强制执行

朱一帆

概述

现有的二进制分析可以大致分为静态分析、动态分析和符号分析。静态分析直接分析可执行文件，而不执行它；动态分析通过执行主题二进制文件获取分析结果；符号（共线性）分析可以生成输入，以探索二进制的不同路径。这些不同类型的分析有其各自的优势和局限性

静态分析无法获得动态计算的库调用参数，当样本打包或混淆时，这些参数不可行。传统的动态分析可以获得参数，并且不受打包和混淆的影响，但是，它只能根据输入和环境来探索一些执行路径。不幸的是，输入通常不知道是否存在恶意软件。符号分析虽然能够根据路径条件构造输入，但在处理复杂或压缩的二进制文件时存在困难。

现代恶意软件已经变得非常复杂，给二进制分析带来了许多新的挑战：
（1）对于零日二进制恶意软件，我们通常对其一无所知，尤其是对其输入的性质，这使得传统的基于执行的分析变得困难；
（2） 恶意软件二进制文件越来越多地配备反分析逻辑，因此即使给定有效输入，也可能拒绝运行；
（3） 恶意软件漏洞可能包含多阶段、条件保护和环境特定的恶意有效载荷，这使得很难显示所有有效载荷，即使有人设法执行它们。
近年来，符号分析和共形分析取得了很大进展。一些可执行程序，可以在二进制文件中探索各种路径。然而，将它们扩展到复杂的真实二进制文件时存在困难，因为它们通过将单个指令建模为符号约束并使用SMT/SAT解算器来解决生成的约束进行操作。尽管最近取得了令人印象深刻的进展，但SMT/SAT仍然很昂贵。虽然符号和具体执行可以同时执行，以便在符号分析遇到困难时具体执行可能会有所帮助，但用户需要提供具体输入，称为种子输入，种子输入的质量对可以探索的执行路径至关重要。由于没有或很少了解恶意软件输入，很难创建此类种子输入。此外，许多现有技术无法处理模糊或自修改的二进制文件

X-Force

恶意软件需要一些特定的环境才能触发恶意功能，以及恶意软件自带一些抗分析功能等。强制执行（Forced Execution）是解决这些困难的一种有效的方法。强制执行通过把程序可执行文件中的条件跳转指令强制取taken/not taken分支的方式，使得程序控制流在不保证数据流正确的情况下强行经过某条路径，并按照路径深度优先的方式，不断探索路径上的分支节点，从而尽可能多地遍历整个程序的控制流，使得一些需要严苛条件才能触发的代码能够被执行到，从而暴露出程序的恶意行为。

实用执行引擎X-Force：背后的核心启用技术是强制执行。强制任意二进制文件沿不同路径执行，而无需任何输入或环境设置。更具体地说，X-Force通过动态二进制检测监控二进制的执行，系统地强制执行可能影响执行路径的一小部分指令（eg：谓词和跳转表访问）来得到特定值，在需要输入时提供随机值。因此，可以系统地研究二进制的具体程序状态。

X-Force- Motivation Example

```
1   void main () {
2       int x=inputInt(...);
3       if (C(x))
4           p=(DNSentry*) malloc(...);
5       if (x & CODE_RED) {
6           genName(x,p);
7           table_put(x, p);
8       }
9       ...
10      table_put(..., o); /*o is of type T*/
11      ...
12      s=table_get(y);  /* y==x through execution */
13      if (s)
14          /*redirection for the domain specified by s*/
    }
```

```
41  mov [0x8004c0], [esp]
42  call ...
```

```
46  mov [0x8004c0], ecx
47  push ecx
48  call ...
```

```
20  void genName(int x, DNSentry * q) {
21          inputDictionary();
22          *(q->name) =... Lookup(x,date())...;
23  }
24
25  void * table_get(int key) {
26      .../* i is derived from key*/
27      if (key==bucket[i])
28          return bucket[i+4];
29  }
30  void table_put(int key, void* value) {
31      ... /* i is derived from key*/
32      bucket[i]=key;
33      bucket[i+4]=value;
    }
```

```
50  mov eax, [edi]
```

```
55  mov [eax+4], eax
56  ret
```

```
58  mov ecx, [ebx+4]
```

X-Force-强制执行语义

本节介绍单个强制执行如何进行。目标是实现一个不会崩溃的执行（通过按需分配存储器）。先讲检测内存错误并从中恢复，然后在后面的章节中逐步介绍强制执行的其他方面，如路径探索和处理库和线程

