Evaluation

We evaluate SYSNAME with respect to its correctness and performance, and as well its effectiveness on real-world programs from IoT devices. First, we evaluate SYSNAME on 4 GNU core utilities programs from LAVA-M dataset to see if SYSNAME can indeed find bugs inside programs. Then, we use the same 4 programs to evaluate its performance by running both SYSNAME patched binaries and AFL compiled binaries for each of the program with a fixed amount of time. To minimize the randomness AFL introduces, each program was evaluated several times. Finally, we run SYSNAME on 3 IoT devices to see how effective binary-level instrumentation is working in a real-world scenario.

1. Correctness

In order to verify the correctness of our method, we must run SYSNAME on a dataset with ground-truth to see if our system can fuzz a program normally and find bugs. LAVA is a dataset widely used in fuzzing system test. It injects large number of bugs into real-world GNU core utilities programs automatically to generate buggy program dataset that could be used in Fuzzing test. Since the bugs are injected manually, we can determine the existence of bugs and therefore evaluate the correctness of the fuzzer.

The LAVA-M dataset in LAVA injects multiple bugs into 4 GNU core utilities, which are uniq, base64, who and md5sum. In other work related to fuzzer testing, we found that AFL does not perform well on LAVA-M dataset, it can only find certain number of bugs within program uniq and almost none within others. SYSNAME does not optimize any of AFL’s fuzzing process, we certainly don’t expect it to work better than AFL. Therefore, we decide to run AFL and SYSNAME only on program uniq. To avoid any environmental differences on the real devices, our experiments are set to run on Ubuntu 16.04 in the QEMU-ARM emulator, each host is configured with 2 cores and 4GB of memory.

We first compile the uniq program with afl-clang-fast, which comes with AFL, and then run AFL for about 20 hours on a single core. In order to eliminate the impact of AFL's random mutation, we repeat the experiment three times to get the following results. The first experiment triggered 6 unique crashes, and the third triggered 4. In other work (Angora), AFL found 9 unique crashes in uniq when the single core was running for 5 hours. Although the result in our experiment is not as good as theirs but noticed that we are running under emulation, which is much slower than a real x86 environment.

|  |  |  |  |
| --- | --- | --- | --- |
| No. | Unique Crashes | Bitmap Coverage | Total Paths |
| 1 | 6 | 0.42% | 75 |
| 2 | 0 | 0.42% | 80 |
| 3 | 4 | 0.42% | 77 |
| 4 | 0 | 0.42% | 78 |
| 5 | 4 | 0.42% | 83 |
| 6 | 4 | 0.42% | 71 |

On the other side, we use clang to compile uniq with the same parameters as afl-clang-fast and then instrument the target with SYSNAME. We test instrumented program for a total of 6 times, giving each fuzz instance about 10 hours of running time on a single core. In 6 sets of tests, we can find crashes every time, and up to 6 unique Crashes can be triggered. The results of the test, including bitmap coverage and total paths found, are basically the same as those of the AFL compiled program. This result proves the correctness of SYSNAME, which is quite the same as AFL with source code.

|  |  |  |  |
| --- | --- | --- | --- |
| No. | Unique Crashes | Bitmap Coverage | Total Paths |
| 1 | 6 | 0.44% | 79 |
| 2 | 1 | 0.44% | 76 |
| 3 | 1 | 0.44% | 74 |
| 4 | 4 | 0.44% | 82 |
| 5 | 3 | 0.44% | 76 |
| 6 | 1 | 0.44% | 78 |

2. Performance

In addition to the correctness, performance is also vital for fuzz. SYSNAME performs direct patching on binary programs, which lacks a view from the source code level, and thus unable to make optimization on the patched code. The program that is patched will suffer from performance downgrade then. In order to test the performance, we need a program which we have access to the source code, and then compare how two versions of the instrumented program perform. Here we again choose the LAVA-M dataset.

