Computers & Security 126 (2023) 103068

Contents lists available at ScienceDirect

Computers & Security

journal homepage: www.elsevier.com/locate/cose

VERI: A Large-scale Open-Source Components Vulnerability Detection in IoT Firmware

Yiran Cheng[a,b], Shouguo Yang[a,b], Zhe Lang[a,b], Zhiqiang Shi[a,b,*], Limin Sun[a,b]

[a] Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China
[b] School of Cyber Security, University of Chinese Academy of Sciences, China

ARTICLE INFO

Article history:
Received 17 July 2022
Revised 8 December 2022
Accepted 15 December 2022
Available online 17 December 2022

Keywords:
IoT Firmware
Open-source component
Vulnerability detection
Version identification

ABSTRACT

IoT device manufacturers integrate open-source components (OSCs) to serve necessary and common functions for facilitating firmware development. However, outdated versions of OSC conceal N-day vulnerabilities and continue to function on IoT devices. The security risks can be predicted once we can identify the OSC versions employed in the firmware. Existing works make attempts at OSC version identification but fail to perform vulnerability detection on a large-scale IoT firmware due to i) unsuitable version identification method for IoT firmware scenario. ii) the lack of a large-scale version-vulnerability relation database. To this end, we propose a system VERI for large-scale vulnerability detection based on lightweight version identification. First, for OSC version identification, VERI leverages symbolic execution with static analysis to identify exact OSC versions even though there are many version-like strings in OSC. Second, VERI employs a deep learning-based method to extract OSC names and vulnerable version ranges from vulnerability descriptions, constructs and maintains an OSC version-vulnerability relation database to serve the vulnerability detection. Finally, VERI polls the relation database to confirm the N-day security risk of the OSC with identified version. The evaluation results show that VERI achieves 96.43% accuracy with high efficiency in OSC version identification. Meanwhile, the deep learning model accurately extracts the OSC names and versions from vulnerability descriptions dataset with 97.19% precision and 96.56% recall. Based on the model, we build a large-scale version-vulnerability relation database. Furthermore, we utilize VERI to conduct a large-scale analysis on 28,890 firmware and find 38,654 vulnerable OSCs with 266,109 N-day vulnerabilities, most of which are with high risks. From the detection results, we find that after the official patch for the vulnerability is released, manufacturers delay an average of 473 days to patch the firmware.

1. Introduction

With the explosion of the Internet-of-Things (IoT), the number of embedded devices, from appliances to smart homes and personal gadgets, has proliferated dramatically. While the benefits of IoT devices can be observed everywhere, their inherent vulnerabilities do create new security risks (Alrawi et al., 2019). These vulnerabilities leave devices open to cyberattacks, which can disrupt industries and economies in dangerous ways. For example, the Mirai botnet compromised millions of IoT devices in late 2016 and overwhelmed targets with massive distributed denial-of-service (DDoS) attacks (Antonakakis et al., 2017).

For facilitating development, IoT device manufacturers use open-source component (OSC) to serve necessary and common functions running on the firmware. From perspective of component, firmware is an integrated component package programmed on IoT device. However, vulnerabilities in the OSC can be introduced into IoT devices and make them vulnerable. For instance, Heartbleed compromised millions of devices and leveraged them in buffer over-read attacks to leak sensitive information, which was reported in OpenSSL (CVE, 2022; Hea, 2021; Ope, 2021).

To address these security issues faced by IoT devices, manufacturers provide new firmware releases for updates, including patches for known OSC vulnerabilities. However, due to the various types of fragmented devices, it is difficult for manufacturers to generate patches timely for each type of device (He et al., 2022; Niesler et al., 2021). Even if patched firmware is released, end users may neglect to update the firmware to mitigate the vulnerabilities. As a result, attackers conduct N-day vulnerability attacks, which are made easier by leveraging patch information (Wang et al., 2019). A 2021 audit report from Synopsys (Bla, 2021) mentioned that more than 65% of manufacturer-issued firmware contains re-

* Corresponding author.
E-mail address: shizhiqiang@iie.ac.cn (Z. Shi).

https://doi.org/10.1016/j.cose.2022.103068
0167-4048/

curring OSC vulnerabilities, many of which were disclosed years ago.

Because N-day vulnerabilities exist in specific OSC versions, identifying the OSC versions allows the security risk to be predicted. Many binary version identification approaches have been proposed (Almanee et al., 2019; Duan et al., 2017; Zhan et al., 2021). However, they present their own set of limitations when applied to OSC version identification in IoT firmware. i) In firmware scenario, OSCs in different architectures and compiler environment can result in significantly different control-flow-related and opcode-related features. However, ATVHUNTER (Zhan et al., 2021) extracts the Control-Flow Graph (CFG) and opcode sequences as the features to identify the version. The application of these features leads to a significant performance loss. ii) OSSPolice and ATVHUNTER (Duan et al., 2017; Zhan et al., 2021) aim to identify the third-party native library and the librarys version in Android apps. These tools are based on Android's library dependency relations, which cannot meet our research scenarios for IoT firmware and their feature extraction methods are incompatible with IoT firmware.

Once the versions of OSCs are identified, the security risk can be predicted by querying the vulnerability databases such as National Vulnerability Database (NVD) (NVD, 2021). The Common Platform Enumeration (CPE) format is used by NVD to store information about vulnerable versions. However, the CPEs in certain CVE are missing or inconsistent (as shown in Section 2.1) with the actual vulnerable version range, which may cause major delays in patching. We examined all CVEs from 2016 to 2018, there are 271 CPEs missing partial or even total vulnerable versions which include frequently-used OSCs.

We summarize the challenges of OSC vulnerability detection in IoT firmware as follows:

- **C1: Multiple version-like strings exist in the same OSC.** Plenty of OSCs include multiple string literals which "look like" real version string (e.g., strings 1.0.0, 1.0.1, 1.0.1*d*, and 1.0.2*g* contained in OpenSSL 1.0.2g at the same time). Therefore, simply using regular-expression matching will not find the real version string to identify the OSC version.
- **C2: Dynamic method is not scalable in IoT scenario.** We assume that running the OSC with specific parameter can produce the OSC version. However, due to the diverse CPU architectures and peripherals of IoT devices, these OSCs are very difficult to run without the devices. Even though QEMU (QEM, 2021) can conduct simple emulation of OSCs, it still cannot perform on a large scale (Zheng et al., 2019).
- **C3: Constructing version-vulnerability relation database.** Due to the inconsistency between the CPE and the actual vulnerable version range (as described in Section 2.1), there is a need to construct a precise version-vulnerability relation database. To this end, we need a method to automatically and accurately extract vulnerable versions from the vulnerability reports.

To address these challenges, we propose VERI, an N-day vulnerability detection system for large-scale IoT firmware OSCs. VERI leverages lightweight symbolic execution with static analysis to run OSC and identify its version (to solve the challenge **C2**). Specifically, it first recovers the whole control flow of the target OSC which consists of a Control-Flow Graph (CFG) and a Call Graph (CG). Next, it locates the entry point and the version-output point to execute the OSC between these two points. To this end, VERI determines the start block of *main* function with its input arguments as entry point, and it collects the version-like string literals and locates the basic blocks which read them as the *candidate version point*. Then it conducts CFG pruning with Depth First Search (DFS) algorithm and static taint analysis to find possible feasible paths to the version point. Finally, it leverages an input-driven symbolic execution to remove infeasible paths ended with wrong version-output point (to solve challenge **C1**), and output the real version string.

VERI maintains an OSC version-vulnerability relation database to check if the OSC version is vulnerable. Since vulnerability description text is unstructured, we utilize a novel entity-relation extraction model, which learns the patterns from the description structures to recognize OSC version range (to solve challenge **C3**). More specifically, given a vulnerability description, we consider the vulnerable version range extraction as two tasks: (i) locating the entities (OSC names and versions) and (ii) extracting relations between the name and version. To consider the information interaction between the two tasks, we take a joint neural network model to perform end-to-end relations extraction. Finally, we convert the relations into unified version range and construct the version-vulnerability relation database.

We implement VERI and evaluate it in OSC version identification and entity-relation joint extraction model, respectively. To evaluate OSC version identification in different compilation settings, consisting of different instruction set architectures (ISA) and optimization levels. We compile open-source projects in different ISAs to build a Cross-ISA dataset with 1044 OSCs. VERI achieves 96.5% accuracy on the dataset for the version identification task. Besides, we build a Cross-optimization level dataset with 1252 OSC binaries and VERI achieves 96.6% accuracy. To evaluate the entity-relation joint extraction model, we collect a large dataset from NVD which covers 23,410 vulnerabilities and their descriptions. Among them, we manually label 3822 descriptions for model training, validating and testing. We show our model achieves high accuracy with 97.19% precision and 96.56% recall in OSC entity recognition, 94.29% precision and 91.86% recall in vulnerable version relation extraction.

We also conduct a large-scale OSC version identification on real-world firmware. Combined with the version-vulnerability relation database, we detect 38,654 vulnerable OSCs in 28,890 firmware, and each Mb binary takes a very short time of 0.572s on average. Furthermore, We gain an interesting insight into firmware security, which is during the firmware iteration, manufacturers do not update vulnerable versions in time for OSCs with 473 outdated days on average.

The main contributions of this work are as follows:

- We construct a large-scale IoT firmware dataset including 28,890 IoT firmware with 31 kinds from 46 IoT device manufacturers, which is representative to understand IoT security.
- We propose a novel system VERI for N-day vulnerability detection in IoT firmware. VERI leverages a lightweight symbolic execution with static analysis for OSC version identification and introduces an end-to-end neural network model to extract the vulnerable version range from unstructured vulnerability descriptions. Both show high accuracy.
- VERI reports the discovery of 38,654 vulnerable OSCs in 24,845 firmware, with 524 distinct CVEs detected. According to the results, we reveal the exploit type and severity of the vulnerabilities. We conduct further analysis on the firmware and find that OSC was outdated by 473 days on average despite the firmware having been published new release.

2. Background

This section aims to explain our motivations and the assumption.

We illustrate our insights for OSC version identification using a real-world OSC *tcpdump*. *Tcpdump* is a freely available, powerful packet analyzer (tcp, 2022) which is widely used in firmware.

