the Morris Worm was a first swallow, but the mass worms appered only in the late nineties. Metamorphism, is the recent example of the same cycle, being implemented on the qualitatively new level it would allow us to move on.

Two-faced undecidability: Why self-optimization is not (and never be) a problem

While discussing the virus-compiler issues with my peers, they had often raised the concerns that a decompiler of a virus engine could be used directly by anti-virus to produce the clean virus sample. “Qué carajo”, I said. I could back this with such thing as *full employment theorem*. It's impossible to write a perfect optimizing compiler. Suppose that such compiler exists. It could "look" at the hanging program and optimize it down to the single dead-loop statement. Or converge two different algorithm's implementations to a single most optimal one (thus *prooving* their identity). Both tasks boils down to the *halting problem*, and thus (according to Church-Turing thesis) undecidable. So it is always possible to improve the compiler, or spam filter, or an anti-virus and their developers would not be out of work.

The word "undecidable" in this context doesn't mean that there is no solution (if it being so there were no compilers at all), but that it's not known is there any solution and how much time does it take to get it. This uncertainty has the diametrically opposite consequences for viruses and anti-viruses. In order to detect virus the anti-virus requires single (or a limited number) of forms of the virus. On the contrary the virus would require *any* possible form with only one requirement - the output's *average* size should be constant (to prevent the virus from endless growing in size).

One of the recent research [6] called such basic form (or pattern) the "zero form", and the process of obtaining such form is called "zeroing". The attentive reader could notice already that zeroing is undecidable. It could be proved in one step - if zeroing is able to reduce all possible forms of the virus to a single "normalized" (not neccessary optimal) form it could do

it to any type of programs as well, thus proving or refuting their identity. That's known to be undecidable. QED. The world is really that bad - the virus which has 10% survival rate is a good one, the anti-virus with 90% detect rate is not. The glass is half empty for an AVers and half-full for us.

The sketch

My first intention was to use some ready compiler (as simple as possible) and I found the extreme example. [7] cc500 is not exactly what I want, but it's suitable to illustrate some concepts. It is simple, one-pass compiler which implements the small subset of C language and directly produces the x86 code. It is able to compile itself and the resulting executable is 16K long. I have tried to fix some limitations like hardcoded main() position and this was amazingly simple. When I changed the code generation routines to slightly improve the quality of the code:

```
<   emit(6, "\x81\xc4...."); /* add $(n * 4),%esp */
<   save_int(code + codepos - 4, n << 2);
---
>   if (n != 0) { /* avoid null stack adjustments */
>     if (n <= 5)
>       while (n != 0) {
>         emit(1, "\x5a"); /* pop edx */
>         n = n - 1;
>       }
>     else {
>       emit(6, "\x81\xc4...."); /* add $(n * 4),%esp */
>       save_int(code + codepos - 4, n << 2);
>     }
>   }
> }
>
> void load_reg(int v)
> {
>   if (v == 0) {
>     emit(2, "\x31\xc0"); /* xor eax, eax */
>   } else
>   if (v <= 127) {
>     emit(3, "\x6a\x00\x58"); /* push byte v / pop eax */
>     code[codepos - 2] = v;
>   } else {
>     emit(5, "\xb8...."); /* mov $x,%eax */
>     save_int(code + codepos - 4, v);
>   }
```

```
> }
>
> void lea_reg(int v)
> {
>   if (v == 0) {
>     emit(2, "\x89\xe0");
>   } else
>   if (v <= 127) {
>     emit(4, "\x8d\x44\x24\x00");
>     code[codepos - 1] = v;
>   } else {
>     emit(7, "\x8d\x84\x24...."); /* lea (n * 4)(%esp),%eax */
>     save_int(code + codepos - 4, v);
>   }
267,268c304
<     emit(7, "\x8d\x84\x24...."); /* lea (n * 4)(%esp),%eax */
<     save_int(code + codepos - 4, k);
---
>     lea_reg(k);
272,273c308
<     emit(7, "\x8d\x84\x24...."); /* lea (n * 4)(%esp),%eax */
<     save_int(code + codepos - 4, k);
---
>     lea_reg(k);
334a370
>
362,363c398
<     emit(5, "\xb8...."); /* mov $x,%eax */
<     save_int(code + codepos - 4, n);
---
>     load_reg(n);
377,378c412
<     emit(5, "\xb8...."); /* mov $x,%eax */
<     save_int(code + codepos - 4, token[1]);
---
>     load_reg(token[1]);
```

Not too much, right? But even this small modification lead to significant changes - size of code reduced by 2K and a lot of things changed through all the binary. Imagine how it would look like if I add some randomness, like this:

```
if (random() || abs(v) > 127)
  emit(5, "\xb8....");            /* mov $x,%eax */
  save_int(code + codepos - 4, v);
} else
if (random() || v != 0) {
  emit(3, "\x6a\x00\x58");        /* push byte v / pop eax */
  code[codepos - 2] = v;
} else
if (random()
  emit(2, "\x31\xc0");            /* xor eax, eax */
} else
  emit(2, "\x29\xc0");            /* sub eax, eax */
```

There are a lot of similar changes could be made. In addition, the registers could be selected randomly like in Vecna's RegSwap. And all this goes at nearly no cost, just as a *side effect*. By adding several hundreds of C lines one could reproduce the results of the best and most complex viruses out there. One don't need any texts on poly- and metamorphism and virus-related experience to achive this. MetaPHOR is 17K lines of assembly code, while a hypothetical virus based on cc500 would take 1-2K lines of straight and clean C and produce the same thing as a by-product.

