1 设计目标

在与对抗软件漏洞对抗的过程中，软件的崩溃信息起着至关重要的作用。通常而言利用软件的崩溃信息定位到导致软件崩溃的原因有两种方法：记录-重放分析与核心转储文件分析。其中记录-重放分析通过详细地记录软件动态运行中的状态来定位崩溃原因。然而这种方法会使软件运行过程中产生巨大的额外开销而较少被采用，特别是大型软件，更难应用此方法进行分析。而另外一种思路是通过分析软件崩溃时产生的核心转储文件与动态运行时的控制流信息，根据所执行的指令语义恢复出软件运行过程中的状态，从而获得数据流信息，以定位到导致崩溃的漏洞。

核心转储文件分析方法是一种十分轻量级的分析方法，仅仅需要记录软件动态运行时的控制流信息便足以支持这种分析，适用于大量的软件崩溃报告的分析工作。核心转储文件是软件在崩溃时留下的信息，其中包括了崩溃时各个线程的上下文、进程的内存映射表等信息，在 Linux 平台下为 core dump 文件。当某个线程执行的指令序列已知时，便可以根据指令的语义恢复出该指令执行之前的状态。后向污点分析技术以软件运行过程中的中间状态为基础，从导致崩溃的值出发，找出异常数据的传递路径，即可据此对漏洞进行分析。

本篇报告使用基于 DynamoRIO 的软件动态二进制插桩技术记录软件动态运行时的控制流，在 ARM 体系结构下实现了对 AARCH64 指令的语义解析，结合后向污点分析技术跟踪数据流传递，实现了 Linux 平台下软件的二进制级自动化漏洞定位工具。在二进制级别进行分析，能够适用于无法获取源码的闭源软件，并且能对由不同语言编写的软件进行统一化的分析。同时，这种逆向的分析方式还能被运用于软件崩溃的分类中，以精确的自动化漏洞分类降低软件供应商对软件崩溃报告分析的工作量。

2 设计思路

一般而言，对程序崩溃原因进行分析的可能性来源于程序崩溃时留下的信息，通常是记录了崩溃时内存状态与每个线程的上下文等信息的核心转储文件。在手工进行正向调试时便可以根据这个状态信息结合代码语义进行漏洞定位。而该自动化工具便是模拟了这个过程，在二进制级别进行逆向数据流分析。

对崩溃程序的分析分为以下几个步骤：1. 对程序的控制流进行动态追踪；2. 提取分析程序崩溃时产生的转储文件，恢复出程序执行的所有指令；3. 依据指令语义进行逆向执行，恢复每一条指令执行前后所使用的内存或寄存器的值；4. 根据逆向执行恢复的结果进行后向污点分析，标记出导致程序崩溃的异常数据的传递路径。下面对以上步骤的设计思路一一进行解释。

2.1 动态二进制插桩

动态二进制插桩技术通常以基本块为单位对程序进行动态监控。在使用 DynamoRIO 进行动态插桩时，每当程序将要执行一个基本块中的代码时，便可以向其中插入 clean call，并在实际执行该基本块代码的过程中调用已插入的 clean call，以此实现对控制流的监控。

通常情形下仅对控制流进行监控时，可以仅在每个基本块的入口处插入 clean call，即仅记录基本块的跳转流程，在后续处理中根据基本块入口信息恢复出控制流。

如果需要作为实验验证的基准，也可以为每一条指令插入 clean call，以记录每条指令执行以前某些寄存器的值，并在后续与逆向执行恢复出来的值进行比较。

```
Basic block 起始
Clean call: 记录入口地址 0x0000ffff7f49e6f0
0x0000ffff7f49e6f0      0x90000250      adrp    x16, #0xffff7f4e6000
0x0000ffff7f49e6f4      0xf945f611      ldr     x17, [x16, #0xbe8]
可选 clean call: 记录此刻 x16 寄存器的值
0x0000ffff7f49e6f8      0x912fa210      add     x16, x16, #0xbe8
0x0000ffff7f49e6fc      0xd61f0220      br      x17
Basic block 结束


Basic block 起始
Clean call: 记录入口地址 0x0000ffff8324f080
0x0000ffff8324f080      0xd503245f      hint    #0x22
0x0000ffff8324f084      0x92402c04      and     x4, x0, #0xfff
0x0000ffff8324f088      0xf13f809f      cmp     x4, #0xfe0
0x0000ffff8324f08c      0x54000728      b.hi    #0xffff8324f170
Basic block 结束
```