当我们用分配的内存替换指向无效地址a的指针时，我们需要更新所有变量具有相同的地址值或表示地址偏移量的值的其他指针变量。

我们通过线性集跟踪语义实现了这一点，这也是强制执行的基本语义。其目标是识别变量集（即二进制级别的内存位置和寄存器），其值具有线性相关性。

X-Force-强制执行语义

Source Code Trace	Binary Code Trace	Linear Set Computation and Memory Safety
6 genName(x,p);	1. ebx= 0x8004c0;	**SR(ebx)**→ {}
	2. eax= R(ebx); /* eax=0 */	**SR(eax)**, SM(0x8004c0) → {0x8004c0}
	3. W(esp, eax); /* esp=0xce0080 */	**SM(0xce0080)**, SR(eax), SM(0x8004c0)→ {0x8004c0, 0xce0080}
	4. …	
	5. call genName;	
/* in *genName*(... DNSentry * q) */	6. …	
22 *(q->name) =... Lookup(...)	7. ebx= ebp;	
	8. ebx= ebx + 8; /* ebx=0xce0080*/	
	9. edi= R(ebx);	**SR(edi)**, SM(0xce0080), SR(eax), SM(0x8004c0)→ {0x8004c0, 0xce0080}
	10. edi =edi + 4; /* edi= 0+4=4*/	**SR(edi)**, SM(0xce0080), SR(eax), SM(0x8004c0)→ {0x8004c0, 0xce0080}
	11. eax=…; /* eax=Lookup(...) */	SR(edi),SM(0xce0080),SM(0x8004c0)→ {0x8004c0, 0xce0080}
	12. W(edi, eax);	Exception! *0xce0080 = *0x8004c0 = malloc (4+BLOCKSIZE) = 0xd34780 edi= 0xd34780 + 4=0xd34784
7 table_put(x,p);	13. …	
	14. ebx= 0x8004c0;	
	15. ecx= R(ebx);	ecx= 0xd34780
	16. W(esp, ecx); /* esp=0xce0080 */	
	17. call table_put;	

```
1  void main () {
2     int x=inputInt(...);
3     if (C(x))
4        p=(DNSentry*) malloc(...);
5     if (x & CODE_RED) {
6        genName(x,p);
7        table_put(x, p);
8     }
9     ...
10    table_put(..., o); /*o is of type T*/
11    ...
12    s=table_get(y); /* y==x through execution */
13    if (s)
14       /*redirection for the domain specified by s*/
   }
```

```
41 mov [0x8004c0], [esp]
42 call ...
46 mov [0x8004c0], ecx
47 push ecx
48 call ...
```

```
20 void genName(int x, DNSentry * q) {
21       inputDictionary();
22       *(q->name) =... Lookup(x,date())...;
23 }
24
25 void * table_get(int key) {
26    .../* i is derived from key*/
27    if (key==bucket[i])
28       return bucket[i+4];
29 }
30 void table_put(int key, void* value) {
31    ... /* i is derived from key*/
32    bucket[i]=key;
33    bucket[i+4]=value;
   }
```

```
50 mov eax, [edi]
55 mov [eax+4], eax
56 ret
58 mov ecx, [ebx+4]
```

X-Force-路径探索

Algorithm 1 Path Exploration Algorithm

Output:	Ex - the set of executions (each denoted by a sequence of switched predicates) achieving a certain given goal (e.g. maximum coverage)
Definition	$switches$: the set of switched predicates in a forced execution, denoted by a sequence of integers. For example, $1 \cdot 3 \cdot 5$ means that the 1st, 3rd, and 5th predicates are switched $WL : \mathcal{P}(\overline{Int})$ - a set of forced executions, each denoted by a sequence of switched predicates $preds : \overline{Predicate \times boolean}$ - the sequence of executed predicates with their branch outcomes

1: $WL \leftarrow \{\text{nil}\}$
2: $Ex \leftarrow$ nil
3: **while** WL **do**
4: $switches \leftarrow WL$.pop()
5: $Ex \leftarrow Ex \cup switches$
6: Execute the program and switch branch outcomes according to $switches$, update fitness function $\mathcal{F}$
7: $preds \leftarrow$ the sequence of executed predicates
8: $t \leftarrow$ the last integer in $switches$
9: $preds \leftarrow$ remove the first t elements in $preds$
10: **for** each $(p,b) \in preds$ **do**
11: **if** $eval(\mathcal{F},p,b)$ **then**
12: update fitness function $\mathcal{F}$
13: $WL \leftarrow WL \cup switches \cdot t$
14: **end if**
15: $t \leftarrow t+1$
16: **end for**
17: **end while**

