Implementation

1. Fuzzer Integration

Recall that our goal is to perform efficient coverage-guided fuzzing on binary programs in Linux-based IoT devices. Our work is mainly focused on how fuzzer is accessing the target program instead of improving fuzzing process. Thus, we adopted AFL directly into our framework, and the instrumentation code uses a path reporting mechanism that is fully compatible with AFL. This also ensures that any improvement from AFL community will also benefits us without any modifications to our framework.

We compiled the AFL initialization code into a shared library, so it can be loaded into target program and the corresponding initialization interface to initialize the fork-server and shared memory will be called by the stub we instrumented when the program is initializing.

For programs that using network, we hooked corresponding library function mentioned before, and perform checks on its parameters to determine whether to intercept or forward the network stream. For these will become daemon, we also using library hook to intercept the whole process.

These library hooks and the instrumentation logic take about 300 lines of python code and 3000 lines of C code.

2. Device Setup

The performance is extremely limited for the most of IoT devices, where most vendors will deploy a streamlined runtime library such as uClibc, and this will cause problems when fuzzing on these devices. When AFL gets compiled, it needs certain symbols in the libraries to executing correctly which some devices cannot provide. Therefore, we compiled AFL statically specifically for every device, so it can run without additional library support.

In addition, the ELF loader that comes with some devices does not have TLS (Thread local storage)-related support enabled and the programs with TLS variables cannot be loaded. But AFL requires TLS support to record the execution path of different threads. We found that although the loader does not have TLS support enabled, the corresponding Linux kernel does have it. So, we compiled another version of ELF loader for each device and switched to the new loader if necessary.

Finally, most devices do not preserve interfaces for shell access. We enabled SSH support on each device in different ways to ensure that we can log in and perform fuzzing on the devices.

3. Basic Block Identification

Most binaries from IoT devices are stripped, which can be challenging for reverse engineering tools to completely identify the code resides in the binary. But as we mentioned above, the loss of identified basic blocks is equivalent to the selective discarding that AFL does when instrumentation ratio is below 100%, which only affects accuracy of path resolution.

We have tried several reversing tools such as Barf, Sibyl and Miasm, but none of these turned out to be accurate enough for ARM instruction set. We end up adopting IDA Pro as our reversing tool, who can retrieve most of the basic blocks in the binary and thus meet our requirements.

4. Instrumentation

As mentioned in Section~\ref{sec:design:wrap}, to guarantee successful execution of the original instruction~(e.g. $inst\\_A$ in Figure~\ref{figs:instrumentation}) at new position, we provide different wrap solutions for various types of instructions. In this section, we demonstrate how we implement these solutions on ARM platform by listing wrap solution for instruction type T3-T5.

For instruction type T3, the instruction reads from PC and writes somewhere else. If the instruction is add r1, pc, r2, and we pick r3 as our pivot register, then we will have wrap stub in listing~1.

|  |  |
| --- | --- |
| loc\_1000: add r1, pc, r2  loc\_1004: …  loc\_1008: … | stmdb sp, {r3}  ldr r3, =loc\_1008  add r1, r3, r2  ldmdb sp, {r3}  b loc\_1004 |

listing~1

For instruction type T4, the instruction reads from PC and store it to the stack. If the instruction is push {r0, r2, r4, pc}, and we pick r1 as pivot, then we will have wrap stub in listing~2.

|  |  |
| --- | --- |
| loc\_1000: push {r0, r2, r4, pc}  loc\_1004: …  loc\_1008: … | sub sp, sp, 4  stmdb sp, {r1}  ldr pivot, =loc\_1008  stmia sp, {r1}  ldmdb sp, {r1}  push {r0, r2, r4}  b loc\_1004 |

listing~2

For instruction type T3, the instruction reads from PC and writes back to PC. If the instruction is ldr pc, [pc, r3, lsl#2], and we pick r1 as pivot, then we will have wrap stub in listing~3

|  |  |
| --- | --- |
| loc\_1000: ldr pc, [pc, r3, lsl#2]  loc\_1004: …  loc\_1008: … | sub sp, sp, 4  stmdb sp, {r1}  add sp, sp, 4  ldr pivot, =loc\_1008  ldr r1, [r1, r3, lsl#2]  stmdb sp, {r1}  ldmdb sp, {r1, pc} |

listing~3

We implemented our instrumentation framework for ELF binaries on top of keystone-engine(<https://www.blackhat.com/us-16/briefings.html#keystone-engine-next-generation-assembler-framework>) and capstone-engine(https://www.blackhat.com/latestintel/05282014-focus-on-reverse-engineering.html), who are assemble and disassemble frameworks that cover a lot widely used architectures. And we designed the instrumentation framework so it can be easily adapted to other architectures. We implement our instrumentation framework with python, and it contains about 5300 lines of python code.