Table 1
The case study of vulnerable version inconsistencies problem. The list illustrates the vulnerability information reported in NVD including CVE ID, OSC name, vulnerable version range of CPE (CPE version range), vulnerable version range described by CVE description (CVE description version range), missing version of CPE compared to CVE description (CPE missing) and the true version range in our manually test (manually test).

CVE ID	OSC	CPE Version Range	CVE Description Version Range	CPE Missing	Manually Test
CVE-2008-5077	OpenSSL	<0.9.8i	0.9.8i and earlier	0.9.8i	≤0.9.8i
CVE-2009-1232	Firefox	3.0–3.0.8	3.0.8 and earlier 3.0.x versions 3.0.10 and earlier (Additional Note)	3.0.9, 3.0.10	3.0–3.0.10
CVE-2011-4089	bzip2	<1.0.5	1.0.5 and earlier	1.0.5	≤1.0.5
CVE-2015-3274	moodle	2.6-2.6.10, 2.7-2.7.8, 2.8-2.8.6, 2.9	through 2.6.11, 2.7.x before 2.7.9, 2.8.x before 2.8.7, and 2.9.x before 2.9.1	2.6.11	2.6-2.6.11, 2.7-2.7.8, 2.8-2.8.6, 2.9
CVE-2015-3419	vBulletin	5.0–5.1.3	5.x through 5.1.6	5.1.4–5.1.6	5.0–5.1.6
CVE-2016-3076	pillow	2.5.0–3.1.0	2.5.0 through 3.1.1	3.1.1	2.5.0–3.1.1
CVE-2016-4477	wpa_supplicant	null	0.4.0 through 2.5	0.4.0–2.5	0.4.0–2.5
CVE-2017-9434	cryptopp	≤5.6.4	through 5.6.5	5.6.5	≤5.6.5
CVE-2018-10771	abcm2ps	≤6.3.9	through 8.13.20	6.3.10-8.13.20	6.3.10-8.13.20
CVE-2019-11445	openkm	6.3.2–6.3.6	6.3.2 through 6.3.7	6.3.7	6.3.2–6.3.7

We compiled *tcpdump* 4.9.0 source code (with compiler gcc 4.7.1 and optimization option O0) to get the OSC binary and take the partial code in Listing as motivation example. OSCs typically contain their version string literals, which are referred to by their version output function. Such a function can be invoked by running the OSC with a specific parameter so that user can get the OSC versions to determine the OSC update. For example, the function *print_program_info* in line 24 of Listing is used to print version information.

However, there are multiple version-like strings exist in the OSC, which has followed problems. First, the patterns and contexts of the string literals are so similar, it is challenging to distinguish the true version string from these version-like strings using regex-expression matching. For example, the true version string of OSC *tcpdump* 4.9.0 is referenced in Line 26. But similarly, the string of other old OSC *tcpdump* 4.7.4 is also referenced in Line 18. Meanwhile, string 1.7.0 of an integrated OSC *libpcap* is printed in function *pcap_find* (Line 30). Second, since code of these version strings is possible to be executed, classic static analysis techniques like data flow analysis cannot be used to prune them. For example, the print code of 4.9.0, 4.7.4 and 1.7.0 string literals are reachable when *user_input* takes the values 118, 187 and 68 respectively. We find that the value of *user_input* is controlled by the argument *argv*, which stores user-specified command option. Furthermore, we find that many paths lead to *fake* version string output code are not determined by the user-specified command option. Inspired by this observation, we leverage the user-specified version print command to prune code execution paths and execute real version print function.

2.1. Vulnerable version range inconsistent

We examined all CVEs from 2016 to 2018, 271 CPEs are missing partial or even total vulnerable versions which include frequently-used OSCs such as *OpenSSL*. CPE records the software name and versions the vulnerability affected. We select 10 distinct CVE reports with inconsistency problem as motivation examples as shown in Table 1, by crawling their CPEs and vulnerability descriptions from the NVD (NVD, 2021). For the affected OSCs in these reports, we collect all versions of them from Github (git, 2022). According to the vulnerable and patched code described detail in the reports, we manually confirm the actual vulnerable versions (ground-truth) by comparing the target version code with vulnerable version and patched version code. Naturally, by comparing the version range of CPE and description with the ground-truth, we find out all inconsistent versions missed by CPE, as displayed in the CPE Missing column. Across the 10 vulnerabilities in Table 1, we find that $Set_{cpe} \subsetneq Set_{des} = Set_{truly}$ (Set_{cpe}, Set_{des}, Set_{truly} denote the vulnerable version range in CPE, CVE description and the manually test). In conclusion, we discover that CVE descriptions always accurately describe the vulnerable version range, whereas the CPE misses vulnerable versions.

The certificate parser in OpenSSL before 1.0.1u and 1.0.2 before 1.0.2i might allow remote attackers to cause a denial of service (out-of-bounds read) via crafted certificate operations, related to s3_clnt.c and s3_srvr.c.
Publish Date : 2016-09-26 Last Update Date : 2021-11-17

Fig. 1. CVE-2016-6306 vulnerability description.

Given the fact we discovered, we can extract vulnerable version ranges from CVE descriptions. A large number of existing Natural Language Processing (NLP) studies are used to extract information from text (Etzioni et al., 2008; Lin et al., 2020; Mausam, 2016). However, their application in vulnerability description has some limitations. The following are the primary reasons: i) CVE description text is unstructured, so the regular expression-based methods are not effective for it. ii) A CVE description often contains multi-head relations, where multi-head relation denotes multiple versions in different relations with the OSC name. For example, as shown in Fig. 1, CVE-2016-6306 description contains i) OSC Name: OpenSSL (red line), ii) three vulnerable version relation: before 1.0.1u and 1.0.2 before 1.0.2i (blue line). There are three relation pairs for *OpenSSL*, the first pair: P_1 = *(OpenSSL, 1.0.1u)*, the second pair: P_2 = *(OpenSSL, 1.0.2)*, the third pair: P_3 = *(OpenSSL, 1.0.2i)*. The relation in P_1 is *"before"*, the relation in P_2 is *"start from"* and the relation in P_3 is *"end without"*. Previous studies (Adel and Schütze, 2017; Gupta et al., 2016; Zheng et al., 2017) feed one entity pair (e.g. (OpenSSL, 1.0.1u)) at a time to a neural network for relation extraction, thus they do not simultaneously infer other relations within the same sentence. This highlights the importance of multi-head relations extraction in a single description.

2.2. Assumption

Due to the frequent iteration of open-source components, OSCs often contain current version information in the binary for developer validation. This work is based on the assumption: OSC contains its version information in the binary. For example, *tcpdump* in the Listing 1 contains its version string 4.9.0.

3. Overview

In this section, we describe the high-level architecture of VERI for detecting N-day security risk of OSC in firmware. The overview of our system VERI is shown in Fig. 2, which consists of three parts.

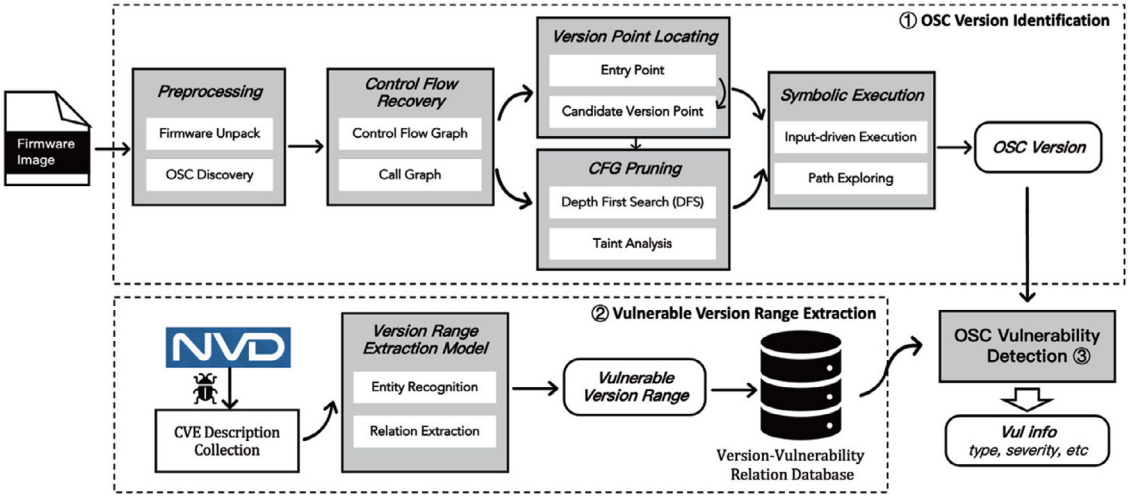


Fig. 2. The Overview of VERI.

VERI's input is an entire firmware image and output is the vulnerable OSCs of firmware and its N-day security risks (vulnerabilities).

OSC Version Identification (part ①). VERI identifies OSC version automatically using a lightweight symbolic execution with static analysis. We unpack the firmware image and then discover all possible OSC from it as the object for subsequent analysis. Given the OSC, we recover the Control Flow Graph (CFG) and the Call Graph (CG). Then we locate the entry point and *candidate version point*. Based on the CFG and point, we leverage a two-phase traverse policy including Depth First Search (DFS) and static taint analysis, to remove irrelevant nodes and edges from CFG. Finally, we use input-driven symbolic execution to reason whether the path from entry point to *candidate version point* is reachable. If the *candidate version point* is reachable, the corresponding string literal is considered as true version.

Vulnerable Version Range Extraction (part ②). To address the aforementioned challenge C3, we collect vulnerability descriptions widely from NVD. Then we train an entity-relation joint extraction model to recognize vulnerable version range from the descriptions, where entity represents OSC name and version, and the relation stands between them. Finally, we convert the extracted information into unified version range and maintain a version-vulnerability relation database.

OSC Vulnerability Detection (part ③). After identifying the OSC version, VERI polls the version-vulnerability relation database (from part ②) to confirm whether the OSC version (from part ①) is contained in the vulnerable version range, reporting the vulnerabilities of the OSC and security risks of the IoT firmware.

4. OSC version identification

This section presents our design and implementation of the OSC version identification method.

