计算机科学
COMPUTER SCIENCE

利用精确中间污点源和危险函数定位加速固件漏洞挖掘

张光华, 陈放, 常继友, 胡勃宁, 王鹤

引用本文

张光华, 陈放, 常继友, 胡勃宁, 王鹤. 利用精确中间污点源和危险函数定位加速固件漏洞挖掘[J]. 计算机科学, 2025, 52(7): 379-387.

ZHANG Guanghua, CHEN Fang, CHANG Jiyou, HU Boning, WANG He. Accelerating Firmware Vulnerability Discovery Through Precise Localization of IntermediateTaint Sources and Dangerous Functions [J]. Computer Science, 2025, 52(7): 379-387.

计算机科学 COMPUTER SCIENCE
http://www.jsjkx.com
DOI:10.11896/jsjkx.240800052

利用精确中间污点源和危险函数定位加速固件漏洞挖掘

张光华[1,2]　陈　放[1]　常继友[1]　胡勃宁[1]　王　鹤[2]
1 河北科技大学信息科学与工程学院　石家庄 050018
2 西安电子科技大学网络与信息安全学院　西安 710126
(xian_software@163.com)

摘　要　先前的固件静态污点分析方案通过识别中间污点源来精确污点分析的起点,过滤部分情况的安全的命令劫持类危险函数调用点以精简污点分析的目标终点,减少了待分析的污点传播路径,缩短了漏洞挖掘的时间。但由于其在识别中间污点源时所用时间过长,以及没有实现充分过滤安全的危险函数调用点,导致固件漏洞挖掘的整体时间依旧较长。为改进这一现状,提出了一种利用精确中间污点源和危险函数定位加速固件漏洞分析方案 ALTSDF(Accurate Locating of intermediate Taint Sources and Dangerous Functions)。在快速精确识别中间污点源作为污点分析的起点时,收集每个函数在程序中不同调用点处使用的参数字符串构成每个函数的函数参数字符串集合,并计算此集合在前后端共享关键字集合中的占比,根据占比对所有函数进行降序排列,占比越高,则此函数越有可能是中间污点源。在过滤安全的危险函数调用点时,通过函数参数静态回溯分析参数类型,排除参数来源是常量的复杂情况的安全的命令劫持类危险函数调用点和安全的缓冲区溢出类危险函数调用点。最终缩短定位中间污点源所用时间,减少由中间污点源到危险函数调用点所构成的污点传播路径数量,进而缩短将污点分析应用于污点传播路径所需的分析时间,达到缩短漏洞挖掘时间的目的。对 21 个真实设备固件的嵌入式 Web 程序进行测试后得出,ALTSDF 相比先进工具 FITS,在中间污点源推断方面所用时间大幅缩短;在安全的危险函数调用点过滤方面,相比先进工具 CINDY,ALTSDF 使污点分析路径减少了 8%,最终使漏洞挖掘时间相比 SaTC 结合 FITS 与 CINDY 的整合方案缩短 32%。结果表明,ALTSDF 可加速识别固件嵌入式 Web 程序中的漏洞。

关键词:物联网安全;固件漏洞静态检测;污点分析;中间污点源
中图分类号　TP309

Accelerating Firmware Vulnerability Discovery Through Precise Localization of Intermediate Taint Sources and Dangerous Functions

ZHANG Guanghua[1,2],CHEN Fang[1],CHANG Jiyou[1],HU Boning[1] and WANG He[2]
1 School of Information Science and Engineering,Hebei University of Science and Technology,Shijiazhuang 050018,China
2 School of Cyber Engineering,Xidian University,Xi'an 710126,China

Abstract　Existing methods aim to accurately identify the starting points of taint analysis by recognizing intermediate taint sources and filter safe command hijacking points in certain cases to streamline endpoint analysis,thus reducing the paths to be analyzed and shortening vulnerability mining time. However,these methods spend excessive time identifying intermediate taint sources and fail to fully filter safe dangerous function call points,leading to prolonged overall vulnerability mining times. The ALTSDF scheme addresses these issues by accurately identifying intermediate taint sources and dangerous function locations. To quickly and accurately identify intermediate taint source as the starting point for taint analysis,it collects the parameter strings used at different call sites of each function to form its parameter string set. We then calculate the proportion of this set that overlaps with the shared keyword set. Functions are ranked in descending order of this proportion—the higher the proportion,the more likely the function is an intermediate taint source. When filtering safe dangerous function call points,it statically back-traces parameter types to exclude points where the parameter source is a constant,thus avoiding safe command hijacking and buffer overflow points. To reduce the time spent identifying intermediate taint sources,minimize taint propagation paths to dangerous function calls,and shorten the analysis time,thus speeding up vulnerability discovery. Testing on embedded Web programs in 21

到稿日期:2024-08-08　返修日期:2024-11-07
基金项目:国家自然科学基金(62072239,62372236);2025 年河北省硕士在读研究生创新能力培养资助项目(CXZZSS2025076)
This work was supported by the National Natural Science Foundation of China(62072239,62372236) and Postgraduate Innovation Fund Project of Hebei Province(CXZZSS2025076).
通信作者:胡勃宁(wwhbn@hebust.edu.cn)

real device firmwares show that ALTSDF significantly reduces the time spent on intermediate taint source inference compared to the FITS tool. It also reduces the taint analysis path by 8% compared to CINDY and ultimately reduces vulnerability mining time by 32% compared to the combined solution of SaTC with FITS and CINDY. These results demonstrate that ALTSDF acce-lerates the identification of vulnerabilities in firmware embedded Web programs.

Keywords IoT security, Static detection of firmware vulnerabilities, Taint analysis, Intermediate taint source

1 引言

在现代社会中,物联网设备的应用变得越来越普遍。到2030年,使用中的物联网设备数量将达到300亿[1]。物联网设备极大地方便了人们的日常生活,但同时,它们固件中的设计缺陷也逐渐成为各种网络攻击(如僵尸网络[2]、隐私信息窃取[3]、高级持续性威胁[4])的目标。发现固件中的漏洞对于防止物联网设备遭受网络攻击至关重要。

固件静态污点分析因其高代码覆盖率和不需要仿真或真实设备而被广泛应用于嵌入式设备固件漏洞挖掘。现有的静态污点分析工具致力于精简污点分析的起始点和目标终点,以减少待分析的污点传播路径,最终提高固件漏洞挖掘效率。然而,在实际工作中仍存在一些问题,具体表现为:1)精确识别中间污点源代替污点源(如接口库函数recv)缩短需分析的数据流路径时,确定中间污点源所用时间较长[5];2)提出并仅实现过滤部分情况的安全的命令劫持类危险函数调用点,而没有排除复杂情况的安全的命令劫持类危险函数调用点以及安全的缓冲区溢出类危险函数调用点,造成应用污点分析在不会导致漏洞的安全的危险函数调用点上消耗太多时间[6]。

