UVscan: Detecting Third-Party Component Usage Violations in IoT Firmware

Binbin Zhao, *Georgia Institute of Technology and Zhejiang University;* Shouling Ji and Xuhong Zhang, *Zhejiang University;* Yuan Tian, *University of California, Los Angeles;* Qinying Wang, Yuwen Pu, and Chenyang Lyu, *Zhejiang University;* Raheem Beyah, *Georgia Institute of Technology*

https://www.usenix.org/conference/usenixsecurity23/presentation/zhao-binbin

This paper is included in the Proceedings of the 32nd USENIX Security Symposium.

August 9–11, 2023 • Anaheim, CA, USA

978-1-939133-37-3

UVSCAN: Detecting Third-Party Component Usage Violations in IoT Firmware

Binbin Zhao[*, †], Shouling Ji[†, ✉], Xuhong Zhang[†], Yuan Tian[‡], Qinying Wang[†], Yuwen Pu[†], Chenyang Lyu[†], and Raheem Beyah[*]

[*]Georgia Institute of Technology, [†]Zhejiang University, [‡]University of California, Los Angeles

E-mails: binbin.zhao@gatech.edu, {sji, zhangxuhong}@zju.edu.cn, yuant@ucla.edu, {wangqinying, yw.pu, puppet}@zju.edu.cn, rbeyah@coe.gatech.edu.

Abstract

Nowadays, IoT devices integrate a wealth of third-party components (TPCs) in firmware to shorten the development cycle. TPCs usually have strict usage specifications, e.g., checking the return value of the function. Violating the usage specifications of TPCs can cause serious consequences, e.g., NULL pointer dereference. Therefore, this massive amount of TPC integrations, if not properly implemented, will lead to pervasive vulnerabilities in IoT devices. Detecting vulnerabilities automatically in TPC integration is challenging from two perspectives: (1) There is a gap between the high-level specifications from TPC documents, and the low-level implementations in the IoT firmware. (2) IoT firmware images are mostly closed-source binaries, which lose lots of information when compiling from the source and have diverse architectures.

To address these challenges, we design and implement UVSCAN, an automated and scalable system to detect TPC usage violations in IoT firmware. In UVSCAN, we first propose a novel natural language processing (NLP)-based rule extraction framework, which extracts API specifications from inconsistently formatted TPC documents. We then design a rule-driven NLP-guided binary analysis engine, which maps the logical information from the high-level TPC document to the low-level binary, and detects TPC usage violations in IoT firmware across different architectures. We evaluate UVSCAN from four perspectives on four popular TPCs and six ground-truth datasets. The results show that UVSCAN achieves more than 70% precision and recall, and has a significant performance improvement compared with even the source-level API misuse detectors. To provide an in-depth status quo understanding of the TPC usage violation problem in IoT firmware, we conduct a large-scale analysis on 4,545 firmware images and detect 27,621 usage violations. Our further case studies, the Denial-of-Service attack and the Man-In-The-Middle attack on several firmware images, demonstrate the serious risks of TPC usage violations. Currently, 206 usage violations have been confirmed by vendors as vulnerabilities, and seven of them have been assigned CVE IDs with high severity.

Shouling Ji is the corresponding author.

1 Introduction

Nowadays, the Internet of Things (IoT) is omnipresent and plays an essential role in our daily lives. According to a recent report [25], 152,200 IoT devices will be connected to the internet per minute by 2025 and the number of active IoT devices will exceed 25.4 billion by 2030. Nevertheless, the growing number of IoT devices poses many serious security risks. For instance, a large number of IoT devices, e.g., routers and IoT gateways, are vulnerable to the NULL pointer dereference vulnerability due to the API misuses in the closed-source Qualcomm QCMAP suite [23], making them ideal compromise targets of various botnets, e.g., Mirai [9, 47].

