A Platform for Secure Static Binary Instrumentation

MAR 16TH, 2016

论文下载

摘要

近来很多软件安全防御中都用了程序插桩。相比于源码插桩，二进制插桩更容易使用和推广。
安全插桩有两个关键点：

- 能被用到所有应用程序中，包括各种系统和应用程序库中。
- 无法被绕过

DBI（动态二进制插桩）容易做到以上两点，但SBI（静态二进制插桩）相对较难。加上易用性，所以之前安全插桩都选用DBI。不过DBI通常开销较大。
文中开发了一个PSI，克服了DBI技术的缺点，并且拥有安全，鲁棒性及易用性的特点。

引言

在源码级别插桩更容易，由于保留有像类型这样的高层信息，也容易优化；二进制插桩则是应用的更广泛。
SBI离线处理程序，插桩前要获得程序完整的CFG。DBI是在程序已经载入内存后才处理。DBI通常在每个BBL第一次执行前才插桩，已经成为对COTS程序进行安全插桩的（首选？）

技术。主要是由于DBI技术的几个特点，这些特点在保证安全性的同时，也简化了插桩代码的开发：

- 插桩无法绕过：在每个控制流转移处都检查target是否被插桩，可以阻止试图逃离插桩安全检查的行为（跳转到数据段、指令中间、插桩代码中间）。
- 完备性：DBI能对所有程序和库进行插桩。
- 易用

以前的SBI不具有以上特点，并且通常只应用在特定方面，如CFI和SFI。

文中SBI同样也拥有DBI这些优点，并且保证开销较小，不需要在程序运行时提供一个虚拟环境。

主要贡献：

- 安全静态插桩：本文SBI具有两个安全插桩的关键特性，完备性和无法绕过性。
- 一个容易、通用的静态插桩平台：提供了易用的上层接口，抽象掉了底层的细节。
- 按需对库进行插桩：SBI要求所有库依赖关系在静态被确定，执行前被插桩。但有的程序是动态载入库函数的，文中提供了这种技术
- 性能不错
- 局限性：不支持混淆过的程序（控制流可能被打乱），不支持自修改的代码

背景

反汇编

静态地对变长指令集下strip过的程序进行反汇编是很难的，这也是为什么DBI比较受欢迎的原因，因为DBI返回值只在BBL级别，且在BBL第一次执行前才进行。

bincfi和binary stirring已经证明了这种情况下静态反汇编的可能性。将线性和递归反汇编结合起来，并且现代编译器越来越严谨，会尽量避免把数据插入代码段中

1. 通过静态分析的技术发掘代码指针，扩展了递归反汇编的范围
2. 插桩的开发中能够容忍反汇编出数据

解决间接转移

插桩会导致代码地址改变，静态计算的函数指针值不再正确。DBI通过运行时的间接控制流target地址转换解决这个问题，binCFI也用了类似的方法，PSI一样。一旦没有在转换表中找到某个地址，就意味着程序视图转移到非法区域，PSI将阻止这个行为。

用了binCFI里的GTT（Global Translation Table）和MTT（Module-specific Translation Table）来解决。