为了解决上述问题,本文提出并实现了一种适用于32位ARM平台固件的ALTSDF原型系统来加速检测嵌入式固件中污点式的命令劫持(Command Injection,cmdi)漏洞和缓冲区溢出漏洞(Buffer Overflow,bof)。主要思路是通过快速并精确地确定中间污点源作为污点分析的起点,将有风险的危险函数调用点作为污点分析的目标终点,实现固件高效漏洞分析。具体而言,ALTSDF首先识别出用于传递用户前端输入到后端处理程序的共享关键字。其次,ALTSDF基于欠约束符号执行[7]生成的控制流图和调用图进行定义可达性分析和调用点分析,以提取程序中每个函数的特征,并计算每个函数的参数字符串在共享关键字集合中的占比,以对可能是中间污点源的函数进行排名。然后利用静态回溯分析过滤掉参数来源是常量的安全的命令劫持类和缓冲区溢出类的危险函数调用点,以精确识别有风险的危险函数调用点。最后,ALTSDF通过污点分析检查中间污点源到有风险的危险函数调用点是否存在路径,以实现漏洞检测。本文评估了ALTSDF在5个流行供应商的21个真实32位ARM平台固件上的快速性。在识别中间污点源方面,将ALTSDF与先进的FITS[5]进行比较,在确保排名前三的函数中至少有一个可以用作中间污点源的情况下,识别中间污点源所用时间大幅下降;在筛选安全的危险函数调用点方面,与先进的CINDY[6]相比,ALTSDF过滤掉了更多的安全的危险函数调用点,使污点分析路径减少8%。通过上述改进,ALTSDF相比先进的SaTC[8]结合FITS和CINDY的整合方案漏洞检测的时间缩短32%。结果表明,ALTSDF是一个实用的工具,可以用来加速检测嵌入式固件中的漏洞。本文的主要贡献包括以下3点:

1)结合用于传递用户前端输入到后端处理程序的共享关键字,提出了一种利用函数参数字符串在共享关键字集合中的占比排序以确定中间污点源的方法,缩短了精确识别中间污点源所用的时间。

2)分析了复杂情况的安全的命令劫持类危险函数调用点以及安全的缓冲区溢出类危险函数调用点的代码结构,提出了基于静态回溯分析技术对参数来源是常量的复杂情况的安全的命令劫持类危险函数调用点和安全的缓冲区溢出类危险函数调用点的筛选方法,减少了有风险的危险函数调用点数量。

3)提出并实现了利用快速中间污点源定位和充分过滤安全的危险函数调用点针对固件嵌入式web程序的漏洞挖掘系统ALTSDF,用于检测命令劫持漏洞和缓冲区溢出漏洞。在5个供应商的21个真实32位ARM平台固件样本上,ALTSDF通过缩短定位中间污点源所用时间以及减少由中间污点源到危险函数调用点所构成的污点传播路径数量,来缩短将污点分析应用于污点传播路径所需的分析时间,相比先进的SaTC结合FITS与CINDY的整合方案漏洞检测的时间缩短了32%。

2 相关工作

在所有物联网设备中,与互联网连接的嵌入式设备(如路由器、网络摄像头)比其他设备更容易受到攻击。此类设备的固件直接暴露在涉及用户交互的复杂的互联网服务中,因此往往包含可利用的漏洞[9]。此外,此类设备作为本地网络的入口点,常被作为桥梁,用于对同一网络内的其他物联网设备(如智能插座和扫地机器人)发起攻击[10]。因此,迫切需要设计高效的技术来自动发现与互联网连接的嵌入式设备固件漏洞。

研究人员已经提出一系列动态或静态的固件分析方法,以检测嵌入式固件中的漏洞。目前提出的许多动态分析方法,如SEmu[11]等,都取得了良好的效果。然而,由于嵌入式设备的种类繁多(如路由器、网络摄像头等)和对不同类型外设(如传感器、定时器、总线控制器等)的极度依赖,动态分析方法难以在多种设备中保持有效性[11]。同时,为了使嵌入式固件漏洞发现技术更具适用性,研究人员提出了许多基于静态污点分析的漏洞发现方法,如HermeScan[12],SaTC[8]和Karonte[13],它们可以直接静态应用于固件程序,实现高代码覆盖率,而无需仿真或使用真实嵌入式设备进行分析。经典的固件污点分析[14]的工作流程包括3个主要步骤:1)将接收

用户数据的接口函数（如 recv，getenv，fgets）识别为污点源；2）将可能导致命令劫持或缓冲区溢出的不安全库函数（如 system，sprintf，strcpy）识别为污点汇聚点；3）分析从污点源到污点汇聚点的数据流，并通过检查污点数据是否在未经过净化处理的情况下到达污点汇聚点，从而判断漏洞是否存在。

对从污点源（例如接口库函数 recv）到汇聚点的大量闭源固件的完整数据流路径进行分析极其困难。嵌入式 Web 程序中的某些自定义函数可以用作中间污点源。与接口库函数相比，使用自定义函数作为中间污点源可以大大缩短分析的数据流路径。然而在固件的嵌入式 Web 程序内，数据别名、通过函数指针或跳转表进行的间接调用、调试信息被剥离等问题增加了分析数据流的复杂性，导致中间污点源难以被快速精准地确定。目前先进的中间污点源识别工具 FITS[5] 使用行为特征（包含 6 个结构特征和 5 个流特征）来表示每个函数，该特征表示能捕捉函数的静态和动态属性，并通过行为聚类和相似性打分按照被判定为中间污点源的可能性对自定义函数进行排序。然而，FITS 在构建程序中每个函数的行为特征时，其构建的特征数量较多且依赖嵌入式 Web 程序本身与标准库，导致推断中间污点源所用时间较长。虽然 FITS 精确中间污点源作为污点分析的起点，缩短了污点分析的路径和所用的时间，但是其确定中间污点源所用的时间过长，导致整体漏洞挖掘时间依旧较长，因此迫切需要对其优化并设计出快速精确的中间污点源定位技术。

在确定污点汇聚点时，大多数方法通过 C 标准库函数的符号名称（如 system，strcpy 等）来识别污点汇聚点，然而这不能确保用户输入的数据流到达污点汇聚点，因此，这些方法在分析不会导致漏洞的安全污点汇聚点时消耗了太多时间。CINDY[6] 仅实现了识别部分情况的安全的命令劫持类危险函数调用点，没有实现识别复杂情况的安全的命令劫持类危险函数调用点以及安全的缓冲区溢出类危险函数调用点，因此没有充分过滤安全的危险函数调用点，导致污点分析时间过长。