Intermediate code considered harmful

It could be natural to move the lexing and parsing stuff offline and replace the direct code generation with some intermediate code translated into machine instructions or executed directly by a VM. But by doing this one would shrink the compiler down to a classical poly/metamorphic engine. I think that it would be better if a front-end will produce the abstract syntax tree of the virus, including the compiler itself (optimization, obfuscation and back-end parts). There are a lot of things that could be done besides simple instruction replacements. For example, the storage class of the variables could be changed from generation to generation, including classes that are not present in the language spec., like common blocks or "temporary globals". The temporary variable could be introduced at any time. The order of the execution could be randomized with very simple rule, like:

```
for each node (v) in tree
        if random() && is_commutative(v->op)
                swap(v->left, v_right)
```

Since, optimization part needs to know if two operations has dependencies on code or data, the same information could be used for the permutation. Being implemented correctly, all hell will let loose to a reverse engineer who would try to understand what the fucking virus is trying to do. All this will be only possible if an "engine" will "knew" what is the *meaning* of code. The more information is available, the more effective could be compiler.

21 ways to shoot yourself in the foot

It's still quite easy to break the whole thing. Especially when you trying to rely on features that are missing from the cross-compiler. Since cc500 has no separate text and data segments (the data is mixed with code) I tried to add the obvious optimization and replace two strings:

```
  if (accept("==")) {
    binary1(type);
    type = binary2(relational_expr(), 9, "\x5b\x39\xc3\x0f\x94\xc0\x0f\xb6\xc0");
  }
  else if (accept("!=")) {
    binary1(type);
    type = binary2(relational_expr(), 9, "\x5b\x39\xc3\x0f\x95\xc0\x0f\xb6\xc0");
```

with a single one:

```
 char *c = "\x5b\x39\xc3\x0f\x94\xc0\x0f\xb6\xc0";
 while (1) {
   if (accept("==")) {
     c[4] = 148; /* SETE */
   } else
   if (accept("!=")) {
     c[4] = 149; /* SETNE */
   }
   else return type;
   binary1(type);
   type = binary2(relational_expr(), 9, c);
```

But gcc put the string constant into .rodata and the first generation of the compiler segfaulted. In this very case I get off easily with #ifdef:

```
#ifdef __GNUC__
char c[]=
#else
char*c =
#endif

    "...
```

but in more complex cases it could lead to the painful process of bootstrapping the compiler. Suppose that you wrote a nice translator from the IR code (as Mental Driller suggested) into highly polymorphic machine code. All is fine except the compiler itself, you'll need to do the

same thing twice and rewrite your translator with byte code. That's why I prefer to use C (rather than some homebrew language) for which the compilers are available and there is no need in bootstrapping. Another trap is bytecode itself - many transformations are just not available at this level and you will remain bound by registers, instructions, memory locations and other low-level mess. While you should think about the program in terms of expressions, symbols and scopes. Think *what* it do rather than *how*.

The silver bullet

So, how this AV soul's resting bullet will look like? An "offline" front-end should produce the serialized AST of the program (it could be stored as a (list (list (or ('lisp 'scheme)))) or a notation polish reverse - that doesn't really matters). The virus is either keeping its source or *decompiling* it from the machine code. Personally, I prefer the latter option, but to properly implement the decompiler one most likely would need additional information (at least about non basic types which couldn't be recovered and propagated) attached to the code. After recovering its own source the virus should apply optimization (as much as it could) when random obfuscation. I think that it is essential that a virus could use the same rules and the same intermediate representations (like SSA) for both phases, when compile it into native code. I am sure that using the advantages of high-level languages and compiler's theory would fulfill the potential hidden in metamorphism.

References

1. The Mental Driller "Metamorphism in practice or "How I made MetaPHOR and what I've learnt"", 29a #6, February 2002
2. Paul Houle, topic "How to measure polymorphism", Risks, February 1993
3. Mad Daemon "Amazing Code Generator", December 1997
4. Adrian Marinescu "ACG in the Hole", Virus Bulletin, Jul 1999, pp.8-9
5. John Aycock "Stux in a rut: Why Stuxnet is boring", Virus Bulletin, September 2011, page 14-17
6. Arun Lakhotia, Moinuddin Mohammed "Imposing Order on Program Statements to Assist Anti-Virus Scanners", In Proceedings of Eleventh Working Conference on Reverse Engineering, Delft, The Netherlands, November 2004, pp. 161-170.
7. Edmund Evans "CC500: a tiny self-hosting C compiler", http://homepage.ntlworld.com/edmund.grimley-evans/cc500/, 2008