4.1. Firmware preprocessing

VERI first unpacks the firmware image using the off-the-shelf firmware unpacking utility binwalk (bin, 2021). Binwalk can extract filesystems from firmware, such as SquashFs, JFFS2, YAFFs, etc. We discover the OSCs from the extracted filesystem through an OSC white list. To construct the white list, we collect all possible OSC from integrated components of IoT Device SDKs, e.g., AWS IoT SDKs, and open-source components list provided by manufacturers (nuk, 2022).

4.2. Control flow recovery

To help our analysis, we recover two graphs of OSC, Control-Flow Graph (CFG) and Call Graph (CG). CFG shows all possible execution paths in the OSC. In the CFG, nodes are basic blocks and edges are transfers of control. We use IDAPython provided by IDA Pro (ida, 2022) to build the CFG for the basic blocks of each binary function. CG is a directed graph which represents the function and their calling relationships in a binary (Ryder, 1979). We extract function information with IDAPython, and connect the block-block with *call* relation in CFG to create the CG for the functions of binary.

4.3. Version point locating

The binary execution requires locating the entry point and the *candidate version point*.

Entry Point. OSCs usually output version information when running with a specific parameter, such as '-v'. However, we cannot simply run the target OSC with the parameter based on symbolic execution. Running OSC from the start will execute through shared library constructors or preinitializers, which is heavy for symbolic execution. So we locate function *main* as our entry point to avoid the initializers. To feed the parameter to the function *main*, we need set values to particular memory location. The parameters of function *main* are filled with i) *argc*, an integer specifying how many arguments are passed to the binary from the command line, ii) *argv*, an array of pointers to char, the first string (*argv*[0]) is the binary name, and each following string is an argument passed to the binary from the command line. We set the stack memory of *argc* to the number of parameters, and *argv* to string array containing specific parameters. We summarize such candidate arguments as 'v, V, version, h, help'. In conclusion, we set concrete value to the parameters of function *main*: $argc = 2$, $argv = [binary\_name, candidate\_argument]$.

Candidate Version Point. String literals are stored in the data section (e.g, *.rodata*) in binary. During compilation, the compiler removes duplicate strings to ensure that only one copy of string exists in the section. We collect candidate version strings by regular-expression matching from data section. The program retrieves the string literal from data section in two ways: i) instructions read the memory address of *.rodata* section directly, ii) the global variables stored in *.data* section are initialized and assigned the memory address of *.rodata* section, and instructions read the global variable to get the string literal. Based on these two ways,

we locate the instructions which read the candidate string literals, e.g., instruction *mov eax*, *off_8055120*, *off_8055120* is the global variables in *.data* section and assigned the memory address of *.rodata* section, where store the string literals 4.9.4. Furthermore, from the generated CFG of executable binary, we locate the basic blocks to which the instructions belong as the *candidate version point*.

Based on the CG, we extract the Partial Call Graph (PCG) from the function where the entry point is located to the function to which the *candidate version point* belongs.

4.4. CFG Pruning

Symbolically executing entire binary is inefficient due to the path exploration and time-consuming constraint solving (Stephens et al., 2016). Instead of executing entire binary, we focus on the portion of binary code that is relevant to binary version output, more specifically, the execution path for version output. We isolate this portion by creating a pruned CFG, which is realized in two-phase traverse policy, Algorithm 1 shows the method to prune CFG.

Algorithm 1: CFG Pruning.

```
   Input  : binary B, entry point ep, candidate version point cvp
            control flow graph CFG, partial call graph PCG
   Output: pruned CFG
1  Function CFGPruning(B, CFG, PCG, ep, cvp):
2      path ← DFS(CFG, ep, cvp)
3      accessible_path ← topological_sort(path)
4      for each block ∈ Accessible_path do
5          bl_IRs ← to_vexir(block)
6          for each ir ∈ bl_IRs do
7              r_node, w_node, op ← extract(ir)
8              E ← E ∪ create_dependency(r_node, w_node)
9              if r_node is taint_node and op is Put or Store then
10                 taint_nodes.append(w_node)
11             end
12         end
13         if bl.jumpkind = "call" then
14             succ_bl, succ_func ← get_succ(bl, CFG)
15             if succ_func ∉ PCG and not
               if_relative_caller(succ_func.args, taint_nodes) then
16                 pruned_cfg ← make_fake_ret(succ_func)
17         end
18     end
19     return pruned_cfg
```

Depth First Search. In the first phase, we apply Depth First Search (DFS) on the CFG to traverse all possible paths between entry point and *candidate version point* (line 2). Then, we prune the nodes and edges of CFG that are not involved in paths. To solve the loop case, we use topological sort to find loops in the path and remove the last edge in the loops (line 3).

Taint analysis. To skip argv-irrelevant and costly path prefix (e.g., the init steps in Listing 1 Line 2) in the paths, we perform static taint analysis. The taint analysis tags the taint function argument at the entry point and tracks the data-flow of the argument. According to the taint propagation with data dependencies, we determine the irrelevant callers and make a pruning for them to reduce time consumption. To conduct taint analysis for each current block in CFG (line 4–18), we lift the binary executables in various architectures to the unified representation VEX IR (Nethercote and Seward, 2007).

The first step is to precisely tag the taint sources, which are the specific arguments of functions (i.e., the second argument *argv* of *main* function). Binary executables with different architectures lead to various calling conventions and stack layouts, while the unified IR code losts the features. For that, we check the architectures of firmware and apply the corresponding calling conventions. The arguments passing rules fall into two situations: i) When the argument is passed with the register, the register is tagged as taint source. ii) For the argument passing based on the stack, we first determine the caller's stack range by the value of stack and based pointer, then we offset the callers stack pointer to locate the argument address in the stack, eventually we tag the corresponding memory location of argument as taint source. For each instruction in the basic block, we extract the read node, write node and operation *op* between them. Memory is accessed through *Store* and *Load* VEX IR operations. *Put* and *Get* operations specify data for writing to and reading from registers. If the *op* is *Put* or *Store*, we build a data dependency between the read node and write node. In the meantime, we dynamically update the list of taint nodes according to the dependency.

To analyze the inter-procedure calls, we check whether taint node is used as the argument of the successive function, if not, the caller is considered unnecessary and then pruned in CFG. The check method is similar to taint source location. Meanwhile, to avoid pruning the inevitable caller, the caller in PCG is reserved by default.

4.5. Symbolic execution

Given the right version-trigger argument, there is a particular module for argument analysis in OSC. To ensure that the argument can be accurately parsed, we develop an argument analysis module. After analyzing the arguments, we utilize the correct argument to direct the execution to version-output code based on symbolic execution, which we call OSC execution module.

Argument Analysis Module. We manually examine the method for parsing parameters in 138 binaries, which fall into two main types, i) 67 binaries parse the parameters through *getopt*-related function. *getopt* is a well-known function in standard C library (get, 2022), ii) 71 binaries parse parameters through customized code. Rather than symbolically execute the complex *getopt*-related functions in standard C library routines which is expensive, we introduce function hook module to automatically hook and replace them with a simplified routine that executes concretely, to complete the expected functionality.

OSC Execution Module. With the pruned CFG and function hooks, we use symbolic execution to reason which *candidate version point* is reachable. First, we set the symbolic state of the entry point with specific function input as mentioned in Section 4.3. Then VERI executes each block, and solves the path constraints according to the concrete value of each symbolic variable. For example, if the block path constraint is *"$a > 20$"*, the symbolic variable *a* satisfying the condition is resolvable and the path is feasible. On the contrary, if *a* conflicts the condition, the path is unfeasible. We select the feasible path and update the symbolic state, continue to solve the constraint on the new path. The *candidate version point*, which print the real OSC version, is reached and executed during the path execution. We collect the strings printed by standard program output. Then version string can be parsed from the collected string. The *fake candidate version points* will be unreachable in two cases: i) The binary exits execution in advance without stepping into it. ii) We set the upper execution time, the *candidate version point* is not stepped into within the time.

Running example. We now present a running example to give a brief overview of our modular analysis. Listing shows our running example. Our goal here is to figure out the real version from the candidate version strings 4.7.4, 4.9.0 and 1.7.0 respectively in line 18, 27 and 32. i) We first construct the CFG and CG of binary *tcpdump*. ii) The entry point is the first block with concrete input $argc = 2, argv = [\text{"}tcpdump\text{"}, \text{"}V\text{"}]$, the *candidate version point* are the blocks where the candidate version strings are located (ba-

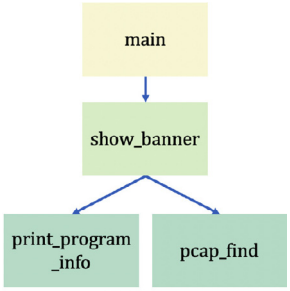


Fig. 3. PCG paths in running example.

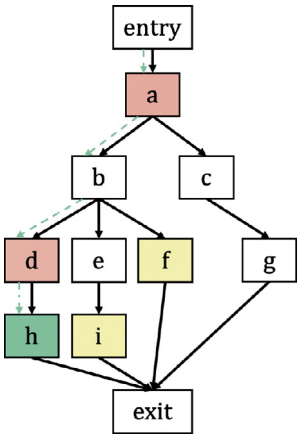


Fig. 4. Partial CFG of running example.

sic block *f*, *i* and *h* in Fig. 4). There are three candidate PCG paths shown in Fig. 3: *main* → *show_banner*, *main* → *show_banner* → *print_program_info* and *main* → *show_banner* → *pcap_find*. iii) We apply DFS on the CFG to get the accessible path between entry point and *candidate version point*, then the nodes and edges outside the path are removed (e.g., node *c*, *g* and their edges in Fig. 4). The argument *argv* of *main* function is set as the taint source. Through taint propagation and analysis, the argv-irrelevant function is pruned (e.g., node *a* and *d* in Fig. 4) and the *argv*-relevant function *parse_program* and *getopt_long* is reserved. Meanwhile, the function *print_program_info* is in PCG, which is reserved in CFG by default though it is *argv*-irrelevant. iv) Finally, we hook the *getopt_long* function and replace it with a simplified routine, then use symbolic execution to confirm whether the *candidate version point* is reachable. The *user_input* is assigned with concrete value *"V"* after the execution of function *getopt_long*. In the path constraint of line 14, the *case* 118 path (green line in Fig. 4) is feasible and eventually the block in *print_program_info* is the reachable block, 4.9.0 is confirmed as OSC version.