X-Force的一个重要功能是能够探索给定二进制文件的不同执行路径，以揭示其行为并获得完整的分析结果。

一个观察结果是，我们不必在低级实用方法中强制设置谓词，因为它们的分支结果通常不受任何输入的影响。因此，在X-Force中，我们使用污染分析来跟踪谓词是否与程序输入相关。X-Force只会强制分支这些受污染谓词的结果

X-Force实际挑战

Jump Tables：在我们前面的讨论中，我们假设控制权的转移仅通过简单的谓词进行。实际上，跳转表允许跳转指令具有两个以上的分支。跳转表被广泛使用。它们通常由源代码级别的Switch语句生成。在X-Force中，我们利用现有的jumptable反向工程技术，为每个间接跳转恢复跳转表。然后，我们的探索算法尝试探索表中所有可能的目标

处理循环和递归：由于X-Force可能会损坏变量，如果循环边界或循环索引损坏，则可能会导致（不正确的）无限循环。类似地，如果X-Force强制保护某个递归函数调用的终止的谓词，则可能会导致无限递归。为了处理无限循环，X-Force利用污点分析来确定是否根据输入来计算循环边界或循环索引。如果是这样，它将 循环绑定/索引值 重置为预定义常数。为了处理finite递归，X-Force不断监视调用堆栈。如果堆栈太深，X-Force会进一步检查调用堆栈中是否存在循环调用路径。如果检测到循环路径，X-Force将通过模拟“ret”指令跳过调用该函数

X-Force实际挑战

保护堆栈内存：由于X-Force可能会破坏影响堆栈访问的变量值，因此此类关键数据可能会被过度写入。因此，我们也需要保护堆栈内存。然而，我们不能简单地防止任何超出当前帧的堆栈写入。X-Force的策略是防止源于当前堆栈帧的任何堆栈写入超出当前帧。具体来说，当堆栈写入尝试重写返回地址时，将跳过写入。此外，该指令被标记。也将跳过指令中访问当前堆栈帧之外的堆栈地址的任何后续实例。当被叫方返回时，标志被清除。

处理库函数调用：X-Force的默认策略是，当我们对用户代码感兴趣时，避免在库调用中切换谓词：

I/O functions：X-Force跳过除文件输入之外的所有输出调用和大多数输入调用。X-Force为文件打开和文件读取提供包装器。如果要打开的文件不存在，X-Force将跳过调用，并返回一个特殊的文件处理程序。在读取文件时，如果文件处理程序具有特殊值，它将返回而不读取文件，因此输入缓冲区包含随机值。支持文件读取允许X-Force在所需文件可用的情况下避免不必要的故障恢复和路径探索。

Memory manipulation functions：为了支持内存安全，X-Force包装了内存分配和取消分配。对于memcpy（）和STRCPY（）等内存复制函数，X-Force包装器首先确定复制操作的有效性。如有必要，在调用实函数之前执行按需分配。这消除了这些函数中内存安全监视、线性集跟踪和内存错误恢复的需要，由于这些函数的特殊结构，这些功能可能会非常重量级。

X-Force实际挑战

对于静态链接的可执行文件，X-Force依靠IDA Pro在预处理步骤中识别库函数。IDA利用一个大的签名字典以非常高的精度重新识别库函数。对于IDA无法识别的函数，X-Force将其剪切为用户代码