Currently, the core part of IoT devices, *firmware*, widely integrates a large number of third-party components (TPCs) to facilitate development efficiency [48, 49]. A TPC always comes with a lot of built-in functions, also referred to as application programming interfaces (APIs). These APIs tend to have complicated specifications, which are typically written with lengthy statements in TPC documents. Nevertheless, developers may fail to strictly follow the API specifications when adopting TPCs, resulting in the *TPC usage violation problem*, which is typically caused by a set of API misuses, e.g., unchecked return value and incorrect invocation sequence. Many previous works have indicated that the API misuse will introduce serious security implications [26, 30, 32, 34, 38, 45]. For instance, the incorrect use of OpenSSL APIs can cause Man-In-The-Middle attacks [19, 22]. Moreover, vendors may call misused APIs multiple times in firmware or reuse TPCs with misused APIs in different firmware, leading to aggravated TPC usage violation problems. Therefore, it is essential to detect TPC usage violations in IoT firmware.

1.1 Challenges

To detect TPC usage violations in IoT firmware automatically, we have the following key challenges.

API specification inference from unstructured TPC document. API specifications describe the requirements for using the API, which is critical for determining if the API usage is correct or wrong. Nevertheless, API specifications are typically implicitly encoded or explicitly presented in massive TPC documents, which are usually loosely organized and unstructured, hindering the API specification inference. Therefore, the first challenge is to effectively and precisely obtain the specifications of all the APIs in TPCs, which is an essential step to enable the usage violation detection. A series of works have leveraged NLP techniques to extract API specifications from TPC documents [30, 32, 34]. Nevertheless, most of them only perform well on well-formatted TPC documents and are hard to handle unusual or ambiguous API specifications. Besides, several works are proposed to infer API specifications through a lot of usage examples [26, 45]. However, the usage examples may be incorrect, not to mention it is hard, if not impossible, to cover all the usage cases of an API, which will result in inaccurate API specifications and introduce many false positives, causing low precision.

Programming expression generation. API specifications extracted from TPC documents are typically natural language, which cannot be directly applied to the usage violation detection. For instance, the API specification *"the X509 object must be explicitly freed using X509_free"* could be easily understood by the human but cannot be handled by the program without extra effort. Therefore, it is vital to translate the natural language-based API specifications into programming expressions, which will be used as input to the usage violation detection system. Nevertheless, it is not trivial to design a practical method to generate programming expressions from API specifications automatically.

Usage violation detection. In practice, IoT firmware images are mostly distributed as the closed-source binaries. Therefore, the third challenge is to perform the usage violation detection at the binary-level. Previous works focus on the source-level API misuses. Therefore, they can rely on existing static analysis tools, e.g., *CodeQL* [2], or straightforward checkers to conduct analysis. Nevertheless, detecting the usage violation at the binary-level is much more complicated than that at the source-level, which will further cause challenges. First, the binary loses much information when compiling from the source code, e.g., missing symbols in the stripped binary, which is essential for the usage violation detection. Second, it is challenging to obtain the exact operations related to API usage at the binary-level due to the complex logical relationships between assembly instructions. Besides, the different architectures used in the firmware, such as x86, ARM, and MIPS, lead to different kinds of assembly instructions, increasing the difficulty of analyzing firmware. Currently, there is no such analysis tool like *CodeQL* for the binary-level detection.

1.2 Methodology

In this paper, we aim to address these challenges to detect TPC usage violations at scale. To this end, we design and implement UVSCAN, the first automated and practical framework to conduct the TPC usage violation detection on binary IoT firmware. Our design philosophy is as follows.