Implementation. We implemented the version identification module with 2337 lines of Python code. Apart from that, we integrated the APIs of existing open-source projects. Especially, Binwalk (bin, 2021) is utilized to unpack firmware images. IDAPython provided by IDA Pro (ida, 2022) is used to create CFG and CG of binary. Python bindings PyVEX (pyv, 2022) lift binary to the unified representation VEX IR (Nethercote and Seward, 2007). Angr (ang, 2022) is adopted as our symbolic execution engine.

5. Vulnerable version range extraction

As shown in Fig. 5, the input of our model is a sequence of words (tokens) from vulnerability description, which are then represented as word vectors through word embeddings. Then we combine the bidirectional sequential LSTM (BILSTM) with the self-attention to extract a more complex representation for each word with context. The softmax layer produces the outputs for entity recognition task: (i) entity labels (e.g., the corresponding output of token *OpenSSL* is B-COM, B denotes the first token in entity, and COM denotes token as an OSC name type). The sigmoid scoring layer produces the outputs for relation extraction task: (ii) a set of tuples comprising the last token of the entity and the relations between them (e.g., the output of token *OpenSSL* is (*1.0.1u, before*), which denotes the relation between *OpenSSL* and *1.0.1u* is *before*).

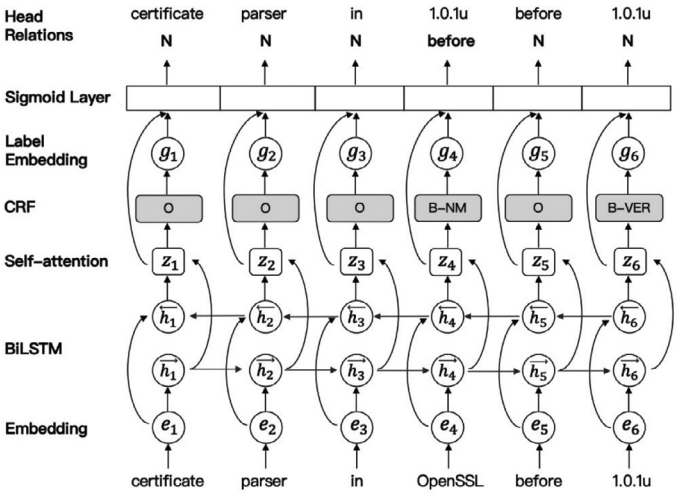


Fig. 5. Entity-Relation Jointing Extraction Model.

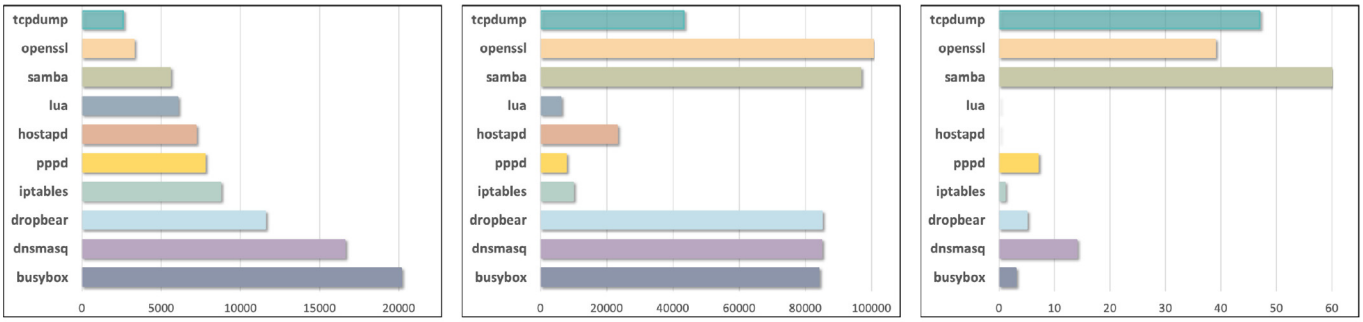
5.1. Word and character embedding

The input sentence from vulnerability description consists of n words $w = [w_1, \ldots, w_n]$, each word w_i is represented by a word vector E_i in the word embedding layer. To be specific, the model utilizes character-level embeddings (Ma and Hovy, 2016) to treat the word as a characters sequence (e.g., word *OpenSSL* corresponding to a sequence [O,p,e,n,S,S,L]). The character-level vectors are concatenated to the word-level vector E_i, which are input to the BiLSTM layer.

5.2. Bilstm encoding and self-attention

Long short-term memory (LSTM) is an artificial recurrent neural network (RNN) architecture widely used in NLP tasks. BiLSTM consists of two LSTMs, which are able to encode sentences from left to right and right to left. For each token, BiLSTM layer is able to link forward $\overrightarrow{h_i}$ and backward $\overleftarrow{h_i}$ in every timestep i, which is represented as: $h_i = [\overrightarrow{h_i} \oplus \overleftarrow{h_i}], i = 0, 1, \ldots, n$. Therefore, BiLSTM can exploit the bidirectional information of the word and the output of it is $H = \{h_1, h_2, \ldots, h_n\}$.

Because long-distance related words are not easily associated, we introduce self-attention after the BiLSTM. Self-attention calculates the attention score between each word and all the other words to obtain the relevancy of them, which better relates context. The output vector H of the BiLSTM layer is input to the Self-attention layer. When dealing with the query word, we need to calculate the attention score of all the key words in the sentence (including the query word itself) to see how relevant it is. z_i represents the output vector for each word in self-attention, $z_i = Attention(W_i^Q H, W_i^K H, W_i^V H)$. So for each word, we create a query vector ($W_i^Q H$), a key vector ($W_i^K H$), and a value vector ($W_i^V H$). where W^Q, W^K, W^V respectively represent weight matrices that we trained during the training process. Finally we concentrate the



(a) The number of CVE in Top 10 OSCs. (b) The number of CVE in affected firmware by Top 10 OSCs. (c) The number of CVE whose CVSS is high and critical in Top 10 OSCs.

Fig. 6. The number of CVE, CVE in affected firmware and CVE whose CVSS is high and critical in Top 10 OSCs.

z_i matrices then multiply with a weight W^O to produce the output of the self-attention layer, the output $Z = W^O[z_1, z_2, \ldots, z_n]$.

5.3. Entity recognition

Entity Recognition aims to identify the OSC names and versions from the descriptions. We consider the entity recognition task as a sequential labeling problem, using the BIO (Begin, Inside, Outside) encoding scheme to make a tag for each token. As shown in Fig. 5, we have two entity types: NM (OSC name), VER (OSC version). We assign B-type tag to the first token of an entity, I-type tag to the other tokens of an entity, and if token does not fall into the entity types above, it will be assigned to O-type tag. In Fig. 5, we assign *B-NM* tag to "OpenSSL", *B-VER* tag to "1.0.1u". We use the Softmax function to calculate which entity tag is the most possible. First, we employ the softmax function to calculate the probabilities of all the candidate tags for each token the:

$$Probability(tag|w_i) = Softmax(kf(m \cdot z_i + b)) \quad (1)$$

where f is an activated function, $k \in \mathbb{R}^{t\times w}$, $m \in \mathbb{R}^{w\times 2d}$, $b \in \mathbb{R}^w$. t is the number of entity tags, w is the layer width, d is the size of LSTM hidden layer. Finally, we select the entity tag with the highest value as the output.

5.4. Relation extraction

The task of relation extraction is to predict relations between OSC names and OSC versions. The relations are denoted as tuples (v_i, r_i) for each OSC name token w_i, where v_i is the version token vector and r_i is the relation token vector. Meanwhile, we treat the relation extraction task as a multi-head problem, which means multiple relations with one token, as mentioned in Section 2.1. The inputs x_i of next layer: Sigmoid layer, include (i) the output state of encoding layer and (ii) the output representations of entity label (output by the Entity Recognition task) embeddings g_i. The input $x_i = [z_i \oplus g_i], i = 0, 1, \ldots, n$. Given the token sequence w and the relational label R, we calculate the score of token w_i and token w_j over relation r_m:

$$Score(x_i, x_j, r_p) = kf(m \cdot x_j + n \cdot x_i + b) \quad (2)$$

here, f is an activated function, $k \in \mathbb{R}^w$, $m \in \mathbb{R}^{w\times(2d+p)}$, $n \in \mathbb{R}^{w\times(2d+p)}$, $b \in \mathbb{R}^w$. p is the size of label embeddings, w is the layer width, d is the size of LSTM hidden layer. Then we use sigmoid function to define the probability of token w_i and token w_j over relation r_m:

$$Probability(w_j, r_p|w_i) = Sigmoid(Score(x_i, x_j, r_p)) \quad (3)$$

We set the threshold to get to the final relation tuples. If the probability of relation r_m between w_i and w_j exceeds the threshold, we confirm that the relation r_m exists.

Table 2
Unified Version Range Conversion.

Relation tuple	Version range
(v, through)	[*oldest version*, v]
(v, in)	[v, v]
(v, before)	[*oldest version*,v)
(v, after)	(v, *latest version*]
(v, and earlier)	[*oldest version*, v]
(v, and later)	[v, *latest version*]

5.5. Version range unification

We apply the jointing extraction model to all descriptions to get the entities and relations. Then we use these results to build a version-vulnerability relation database. Due to the diversity of words representing relations, we unify the relation type before building the database. We summarize the relations into two types *start* and *end*. For example, in the description "OpenSSL before 1.0.1u" we can get the relation tuple for "OpenSSL" is (1.27.2, *before*), which means the version up to and including 1.27.2 is vulnerable. We convert this tuple into unified version range form (*oldest version*, 1.27.2), simultaneously we record whether the border version 1.27.2 is not contained and the *oldest version* is contained. The unified version range conversion is listed in Table 2. In conclusion, a unified vulnerable version range entry is composed of CVE number, vulnerability description, OSC name, start version, end version, and border version is contained or not.

Model Implementation. We use 200-dimensional word embeddings. Our training is performed using the Adam optimizer (Kingma and Ba, 2014) which calculates the adaptive learning rate of each parameter. We also use dropout to normalize our network and prevent overfitting (Srivastava et al., 2014).

6. Evaluation

In this evaluation, we aim to answer following research questions:

RQ1: What is the accuracy of VERI for OSC version identification?

RQ2: What is the quality of the vulnerable version range extraction model in VERI for constructing database?