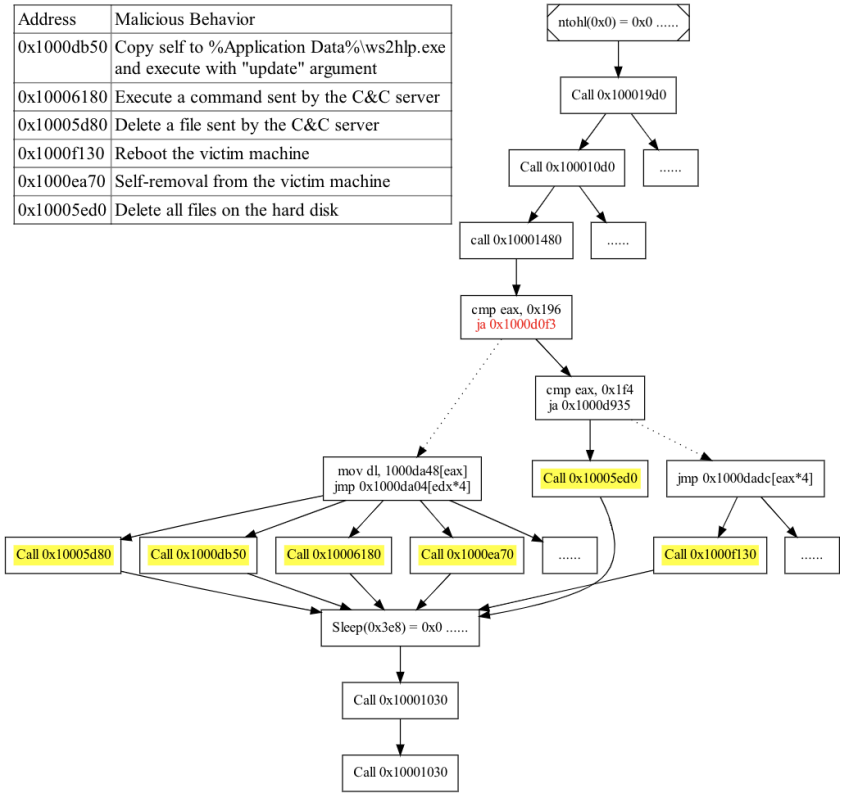
处理线程：有些程序在执行过程中会产生额外的线程。X-Force很难将多个线程建模为单个执行，因为它们的执行顺序是不确定的。为了解决这个问题，我们采用了一种简单而有效的串行化线程执行的方法。对线程创建库函数的调用被替换为对线程的启动函数的直接函数调用，这避免了创建多个线程，并保证了代码在同一时间的覆盖率。请注意，因此，X-Force无法分析对时间表敏感的行为。

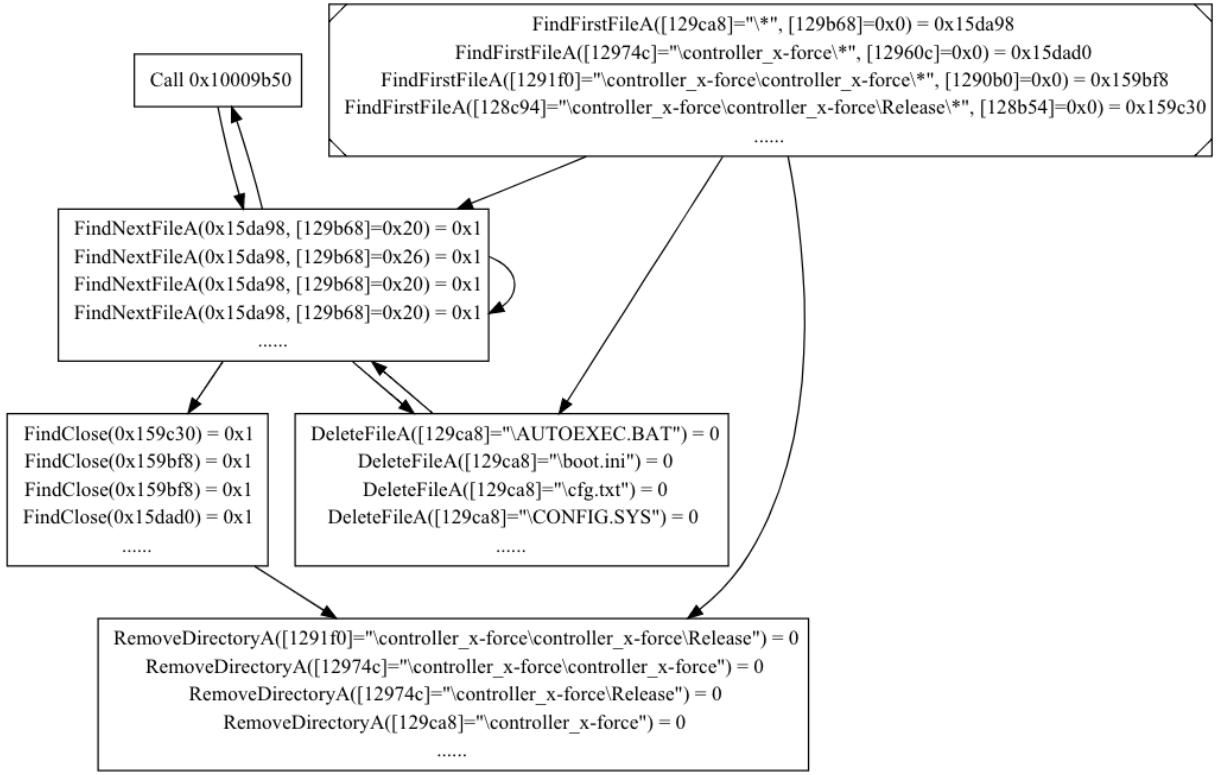
X-Force-恶意样本分析

传统的动态分析可以建立在X-Force上，以探索各种执行路径，而无需提供任何输入或环境
给定一个恶意软件样本，我们使用X-Force来探索路径。我们使用线性搜索算法，因为它在效率和覆盖率之间提供了良好的平衡。在每次执行期间，我们都会记录函数调用的跟踪。对于库调用，我们还记录参数值。然后将跟踪转换为一个内部流程图，该图将控制传入指令（包括跳转和调用）作为其节点，并将控制流/调用边作为其边。库调用的参数也在图上标注。在多个执行中生成的图被合并以生成最终图。然后，我们手动检查最终图形以了解恶意软件