This time we test on all 4 programs, the evaluation environment is the same as the previous one, which is configured as dual-core CPU with 4GB memory under the QEMU-ARM emulator. The two sides of the comparison are the programs compiled from the source code by afl-clang-fast, and the programs compiled by clang with the same parameters and instrumented by SYSNAME. For each of the 4 programs, we have two different versions. To eliminate the randomness, we run each version of the program for 4 times and average the results of each run. Each running instance runs for 1 hour in a single core environment.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Program | Version | Average Bitmap Coverage | Coverage Less(%) | Average Total Paths | Paths Less(%) | Average Execution Speed(exec/s) | Slowdown(%) |
| base64 | AFL | 0.58% | 1.72 | 94.50 | 3.85 | 45.73 | 9.80 |
| Patch | 0.57% | 91.00 | 41.65 |
| md5sum | AFL | 0.40% | 0.00 | 13.25 | 6.00 | 52.55 | 13.1 |
| Patch | 0.40% | 12.50 | 46.46 |
| uniq | AFL | 0.42% | -4.76 | 70.25 | 5.24 | 51.07 | 7.51 |
| Patch | 0.44% | 66.75 | 47.50 |
| who | AFL | 3.69% | 0.00 | 40.25 | 5.23 | 38.02 | 9.05 |
| Patch | 3.69% | 38.25 | 34.86 |

We choose bitmap coverage, total number of paths, and execution speed as indicators of performance. As can be seen from the figure, the program after the patch and the program AFL compiled are almost consistent in terms of bitmap coverage. In the uniq test, the patch version even gets a higher bitmap coverage. It should be noted that because SYSNAME uses the disassembly tool to locate basic blocks within the program, the result obtained is different from the what clang gets. Therefore, the ”bits” in the bitmap are not exactly the same of these two, it can only be a rough estimate of the size.

In terms of the number of paths, the patched program has a certain degree of reduction compared to the native compiled program. The main reason here is that the patched program runs slower than the other one, resulting in a decrease in the number of instances that AFL can execute in a same amount of time. This will eventually affect how many inputs AFL can mutate and causing the number of paths to drop.

Among all 4 programs, the patched version has an average speed drop of about 10% compare to the other one. This performance degradation can be roughly attributed to the following three reasons:

1. As we are doing the instrumentation on the binary level, we do not have the corresponding context information, which makes us unable to optimize the stub we instrumented using the context. Therefore, we must save the used registers every time entering the instrumented stub. In contrast, AFL's source-based instrumentation can take full advantage of the compiler's optimization capabilities. In addition to avoiding frequent context saves, it can also use idle registers to store data such as pointers globally, thus yield a faster stub.

2. As stated in the implementation, we need to wrap the original instruction at the patch target address during the patch to ensure that the overwritten instruction can be executed normally in the patch code segment. This introduces pivot-related operations and some expensive memory access instructions, which will slow down the execution of the program comparing to the original simple instruction.

3. Finally, in order not to destroy the integrity of the original program, we place the patch code and data at the end of the entire program in memory. During execution, the program needs to jump to the patch code frequently to record the path, and since the original code segment of the program is far away from the patch code segment in memory, they cannot be loaded into the cache at the same time. This frequent back and forth jump can cause a large number of instruction cache misses, which further drags the execution speed of the program.

These 3 reasons mentioned are quite difficult to solve, but since SYSNAME patched program is only about 10% slower than the native compiled version, this is still acceptable.

3. Effectiveness

We evaluate SYSNAME on program with source code for correctness and performance, but the ultimate goal of SYSNAME is to test programs in COTS IoT devices without source code. Considering the popularity and the computational power the device has, we have picked 3 real-world router devices, which are ASUS AC1700, Xiaomi R1D and Netgear R7000, as our experiment targets.

We upgraded all 3 devices to the latest version of firmware, and then selected dozens of command-line programs and dozens of server programs as well from these devices. By manually analyzing the program, we can determine how to interact with these programs, including direct input from standard input, reading content from a file, and obtaining data from network. Programs that use Standard input or file reading are easy to patch, but we need the socket interception mechanism we mentioned to redirect the input for server programs. We analyzed all the programs, instrumented them, and finally started fuzzing on the original device. For each fuzz instance, we use a single-core CPU that runs no more than 24 hours.