RQ3: How practical is VERI to detect N-day vulnerabilities in large-scale firmware? What about the outdatedness of the OSC in firmware?

We implement VERI in Python with 7038 lines of code. All the evaluations are conducted at ubuntu server with 56 cores CPU of Intel Xeon E5-2697@2.6GHz and 256 GB memory.

Table 3
The list of 9 open-source components (313 different versions) used to evaluate the version identification accuracy in different optimization levels.

OSC Name	OSC Versions
openssl	1.0.1, 1.0.1a-1.0.1u, 1.1.0, 1.1.0a-1.1.0l
ffmpeg	3.0.1–3.4.8
sqlite3	3.8.8.3, 3.8.9, 3.19.3-3.23.0
freetype	2.4.0–2.5.5, 2.7.1, 2.8.0–2.8.1
openssh	1.2.1pre18-1.2.1pre21, 3.2.2p1-3.6p1, 4.6p1-8.4p1
tcpdump	3.5.2–4.8.0, 4.9.0–4.9.3
binutils	2.10-2.14, 2.22-2.35.1
openvpn	2.3.4-2.4.4
libexpat	2.1.0–2.2.1

6.1. Dataset preparation

We prepare three different datasets for VERI: open-source component dataset, vulnerability description dataset, and firmware dataset.

1) The open-source component dataset contains thousands of open-source project binaries with known versions and is used for testing the accuracy of version identification of VERI.

2) The vulnerability description dataset is used for model training, model testing, and version-vulnerability relation database construction.

3) The firmware dataset collects thousands of firmware and is used to test the practicality of VERI.

Open-source Component Dataset. OSCs in the dataset all comply with the assumption in 2.2, i.e., all OSCs contain version information. The dataset consists of two subsets: i) a typical OSC dataset in Cross-optimization (Table 3) and ii) a Cross-ISA dataset. The different two sets are used for testing the version identification accuracy in typical binaries and different compiler environments. i) We selected open-source projects in different versions shown in Table 3 and compiled these projects with four different compiler optimization levels: O0, O1, O2, and O3. As a result, we obtain 1252 OSC binaries. ii) Cross-architecture dataset contains 87 different open-source projects. We compile these projects with three ISA compilers to obtain 1044 OSC binaries. Specifically, three ISAs include x86, ARM, x64.

Vulnerability Description Dataset. We crawl vulnerability description entries from NVD. Specifically, 23,410 description entries are collected from NVD, which cover 586 kinds of open-source components, with vulnerabilities spanning from November 1988 to August 2021. We take 3822 descriptions with a ratio of 6:2:2 as training dataset, validating dataset, and test dataset. For the ground truth, we first automatically label the OSC name entity and version entity in vulnerability descriptions through string regex matching. Then we manually check for unlabeled entities and label them, we also manually label the relation between the entities. The ground truth labels include (i) two entity types: *name, version* and (ii) six relations types: the relations between name and version include *before, through, in, and earlier, start, end*.

Firmware Dataset. We first crawled firmware on the Internet, then we unpack the firmware with Binwalk (bin, 2021). We successfully unpacked 28,890 firmware from 46 IoT device manufacturers. The firmware is used in devices including routers, switches, IPCams, printers, cameras, etc. Most firmware that unpacks successfully is router or IPCam because their images consist of standard filesystems, such as Squashfs, YAFFS2 and JFFS2.

6.2. RQ1: Effectiveness of version identification

Accuracy of Our Method. Table 4 shows the version identification accuracy of VERI evaluated on the open-source component dataset. For the Cross-optimization dataset, VERI achieves high accuracy at all the optimization levels from 0.952 to 0.974. Among them, the version accuracies of *binutils, openssl, tcpdump and openvpn* reach 100%. For the Cross-architecture dataset, the identification accuracies in x86, ARM and x64 architectures are 0.966, 0.943 and 0.968. The accuracy in ARM is slightly lower than the x86 and x64. The test binaries size is 534Kb-23.5Mb, and the average time of every Mb binary is between 0.543s/Mb and 0.612s/Mb. Even for large binaries, the performance is acceptable.

Table 4
Version identification accuracy in cross-optimization and cross-architecture. For each compiler optimization (OP) and each ISA, we report the number of OSCs whose versions are correctly identified (# True), the number of OSCs that are misidentified due to the exit or explosion of symbolic execution (# Fs), the number of OSCs that are misidentified due to the existing of multiple reachable points (# Fc), the identification accuracy (Acc.(%)) and the average time of every Mb binary.

Dataset	CO/ISA	# True	# Fs	# Fc	Acc.(%)	Time
Cross-op (313)	O0	305	7	1	97.4%	0.532s
	O1	301	10	2	96.2%	0.596s
	O2	298	11	4	95.2%	0.608s
	O3	305	6	2	97.4%	0.573s
Cross-arch (348)	x86	336	9	3	96.6%	0.564s
	ARM	332	10	6	95.4%	0.543s
	x64	340	7	1	97.7%	0.589s

Table 5
Version identification comparison in i) Regex method, ii) state-of-the-art tools and iii) VERI without pruning and with pruning. For each method, we report the number of OSCs whose versions are correctly identified (# True), the number of OSCs that are misidentified due to the exit or explosion of symbolic execution (# Fs), the identification accuracy (Acc.(%)) and the average time of every Mb binary.

Approach	# True	# Fs	Acc.(%)	Time
Regex+random(313)	129	-	41.2%	0.089s
Regex+max(313)	163	-	52.1%	0.093s
OSSPolice(313)	237	-	75.7%	0.825s
B2SFinder(313)	274	-	87.5%	14.823s
VERI-pruning(313)	219	76	67.0%	5.380s
VERI(313)	**305**	**6**	**97.4%**	0.564s

False Positive Cause. We examined the false positives and manually analyzed them, which have two main false causes (Fs and Fc columns in Table 4). First, the true version point is unreachable when the binary exits execution in advance or steps into path explosions during execution. Though we prune the CFG before symbolic execution, there are still a few binaries have path explosions. Path explosion usually occurs when there are too many branch paths. Second, since some symbolic variables are assigned unconstrained symbol values during execution, leads to two or more branches are feasible and ultimately more than one *candidate version point* reachable.

Comparison in Version Identification. We compare our approach with regular-expression matching method and two state-of-the-art tools, OSSPolice (Duan et al., 2017) and B2SFinder (Yuan et al., 2019). As there are multiple candidate version strings matched by regex, we adopt two ways to select them, i) random selection, ii) selecting the max version. OSSPolice compares the similarities of binaries against the source code of open-source software to identify binary versions. B2SFinder reasons about seven types of features from both binary and source code and employs a weighted feature matching algorithm to detect open-source binary code reuse. We collect the source code of the binaries in Cross-optimization dataset and build the source code feature index. Table 5 presents the comparison results based on the Cross-optimization dataset. VERI performs better than other methods and tools with high accuracy and identification rate. There are 4 candidate version strings on average in each binary, and regex method with randomly selection correctly identified 129 (41.2%) binaries and regex method with maximal selection correctly identi-

Table 6
The performance of entity recognition and relation extraction tasks in our model. We report results in terms of Precision, Recall, F_1 for the two tasks.

Type	Name	Metric(%)		
		Precision	Recall	F_1 **score**
Entity	OSC name	95.24	94.86	95.05
	OSC version	99.52	98.58	99.05
	Overall	**97.19**	**96.56**	**96.87**
Relation	before	93.51	87.80	90.57
	and earlier	95.00	90.48	92.69
	through	96.15	80.65	87.72
	in	93.59	91.25	92.41
	start	95.20	95.20	95.20
	end	93.65	94.40	94.02
	Overall	**94.29**	**91.86**	**93.06**

fied 163 (52.1%) binaries, which reports that regex matching is not effective in distinguishing candidate version strings. Because the C/C++ code feature of OSSPolice is coarse (i.e., string literal and exported function), which would generate the same signature for different versions if the two versions have minor differences, resulting in plenty of false positives. Compared to OSSPolice, B2SFinder introduces new features (constants in "switch/case" and "if/else" statements) and new weighted matching methods, which brings an improvement in accuracy (87.5%), but is more time-consuming (26 times longer than VERI).

Comparison in Method without and with Pruning. Furthermore, we evaluated the importance of CFG pruning by comparing whether or not the pruning method was used. From Table 5, we can see that the accuracy is increased and the false positive caused by path explosion is reduced significantly. Simultaneously, the average execution time without CFG pruning is 9 times longer than that execution with CFG pruning.

Answer RQ1: VERI can identify OSC versions with high effectiveness. In different optimization levels and architectures, VERI achieves 96.6% and 96.5% accuracy on average respectively, with an average time of 0.572s in every Mb binary.

6.3. RQ2: Quality of the version range extraction model

Metrics. We adopt three metrics to evaluate the quality of model: Precision, Recall, and F_1 score. The terms True Positive (TP), False Positive (FP), True Negative (TN), and False Negative (FN) refer to the result of a test and the correctness of the classification. For example, in the "OSC name" classification task, TP means correctly classified as "OSC name", FP means incorrectly classified as "OSC name", TN means correctly classified as other entities, FN means incorrectly classified as other entities.

Baseline Methods. *1)* ***Table Filling Multi-Task Model***. This model utilizes entity recognition as a secondary task to help with relation extraction (Gupta et al., 2016). *2)* ***Novel Tagging Scheme Model***. This model transforms the joint extraction problem into a sequence labeling problem (Zheng et al., 2017). After the annotation sequence is generated, the tags are merged into entity relation triples. *3)* ***VIEM***. It utilizes entity recognition model and relation extraction model separately to extract information from unstructured vulnerability descriptions, ignores the interactions between the two tasks (Dong et al., 2019).

Quality of Our Model. The version range extraction model is composed of entity recognition and relation extraction as evaluated in Table 6. Table 6 shows the performance of entity recognition for OSC names and OSC versions. The precision, recall and F_1 score in OSC names recognition are 0.9524, 0.9486 and 0.9505, and in OSC versions recognition are 0.9952, 0.9858 and 0.9905. The overall precision, recall, F_1 score are as high as 0.9719, 0.9656, and 0.9687. Table 6 shows the detailed performance of different relation types. The overall precision, recall, F_1 score in relation extraction are 0.9429, 0.9186, and 0.9306. The overall extraction result combines the results of entity recognition and relation extraction. A correct extraction result means the results of entity recognition and relation extraction are all correct. The overall accuracy of version range extraction model is 92.40%.