X-Force-恶意样本分析

Address	Malicious Behavior
0x1000db50	Copy self to %Application Data%\ws2hlp.exe and execute with "update" argument
0x10006180	Execute a command sent by the C&C server
0x10005d80	Delete a file sent by the C&C server
0x1000f130	Reboot the victim machine
0x1000ea70	Self-removal from the victim machine
0x10005ed0	Delete all files on the hard disk





Dg003.exe：这是一个典型的APT恶意软件样本，具有多阶段、条件保护和环境特定的有效载荷。如左所示，图中每个突出显示的函数调用对应于先前未披露的恶意行为。每个行为都通过相应函数中的库调用进行标识。例右图所示，库调用和函数AT0x100009B50中的参数表明，它递归地枚举和删除从根目录开始的文件和目录，这表明它的行为是删除磁盘上的所有文件。

PMP-具有概率内存预规划的强制执行

一种重量轻、实用的强制执行技术。在不丢失分析精度的情况下，它避免了跟踪单个指令和按需定位。在我们的方案中，强制执行与本地执行非常相似。它具有一个新的内存预规划阶段，预先分配一个大内存缓冲区，然后初始化缓冲区和主体二进制中的变量，在实际执行之前以随机方式地给予值。

预规划的设计方式是，取消引用无效指针，它有很大的机会落入预分配区域，因此不会导致任何异常，而语义无关的无效指针解引用极有可能访问不相交（预分配）的内存区域，以概率保证避免状态损坏。我们的实验表明，PMP技术比X-Force快84倍，程序相关性检测的误报率和误报率分别减少6.5倍和10%，并且可以在最近的400个恶意软件样本中暴露98%以上的恶意行为

PMP

X-Force是一种非常重的技术，很难在实践中部署。具体而言，为了尊重程序语义，当X-Force修复无效指针变量时（通过为该变量分配新分配的内存块），它必须更新所有相关指针变量（例如，与原始无效指针具有常量的指针变量）。为此，它必须跟踪所有内存操作（检测无效访问）和所有移动/加法/减法操作（跟踪指针变量相关性/别名）。这种跟踪不仅会带来巨大的开销，而且由于指令集的复杂性和需要考虑的大量角点情况（例如，不计算指针关系），很难正确实现。

PMP提出了一种实用的强制执行技术。它不需要跟踪单个内存或算术指令。它也不需要按需内存分配。因此，强制执行与本机执行非常相似，自然处理库和动态生成的代码。具体而言，它通过新的内存预规划阶段实现无崩溃执行（具有概率保证），在该阶段，它从地址0开始预分配一个内存区域，并用精心编制的随机值填充该区域。

PMP-MOTIVATION

```
01 typedef struct{char ip[16]; long port;} Dest;
02 typedef struct{long act; Dest* dests[0];} Cmd;
03
04 int main(int argc, char *argv[]) {
05   Cmd *cmd = NULL;
06   int max = 0;
07
08   if (config_file_exists()) {
09     max = read_from_config_file();
10     cmd = malloc(sizeof(Cmd) + max*sizeof(Dest*));
11     for (int i = 0; i < max; i++)
12       cmd->dests[i] = malloc(sizeof(Dest));
13   }
14   ...
15   if (cnc_server_connectable()) {
16     scan_intranet_hosts(cmd, max);
17     cmd->act = get_action_from_cc_server();
18     switch (cmd->act) {
19       case 1: do_action_1(cmd->dest, max); break;
20       case 2: do_action_2(cmd->dest, max); break;
21       ...
22     }
23   }
24   ...
25 }
```

```
26 void scan_intranet_hosts(Cmd *cmd, int max) {
27   Dest **dests = cmd->dests;
28   for (int i = 0; i < max; i++) {
29     struct sockaddr_in *host = iterate_host();
30     inet_ntop(host->ip, dests[i]->ip);
31     dests[i]->port = ntohl(host->port);
32   }
33 }
```