3 方案设计

3.1 动机示例

图 1(a)给出了通过对 Tenda AC15 路由器固件的 httpd 程序反编译得到的缓冲区溢出漏洞示例的代码。函数 sub_2BABC 从接收到的用户请求 a1 中提取参数名 ssid 对应的参数值，将其保存在变量 src 中，并由 strcpy 使用。如果变量 src 的长度未进行检查且超过了变量缓冲区的大小，则会出现缓冲区溢出漏洞。由于嵌入式设备的处理能力和内存有限，因此工作人员在开发过程中更重视功能和成本，而忽略了安全检查，故嵌入式设备极易受到漏洞攻击。图 1(b)给出了多个中间污点源的调用示例。中间污点源具有两个显著的特征：1）中间污点源从用户请求中提取部分内容，并通过返回数值、指针、全局变量等方式传递结果；2）使用不同的用于传递信息的前后端共享关键字作为参数。基于上述特征，本文提出了更快速地确定中间污点源的方法：函数参数的不同字符串在用于传递信息的前后端共享关键字中占比最高的函数最有可能是中间污点源。

```
1. src =(char *)sub_2BABC(a1,"ssid",&unk_E35DC);
2. if( *src ) {
3.   strcpy(s,src);
4.   strcpy(dest,src);
5.   ……
```

(a)典型缓冲区溢出漏洞示例

```
1. v2 =(char *)sub_2BABC(a1,"deviceId",&unk_EFC98);
2. v6 =(char *)sub_2BABC(a1,"password",&unk_DC470);
3. v7 =(char *)sub_2BABC(a1,"username",&unk_DC470);
4. v9 =(char *)sub_2BABC(a1,"speed_dir","0");
5. s2 =(char *)sub_2BABC(a1,"mac",&unk_EDB68);
```

(b)中间污点源调用示例

图 1 从中间污点源到达危险函数的缓冲区溢出漏洞示例

Fig. 1 Example of buffer overflow vulnerability reaching dangerous function from intermediate taint source

Netgear 的 R6400 路由器的 httpd 程序的反编译代码片段如图 2 和图 3 所示。

```
1. int sub_7E49C(int a1){
2.     ……
3.     system("/usr/bin/killall-9 bftpd 2> /dev/null");
4.     v6="bftpd-D-c /tmp/bftpd.conf &";
5.     system(v6);
6. }
7. int sub_9322C(){
8.     sprintf(&v1,"bzip2-d%s","/tmp/langtbl.bz2");
9.     system((const char *)&v1);
10.    …
11. }
12. int sub_3C53C(char *a1,int a2){
13.    ……
14.    if(! strcmp(v34,"dmz_enable")){
15.    v7=1;
16.    }
17.    else{
18.     v7=0;
19.    }
20.    sprintf(v39,"nvram set%s=%d","fw_dmz_enab",v7);
21.    system(v39);
22.    ……
23. }
```

图 2 安全的命令劫持类函数调用示例

Fig. 2 Example of safe command hijacking class function call

在图 2 的 sub_7E49C 中有两处命令劫持类危险函数调用点，第一处的 system 函数的参数是一个固定的字符串，第二处的 system 函数的参数是变量 v6，但是回溯变量 v6，发现它的值是常量字符串；在图 2 中的 sub_9322C 函数中，变量 v1 传递给 system 函数，但当回溯变量 v1 后，发现它的值是调用 sprintf 函数得到的，而传递给 sprintf 的源参数是常量字符串 bzip2 -d%s 和/tmp/langtbl.bz2，因此最终存储在变量 v1 中的值是常量字符串 bzip2 -d /tmp/langtbl.bz2；在图 2 的 sub_3C53C 函数中，变量 v39 传递给 system 函数，但当回溯变量 v39 后，发现它的值是调用 sprintf 函数得到的，而传递给 sprintf 的源参数是常量字符串 nvram set%s=%d，fw_dmz_enab 和变量 v7，进一步回溯变量 v7 发现它只可能是常量数字，因此最终存储在变量 v39 中的值是常量字符串。

因此，sub_7E49C 函数、sub_9322C 函数和 sub_3C53C 函数涉及到的命令劫持类危险函数调用点是安全的。3 类安全的命令劫持类危险函数调用点代码结构包括：1）参数是常量字符串，对应 sub_7E49C 函数中第一处 system 函数调用点代码结构；2）参数是变量，但该变量在调用危险函数之前被赋予常量字符串，对应 sub_7E49C 函数中第二处 system 函数调用点代码结构；3）传入危险函数调用点的参数是变量，此变量是前一个执行函数返回的常量结果，对应 sub_9322C 和 sub_3C53C 函数中 system 函数调用点代码结构。

图 3 给出了在 sub_FBE0 函数中的两处缓冲区溢出类危险函数调用点，变量 v23 传递给第一处缓冲区溢出类危险函数 strcpy，但通过回溯变量 v23，发现它只可能是两个常量字符串中的一个，因此此处的缓冲区溢出类危险函数调用点是安全的；sub_FBE0 函数中的第二处缓冲区溢出类危险函数调用点的源参数是常量字符串，因此其也是安全的危险函数调用点。2 类安全的缓冲区溢出类危险函数调用点代码结构包含：1）参数是常量字符串，对应 sub_FBE0 函数中第二处 strcpy 函数调用点代码结构；2）参数是变量，但该变量在调用危险函数之前被赋予常量字符串，对应 sub_FBE0 函数中第一处 strcpy 函数调用点代码结构。

```
1. char sub_FBE0(char *a1,int a2,int *a3,int a4){
2.     ……
3.     if( stristr(v22,"www.routerlogin.net") ){
4.     v23="www.routerlogin.net";
5.     }
6.     else{
7.     v23="routerlogin.com";
8.     }
9.     strcpy(dest,v23);
10.    ……
11.    strcpy((char *)&v303,"Chinese");
12.    ……
13.}
```

图 3 安全的缓冲区溢出类函数调用示例

Fig. 3 Example of safe buffer overflow class function call

对以上 3 类安全的命令劫持类危险函数调用点和 2 类安全的缓冲区溢出类危险函数调用点进行分析可知，只通过函数名便将其视为一个有风险的危险函数调用点，并应用基于符号执行的污点分析将浪费大量时间。CINDY 实现了前两类安全的命令劫持类危险函数调用点的过滤，ALTSDF 实现了以上 3 类安全的命令劫持类危险函数调用点和 2 类安全的缓冲区溢出类危险函数调用点的过滤，且可以节省大量时间和计算资源。