Table 7
Comparison with Baseline Methods. We report results in terms of Precision, Recall, F_1 for the two tasks.

Method	Entity			Relation		
	Pr(%)	Re(%)	F_1**(%)**	Pr(%)	Re(%)	F_1**(%)**
Table Filling	88.63	85.92	87.25	64.65	53.14	58.33
Novel Tagging	90.50	87.10	88.77	78.50	73.60	75.97
VIEM	95.54	94.32	95.42	77.27	84.09	80.54
Our model	**97.19**	**96.56**	**96.87**	**94.29**	**91.86**	**93.06**

Compared with Baseline Methods. We compare our model with two baseline methods as shown in Table 7. As shown in the table, our model outperforms baseline methods by a large margin. For entity recognition, VERI exceeds Table Filling model 9.7%, and Novel Tagging Scheme model 7.4% in precision. For relation extraction, VERI exceeds Table filling model 45.0%, and Novel Tagging Scheme model 19.4% in precision. Although VIEM performs well in entity recognition, it cannot extract multi-head relations in a sentence precisely, which underperforms VERI 17.0% in relation extraction task.

We apply VERI to 23,410 vulnerability descriptions to extract the vulnerability structured information and establish a version-vulnerability relation database.

Answer RQ2: The version range extraction model in VERI outperforms state-of-the-art model, achieving 96.87% F_1 score in entity recognition, 93.06% F_1 score in relation extraction and 92.40% overall accuracy.

6.4. RQ3: Firmware vulnerability detection

We use VERI to conduct N-day vulnerability detection on firmware dataset. For the version identification phase, VERI identifies 177,379 OSC versions of 512 different kinds of OSCs from firmware set. Among them, 38,654 (21.8% of identified OSC) versions are vulnerable, affected by 23,410 different CVE vulnerabilities. These vulnerable OSCs are distributed among 24,845 (86.0% of the total) firmware.

Top 10 OSCs. The Top 10 OSCs affecting most firmware has shown in Table 8, these OSCs were reported vulnerabilities between the period of July 2006 August 2021. *Busybox* affect the most firmware, which has 15 vulnerable versions have been found in 20,130 firmware. The CVE-2016-2147 of *Busybox* is the most widely affected vulnerability in tested firmware, which affected busybox version up to and including 1.24.2 and was found in 16,918 firmware. The second most affected OSC is *Dnsmasq*, whose vulnerable versions has been found in a total of 16,583 firmware, the latest affected firmware up to Dec. 2019. *OpenSSL* has the most vulnerable versions in a total of 15 versions, which included in 3271 firmware. A large number of newly released firmware still uses old *OpenSSL* vulnerable versions like DAP-1522 of Dlink. The count of OSC's existing versions is larger than the vulnerable versions, which illustrates that many firmware manufacturers do not upgrade the OSC to the latest version immediately but upgrade firmware periodically, and the OSC may have gone through several version iterations during this time.

Vulnerability Result. After obtaining the OSC versions of firmware, we polled the pre-built version range database to search for the corresponding vulnerabilities. Out of 28,890 firmware, we

Table 8
Top 10 OSCs affecting most firmware.

OSC Name	# Vul. Vers	Vul. OSC Versions	# Affected Firm
busybox	15	1.00, 1.01, 1.1.3, 1.10.4, 1.11.2, 1.12.1, 1.15.2, 1.17.2, 1.19.3–4, 1.20.2, 1.21.1, 1.22.1, 1.24.2	20,130
dnsmasq	9	2.36, 2.40, 2.45 2.47, 2.49, 2.62, 2.66, 2.71, 2.76-1	16,583
dropbear	13	2011.54, 2012.55, 2013.58–60, 2013.62, 2014.63, 2014.65–66, 2015.67–68, 2016.74	11,566
iptables	12	1.3.5, 1.3.7, 1.3.8, 1.4.0, 1.4.0rc1, 1.4.4, 1.4.5, 1.4.6, 1.4.7, 1.4.10, 1.4.12, 1.4.21	8725
pppd	7	2.4.1, 2.4.2b1, 2.4.3, 2.4.4b1, 2.4.5, 2.4.6, 2.4.7	7743
hostapd	8	0.5.9, 0.6.10, 0.7.3, 2.0, 2.0-devel, 2.3-devel, 2.5-devel, 2.6-devel	7186
lua	4	5.1.0, 5.1.2, 5.1.4, 5.1.5	6048
samba	13	3.0.3, 3.0.13, 3.0.14a, 3.0.22, 3.0.23, 3.0.24, 3.0.33, 3.0.37, 3.2.2, 3.2.8, 3.6.6, 4.1.7	5558
openssl	15	0.9.7f, 0.9.8b, 0.9.8g, 0.9.8k, 0.9.8za, 1.0.0a, 1.0.0g, 1.0.0m, 1.0.1c, 1.0.1g, 1.0.1h, 1.0.2d, 1.0.2h, 1.0.2j	3271
tcpdump	14	3.5.2, 3.7.1, 3.9.1, 3.9.2, 3.9.5, 3.9.8, 4.0.0, 4.1.1, 4.2.1, 4.5.0, 4.5.1, 4.7.2, 4.8.1, 4.9.0	2593

Table 9
A list which includes firmware (Firmware Name) and its release date (Firm. Release(Date)), vulnerable OSC versions contained in the firmware (Vul. OSC Version), the date in which a patch was made available for vulnerable OSC versions (Patch Date), the firmware release and its date at which the vulnerable OSC version was fixed (Firm. Fixed Release(Date)), and the number of days elapsed before the fix was applied (Outdated Days).

Firmware Name	Vul. OSC Version	Firm. Release(Date)	Patch Date	Firm. Fixed Release(Date)	Outdated Days
WR741ND (TP-LINK)	iptables 1.3.7	V1 2010-09-10	2012-05-30	V4 2013-03-25	299
MR3420 (TP-LINK)	pppd 2.4.3	V1.3 2012-11-23	2006-07-10	V5.0 2018-07-12	4385
DSP-W215 (Dlink)	lighttpd 1.4.34	v2.01 2014-10-03	2014-03-12	v2.23 2015-02-24	349
DSP-W215 (Dlink)	hostapd 2.0	v2.01 2014-10-03	2014-10-15	v2.23 2015-02-24	132
DAP-1522 (Dlink)	openssl 0.9.8g	v1.00 2008-02-27	2008-05-13	v2.00 2011-06-01	1190
E1000 (Openwrt)	pppd 2.4.7	v1.0 2018-06-09	2020-02-03	v2.0 2020-03-04	30
E1000 (Openwrt)	hostapd 2.7-devel	v1.0 2018-06-09	2019-04-10	v2.0 2020-03-04	329
E1000 (Openwrt)	dropbear 2017.75	v1.0 2018-06-09	2018-02-27	v2.0 2020-03-04	736
E1000 (Openwrt)	dnsmasq 2.80	v1.0 2018-06-09	2019-10-23	v2.0 2020-03-04	133
ER-e200 (Ubiquiti)	pppd 2.4.4	v1.3.0 2013-10-14	2006-07-10	v2.0.3 2019-05-31	2055
ER-e200 (Ubiquiti)	iptables 1.4.20	v1.6.0 2014-10-31	2012-05-30	v1.10.0 2018-02-15	2087
ER-e200 (Ubiquiti)	openssl 1.0.1e	v1.6.0 2014-10-31	2013-12-29	v2.0.3 2019-05-31	1979
ER-e200 (Ubiquiti)	lighttpd 1.4.35	v1.4.1 2014-06-20	2015-06-09	v2.0.3 2019-05-31	1806
ER-e200 (Ubiquiti)	dnsmasq 2.76-1	v1.9.1 2016-12-14	2017-09-07	v1.10.0 2018-02-15	161
ER-e200 (Ubiquiti)	tcpdump 4.3.0	v1.6.0 2014-10-31	2015-03-24	v1.9.1 2016-12-14	631
ER-e200 (Ubiquiti)	tcpdump 4.7.4	v1.9.1 2016-12-14	2017-01-27	v2.0.3 2019-05-31	854

detected a total of 266,109 potential vulnerabilities involving 524 CVEs. Most vulnerabilities are concentrated in a few OSCs, as shown in Fig., 445 CVEs were identified from Top 10 OSCs, accounting for 84.9% of all CVEs detected. Fig. reports that *OpenSSL* contains the most CVEs in the Top 10 OSCs, 98 in all, it affects 3271 firmware and resulting in 100,559 vulnerabilities. The affected most OSC *Busybox* has 33 distinct CVEs and results in 84,022 vulnerabilities. Although vulnerable versions of *iptables* and *pppd* are found in a large number of firmware, they only lead to few vulnerabilities.

Vulnerability Exploit Type Analysis. Table 11 lists the 10 summarized vulnerability types of the Top 10 OSCs and their number of CVEs, which illustrates that firmware is exposed to a variety of security risks. The *Denial of Service* is the most vulnerable type, 162 in all, accounts for 33.9% of overall CVEs. *OpenSSL* has the most *Denial of Service* CVEs, 56 in all, including 6 critical vulnerabilities with CVSS Score of 10. Besides, Top 10 OSCs all contain *Overflow* type vulnerabilities and 83.1% vulnerabilities in *tcpdump* are of *Overflow* type. The uncommon *Sql injection* type vulnerability occurs only in *pppd*.

Vulnerability Severity Analysis. We collected CVSS scores of Top 10 OSCs' CVEs shown in Table 12. CVEs with CVSS Score in 4.0–6.9 (Medium) accounted for the most, 52.7%. There are 45 CVEs are the critical (CVSS Score in 9.0–10.0) vulnerabilities which cause serious security issues, and the critical CVEs mainly distributed in *OpenSSL*, *Samba*, and *Dropbear*. As shown in Fig., *Samba, OpenSSL*, and *tcpdump* have the most CVEs whose CVSS are high or critical. *hostapd* and *lua* only contain Low and Medium vulnerabilities and the affected most widely OSC *Busybox* includes 6 Low CVEs, 31 Medium CVEs and 3 High CVEs.