图 1 动态二进制插桩示意

2.2 转储文件分析

在 Linux 平台下，软件崩溃产生的核心转储文件为 core dump 文件，为 ELF 格式的文件。通过分析该文件要得到的信息主要有：1. ELF 文件映射表； 2. 程序崩溃时内存空间各段的值与访问权限（读、写、执行）；3. 崩溃时每个线程的上下文信息。

ELF 文件映射表：为了与其他指令追踪方式（如 Intel PT 硬件追踪技术）兼容，该工具接受使用程序计数器（PC 寄存器）所指示的地址作为控制流的描述，并根据该地址从内存空间提取指令的机器码并进行反汇编。在内核开启 ASLR 的情况下，各个 ELF 文件在内存空间的填装的起始地址不同，因此无法直接通过程序计数器找到对应的指令。而 core dump 文件的 Note 段存储了各个文件的起始地址，因此可以根据程序计数器判断该指

令的来源，计算出指令地址的偏移量，并从相应的 ELF 文件中提取指令的机器码。

由于记录下的控制流信息不包含所有指令的地址，因此要根据基本块去分析并还原指令序列。一般而言，在处理基本块入口地址序列时，从入口地址开始到下一个控制流转移指令或下一个基本块的入口地址的指令均属于该基本块中的指令，即实际被执行的指令。

内存信息：core dump 里包含了崩溃时整个程序内存空间的值，具体的值将会在逆向执行的过程中用于恢复程序中间状态。每个段的访问权限将用于确定指令是否合法。

各个线程的上下文信息：逆向执行必须从崩溃时的状态出发进行解析，而 core dump 中记录的上下文信息便是分析的起点。对于具有多个线程的程序，需要从 core dump 里包含的多个线程的信息中确定哪个线程实际崩溃。对于一般的内存错误导致的崩溃，根据每个线程执行的最后一条指令及其访问内存段访问权限便可以判断出该指令是否合法，从而给出崩溃线程。该工具不支持多线程交互导致的内存错误分析（如竞态条件与数据竞争），因此将仅对崩溃的线程进行逆向执行的分析。

2.3 逆向执行

逆向执行指的是根据一条指令执行后的状态以及该指令的语义，恢复出指令执行前的状态。对崩溃程序进行逆向执行分析，在最理想的情况下即可以从最后一条指令执行后的状态开始逐条恢复出每一条指令执行前的状态。

2.3.1 可逆指令与不可逆指令

在逆向执行时，有一部分指令不会使执行前的状态信息丢失（如多数的算数指令），此时便可以根据指令的语义直接获得执行前的状态。如下图中显示的地址为 A3 处的指令，已知执行该指令后 sp 寄存器的值为 0x7e20，而该指令的效果是将 sp 的值加 0x10 再存储到 sp 中，因此可以推断出执行