3.2 系统架构

如图 4 所示，ALTSDF 由前后端共享关键字提取、中间污点源精确定位、危险函数精确定位、污点分析 4 个阶段组成。本文的创新性体现在中间污点源精确定位阶段和危险函数精确定位阶段。

1）前后端共享关键字提取。此阶段通过 binwalk[15] 解压固件以获取前端文件和后端二进制文件，然后 ALTSDF 采用与 SaTC 相同的解决方法提取用于标记传递用户输入的前后端共享关键字，其中包括前端文件和后端程序中共有且相同的前后端共享关键字和仅在后端程序中出现的隐式前后端共享关键字[8]。因此，ALTSDF 收集了更全的前后端共享关键字，以应对前后端信息传递时依赖复杂数据导致可能出现的共享关键字不一致的情况。

2）中间污点源精确定位。网络通信是物联网设备被威胁的主要来源，因此在解压固件获取的二进制文件中，选择嵌入式 Web 程序作为分析目标。为了解决在剥离符号表和调试信息的嵌入式 Web 程序中快速推断中间污点源的问题，设计了一种包括 5 个特征的函数特征表示，以计算每个函数的参数不同字符串在前后端共享关键字集合中的占比并进行排序，占比最高的前三个函数是潜在的中间污点源，具体见 3.3 节。

3）危险函数精确定位。为了消除不必要的路径，减少污点分析工作量以提高效率，在此阶段利用静态回溯分析实现了对 3 类安全的命令劫持类危险函数调用点以及 2 类安全的缓冲区溢出类危险函数调用点的过滤。具体见 3.4 节。

4）污点分析。ALTSDF 的污点分析依赖符号执行，并采用 SaTC 方案的输入敏感的污点分析，其中包括粗粒度污点分析、敏感路径引导和调用路径合并的高效路径探索策略以及路径优先级策略[8]。

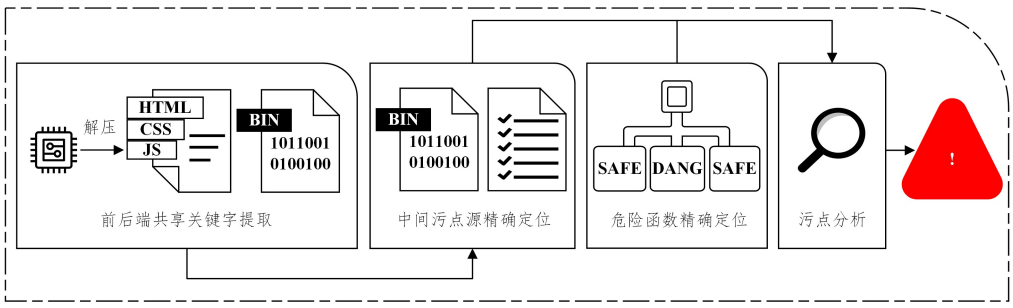

图 4 ALTSDF 架构

Fig. 4 Architecture of ALTSDF

3.3 中间污点源精确定位

本文提出了利用函数的 5 个特征快速识别中间污点源的方法。具体包括以下 5 个特征：1）函数名；2）此函数在嵌入式 Web 程序中被调用的次数；3）此函数在嵌入式 Web 程序中所有调用点处使用的参数字符串组成的列表中不同字符串的个数；4）此函数的参数的不同字符串在前后端共享关键字集合的占比；5）此函数所有参数字符串组成的列表。由于网络通信数据是结构化的，用户请求通常以“关键字”与“用户输入”

结合的形式存储在内存中，当需要处理某些用户输入时，例如 name 和 password 之类的关键字被用作索引，以获取用户输入的相应部分。因此，将函数参数是字符串的情况作为特征是合理的。

为了解决多样化寻址模式和变量类型丢失给确定函数参数是否为字符串带来的挑战，ALTSDF 递归收集所有函数的程序调用点以及存储参数的寄存器，然后向后跟踪寄存器，并在寄存器可以用常量表示时结束跟踪。下文分析常量所在的节以获取参数字符串的具体内容。如果寄存器中保存的地址指向程序中的只读节(rodata)，ALTSDF 将直接获取参数字符串的内容。在某些情况下，字符串是在程序执行的过程中动态生成的，因此它们存储在数据节(data)中。这些字符串的引用方法类似于全局偏移表[16]，它提供了一个中间表来存储指向这些字符串的指针，因此 ALTSDF 通过检索数据节地址指向的内容来获取参数字符串的内容。

结合 5 个特征快速识别中间污点源的整体算法如算法 1 所示。

算法 1 快速识别中间污点源算法

输入：嵌入式 Web 程序 binProg；前后端共享关键字集合 KwsFrontend；函数特征 funcChars，即函数名 name、函数被调用次数 callCnt、函数参数不同字符串数量 uniqParamCnt、函数参数字符串在前后端共享关键字集合中的占比 KwdRatio、函数参数不同字符串列表 uniqParamList

输出：最有可能是中间污点源的 3 个函数 PotentialSources[3]

```
funcList←extract_functions_from(binProg)
PotentialSources←[]
for each func in funcList do
    if isStandardLibraryFunction(func.name) then
        func.KwdRatio←0
        continue
    end if
    if func.callCnt == 0 then
        func.KwdRatio←0
        continue
    end if
    if func.uniqParamCnt == 0 then
        func.KwdRatio←0
        continue
    end if
    KwCnt←0
    for each paramStr in func.uniqParamList do
        if paramStr in KwsFrontend then
            KwCnt←KwCnt+1
        end if
    end for
    func.KwdRatio←KwCnt/len(KwsFrontend) /*计算函数参数字符串在前后端共享关键字集合中占比*/
end for
Sort funcList by func.KwdRatio in descending order /*执行降序排列*/
PotentialSources←funcList[:3]
returnPotentialSources /*输出最有可能是中间污点源的 3 个函数*/
```

为了快速推断出中间污点源，ALTSDF 需要对函数进行过滤。中间污点源是一种自定义函数，而非标准库函数，因此首先将函数名为标准库函数名(如 printf，strcpy 等)的函数过滤。其次，中间污点源需要在程序中被反复调用以提取和返回用户输入的一部分，因此它在程序中被调用的次数必须大于 0。然后，由 3.1 节所述的中间污点源的特征可知，中间污点源使用不同的用于传递信息的前后端共享关键字作为参数，因此它的不同参数字符串个数必须大于 0。通过这 3 条准则对自定义函数进行过滤，留下候选自定义函数。最后将候选自定义函数的函数参数的不同字符串在前后端共享关键字集合的占比进行降序排列，所占比例越高，则此函数越可能是中间污点源。