Outdatedness Analysis of Firmware OSC. To illustrate the outdatedness of OSC during the firmware iteration, we selected 38 group firmware to analyze the update of OSC, the analysis of example firmware is shown in Table 9. Table 9 illustrates that during the firmware iteration, manufacturers do not update vulnerable versions in time for OSC in firmware such as *iptables, openssl, dnsmasq, and tcpdump*. We confirmed patched date for each vulnerable version and compute the outdated days. MR3420 and ER-e200 include vulnerable versions of 2.4.3, 2.4.4 of *pppd* which was patched for a privilege escalation vulnerability in July 2006. However, the firmware containing the vulnerable *pppd* versions took a long time to apply the fix, MR3420 took 4385 days with 5 firmware release iterations and ER-e200 took 2055 days with 8 firmware release iterations. The E1000 v1.0 of Openwrt also contains vulnerable *pppd*, but it was fixed after only 30 outdated days.

Table 10
Top5 most outdated OSC and the average outdated days.

OSC Name	Genre	# AVG. Outdated Days
iptables	IPv4 packet administration tool	425
dnsmasq	Network Service	527
tcpdump	Command-line Packet Analyzer	201
openssl	Secure communication toolkit	685
pppd	Point-to-point protocol for network communication	992

Table 10 lists the Top 5 most outdated vulnerable OSCs across all analyzed firmware iterations. *pppd* is the most outdated OSC with an average time-to-fix of 992 days. Each of the five OSCs is an integral component for network communication, whose security is critical. However, these outdated OSCs provide plenty of time for attackers to exploit N-day vulnerabilities.

Answer RQ3: VERI can detect vulnerability in large-scale firmware. For 28,890 firmware, VERI gets 177,379 OSC versions and

Table 11
Number of CVE in different vulnerability types.

No.	Vulnerability Type	# CVEs
1.	Bypass Something	19
2.	Denial of Service	162
3.	Execute Code	39
4.	Overflow	130
5.	Gain Information	33
6.	Memory Corruption	11
7.	Execute Code	30
8.	Directory Traversal	6
9.	Sql Injection	1
10.	Gain Privilege	5

Table 12
Number of CVE in different vulnerability CVSS scores.

CVSS Score	Qualitative Rating	# CVEs
0.1 - 3.9	Low	36
4.0 - 6.9	Medium	225
7.0 - 8.9	High	121
9.0 - 10.0	Critical	45

finds 38,654 vulnerable OSCs with 266,109 vulnerabilities. VERI evaluates the severity and type of these vulnerabilities, most of which are with high risks.

7. Discussion

In this section, we discuss the limitations of VERI and future research work.

Firmware Complexity. The implementation of our scheme depends on the unpacking of large-scale firmware. The IoT firmware is made of multiple components and has different filesystems. Linux-based system is the most frequently encountered embedded operating system (Costin et al., 2014), our firmware processing method can unpack it well. However, blob firmware is a large, single-binary embedded OS compiled from components in the form of different modules (Redini et al., 2020), for which our firmware processing method cannot handle and we will address it in our future work.

Version Information in OSC. Due to the frequent iteration of open-source components, OSCs often contain current version information in the binary for developer validation. Our work can identify the versions of popular open-source components. However, the version information for some uncommon components such as *crda* (a udev helper for communication between the kernel and userspace) only appears in the documentation and is not linked to the binary, which our tool cannot handle. We manually check 182 distinct OSCs, of which 37 uncommon OSCs do not contain version information. We plan to leverage code similarity analysis in binary code and source code to find the version of these OSCs and combine the technique with VERI.

Continuous Database Building. We will continue to collect more vulnerability reports to extend our version-vulnerability relation database. Though we have collected 23,410 OSC vulnerabilities, the vulnerabilities are updated in large numbers every day. Second, we will continually enrich our firmware database, though we have collected 28,890 firmware images with 31 kinds, we still lack some latest IoT firmware.

8. Related work

8.1. Information extraction from vulnerability reports

A large number of vulnerabilities require automated methods to assist in analyzing vulnerability reports. Dong et al. (2019) utilized a pipeline model (VIEM) to extract structured information from reports of multiple vulnerability databases to examine the information consistency. VIEM has some limitations: i) Pipeline model treats entity recognition and relation extraction as two separate tasks, ignoring the interactions between the two tasks. ii) VIEM cannot directly extract the relation between names and versions (e.g., before, through, and earlier), it only has two relations Y and N, which denote that there has a relation (Y) or no relation (N) between two entities. iii) VIEM cannot learn multiple relations in a sentence synchronously, so the relation extraction model of VIEM works poorly when dealing with multiple relations in a vulnerability description. For these limitations, we improve our model: i) Different from the separate models of VIEM, we adopted an end-to-end joint extraction model to realize information interaction between two tasks. ii) Our model implements multi-type relation extraction between names and versions (e.g., before, through, and earlier). iii) We treat relation extraction task as a multi-headed problem to learn multiple relations in a sentence synchronously (We use sigmoid function to define the probability of two tokens over relation r, if the probability of relation r between two tokens exceeds the threshold we set, we confirm that the relation r exists). As shown in TABLE VII, Although VIEM performs well in entity recognition task (underperforms our model 1.65%), it cannot extract multi-head relations in a sentence precisely. Our model exceeds VIEM 17.0% in relation extraction task. Han et al. (2017) used word embeddings and a Convolutional Neural Network (CNN) to predict the vulnerability severity from the CVE descriptions. Xiao et al. (2019) developed a knowledge graph embedding approach to learn the relationships between software vulnerabilities and weaknesses. Bettenburg et al. (2008) proposed a method to merge the additional information in vulnerability duplicates, helping resolve bugs quicker. Bozorgi et al. (2010) trained a classifier with features from vulnerability reports to predict the exploitability of vulnerabilities. Mu et al. (2018) found that the prevalence of missing information in vulnerability reports leads to poor vulnerability reproducibility. Anwar et al. (2021) remedied the data quality issues in NVD using a machine learning-based automation to provide a more reliable source of vulnerability information. Nappa et al. (2015) proposed a novel approach to quantify the race between exploit creators and patch development, and identify several errors in the existing vulnerability databases. Guo et al. (2022) adopted a rule-based method to extract the six key aspects from the CVE descriptions and a neural-network-based method to automatically recommend and augment the missing aspects in CVE descriptions.

8.2. Vulnerable open-source components assessing

Many researchers assess the vulnerable components based on different approaches. Perl et al. (2015) trained a machine learning-based classifier to predict suspicious commits of open-source components based on constructed vulnerable commit database from the mapping of CVEs to GitHub commits. Shin et al. (2010) investigated the applicability of three types of component metrics to build vulnerability prediction models: complexity, code churn, and developer activity. Some researchers presented an approach based on machine learning with source code text mining technique to predict which components contain security vulnerabilities (Li et al., 2017; Scandariato et al., 2014; Walden et al., 2014).

8.3. Binary version identification

There are some version identification methods for Android third-party libraries. ATVHUNTER (Zhan et al., 2021) extracted the CFG as the coarse-grained feature and opcode of each basic block in CFG as the fine-grained feature to identify

the version by employing the similarity comparison method. Almanee et al. (2019) leveraged different features from library metadata and strings in read-only sections to identify native library versions based on their similarity metric. Duan et al. (2017) used software similarity comparison which extracted inferent characteristic features to identify free software license violations and vulnerable versions of open-source third-party libraries. OSSPolice Duan et al. (2017) is a binary-to-source matching tool targeting open-source libraries, which indexes a repository by structural layout of the tree of files and directories. B2SFinder (Yuan et al., 2019) reasons about seven types of features from both binary and source code and employs a weighted feature matching algorithm to detect open-source binary code reuse. Yasumatsu et al. (2019) collected large-scale third-party libraries in Android to study the vulnerable versions and corresponding security fix, which found that outdated libraries are widely used in apps. Asteria (Yang et al., 2021) used Tree-LSTM network to encode ASTs at binary level into semantic representation vectors for binary code similarity detection. Zhang et al. (2019) leveraged a static analysis of app binaries coupled with a database of third-party libraries to identify versions of third-party libraries in android apps. Then they proposed a novel approach to generate synthetic apps to tune the detection thresholds.

9. Conclusion

In this paper, we propose VERI, an automatic N-day vulnerability detection system, which precisely identifies the OSC version, extracts vulnerable version range by an entity-relation joint extraction model and maintains a database. Evaluation results show that the version identification method is accurate and effective, meanwhile, the joint extraction model performs well in extracting version range information. Based on application results, VERI is able to conduct large-scale vulnerability detection in a fast and accurate manner. For vulnerabilities, we find there are 24,845 firmware (86.0%) containing vulnerable OSC versions and a total of 266,109 potential vulnerabilities involving 524 CVEs. We conduct further analysis on the firmware and find that OSC was outdated by 473 days on average despite the firmware having been published in a new release.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

CRediT authorship contribution statement

Yiran Cheng: Conceptualization, Methodology, Formal analysis, Writing – original draft. **Shouguo Yang:** Data curation, Formal analysis, Writing – review & editing. **Zhe Lang:** Data curation, Validation. **Zhiqiang Shi:** Conceptualization, Resources, Writing – review & editing. **Limin Sun:** Supervision, Writing – review & editing.

Data availability

Data will be made available on request.

Acknowledgement

This research has been partially funded by the Nature Science Associate Foundation of China under Grants No. U1766215.