该指令前 sp 的值为 0x7e10。

而地址为 A1 的指令将内存中起始地址为 sp + 0x20 且长度为 8bytes 的数据读入 x0 寄存器中，这一行为将导致指令执行前 x0 的值无法仅通过现在的状态以及该指令的语义恢复出来。此类指令即为不可逆指令。通过观察可知，A4 指令将 x0 寄存器的值设置成了 0，而此后在执行 A1 指令之前未改变 x0 的值，因此可推断出执行 A1 指令之前 x0 的值为 0。通常而言，不可逆指令的恢复需要借助其前后的指令及状态进行辅助。

```
long global = 0;


long child(long *a)
{
        a[0] = 1;
        a[1] = 2;
        return 0;
}


int main(void)
{
        long *globalp;
        long var;
        globalp = &global;
        child(&var);
        *globalp = *globalp + 1;
        return 0;
}
```

```
sub     sp, sp, #0x10
str     x0, [sp, #8]
ldr     x0, [sp, #8]
A10: movz    x1, #0x1
A9:  str     x1, [x0]
A8:  ldr     x0, [sp, #8]
       ; sp: 0x7e10 由 A3 指令获得该值
A7:  add     x0, x0, #8
A6:  movz    x1, #0x2
       ; x1: 2
A5:  str     x1, [x0]
       ; x0:?  x1: 2
A4:  movz    x0, #0
       ; x0: 0,  sp: 0x7e10
A3:  add     sp, sp, #0x10
A2:  ret
       ; x0: 0x0, sp: 0x7e20
A1:  ldr     x0, [sp, #0x20]
       ; x0: 0x2,  sp: 0x7e20 从 core dump 中获取的值
A0:  ldr     x0, [x0] ; 导致崩溃的指令
```

图 2 逆向执行示例

2.3.2 内存别名

在逐指令向前恢复过程中，如果遇到了不可逆指令，则可能出现某些寄存器的值未知的情况。如在分析 A4 指令时，便无法获知 A4 执行之前 x0 寄存器的值。此时再分析 A5 指令，由于 x0 的值未知，该指令访问的内存地址也未知，即无法判断该指令向哪个内存块写入了 0x2。如果我们想要得到 x0 的值，则需要向前找到对 x0 引用的指令，可以发现指令 A8 将起始地址为 sp + 0x8 且长度为 8bytes 的内存块写入 x0 中，且该内存地址

已知，为 0x7e18。

由于 x0 的值未知，因此在分析过程需要考虑两种情形：

寄存器 x0 的值等于 sp + 0x8：此时可以确定 A5 与 A8 两条指令访问了相同的内存区域。而由于 A5 指令对该内存区域进行了写操作，改变了其中的值，因此 core dump 里存储的这一区域的值将不再与 A8 指令执行时该区域的值相同，故无法通过 core dump 中的值恢复 x0 的值。由于假设 x0 = sp + 0x8 = 0x7e18，可根据此关系继续进行逆向执行的恢复。这种两个符号（[x0] 和 [sp + 0x8]）表示相同地址（0x7e18）的情况称为内存别名。

寄存器 x0 的值不等于 sp + 0x8：此时可以认为 A5 和 A8 两条指令访问了不同的内存区域，由此可知 x0 的值为起始地址为 sp + 0x8 且长度为 8bytes 的内存块的值，并且可以从 core dump 中存储的内存状态将 x0 的值恢复出来，同样可以继续进行逆向执行的恢复。

假设 [x0] 与 [sp + 0x8] 存在内存别名，由 A5 指令的语义可知，在 core dump 中 [x0] 处存储的值应当为 0x2，而这与实际情形不符，因此 x0 的值不可能等于 sp + 0x8，即可知该处不存在内存别名，由此可以计算出 x0 的值应当为 0x7e40。实际分析中，内存需要被细化成 byte 粒度进行分析，以顺利处理两个内存块交叠或包含等较为复杂的关系。

通过以上分析可知，当涉及到需要对两个内存区域进行比较的场景时，如果其中存在未知的内存操作地址，其相对关系的判断将影响恢复的结果。而由于程序的执行状态已经确定，因此仅存在一种正确情况，如果从错误的相对关系判断出发则会导致后续的恢复产生冲突。

2.3.3 Use-def 链

Use-def 链是一种数据关系的分析方法。使用这种方法对指令中的各种操作数进行解析并找到其中的相互关系，以解出逆向执行中各种数据的值以及数据的传递关系。

逆向执行主要关注的指令是会导致寄存器或内存的值被更改的指令，如算数指令、存取指令、移动指令等，而这些指令可以被抽象成对某些对象的 use 与 define 的组合。如 A5 指令，可以被解释成首先引用了 x1 寄存