3.4 危险函数精确定位

利用符号执行的污点分析对固件程序进行漏洞检测取得了良好的结果，然而其在分析安全的危险函数调用点上浪费了太多时间，导致漏洞检测效率较低。因此，在已有精确中间污点源的情况下，识别并过滤安全的危险函数调用点可进一步优化污点分析路径。先前的工具 CINDY 已经针对安全的命令劫持类危险函数调用点的识别，实现了以下 2 类情况：1)参数是常量字符串；2)参数是变量，但该变量在调用危险函数之前被赋予常量字符串。具体而言，CINDY 首先使用开源工具 Ghidra 反汇编固件嵌入式 Web 程序，并遍历反汇编代码，根据危险函数名找到所有的危险函数调用点，然后为每个危险函数调用点生成 PcodeOpAST 数据结构，并使用此数据结构跟踪危险函数的参数来源。对于第一种情况，直接判断参数的属性是否为字符串；对于第二种情况，当发现传递给函数的参数是变量时，使用定义-使用链(Def-Use Chain)来确定它的来源是否是常量字符串[6]。

受此启发，针对 3.1 节中安全的命令劫持类危险函数调用点代码结构的第 3 类情况(即传入危险函数调用点的参数是变量，此变量是前一个执行函数返回的常量结果)，ALTSDF 利用开源工具 Ghidra 首先判断命令劫持类危险函数的参数来源是前一个执行函数的返回结果，然后获取危险函数调用点的前一句反编译代码结果，查找反编译代码结果中是否存在用于拼接或复制命令劫持类危险函数所执行命令的函数(即 strcpy，memcpy，strlcpy，sprintf，sscanf，strncpy，snprintf)。进一步探究用于拼接或复制命令劫持类危险函数所执行命令的函数可知，此类函数将数据从源地址的内存空间复制到目标地址的内存空间，因此目标地址内存空间的数据具有与源地址内存空间数据相同的属性，故只需要对前一个执行函数的参数来源进行检查，就可知前一个执行函数返回结果的属性。如果前一个执行函数的参数来源均为常量，则前一个执行函数的返回结果为常量，进而命令劫持类危险函数的参数来源是常量，因此此处是安全的命令劫持类危险函数调用点；如果前一个执行函数的参数来源中存在变量，则前一个执行函数的返回结果不是固定的字符串，进而命令劫持类危险函数的参数来源不是固定的字符串，因此此处是有风险的命令劫持类危险函数调用点。针对前一个执行函数参数来源是常量的识别，考虑以下 2 种情况：1)参数是常量字符串；2)参数是变量，但该变量在调用危险函数之前被赋予常量字符串。

对上述理论进行扩展，ALTSDF 利用静态回溯分析识别安全的缓冲区溢出类危险函数调用点，具体将以下两类情况视为安全的传递参数：1)参数是常量字符串；2)参数是变量，但该变量在调用危险函数之前被赋予常量字符串。结合缓冲区溢出漏洞触发特点，考虑对 4 个缓冲区溢出类函数(即 strcpy，sprintf，sscanf，strcat)进行回溯分析，判断函数参数类型。

3.5 系统实现

本文使用 Python3(v3.10.12)为 32 位 ARM 架构固件程序实现了 ALTSDF 的原型系统。固件解包基于 binwalk[15]来提取固件中的文件系统。基于 angr(v9.2.100)提取函数的控制流图和函数调用图，并分析数据流和控制流以实现函数特征的提取，从而对中间污点源的参数字符串在前后端共享关键字集合中的占比进行排序，快速精确地实现语义无关的中间污点源推断。在排除安全的危险函数调用点阶段，使用 Ghidra[17]提供的反编译器，并通过其在危险函数调用点生成的 PcodeOpAST 数据结构实现对函数参数来源的静态回溯，同时支持 3 类安全的命令劫持类危险函数调用点和 2 类安全的缓冲区溢出类危险函数调用点的识别。前端文件共享关键字提取方法和污点分析引擎，采用了 SaTC 发布的方案。

4 实验及结果分析

本文设计了对 ALTSDF 的评估实验，以回答以下 3 个研究问题。

问题 1 ALTSDF 推断中间污点源的准确性和快速性。

问题 2 ALTSDF 识别安全的危险函数调用点的准确性，以及在减少有风险的危险函数调用点数量和减少污点分析路径数量方面的有效性。

问题 3 ALTSDF 相比现有的先进工具在固件漏洞检测中的快速性。

4.1 评估设置

为了评估 ALTSDF 工具，选择基于 Karonte[13]数据集的 5 个知名的物联网供应商(即 Tenda，D-Link，NetGear，TP-Link，Cisco)的 21 个固件样本，它们全部采用了 32 位的 ARM 架构。所有实验都是在 Linux 工作站上进行的，该工作站配有 Intel Core i9-12900H CPU 和 64 GB RAM。

为了回答提出的研究问题，在实验评估时选择了不同的对比标准。

针对问题 1，选择 FITS[5]对每个自定义函数构建的 11 个函数特征进行打分，将排序前三的函数确定为潜在的中间污点源，若潜在的中间污点源中存在经人工分析后确定的中间污点源，则证明 FITS 工具可精确定位中间污点源。ALTSDF 采用相同的标准来对比推断中间污点源的准确性，即获取到的 3 个潜在的中间污点源中存在经人工分析后确认的中间污点源，则证明工具是精确的。同时，ALTSDF 与 FITS 将比较中间污点源定位的快速性。

针对问题 2，首先评估 ALTSDF 识别安全的危险函数调用点的准确性。为了评估 ALTSDF 在减少有风险的危险函数调用点数量以及减少污点分析路径数量方面的有效性，将其与 CINDY[6]进行比较。

针对问题 3，比较 ALTSDF 和先进的 SaTC[8]结合 FITS 与 CINDY 的整合方案进行漏洞分析所用的时间，展示 ALTSDF 在检测漏洞方面的快速性。

4.2 高效中间污点源推断

根据系统的开发规范，开发人员不可能复制实现相同目的的函数，因此在一个二进制文件中最多有一个或两个函数是有效的中间污点源。在分别获取 ALTSDF 和 FITS 的排名结果后，需要验证函数是否可以用作中间污点源。将如 3.1 节所述的中间污点源的两个显著的特征作为中间污点源确定的原则，针对不同固件程序的特定情况使用了 3 种验证方法：先前漏洞信息细节分析、固件重新托管验证和固件历史相似版本信息分析。表 1 列出了 ALTSDF 和 FITS 推断中间污点源的结果，因为 ALTSDF 和 FITS 在输出结果时，均输出 3 个函数作为潜在的中间污点源，且输出结果按照各自工具经过计算得出此函数被确定是中间污点源的可能性从高到低进行排序，所以表 1 中的排名表示在由工具输出的 3 个潜在中间污点源中排名数字对应的函数是经人工分析后确认的中间污点源。结果表明 ALTSDF 能精确且快速地推断出中间污点源。FITS 在 D-Link 品牌的 5 个固件样本中，虽然有推断结果，但经人工分析后均不是中间污点源，进一步探究发现 FITS 在寻找中间污点源时关注的是自定义非动态链接库函数，但 D-Link 品牌的 5 个固件样本中的中间污点源都是动态链接库函数，因此 FITS 的方法失效。