As shown in the table, we found crashes in 10 programs. The first 7 are archive tools included in Busybox, and we found up to 30 unique crashes in these 7 programs. After analyzing these crashes, it was found that most of the bugs have been fixed in the latest version of Busybox. 1 crash from lzcat and 2 crashes of unlzma were just fixed a few months before the experiment, which also proves SYSNAME's ability to find actual bugs. In addition to the bugs in Busybox, we also found 12 unique crashes in the independent archive program unzip.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| No. | Program | Execution Time(H) | Bitmap Coverage | Total Paths | Execution Speed (exec/s) | Unique Crashes |
| 1 | bunzip2 | 0.34 | 0.56% | 148 | 348.88 | 8 |
| 2 | bzcat | 12 | 0.57% | 202 | 28.07 | 13 |
| 3 | gunzip | 0.34 | 0.71% | 98 | 289.21 | 11 |
| 4 | gzip | 12 | 0.85% | 255 | 337.07 | 30 |
| 5 | lzcat | 14.18 | 0.24% | 78 | 37.45 | 1 |
| 6 | unlzma | 12.12 | 0.25% | 79 | 44.08 | 4 |
| 7 | zcat | 12.28 | 0.81% | 224 | 132.81 | 30 |
| 8 | unzip | 12 | 2.23% | 626 | 349.43 | 12 |
| 9 | tsar | 0.17 | 0.46% | 27 | 55.26 | 8 |
| 10 | plugincenter | 7.58 | 2.06% | 179 | 15.76 | 17 |

The ninth program in the list is a third-party version of sar (System Activity Reporter) which called tsar, in which SYSNAME found 8 unique crashes. The last one, plugincenter, is a server program from Xiaomi. Note that because the server needs to do a series of initialization work when it gets started, which slows down the execution speed. SYSNAME is able to find 17 unique crashes in it.

The fuzzing result on the IoT devices shows SYSNAME's ability to find crash in COTS programs that without source code accessibility, which is limited by the computational power the actual device has but still effective and accurate.

1. Correctness

为了验证我们的方法的有效性，我们需要在一个ground-truth数据集上运行我们的系统，以观察系统是否可以在一般程序中正常的执行Fuzz，并且找到其中存在的Bug。LAVA是一个在测试Fuzz系统时经常被使用的数据集，其通过将大量的BUG自动化插入到真实的GNU core utilities程序中，来生成具备ground-truth的用于测试Fuzzer效果的BUGGY程序数据集。因为BUG都是人工插入的，所以我们可以确定BUG的存在。其中 LAVA-M数据集将大量BUG插入到4个GNU core utilities程序中，分别是uniq、base64、who、md5sum，我们就可以选用这个数据集来确定我们系统的正确性。

我们在这里没有在全部4个程序上测试我们的系统，因为在其他Fuzzer相关的工作中，我们发现AFL在LAVA-M数据集上的效果并不是非常出众，其只在uniq程序上找到了一定数量的BUG，而我们的系统并没有对AFL的Fuzz过程进行任何的优化，所以我们决定同样只在uniq程序上进行测试，以便于AFL的结果进行对比，以说明我们的正确性。

为了避免真实设备上环境差异对实验的影响，我们的实验环境设定为运行在QEMU-ARM模拟器中的Ubuntu 16.04，每个实验主机配置为2个核心及4GB内存。

我们首先用AFL自带的afl-clang-fast对uniq程序进行编译，然后在单核心的情况下运行AFL大约20小时来查看Fuzz的结果。因为AFL的mutation具备随机性，所以我们将实验重复三次，得到如下的运行结果。其中第1次实验触发了6个unique crashes，第3次触发了4个。在别的工作（Angora）中，AFL在单核心运行5个小时的情况下，曾在uniq中找到9个unique crashes，虽然我们试验中的AFL找到的crash数量不如他，但注意到我们在模拟器中的运行效率远远低于真实x86环境，所以这样的实验结果基本符合AFL实际的效果。

|  |  |  |  |
| --- | --- | --- | --- |
| No. | Unique Crashes | Bitmap Coverage | Total Paths |
| 1 | 6 | 0.42% | 75 |
| 2 | 0 | 0.42% | 80 |
| 3 | 4 | 0.42% | 77 |
| 4 | 0 | 0.42% | 78 |
| 5 | 4 | 0.42% | 83 |
| 6 | 4 | 0.42% | 71 |