器的值，再引用寄存器 x0 的值，最后对 x0 指向的长度为 8bytes 的内存块进行定义，即 [x0]＝x1。通过类似的方法可以将指令的执行序列构建成一个 use-def 链，把连续指令的 use 与 define 节点串联起来，以便对值与值之间的关系进行分析，如下图。

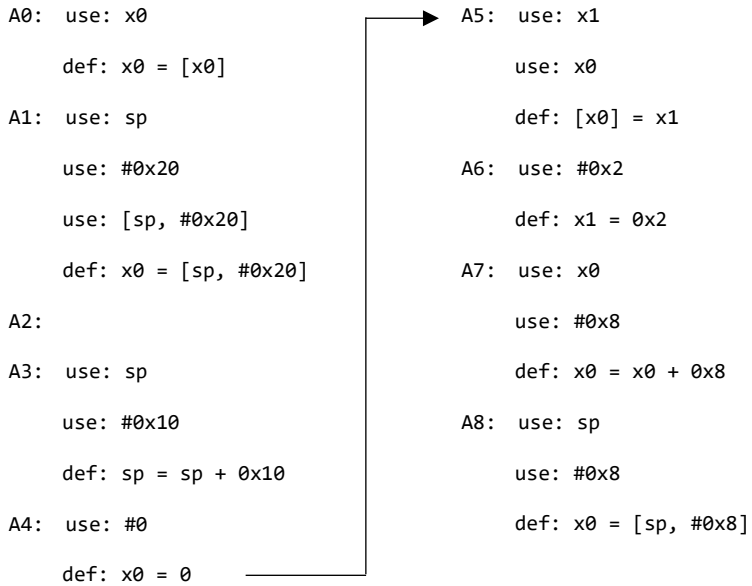

图 3 Use-def 链示意

Use-def 链中的每一个节点都代表对一个符号的引用或者赋值。只有 def 节点才会改变一个符号的值，因此同一个符号的 def 节点之间的任意节点中，该符号都具有相同的值。

如 A4 指令的 def: x0 = 0 节点对 x0 进行了定义，直到 A1 中的 def: x0 = [sp, #0x20] 都没有出现对 x0 的 def 节点，因此这两个 def 节点之间 x0 寄存器都具有值 0x0，由此可以推知 A1 指令执行前 x0 具有值 0x0。

在 A8 指令的 def 节点中，如果想要恢复出 [sp, #0x8] 所包含的值，则需要确认该值是否等于 core dump 里存储的值。向前查找对该内存区域的 def 节点，如果未找到，则表明从 A8 指令开始到程序崩溃时该内存区域的值未被改变，从而可以从 core dump 中直接取值。然而对于 A5 指令的 def 节点，其指向的内存地址未知，无法判断该处是否是对 [sp, #0x8] 的另

一个 def，因此其值无法继续向前传递，即内存别名问题导致的不确定性。

通过 use-def 链分析，可以清晰地将可以被确定的值在该链上向前或向后传递，尽可能的恢复出更多寄存器或内存的值。

2.4 后向污点分析

在完成了逆向执行的分析之后，利用已经恢复出的程序执行中间状态，便可以进行后向污点分析以确定异常数据的传递路径。后向污点分析将基于 use-def 链进行，以 use 或 def 节点为单位进行污点传播。

在图 3 所示的 use-def 链中，导致程序崩溃的节点是 A0 指令的 def: x0 = [x0] ，此时 x0 的值为 0x2，将导致程序访问无读取权限的内存段（0x2）并触发段错误，因此认为当前 x0 的值即为异常数据。