表 1 中间污点源推断结果的对比

Table 1 Comparison of intermediate taint source inference results

品牌	固件	程序	中间污点源	ALTSDF 排名	ALTSDF 时间(hh:mm:ss)	FITS 排名	FITS 时间(hh:mm:ss)
Tenda	AC15_V15.03.05.18	httpd	sub_2babc	2	00:04:53	3	12:54:59
Tenda	AC18_V15.03.05.05	httpd	sub_2b884	2	00:05:50	3	12:44:32
Tenda	AC9_V15.03.05.14	httpd	sub_2b9fc	2	00:05:37	3	12:49:27
Tenda	AC6_V15.03.05.16	httpd	sub_2b7c4	2	00:05:27	3	12:52:26
D-Link	DIR868LA1_FW112b04	cgibin	getenv	1	00:01:37	-	03:36:50
D-Link	DIR880A1_FW107WWb08	cgibin	getenv	1	00:01:45	-	03:44:00
D-Link	DIR885LA1_FW113b03	cgibin	getenv	1	00:01:50	-	02:20:48
D-Link	DIR890LA1_FW111b01	cgibin	getenv	1	00:01:42	-	03:41:39
D-Link	DIR895LA1_FW113b03	cgibin	getenv	1	00:01:55	-	01:16:45
NETGEAR	AC1450-V1.0.0.36	httpd	sub_1654c	1	00:05:47	3	18:04:15
NETGEAR	R6200v2-V1.0.3.12	httpd	sub_15c34	1	00:05:49	3	11:15:26
NETGEAR	R6300v2-V1.0.4.18	httpd	sub_17190	1	00:04:45	1	15:52:43

（续表）

品牌	固件	程序	中间污点源	ALTSDF排名	ALTSDF时间(hh:mm:ss)	FITS排名	FITS时间(hh:mm:ss)
NETGEAR	R6400v2-V1.0.2.46	httpd	sub_18764	1	00:24:19	1	19:24:22
NETGEAR	R6700-V1.0.1.36	httpd	sub_19164	1	00:37:34	1	21:46:05
NETGEAR	R7000P-V1.3.0.8	httpd	sub_19090	1	00:37:10	1	20:48:08
NETGEAR	R7300-V1.0.0.56	httpd	sub_18128	1	00:25:50	1	16:49:51
NETGEAR	R7900-V1.0.1.26	httpd	sub_183b8	1	00:27:20	1	17:04:43
NETGEAR	R8000-V1.0.4.4	httpd	sub_184c0	1	00:26:15	1	17:28:39
TP-Link	Archer_C3200v1_0.9.1_0.1	httpd	sub_ad58	1	00:00:57	1	00:03:24
TP-Link	TX-VG1530V1	httpd	sub_131d8	1	00:01:11	1	00:03:15
Cisco	RV130X_FW_1.0.3.44	httpd	sub_1d170	1	00:03:36	2	05:18:23

回答问题1：将ALTSDF与FITS进行比较。结果显示，一方面ALTSDF精确推断出中间污点源；另一方面，ALTSDF比FITS所用时间短证明了ALTSDF推断中间污点源的准确性和快速性。

4.3 精确筛选危险函数调用点

为了筛选安全的危险函数调用点并评估ALTSDF的准确性，本节选取ALTSDF对NETGEAR品牌的R6300路由器固件程序筛选安全的危险函数调用点的结果进行验证。人工分析ALTSDF所报告的37个安全的命令劫持类危险函数调用点和273个安全的缓冲区溢出类危险函数调用点，并按照3.1节所述3类安全的命令劫持类危险函数调用点代码结构和2类安全的缓冲区溢出类危险函数调用点代码结构对结果进行分类。如表2所列，在所报告的37个安全的命令劫持类危险函数调用点中有34个危险函数调用点被证实是安全的，准确率为92%；在所报告的273个安全的缓冲区溢出类危险函数调用点中有253个危险函数调用点被证实是安全，准确率为93%。因此，ALTSDF可准确筛选安全的危险函数调用点。

表2 人工分析NETGEAR R6300路由器固件程序的安全的危险函数调用点的识别结果

Table 2 Results of manual analysis on secure identification of dangerous function call points in the NETGEAR R6300 router firmware

（个）

安全命令劫持函数调用点数量				安全缓冲区溢出函数调用点数量		
类一	类二	类三	误报	类一	类二	误报
0	4	30	3	70	183	20

为了对比ALTSDF和CINDY在减少危险函数调用点数量以及减少待分析污点传播路径数量方面的有效性，使用两个指标R_S和R_p，分别代表有风险的危险函数调用点数量的百分比减少和污点传播路径数量的百分比减少。其中，S是有风险的危险函数调用点的数量，P是待分析的污点传播路径数量，如式(1)、式(2)所示。为了准确评估筛选安全的危险函数调用点对减少待分析污点传播路径数量的影响，ALTSDF与CINDY均未限制以中间污点源作为待分析的污点传播路径的起点。

$$R_S=\frac{S_{\mathrm{CINDY}}-S_{\mathrm{ALTSDF}}}{S_{\mathrm{CINDY}}} \tag{1}$$

$$R_p=\frac{P_{\mathrm{CINDY}}-P_{\mathrm{ALTSDF}}}{P_{\mathrm{CINDY}}} \tag{2}$$

表3对比了ALTSDF与CINDY分别应用于5个物联网供应商的21个固件程序后，有风险的危险函数调用点数量以及待分析污点传播路径数量。ALTSDF在识别有风险的危险函数调用点数量方面比CINDY少36%，在待分析的污点传播路径数量方面比CINDY少8%。原因是CINDY只实现了针对安全的命令劫持类危险函数调用点的第一类和第二类情况的识别，因此CINDY存在对命令劫持类危险函数调用点的第三类情况的漏报，且其没有对安全的缓冲区溢出类危险函数调用点的排除造成漏报，故CINDY产生了更多待分析的污点传播路径数量。在表3中，ALTSDF针对D-Link品牌和TP-Link品牌的固件程序虽然排除了更多的安全缓冲区溢出类危险函数调用点，但是待分析污点传播路径数量没有减少。详细探究原因后发现ALTSDF和CINDY选择的污点分析引擎采用调用路径合并策略，以精简污点传播路径。如图5所示，调用路径合并策略以不同污点源为起点构建污点传播路径，当污点传播路径对应的危险函数调用点全部安全时，此条污点传播路径才不被构建。因此在D-Link品牌和TP-Link品牌的固件程序中，虽然大量减少了有风险的危险函数调用点，但每条污点传播路径对应的危险函数调用点中仅有部分危险函数调用点被识别为安全，导致此条污点传播路径仍会被保留，最终待分析的污点传播路径总数量不变。