α. mov rbx, [rbp - 0x10] // rbx = [rbp - 0x10] = [0x7ffdfffffed0] = 0x8
/* Validate Memory Address: get_accessible(0x7ffdfffffed0) = true */
/* Update Linear Set: SR(rbx) $\leftarrow$ SM(&dests) = {0x7ffdfffffed0} */

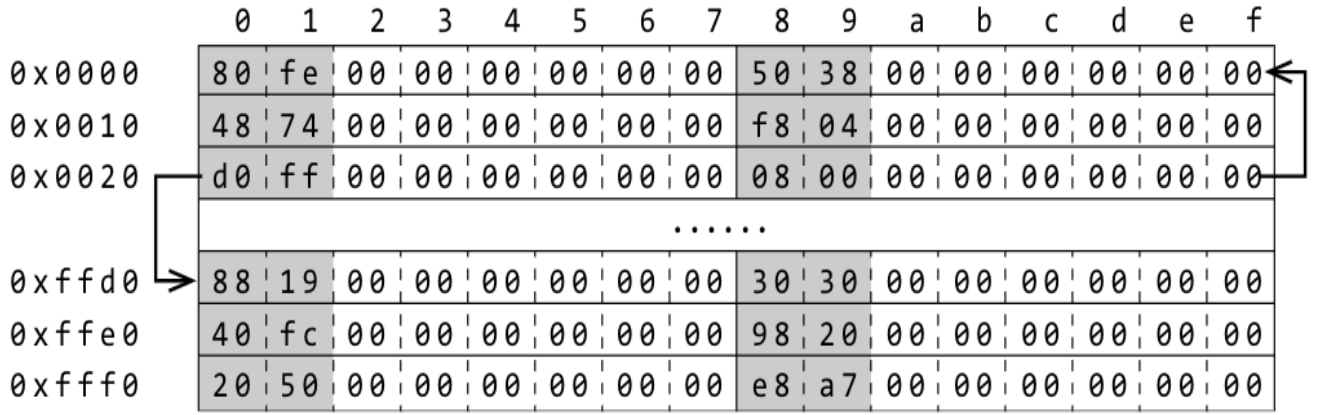
β. mov ecx, [rbp - 0x14] // ecx = [rbp - 0x14] = [0x7ffdfffffecc] = 0x0
/* Validate Memory Address: get_accessible(0x7ffdfffffed4) = true */
/* Update Linear Set: SR(rcx) $\leftarrow$ SM(&i) = {0x7ffdfffffecc} */

γ. lea rdx, [rbx + 8*rcx] // rdx = rbx + 8*rcx = 0x8
/* Update Linear Set: SR(rdx) $\leftarrow$ SR(rbx) = {0x7ffdfffffed0} */

δ. mov rax, [rdx] // rax = [rdx] = [0x8]
/* Validate Memory Address: get_accessible(0x8) = false (invalid read on 0x8) */
/* Allocate Memory Block: malloc(BLOCK_SIZE) = 0x2531000 */
/* Update Reference: rdx = *(0x7ffdfffffed0) = 0x2531000 + 0x8 = 0x2531008 */

ϵ. mov rax, [rax] // rax = [rax] = [0x0]
/* Validate Memory Address: get_accessible(0x0) = false (invalid read on 0x0) */
/* Allocate Memory Block: malloc(BLOCK_SIZE) = 0x2532000 */
/* Update Reference: rdx = *(0x7ffdfffffed0) = 0x2532000 + 0x8 = 0x2532008 */

PMP-MOTIVATION



自包含内存行为（SCMB）

自消歧记忆行为（SDMB）

上图显示了一个 64 KB 的预分配内存区域，映射在从 0x0 到 0xffff 的地址空间中。请注意，尽管此内存区域可能与某些保留的地址范围重叠，但我们利用 QEMU 的地址映射来避免这种重叠（。它充满了精心设计的随机值，以确保 SCMB 和 SDMB。对于我们的动机示例，指令 δ 读取地址 0x8（即 &dests[0]）处的内存单元并获取值 0x3850。随后，该指令使用 0x3850 作为访问 dests[0]->ip 的地址。这两个访问地址（0x8，0x3850）包含在 PAMA 中，因此不会发生内存异常。这两个地址之间的数据依赖关系也被忠实地暴露出来，没有不希望的地址冲突。观察到不需要内存验证和线性集跟踪。