另一边我们使用clang以跟afl-clang-fast相同的参数正常编译了uniq，然后使用我们的系统对目标进行插桩，同样在模拟器环境下单核运行AFL。同样为了避免随机性的影响，我们一共测试了6次，给每个实例大约10小时的运行时间，在6组测试中，我们每一次执行都可以找到crashes，最多的一次可以触发到6个unique crashes。这样的测试结果与AFL编译的程序相比基本保持一致，并且我们的系统所能发现的路径数以及Bitmap覆盖率都与之前的AFL编译的程序基本一致，这个结果证明了我们的系统在正确性上与AFL基于源代码的编译结果是相同的。

|  |  |  |  |
| --- | --- | --- | --- |
| No. | Unique Crashes | Bitmap Coverage | Total Paths |
| 1 | 6 | 0.44% | 79 |
| 2 | 1 | 0.44% | 76 |
| 3 | 1 | 0.44% | 74 |
| 4 | 4 | 0.44% | 82 |
| 5 | 3 | 0.44% | 76 |
| 6 | 1 | 0.44% | 78 |

2. Performance

除了系统的正确性外，性能表现对于Fuzz而言也是非常重要的。我们的系统针对二进制程序进行直接的Patch，这使得我们不具备源代码层面的视野，Patch出来的程序在性能上必定会有所损失。为了进行性能方面的测试，我们需要对可以访问到源代码的程序进行直接编译和编译后插桩的两个版本的运行比较，在这里我们还是选择了LAVA-M数据集作为测试样例。

这次我们在全部4个程序上进行了测试，运行环境同上一节相同，也是在QEMU-ARM模拟器下的双核4GB内存。测试对比的双方分别为由afl-clang-fast从源代码编译的程序，以及由clang用相同参数编译并由我们的系统插桩得到的程序。对于4个程序中的每一个，我们有两个不同的版本，为了消除随机性，每个版本的程序我们都会运行4次，并对运行的结果取平均值。每一个运行实例会在单核环境下持续运行1个小时，运行得到的结果如下表。

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Program | Version | Average Bitmap Coverage | Coverage Less(%) | Average Total Paths | Paths Less(%) | Average Execution Speed(exec/s) | Slowdown(%) |
| base64 | AFL | 0.58% | 1.72 | 94.50 | 3.85 | 45.73 | 9.80 |
| Patch | 0.57% | 91.00 | 41.65 |
| md5sum | AFL | 0.40% | 0.00 | 13.25 | 6.00 | 52.55 | 13.1 |
| Patch | 0.40% | 12.50 | 46.46 |
| uniq | AFL | 0.42% | -4.76 | 70.25 | 5.24 | 51.07 | 7.51 |
| Patch | 0.44% | 66.75 | 47.50 |
| who | AFL | 3.69% | 0.00 | 40.25 | 5.23 | 38.02 | 9.05 |
| Patch | 3.69% | 38.25 | 34.86 |

我们选用了Bitmap覆盖率，总共的路径数，以及执行速度三者作为性能比较的指标。可以从表中看出来，Patch之后的程序与AFL原生编译的程序在Bitmap覆盖率方面几乎保持一致，在uniq的测试中，Patch后的版本获得了更高的Bitmap覆盖率。需要说明的是，因为我们的系统通过反汇编工具来实现Basic Block的定位，其所获得的结果与clang直接编译时的Basic Block存在一定的差异，所以这里两者的Bitmap并不是一一对应的关系，只能是在总量上做一个大概的估算比较。

在路径数方面，Patch之后的程序相比原生编译的程序都有一定程度的减少，主要原因是Patch之后的程序运行速度有所下降，导致在相同时间内Fuzz所能执行的实例数有所下降，这最终会影响到Input的mutate次数，使得路径数出现下降。

最后看到的是Fuzz过程中的执行速度。在全部4个程序中，我们系统处理过的版本与AFL原生编译的版本有平均在10%左右的速度下降。这个性能下降大致可以归结为以下三个原因：

1、因为我们是在二进制层面做的插桩，所以我们不具备相应的上下文信息，这使得我们无法利用上下文做出优化，并且需要在每次进入到插桩代码时都要保存可能被用到寄存器等信息。而反观AFL基于源代码的插桩则可以充分利用编译器的优化能力，除了避免频繁的上下文现场保存之外，其还可以利用闲置的寄存器长时保存共享内存的指针等数据，以获得更快更短的桩代码。