污点传播将在 def 节点之间进行。寻找上一条对 x0 进行 define 的节点可以发现，A1 中的 def: x0 = [sp, #0x20] 对 x0 的值进行了修改，因此污点将传播到该节点上。此时可认为异常数据来源于内存块 [sp, #0x20]。再寻找对该内存块进行 define 的节点，由于 A5 指令中 x0 的值已知且为 0x7e40，因此可确认 def: [x0] = x1 节点对 0x7e40 处的内存进行了修改，即污点也将传播到该节点。可以找到该指令对应于程序源代码中的 a[1] = 2;语句，该语句导致了栈上的数据（globalp 指针）被覆盖为异常的值，从而导致程序崩溃。即污点分析的结果中包含了导致程序崩溃的根本原因，通过该方法可快速定位到导致程序崩溃的原因。

3 实现

3.1 指令追踪工具

指令追踪工具基于 DynamoRIO 编写，作为 DynamoRIO 的一个 client 对程序进行动态插桩，记录所有执行的指令以及每条指令执行前相关寄存器的值。运行效果如下（示例程序为 latex2rtf，取自 CVE-2004-2167）：

```
zyl@hustsec-raspi1:~/toyexample/latex2rtf$ start_trace -t -- /usr/local/bin/latex2rtf ~/tested
_cve/CVE-2004-2167/poc.tex
Program counter, register information will be recorded, human-readable text can be generated.
DR client start.
Thread 1160066 start.
Thread 1160066 recieves signal 11
Thread 1160066 exit. 2767732 instructions have been executed.
/usr/local/bin/start_trace: line 10: 1160066 Segmentation fault      (core dumped) ${0%/*}/../
libexec/DynamoRIO-AArch64-Linux-8.0.18852/bin64/drrun $shrink_bb -c ${0%/*}/../lib/libinstrace
_aarch64.so $@
zyl@hustsec-raspi1:~/toyexample/latex2rtf$ l
core  trace_1160066_raw.out
zyl@hustsec-raspi1:~/toyexample/latex2rtf$ dump
dumpe2fs    dumpimage    dumpkeys    dump_trace
zyl@hustsec-raspi1:~/toyexample/latex2rtf$ dump_trace trace_1160066_raw.out
Target file infomation:
        Thread ID: 1160066
        Trace length: 2767732
        Trace type: 7
Start processing infomation of 2767732 instructions in file trace_1160066_raw.out
Program pointers information file trace_pc_1160066_bin.log has been created.
Registers used information file trace_reg_1160066_bin.log has been created.
Text information file trace_all_1160066_text.log has been created.
```

图 4 追踪工具输出截图

产生的文件有 core dump 文件以及保存了所有执行过指令地址的文件，以及用于验证逆向执行结果正确性的寄存器值信息。

```
0x0000ffffbc820090      0xa9400c02      ldp     x2, x3, [x0]
Total: 3     R0: 0x9090909090909090   R2: 0x0000ffffb8aefef0   R3: 0x0000ffffe09d5f58

0x0000ffffbc82008c      0x54000728      b.hi    #0xffffbc820170
Total: 0

0x0000ffffbc820088      0xf13f809f      cmp     x4, #0xfe0
Total: 2     R4: 0x0000000000000090   R31: 0x0000ffffe09d59b0

0x0000ffffbc820084      0x92402c04      and     x4, x0, #0xfff
Total: 2     R0: 0x9090909090909090   R4: 0x0000000000000001

0x0000ffffbc820080      0xd503245f      hint    #0x22
Total: 4     R2: 0x0000ffffb8aefef0   R3: 0x0000ffffe09d5f58   R9: 0x0000000000000000   R31: 0
x0000ffffe09d59b0

0x0000ffffb8a816fc      0xd61f0220      br      x17
Total: 1     R17: 0x0000ffffbc820080

0x0000ffffb8a816f8      0x912fa210      add     x16, x16, #0xbe8
Total: 1     R16: 0x0000ffffb8ac9000

0x0000ffffb8a816f4      0xf945f611      ldr     x17, [x16, #0xbe8]
Total: 2     R16: 0x0000ffffb8ac9000   R17: 0x0000ffffbc816e40

0x0000ffffb8a816f0      0x90000250      adrp    x16, #0xffffb8ac9000
Total: 1     R16: 0x0000ffffb8ac9d30
```