表3 有风险的危险函数调用点数量及污点传播路径数量对比

Table 3 Comparison of the number of dangerous function call points and the number of taint propagation paths

固件品牌及数量	S_{ALTSDF}/个			S_{CINDY}/个			R_S/%	P_{ALTSDF}/个			P_{CINDY}/个			R_P/%
	cmdi	bof	合计	cmdi	bof	合计		cmdi	bof	合计	cmdi	bof	合计	
Tenda(4)	128	723	851	135	1189	1324	36	357	5671	6028	369	6155	6524	8
D-Link(5)	15	151	166	45	273	318	48	55	361	416	113	361	474	12
NETGEAR(9)	324	7008	7332	670	10681	11351	35	4875	24522	29397	5616	26131	31747	7
TP-Link(2)	0	3	3	0	6	6	50	0	3	3	0	3	3	0
Cisco(1)	3	83	86	4	160	164	48	18	168	186	19	266	285	35
合计	470	7968	8438	854	12309	13163	36	5305	30725	36030	6117	32916	39033	8

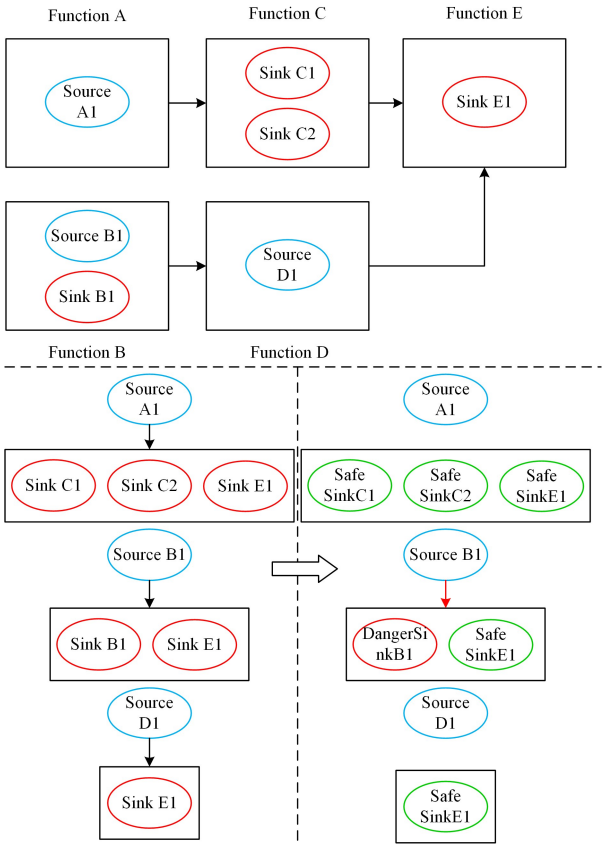

图 5 SaTC 污点传播路径合并策略

Fig. 5 SaTC taint propagation path merging strategy

回答问题 2：ALTSDF 实现了对 3 类安全的命令劫持类危险函数调用点以及对 2 类安全的缓冲区溢出类危险函数调用点的精确识别，ALTSDF 相比 CINDY 在有风险的危险函数调用点数量上减少 36%，待分析的污点传播路径数量减少 8%。结果证明 ALTSDF 有效地减少了有风险的危险函数调用点数量和待分析的污点传播路径数量。

4.4 漏洞检测的快速性

为了证明 ALTSDF 的快速性，将 ALTSDF 和先进的 SaTC 结合 FITS 与 CINDY 的整合方案（表示为 SaTC_FC）进行对比。FITS 通过识别程序的中间污点源来减少污点分析的起点，CINDY 排除部分情况的安全的命令劫持类危险函数调用点来减少污点分析的终点，从而减少待分析的污点传播路径，确保将污点分析应用于有效的污点传播路径，最终缩短漏洞挖掘所用的时间，因此 SaTC_FC 是先进且高效的固件漏洞挖掘整合方案，将 ALTSDF 与其比较是合理的。如表 4 所列，在 ALTSDF 报告的 1 078 个警报中，经人工确认后有 294 个是真实漏洞；在 SaTC_FC 报告的 1 257 个警报中，经人工确认后有 322 个是真实漏洞。两者在真实漏洞发现方面的重合情况如图 6(a)所示，ALTSDF 与 SaTC_FC 重合发现 274 个真实漏洞。进一步分析两者在真实漏洞发现方面的差异情况后发现，ALTSDF 独有的 20 个真实漏洞的污点传播路径在 SaTC_FC 中同样存在，SaTC_FC 独有的 48 个真实漏洞中有 7 个真实漏洞的污点传播路径在 ALTSDF 中同样存在。原因是 ALTSDF 和 CINDY 选择的污点分析引擎为了防止符号执行路径爆炸所导致的运行时间过长和运行占用内存过大而采用限制运行时间策略，即针对每条污点传播路径的分析时间最多不超过固定时间，因此在固定分析时间内，基于符号执行的污点分析探索的随机性导致即使应用相同的污点分析引擎在相同的污点传播路径上，仍会出现漏洞发现结果的差异。因此如图 6(b)所示，ALTSDF 理论上发现漏洞的数量为 301 个（即 20+274+7），SaTC_FC 理论上发现漏洞的数量为 342 个（即 20+274+7+41），仅有 41 个漏洞由于 ALTSDF 安全的危险函数调用点定位的限制，导致有风险的危险函数调用点被错误排除，从而造成漏洞漏报，漏报率仅为 11%。

表 4 漏洞检测效果对比

Table 4 Comparison of vulnerability detection effect

固件品牌及数量	ALTSDF						SaTC_FC					
	警报/个			漏洞/个			警报/个			漏洞/个		
	cmdi	bof	合计	cmdi	bof	合计	cmdi	bof	合计	cmdi	bof	合计
Tenda(4)	3	107	110	3	86	89	4	107	111	3	80	83
D-Link(5)	0	20	20	0	5	5	0	17	17	0	5	5
NETGEAR(9)	28	915	943	4	196	200	84	1 018	1 102	4	225	229
TP-Link(2)	0	0	0	0	0	0	0	0	0	0	0	0
Cisco(1)	0	5	5	0	0	0	0	27	27	0	5	5
合计	31	1 047	1 078	7	287	294	88	1 169	1 257	7	315	322