2、如implementation一章中所说的，Patch时我们需要对Patch目标地址处原有的指令进行wrap，以保证被覆盖的指令可以在Patch代码段正常执行。这个相对于原先简单地一条指令，引入了新的pivot相关的操作以及一些昂贵的内存访问指令，这也拖慢了程序的执行速度。

3、最后，我们为了不破坏原有程序的完整性，将Patch代码和数据放在了整个程序在内存中的最后面。在程序开始执行之后，其需要频繁的跳转到Patch代码处来进行路径的记录，因为程序原有代码段与Patch代码段在内存中的距离比较远，使得其不能同时被加载到Cache中，这种频繁的前后跳动就有可能造成大量的Instruction cache miss，从而进一步拖累程序的执行速度。

这里提到的三个速度下降的原因都相当难解决，不过在这种情况下，我们的系统相对于原生编译的版本还是基本只有10%的速度下降，在实际Fuzz中还是可以接收的。

3. Effectiveness

正确性和性能的测试我们需要在有源码的程序上进行测试，而我们系统的最终目的是要在没有源码的情况下测试IoT设备中的程序，所以我们最后在真实设备上进行了测试。我们挑选了3个路由器设备作为测试的目标，一个是考虑到他们的广泛使用，第二个是他们普遍具有相对其他简单IoT设备较好的性能，可以更好的用于Fuzz测试。

我们将全部3个设备都升级到最新的固件版本，然后从这些设备中挑选出数十个命令行程序，以及数十个监听端口的服务端程序。通过人工对程序进行简单分析的方法，我们可以确定跟这些程序交互的途径，包括从标准输入直接给输入，从文件读取内容，以及从端口获得数据等不同方式。我们对所有的程序进行分析并插桩，然后放回到原先的设备上进行Fuzz。标准输入或者读取文件的程序我们只需要进行基础的Patch就可以进行Fuzz，而服务端程序则需要使用我们提到的socket拦截机制来进行输入的重定向。对于每个Fuzz实例，我们使用单核CPU测试，运行不超过24小时。

如表所示的，我们在10个程序中找到了crash。前7个是包含在Busybox中的archive工具，我们在这7个程序中找到了1个到30个不等的unique crashes。对这些crashes进行分析之后发现，因为路由器的固件没有采用最新的Busybox版本，所以绝大多数的crash都已经在最新版中被修复。其中lzcat的1个crash和unlzma的2个crash所对应的漏洞在实验的几个月前刚刚被修复，这也证明了我们系统寻找实际BUG的能力。除了Busybox中问题之外，我们还在独立的archive程序unzip中找到了12个crashes。

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| No. | Program | Execution Time(H) | Bitmap Coverage | Total Paths | Execution Speed (exec/s) | Unique Crashes |
| 1 | bunzip2 | 0.34 | 0.56% | 148 | 348.88 | 8 |
| 2 | bzcat | 12 | 0.57% | 202 | 28.07 | 13 |
| 3 | gunzip | 0.34 | 0.71% | 98 | 289.21 | 11 |
| 4 | gzip | 12 | 0.85% | 255 | 337.07 | 30 |
| 5 | lzcat | 14.18 | 0.24% | 78 | 37.45 | 1 |
| 6 | unlzma | 12.12 | 0.25% | 79 | 44.08 | 4 |
| 7 | zcat | 12.28 | 0.81% | 224 | 132.81 | 30 |
| 8 | unzip | 12 | 2.23% | 626 | 349.43 | 12 |
| 9 | tsar | 0.17 | 0.46% | 27 | 55.26 | 8 |
| 10 | plugincenter | 7.58 | 2.06% | 179 | 15.76 | 17 |

列表中的第9项tsar是第三方版本的sar（System Activity Reporter），我们的系统可以在其中找到8个unique crashes。最后一项plugincenter是一个服务端程序，注意到其执行时间要远慢于其他的程序，因为server需要做一系列的初始化工作而拖慢了执行速度，但是我们的系统可以在其中找到17个unique crashes。

以上对于真实的IoT设备上的程序的Fuzz可以展现出我们的系统在寻找Crash上的能力达到了传统PC架构上的Fuzzer的水平，可以有效地在命令行程序、服务端程序等中寻找到BUG。