1. GTT，由修改过的Loader维护，用于将每个地址的高20bit翻译成到MTT的入口点
2. MTT，在PSI处理这个module时生成，使用剩余的bit在表中寻找对应的地址

系统概况

PSI可对程序及其所有用到的共享库插桩。输入binary，输出插桩后的Bianry。插桩可以在执行前，也可以在程序执行过程中。

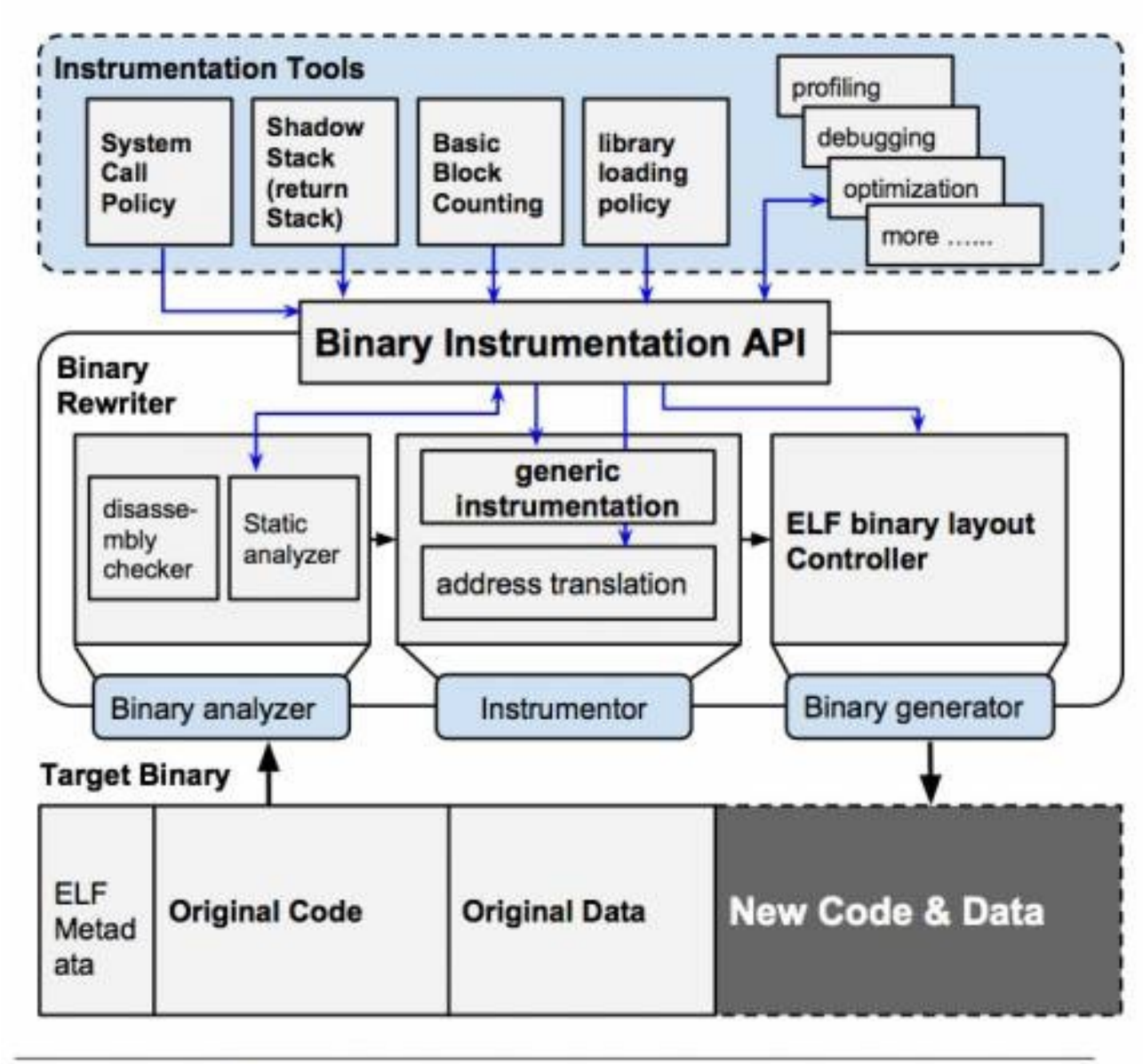

Figure 1. Architecture Overview of PSI

- binary分析器：输入binary，反汇编并输出CFG，CFG作为静态分析和插桩组件的输入
- 插桩器（核心）：将插桩代码编译成共享库（与pin类似），保证这些用高层插桩API插的桩，能够以一种安全的方式被调用
    - 底层指令级插桩：直接插入汇编片段
    - 高层函数级插桩：插入高级语言编写的函数
- binary生成器

实际的插桩工作是在上面的Instrumentation tools里做的，需要你自己用他的API去开发工具（类似pintool）

无法绕过的插桩

PSI通过实施以下属性保证添加的插桩无法被绕过：

- 原来代码中的所有直接和间接控制流转移的target，必须是能够被反汇编器反汇编的代码地址
- 如果一个指令I前通过插桩被插入了新的指令，则原来所有跳转到I的转移指令，现在都跳到桩上的第一条指令
- 只有插了桩的代码能把控制流转移到包含了支持插桩函数的库里

程序运行时会对所有间接跳转、间接调用和返回做上述检查，保证以上属性。直接跳转会在离线生成插桩binary时做检查。

如果开了on-demand开关，则loader会在请求载入未插桩的库时对其进行插桩，否则直接拒绝载入。

程序添加了以下检查保证控制流无法“逃脱”插桩：

- 跳转到数据段：合法target表中只有被合法插桩的代码地址
- 跳转到未知或未被插桩的代码段：如果反汇编器没有识别一些代码片段，这些片段不过会插桩，PSI也会阻止执行这些片段
- 跳转到指令中间：ROP常用攻击手法
- 绕过插桩代码：如果一个指令前被插桩，则他就已经从原来的合法列表里被删掉，取而代之的是桩的第一条指令作为target
- 跳转到桩中间，或者直接访问插桩时才能使用的函数，借此来破坏完整性：会在分支上做检查，排除这些target

插桩API

- getCFG: 获取程序完整CFG
- getBBs: 获取BBL列表
- getInsns: 获取指令列表

这几个API还能迭代遍历。

基于Intel的xed2指令编码/解码库，还能检测指令，isCall, isRet, isTest, isSysCall, isMemRead, isMemWrite, getTarget, getSrc.