图 5 执行的指令序列信息

3.2 核心分析工具

分析工具接受的数据文件有 core dump 文件、崩溃程序所用的所有 ELF 文件、指令序列文件以及寄存器值信息文件。基于 libelf 对 ELF 文件进行处理，从中提取出寄存器、线程以及代码等信息；基于 capstone 反汇编引擎对执行指令序列进行语义解析并建立 use-def 链。为了通过指令语义建立 use-def 链，本工具还为大多数 AARCH64 常规指令实现了对应的解析函数，以从指令中提取 use 及 def 节点并通过指令语义逆向恢复出寄存器的值。

运行分析工具的输出结果为污点传播的路径：

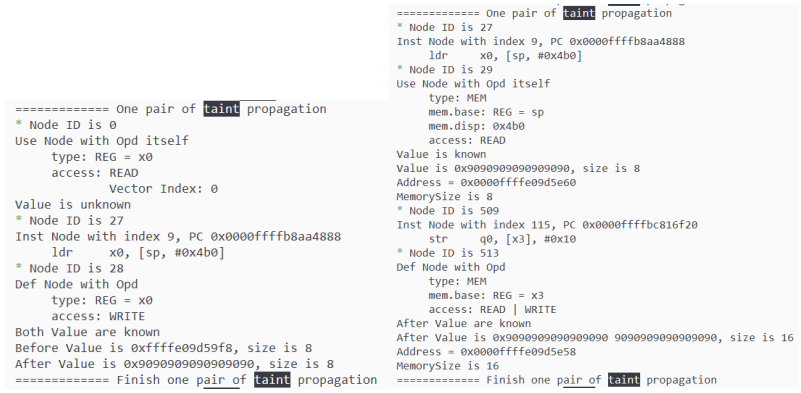

图 6 分析结果

结果显示了每一对节点的污点传递中所涉及到的指令地址以及传递的节点。通过该信息便可获取异常数据的传递流，以辅助漏洞的定位工作。

4 实验验证

目前已经使用该工具对数十个漏洞程序进行自动化分析，结果显示该工具在 ARM 体系结构下对空指针解引用以及栈溢出的漏洞具有较好的分析效果，能够通过反向污点分析覆盖到漏洞所在的代码，下表是一部分已经成功定位到漏洞的崩溃程序。

漏洞编号	软件名称	漏洞类型
CVE-2004-0597	libpng-1.2.5	Stack overflow
CVE-2004-1255	2fax-2.04	Stack overflow
CVE-2004-1257	abc2mtex1.6.1	Stack overflow
CVE-2004-1287	nasm-0.98.38	Stack overflow
CVE-2004-1288	o3read-0.0.3	Stack overflow
CVE-2004-1292	ringtonetools-2.22	Stack overflow
CVE-2004-2167	latex2rtf-1.9.15	Stack overflow
CVE-2005-3862	unalz-0.52	Stack overflow
CVE-2006-2362	binutils-2.15	Stack overflow
CVE-2006-2656	tiff-3.8.2	Stack overflow
CVE-2006-4182	clamav-0.88.2	Integer overflow
CVE-2007-1001	php-5.1.6	Integer overflow
CVE-2007-2683	mutt-1.4.2.2	Stack overflow

表 1 分析程序部分成功案例

5 总结

该工具是 POMP（Postmortem Program Analysis with Hardware-Enhanced Post-Crash Artifacts）在 ARM 体系结构下的移植，实现了较为高效地针对 ARM 体系结构的轻量级软件崩溃分析工具，适用于无源代码情形下的崩溃分析，且不受编程语言的限制。主要缺点是难以对多线程交互的程序进行分析，并且当栈中包含大量无用数据时，难以恢复出较多寄存器的值。