References

angr - home. https://www.angr.io/. (Accessed on 05/14/2022).
binwalk - home. https://www.github.com/ReFirmLabs/binwalk. (Accessed on 05/11/2021).
Blackduck - home. https://www.synopsys.com/software-integrity.html. (Accessed on 02/16/2021).
bzip2 - home. https://www.man7.org/linux/man-pages/man3/getopt.3.html. (Accessed on 05/06/2022).
Cve-2014-0160. https://nvd.nist.gov/vuln/detail/cve-2014-0160. (Accessed on 03/24/2022).
Heartbleed - home. https://heartbleed.com/. (Accessed on 02/16/2021).
Ida pro - home. https://www.hex-rays.com/. (Accessed on 02/16/2022).
Nvd - home. https://nvd.nist.gov/. (Accessed on 05/11/2021).
Open-source components included in nuke. https://www.learn.foundry.com/nuke/content/misc/third_party_software.html, (Accessed on 02/16/2022).
openssl - home. https://www.openssl.org/. (Accessed on 02/16/2021).
openssl - home. https://www.github.com/. (Accessed on 05/01/2022).
Pyvex - home. https://github.com/angr/pyvex. (Accessed on 02/16/2022).
Qemu - home. https://www.qemu.org/. (Accessed on 05/11/2021).
tcpdump - home. https://www.tcpdump.org/. (Accessed on 05/06/2022).
Adel, H., Schütze, H., 2017. Global normalization of convolutional neural networks for joint entity and relation classification. arXiv preprint: 1707.07719
Almanee, S., Payer, M., Garcia, J., 2019. Too quiet in the library: A study of native third-party libraries in android. arXiv preprint: 1911.09716
Alrawi, O., Lever, C., Antonakakis, M., Monrose, F., 2019. Sok: Security evaluation of home-based iot deployments. In: 2019 IEEE symposium on security and privacy (sp). IEEE, pp. 1362–1380.
Antonakakis, M., April, T., Bailey, M., Bernhard, M., Bursztein, E., Cochran, J., Durumeric, Z., Halderman, J.A., Invernizzi, L., Kallitsis, M., et al., 2017. Understanding the mirai botnet. In: 26th USENIX security symposium (USENIX Security 17), pp. 1093–1110.
Anwar, A., Abusnaina, A., Chen, S., Li, F., Mohaisen, D., 2021. Cleaning the nvd: comprehensive quality assessment, improvements, and analyses. IEEE Trans. Depend. Secure Comput..
Bettenburg, N., Premraj, R., Zimmermann, T., Kim, S., 2008. Duplicate bug reports considered harmful really? In: 2008 IEEE International Conference on Software Maintenance. IEEE, pp. 337–345.
Bozorgi, M., Saul, L.K., Savage, S., Voelker, G.M., 2010. Beyond heuristics: learning to classify vulnerabilities and predict exploits. In: Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining, pp. 105–114.
Costin, A., Zaddach, J., Francillon, A., Balzarotti, D., 2014. A large-scale analysis of the security of embedded firmwares. In: 23rd USENIX Security Symposium (USENIX Security 14), pp. 95–110.
Dong, Y., Guo, W., Chen, Y., Xing, X., Zhang, Y., Wang, G., 2019. Towards the Detection of Inconsistencies in Public Security Vulnerability Reports. In: 28th USENIX Security Symposium (USENIX Security 19), pp. 869–885.
Duan, R., Bijlani, A., Xu, M., Kim, T., Lee, W., 2017. Identifying open-source license violation and 1-day security risk at large scale. In: Proceedings of the 2017 ACM SIGSAC Conference on computer and communications security, pp. 2169–2185.
Etzioni, O., Banko, M., Soderland, S., Weld, D.S., 2008. Open information extraction from the web. Commun. ACM 51 (12), 68–74.
Guo, H., Chen, S., Xing, Z., Li, X., Bai, Y., Sun, J., 2022. Detecting and augmenting missing key aspects in vulnerability descriptions. ACM Trans. Softw. Eng. Methodol.(TOSEM) 31 (3), 1–27.
Gupta, P., Schütze, H., Andrassy, B., 2016. Table Filling Multi-task Recurrent Neural Network for Joint Entity and Relation Extraction. In: Proceedings of COLING 2016, the 26th International Conference on Computational Linguistics: Technical Papers, pp. 2537–2547.
Han, Z., Li, X., Xing, Z., Liu, H., Feng, Z., 2017. Learning to predict severity of software vulnerability using only vulnerability description. In: 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 125–136. doi:10.1109/ICSME.2017.52.
He, Y., Zou, Z., Sun, K., Liu, Z., Xu, K., Wang, Q., Shen, C., Wang, Z., Li, Q., 2022. Rapidpatch: Firmware hotpatching for real-time embedded devices. 31th USENIX Security Symposium (USENIX Security 22).
Kingma, D.P., Ba, J., 2014. Adam: a method for stochastic optimization. arXiv preprint arXiv:1412.6980.
Li, X., Chen, J., Lin, Z., Zhang, L., Wang, Z., Zhou, M., Xie, W., 2017. A mining approach to obtain the software vulnerability characteristics. In: 2017 Fifth International Conference on Advanced Cloud and Big Data (CBD). IEEE, pp. 296–301.
Lin, Y., Ji, H., Huang, F., Wu, L., 2020. A joint neural model for information extraction with global features. In: Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, pp. 7999–8009.
Ma, X., Hovy, E., 2016. End-to-end sequence labeling via bi-directional lstm-cnns-crf. arXiv preprint arXiv:1603.01354.
Mausam, M., 2016. Open information extraction systems and downstream applications. In: Proceedings of the twenty-fifth international joint conference on artificial intelligence, pp. 4074–4077.
Mu, D., Cuevas, A., Yang, L., Hu, H., Xing, X., Mao, B., Wang, G., 2018. Understanding the reproducibility of crowd-reported security vulnerabilities. In: 27th USENIX Security Symposium (USENIX Security 18), pp. 919–936.
Nappa, A., Johnson, R., Bilge, L., Caballero, J., Dumitras, T., 2015. The attack of the clones: A study of the impact of shared code on vulnerability patching. In: 2015 IEEE symposium on security and privacy. IEEE, pp. 692–708.

Nethercote, N., Seward, J., 2007. Valgrind: a framework for heavyweight dynamic binary instrumentation. ACM Sigplan Notices 42 (6), 89–100.
Niesler, C., Surminski, S., Davi, L., 2021. Hera: Hotpatching of embedded real-time applications. NDSS.
Perl, H., Dechand, S., Smith, M., Arp, D., Yamaguchi, F., Rieck, K., Fahl, S., Acar, Y., 2015. Vccfinder: Finding potential vulnerabilities in open-source projects to assist code audits. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, pp. 426–437.
Redini, N., Machiry, A., Wang, R., Spensky, C., Continella, A., Shoshitaishvili, Y., Kruegel, C., Vigna, G., 2020. Karonte: Detecting insecure multi-binary interactions in embedded firmware. In: 2020 IEEE Symposium on Security and Privacy (SP). IEEE, pp. 1544–1561.
Ryder, B.G., 1979. Constructing the call graph of a program. IEEE Trans. Softw. Eng. (3) 216–226.
Scandariato, R., Walden, J., Hovsepyan, A., Joosen, W., 2014. Predicting vulnerable software components via text mining. IEEE Trans. Softw. Eng. 40 (10), 993–1006.
Shin, Y., Meneely, A., Williams, L., Osborne, J.A., 2010. Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. IEEE Trans. Softw. Eng. 37 (6), 772–787.
Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., Salakhutdinov, R., 2014. Dropout: a simple way to prevent neural networks from overfitting. J. Mach. Learn. Res. 15 (1), 1929–1958.
Stephens, N., Grosen, J., Salls, C., Dutcher, A., Wang, R., Corbetta, J., Shoshitaishvili, Y., Kruegel, C., Vigna, G., 2016. Driller: Augmenting fuzzing through selective symbolic execution. In: NDSS, Vol. 16, pp. 1–16.
Walden, J., Stuckman, J., Scandariato, R., 2014. Predicting vulnerable components: Software metrics vs text mining. In: 2014 IEEE 25th international symposium on software reliability engineering. IEEE, pp. 23–33.
Wang, X., Sun, K., Batcheller, A., Jajodia, S., 2019. Detecting" 0-day" vulnerability: An empirical study of secret security patch in oss. In: 2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). IEEE, pp. 485–492.
Xiao, H., Xing, Z., Li, X., Guo, H., 2019. Embedding and predicting software security entity relationships: A knowledge graph based approach. In: International Conference on Neural Information Processing. Springer, pp. 50–63.
Yang, S., Cheng, L., Zeng, Y., Lang, Z., Zhu, H., Shi, Z., 2021. Asteria: Deep learning-based ast-encoding for cross-platform binary code similarity detection. In: 2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). IEEE, pp. 224–236.
Yasumatsu, T., Watanabe, T., Kanei, F., Shioji, E., Akiyama, M., Mori, T., 2019. Understanding the responsiveness of mobile app developers to software library updates. In: Proceedings of the Ninth ACM Conference on Data and Application Security and Privacy, pp. 13–24.
Yuan, Z., Feng, M., Li, F., Ban, G., Xiao, Y., Wang, S., Tang, Q., Su, H., Yu, C., Xu, J., et al., 2019. B2sfinder: detecting open-source software reuse in cots software. In: 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, pp. 1038–1049.
Zhan, X., Fan, L., Chen, S., We, F., Liu, T., Luo, X., Liu, Y., 2021. Atvhunter: Reliable version detection of third-party libraries for vulnerability identification in android applications. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE). IEEE, pp. 1695–1707.
Zhang, J., Beresford, A.R., Kollmann, S.A., 2019. Libid: reliable identification of obfuscated third-party android libraries. In: Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 55–65.
Zheng, S., Hao, Y., Lu, D., Bao, H., Xu, J., Hao, H., Xu, B., 2017. Joint entity and relation extraction based on a hybrid neural network. Neurocomputing 257, 59–66.
Zheng, Y., Davanian, A., Yin, H., Song, C., Zhu, H., Sun, L., 2019. Firm-afl: High-throughput greybox fuzzing of iot firmware via augmented process emulation. In: 28th USENIX Security Symposium (USENIX Security 19), pp. 1099–1114.

Yiran Cheng: Ph.D. student from the Institute of Information Engineering, Chinese Academy of Sciences, China. Her research interests include IoT security and program analysis.

Shouguo Yang: Ph.D. student from the Institute of Information Engineering, Chinese Academy of Sciences, China. His research interests include binary code similarity detection, patch detection, and vulnerability detection based on static analysis.

Zhe Lang: Ph.D. student from the Institute of Information Engineering, Chinese Academy of Sciences, China. His research interests include software security and program analysis.

Zhiqiang Shi: Ph.D. professorate senior engineer, Ph.D. supervisor from the Institute of Information Engineering, Chinese Academy of Sciences, China. His main research interests include network system and ICS security.

Limin Sun: Ph.D. professor, Ph.D. supervisor from the Institute of Information Engineering, Chinese Academy of Sciences, China. His main research interests include ICS security and IoT security.