PMP-DESIGN

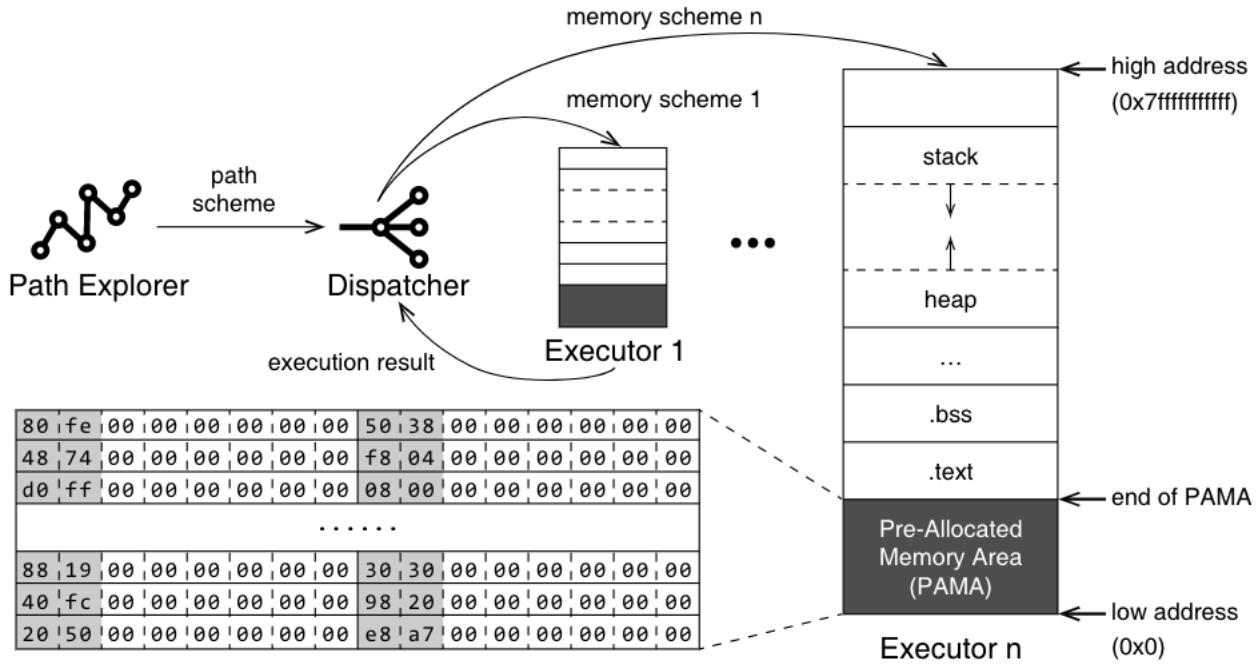
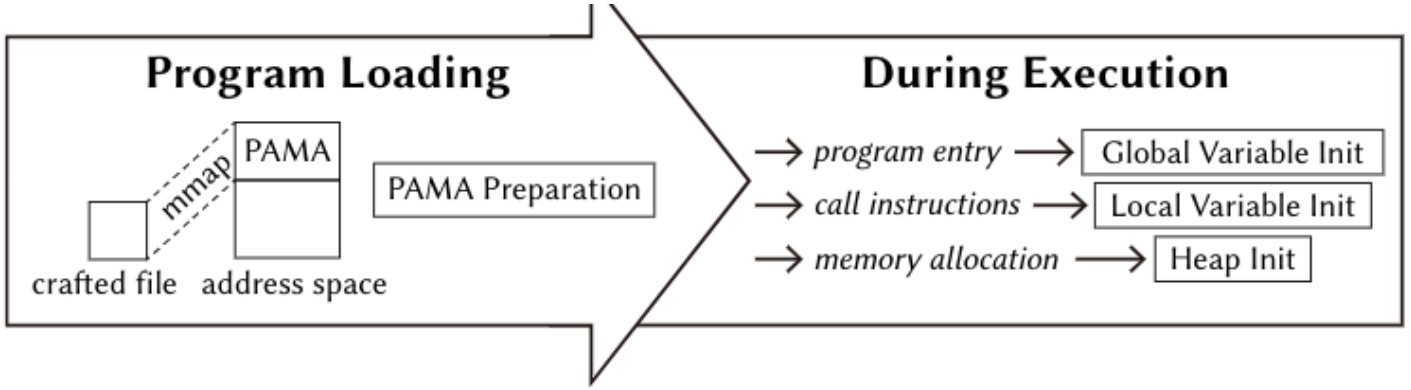


Fig. 3: Architecture of PMP.