插入汇编片段

```
ins_snippet(target, location, snippet)
```

target指向指令或基本块，使用相应对象的引用或标签。location可以是BEFORE, AFTER, AFTER_CALL(call被转换成push和jmp两条指令，AFTER_CALL则插在push和jmp之间)。

```
replace_ins(target, new_snippet)
```

有些程序需要替换已有指令。

PSI还提供了一个私有TLS，在插入的汇编片段中可用这个区域来存储数据。这个TLS不是glibc的那个，而是由TS和GS这两个初始化为0的数组组成。数组大小可配置，通常为一个内存页面。汇编片段中直接用TS_n和GS_n来访问数组中的第n个元素。

插入对插桩函数的调用

插入汇编高效，但是复杂难写。用高层API简化了插桩，但相对低效。

通常调用共享库里的一个handler，不需要再关心很多底层细节（比如存储寄存器和flag寄存器，切换不同栈，解决函数中的符号问题，直接通过高层的数据结构Context来访问程序当前状态），这样简化了插桩任务，能用高级语言来实现Handler。

```
ins_call(target, location, name, args)
void handler(struct Context* c, ...)
```

name是字符串的函数名。Context包括所有寄存器值，栈等。

控制流地址转换

PSI会自动处理间接转移的target地址。PSI提供了一些控制ICF target的插桩API，开发者可以用来做更复杂的分析和监控ICF转移，限制target。

```
rm_indirect_target(src_addrs, target_addrs)   //strict
add_indirect_target(src_addrs, target_addrs)   //relax
```

src_addrs里给出需要限制的ICF指令的标签。如果为空，则对模块中所有的ICF转移指令做限制。

target_addrs也是一个标签列表，但可以包含一些特殊的标签，比如NONLOCAL（在rm里，会将所有的非本地地址从合法的target列表中删去）。

平台会跟踪每个源地址的所有可能target，对于所有共享同一个可能target集合的source addr集合，都会生成一个唯一的地址转换“跳板”。

运行时事件处理

```
register_pre_syscall_handler()
register_post_syscall_handler()
register_library_load_handler()
register_thread_start_handler()
register_thread_terminate_handler()
register_program_start_handler()
register_program_terminate_handler()
```

开发 Instrumentation tools

看不懂

On-demand插桩

运行时需要动态载入库函数的话，很难搞。PSI修改了loader，只要在配置文件里开起来运行时对载入的库进行插桩的选项，就能在载入时对库插桩，不过还需要在配置文件里指定tool code, client library, mapping file。接着Loader就会调用PSI来生成一个插桩后的库，然后载入这个库。
然后这个库存在一个磁盘缓存里，可以同时存多个版本.这个也能用在调用运行多个可执行文件的情况下。

插桩应用

BBL计数

```
unopt = "mov%eax, TS_0;
         lahf;
         incl TS_1;
         sahf;
         mov TS_0, %eax"
opt = "incl TS_1"
foreach bb in getBBs() {
  found = false
  foreach insn in bb {
    if isTest(insn) or isCmp(insn) {
      found = true
      ins_snippet(insn, BEFORE, opt)
      break
    }
  }
  if !found
    ins_snippet(bb, BEGIN, unopt)
}
```

Figure 2. An instrumentation tool for Basic Block Counting

opt是优化后的，这里放在test或cmp前面，就不会因为加法的原因对标志寄存器造成影响了，因为后面一条test或cmp肯定会重新修改标志寄存器。

系统调用

快速识别int x80和sysenter这样的系统调用，handler根据Context来确定系统调用参数。

库载入

shadow stack

```
/* shadow stack pointer is stored in TS_2 */
chk_init_shadowstk= "
            cmp  $0x0, TS_2;
            jnz  L001;
            call $alloc_stack;
        L001: ";
push_shadowstk = "
            mov  %eax, TS_0; mov %ebx, TS_1;
            subl $4, TS_2;
            mov  TS_2, %eax;
            mov  (%esp), %ebx; mov %ebx, (%eax)
            mov  TS_0, %eax; mov TS_1, %ebx;"
check_return(Context*ctxt) {
  shadow_sp = ctxt->TS[2]
  ret = getmem(ctxt->ESP)
  while !empty(shadow_sp)
    if (pop(shadow_sp) == ret) {
      ctxt->TS[2] = shadow_sp
      return
    }
  abort()
}
foreach insn in getInsns()
  if isCall(insn) {
    ins_snippet(insn, BEFORE, chk_init_shadowstk)
    ins_snippet(insn, BEFORE, push_shadowstk)
  }
  else if isRet(insn)
    ins_call(insn, AFTER_CALL, check_return)
```

Figure 3. Shadow Stack Defense

longjmp可能会导致两个栈的返回地址匹配失效，由于这部分栈帧已经从主栈中被Pop出去，解决办法是连续从shadow

stack里pop，直到两个栈当前返回地址匹配，如果直接到达了shadow stack的栈底，说明可能存在攻击，直接中止。
其他可能导致栈不匹配的情况还有 动态加载器的延迟绑定，C++异常处理，UNIX信号，System V的线程上下文切换（如setcontext, getcontext）。
解决办法是找到binary中导致这个异常的返回指令，这样的返回指令都会在加载器或Libc的一个特定routine里，修改对这些指令的插桩。