图 6 真实漏洞数量重合和差异情况对比

Fig. 6 Comparison of overlap and difference of the number of actual vulnerabilities

在对比 ALTSDF 和 SaTC_FC 检测漏洞的时间时，采用指标R_T表示漏洞检测所用时间减少百分比，如式(3)所示。

$$R_T = \frac{T_{\text{SaTC\_FC}} - T_{\text{ALTSDF}}}{T_{\text{SaTC\_FC}}} \quad (3)$$

如表 5 所列，通过缩短定位中间污点源所用时间以及减少污点传播路径数量，来缩短污点分析时间。ALTSDF 相比 SaTC_FC 检测漏洞总时间缩短 32%。

回答问题 3：得益于对中间污点源的精确识别和对安全的危险函数调用点的充分过滤，ALTSDF 相比 SaTC_FC 漏洞检测更快速，检测时间缩短 32%。

表 5 漏洞检测时间对比

Table 5 Comparison of vulnerability detection time

固件品牌及数量	ALTSDF				SaTC_FC				R_T/%
	T_{ITS}/s	T_{cmdi}/s	T_{bof}/s	T_{ALTSDF}/s	T_{ITS}/s	T_{cmdi}/s	T_{bof}/s	T_{ALTSDF}/s	
Tenda(4)	1307	34525	115108	150940	184884	34199	151178	370261	59
D-Link(5)	529	5592	32379	38500	52802	11757	31811	96370	60
NETGEAR(9)	11689	165780	1580523	1757992	573292	223174	1581048	2377514	26
TP-Link(2)	128	140	172	440	399	138	151	688	36
Cisco(1)	216	423	4869	5508	19103	464	11192	30759	82
合计	13869	206460	1733051	1953380	830480	269732	1775380	2875592	32

结束语 ALTSDF 能快速精确地识别中间污点源，并对 3 类安全的命令劫持类和 2 类安全的缓冲区溢出类危险函数调用点进行识别和过滤，最后利用污点分析检测用户输入的危险使用。ALTSDF 在中间污点源推断方面相比 FITS 所用时间大幅降低，在识别有风险的危险函数调用点数量方面相比 CINDY 减少 36%，在待分析的污点传播路径数量方面相比 CINDY 减少 8%。相比先进的 SaTC 结合 FITS 与 CINDY 的整合方案，ALTSDF 在漏洞检测所用时间方面缩短 32%。因此，ALTSDF 可加速识别嵌入式 Web 程序的污点式漏洞。下一步的主要工作包括更精确地识别更复杂的、安全的危险函数调用以及更新污点分析引擎。

参 考 文 献

[1] VAILSHERY L S. Internet of Things(IoT) - statistics & facts [EB/OL]. (2024-06-04) [2024-08-03]. https://www.statista.com/topics/2637/internet-of-things/.

[2] ANTONAKAKIS M, APRIL T, BAILEY M, et al. Understanding the mirai botnet[C]// 26th USENIX Security Symposium (USENIX Security 17). USENIX Association, 2017: 1093-1110.

[3] TEAM T I. 150000 Verkada security cameras hacked—to make a point[EB/OL]. (2021-03-12) [2024-06-28]. https://www.threatdown.com/blog/150000-verkada-security-cameras-hacked-to-make-a-point/.

[4] LANGNER R. Stuxnet: Dissecting a cyberwarfare weapon [J]. IEEE Security & Privacy, 2011, 9(3): 49-51.

[5] LIU P, ZHENG Y, SUN C, et al. FITS: Inferring Intermediate Taint Sources for Effective Vulnerability Analysis of IoT Device Firmware[C]// the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems. ACM, 2023: 138-152.

[6] YIN X, CAI R, ZHANG Y, et al. Accelerating Command Injection Vulnerability Discovery in Embedded Firmware with Static Backtracking Analysis[C]// The 12th International Conference on the Internet of Things. IEEE, 2022: 65-72.

[7] RAMOS D A, ENGLER D. Under-Constrained symbolic execution: Correctness checking for real code[C]// 24th USENIX Security Symposium (USENIX Security 15). USENIX Association, 2015: 49-64.

[8] CHEN L, WANG Y, CAI Q, et al. Sharing more and checking less: Leveraging common input keywords to detect bugs in embedded systems[C]// 30th USENIX Security Symposium (USENIX Security 21). USENIX Association, 2021: 303-319.

[9] QASEM A, SHIRANI P, DEBBABI M, et al. Automatic Vulnerability Detection in Embedded Devices and Firmware: Survey and Layered Taxonomies [J]. ACM Computing Surveys, 2021, 54(2): 1-42.

[10] YAO Y, ZHOU W, JIA Y, et al. Identifying Privilege Separation Vulnerabilities in IoT Firmware with Symbolic Execution[C]// Computer Security-ESORICS 2019: 24th European Symposium on Research in Computer Security. Springer, 2019: 638-657.

[11] ZHOU W, ZHANG L, GUAN L, et al. What Your Firmware Tells You Is Not How You Should Emulate It: A Specification-Guided Approach for Firmware Emulation[C]// the ACM Conference on Computer and Communications Security 2022. ACM, 2022: 3269-3283.

[12] GAO Z, ZHANG C, LIU H, et al. Faster and Better: Detecting Vulnerabilities in Linux-based IoT Firmware with Optimized Reaching Definition Analysis[C]// NDSS2024. ISOC, 2024: 1-16.

[13] REDINI N, MACHIRY A, WANG R, et al. Karonte: Detecting insecure multi-binary interactions in embedded firmware[C]// 2020 IEEE Symposium on Security and Privacy (SP). IEEE, 2020: 1544-1561.

[14] CHENG K, LI Q, WANG L, et al. DTaint: detecting the taint-style vulnerability in embedded device firmware[C]// 2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). IEEE, 2018: 430-441.

[15] REFIRMLAB S. binwalk [EB/OL]. (2023-02-02) [2024-06-20]. https://github.com/ReFirmLabs/binwalk.

[16] WIKIPEDI A . Global Offset Table [EB/OL]. (2024-09-25) [2024-06-20]. https://en.wikipedia.org/wiki/Global_Offset_Table.

[17] AGENCY N S. Ghidra[EB/OL]. (2024-06-14) [2024-06-20]. https://github.com/NationalSecurityAgency/ghidra.

ZHANG Guanghua, born in 1979, Ph.D, professor, master supervisor, is a member of CCF(No. 51334S). His main research interest is network and information security.

HU Boning, born in 1978, master, lecturer. Her main research interest is communication network security.

（责任编辑：喻藜）