实现

1. Fuzzer Integration

我们需要在Linux平台IoT设备上进行高效的覆盖率导向的二进制程序Fuzz，所以我们的工作主要集中在Fuzzer的向下兼容，在上面直接采用了原生的AFL，我们的插桩代码采用了与AFL完全兼容的路径报告机制，所以上面只需要采用AFL即可开始运行。这同时也保证了如果AFL在上层做出任何改进，可以直接适配到我们的环境中，而不需要对我们的插桩做任何的改动。我们插桩代码与AFL默认插桩代码功能相同，只是上报路径。

我们将用于AFL初始化的代码编译成库的形式来加载到程序中，并在程序初始化时调用对应的库函数来初始化fork-server和shared memory。

对从网络获得数据的程序和使用了daemon机制的程序我们会在库函数层面进行hook，进行一定的检查之后来决定是否对网络流进行拦截和转发，或者是否拦截变成daemon的操作。

这些库与插桩的上层逻辑使用了大约300行python和3000行的C程序。

2. Device Setup

因为IoT设备的性能问题，大多数厂商会在其上部署uClibc等精简过的运行库，这使得在设备上进行fuzz有一些特定的问题，比如直接编译的AFL在设备上会因为库不支持而无法运行，所以我们针对每个设备静态编译了AFL，静态编译将运行库打包到程序体重以绕过运行库的问题。

另外，部分设备自带的loader没有开启TLS相关的支持，无法加载有TLS变量的程序。而AFL运行需要程序有TLS支持，因为这样才可以利用TLS记录不同线程的执行路径。我们发现虽然loader没有开启TLS支持，但是对应的Linux内核都是有TLS相关支持的，所以我们又编译了另一个版本的loader，如果目标不支持TLS则切换过来。

最后，大部分设备其实并没有预留访问的接口，我们需要通过不同的途径在每个设备上开启SSH支持，以保证我们可以登陆并且在设备上进行测试。

3. Basic Block Identification

完整的识别没有符号的二进制程序中的代码和数据一直是逆向工具能力高低的表现，我们需要相对比较精准的逆向工具来识别程序中的Basic Block。如上所说，我们不需要获取到所有的Basic Block，因为丢失掉部分仅仅相当于AFL插桩时选择性的丢弃部分插桩点一样，只是对路径分辨精确度的影响。

在识别Basic Block时，我们尝试了Barf、Sibyl、Miasm等框架，但在ARM指令集下，他们的识别率有时相当低下，所以最后我们采用了IDA Pro，其可以找到程序中绝大多数的Basic Block，基本符合我们的要求。

4. Instrumentation

我们的插桩框架依赖于keystone-engine作为汇编工具，以及capstone-engine作为反汇编工具，在这两个框架的基础上，我们实现了ARM架构下的ELF插桩框架，并且可以轻松的在此基础上实现针对其他架构的插桩。

如上所说的，为了让插桩的目标指令可以在新的地址执行，我们针对不同类型的指令设计了不同的wrap方案，这节中，我们将展示具体是如何在ARM平台上进行wrap的。我们在这里列出来三种需要特殊wrap的情况，即上面提到的T3-T5，分别列举一个例子来说明。

第一种只是从PC中读值：

|  |  |
| --- | --- |
| loc\_1000: add r1, pc, r2  loc\_1004: …  loc\_1008: … | stmdb sp, {r3}  ldr r3, =loc\_1008  add r1, r3, r2  ldmdb sp, {r3}  b loc\_1004 |

第二种从PC中读值且涉及到栈操作：

|  |  |
| --- | --- |
| loc\_1000: push {r0, r2, r4, pc}  loc\_1004: …  loc\_1008: … | sub sp, sp, 4  stmdb sp, {r1}  ldr pivot, =loc\_1008  stmia sp, {r1}  ldmdb sp, {r1}  push {r0, r2, r4}  b loc\_1004 |

第三种涉及到PC的读写

|  |  |
| --- | --- |
| loc\_1000: ldr pc, [pc, r3, lsl#2]  loc\_1004: …  loc\_1008: … | sub sp, sp, 4  stmdb sp, {r1}  add sp, sp, 4  ldr pivot, =loc\_1008  ldr r1, [r1, r3, lsl#2]  stmdb sp, {r1}  ldmdb sp, {r1, pc} |

我们的插桩框架由python实现，包含大约5300行python代码。