PMP-DESIGN

北京大学 PEKING UNIVERSITY

内存预规划工作流程



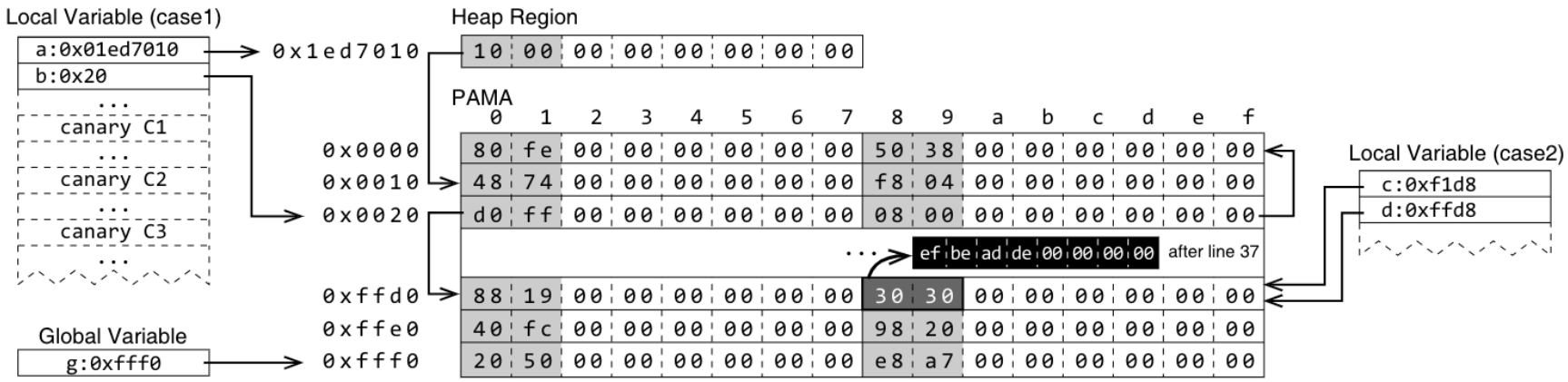
上图展示了内存预规划的工作流程。 加载程序时，会通过调用它们的映射系统调用来准备预分配内存区域 (PAMA)，以将精心制作的文件映射到程序地址空间。 文件内容是事先随机生成的。 在执行过程中，程序变量（包括全局变量、局部变量和堆区域）由 PMP 用指向 PAMA 的随机八倍值初始化。 具体来说，PMP截取： 1) 初始化全局变量的程序入口点； 2) 初始化局部变量的调用指令； 3) 用于初始化堆区域的内存分配。 请注意，PAMA 准备是先验发生的，并且产生的运行时开销可以忽略不计，而变量初始化是在执行期间即时发生的。

PMP-DESIGN

```
01 typedef struct{double *f1; long *f2;} T;
02 typedef struct{char f3; long *f4; long *f5;} G;
03 G *g;
04
05 void case3() {
06   long *e = NULL, *f = NULL;
07   if (cond1()) init(e, f);
08   if (cond2()) {
09     *e = 0x6038; // [0x0000] = 0x6038
10     long tmp = *f; // tmp = [0x0000]: bogus dep!
11   }
12 }
13
14 void case4() {
15   if (cond1()) init(g);
16   if (cond2()) {
17     *(g->f4) = 0x0830;
18     long tmp = *(g->f5); // &(g->f5) = 0x10000
19   }
20 }
```

```
21 void case1() {
22   long **a = malloc(...);
23   T *b;
24   if (cond1()) init(b);
25   if (cond2()) {
26     long *alias = b->f2;
27     *(b->f2) = **a; // [0x0008] = [0x0010]
28     *(b->f1) = 0.1; // [0xffd0] = 0.1
29     long tmp = *alias;
30   }
31 }
32
33 void case2() {
34   long *c; double **d;
35   if (cond1()) init(c, d);
36   if (cond2()) {
37     *c = 0xdeadbeef;  // [0xffd8] = 0xdeadbeef
38     double tmp = **d; // [0xdeadbeef]: error!
39   }
40 }
```

(a) code snippet.



(b) memory scheme.

PMP-Implementation

PMP 是基于 QEMU 用户模式仿真器实现的。 具体来说，PMP 使用条件跳转和间接跳转来执行路径方案。 路径方案是需要强制执行的一系列分支结果。 例如， “401a4c:T, 4094fc:F, 40a322#40a566”是一个包含三个要按顺序执行的分支结果的路径方案。 特别是 0x401a4c 和 0x4094fcs 的谓词应该分别取真分支和假分支，0x40a322 的跳转表应该取 0x40a566 的入口。目前 PMP 在 x8664 平台上支持 ELF 二进制。由于跨平台，它可以很容易扩展以支持其他架构 QEMU 的特点。

PMP比X-Force快，并产生长度比X-Force长的路径方案。路径方案越长，对代码的探索就越深入。

Thanks