First, to solve the challenge of inferring API specifications, we start from designing a sentiment-based model to filter out irrelevant API descriptions from TPC documents by leveraging a coreference resolution model and a customized BiLSTM model with the multi-head self-attention mechanism. Next, we propose a Machine Reading Comprehension (MRC)-driven approach to extract API specifications from the processed TPC document by customizing an MRC system with our well-designed query questions and the manually constructed dataset. **Second**, to address the problem of generating programming expressions, we design an NLP-guided approach by constructing the dependency tree for each API specification according to the part of speech (POS) and then map the phrases in nodes into programming expressions based on semantic patterns, which are collected from previous works and enriched by using synonym replacement and changing the voice of patterns. **Third**, we adopt *FirmSec* [48] for firmware processing, incorporating various enhancements. *FirmSec* supports extracting multiple kinds of firmware and recognizing the TPCs in firmware by leveraging syntactical features and control flow graph (CFG) features. **Finally**, to solve the challenge of detecting usage violations in IoT firmware, we design a rule-driven analysis engine with customized violation-targeted checkers. The main idea behind our design is to encode the binary code into Datalog facts [13] and extract the corresponding implicit and explicit contextual logic relationships. By parsing programming expressions into our analysis engine, we successfully detect TPC usage violations in IoT firmware with high precision and recall.

1.3 Contributions

We summarize our main contributions as follows.

• We propose UVSCAN, the first automated and practical system to detect TPC usage violations in binary IoT firmware, which fills the gap in mapping the high-level specifications from TPC documents to the binary-level violation analysis. UVSCAN achieves over 70% precision and recall in detecting TPC usage violations on ground-truth datasets, including 146 usage violations in three architectures. Though UVSCAN targets binary-level, it has an excellent performance improvement even compared to state-of-the-art source-level works: *Advance* [30], *APISAN* [45], and *APEx* [26]. To facilitate future IoT security research, we will open-source UVSCAN at `https://github.com/BBge/IoT-CVE`.

• To the best of our knowledge, we are the first to work on the TPC usage violation problem in IoT firmware and conduct the first large-scale analysis on this problem. Based on UVSCAN, we detect 27,621 usage violations caused by four TPCs in 4,545 firmware images. According to the results, we uncover the widespread TPC usage violations in IoT firmware. Our further case studies, the Denial-of-Service attack and the Man-In-The-Middle attack on several firmware images, demonstrate the potential serious risk of TPC usage violations. Up to now, 206 usage violations have been confirmed by vendors as vulnerabilities and seven of them have been assigned CVE IDs with high severity.

2 Background

2.1 NLP Technique

The NLP technique is an essential cornerstone in our system. Our NLP-based API specification extraction framework and NLP-guided programming expression generation approach involve multiple NLP concepts. We give a brief introduction to these concepts as follows.

Coreference resolution is a task that aims to automatically identify all mentions that refer to the same entity in a given text. Entities may be various kinds of names (e.g., API names), dates, locations, etc. Mentions are pronouns, named entities, and noun phrases. Currently, coreference resolution has lots of applications, including machine translation, full-text understanding, and so on. In this paper, coreference resolution is an essential step for extracting API specifications from documents since a TPC document usually has many entities, e.g., the return value and the arguments of an API.

Multi-head self-attention is derived from the self-attention mechanism which applies self-attention multiple times in parallel. The self-attention mechanism is a variant of the attention mechanism that relies less on external information and is better at capturing the inter-word dependencies of the sentence itself, such as common phrases, and entities referred to by pronouns. In this paper, the multi-head self-attention mechanism serves as an essential role in our customized sentiment-based model in extracting relevant API descriptions from TPC documents.

Machine Reading Comprehension (MRC) is a hot topic in NLP and has many real-world applications, e.g., information retrieval. An MRC system is designed to enable computers to understand the semantics of text and respond to natural language questions with precise answers by retrieving and reasoning through relevant knowledge, which is usually implicitly encoded or explicitly presented in the text. In this paper, we design an MRC-driven extraction method to extract precise API specifications from relevant API descriptions with our well-designed query questions and the manually annotated dataset.

2.2 Datalog

Datalog [13] is a declarative programming language that has been used in many tasks, including dataflow analysis [39], binary disassembly [18], and program trace encoding [28]. Datalog utilizes the first-order predicate logic to represent the computation of logical relations. A Datalog program has two parts, an intensional database and an extensional database. The intensional database is composed of a set of Datalog rules while the extensional database is defined by Datalog facts. A Datalog rule is in the form of Horn clauses: $Y : -X_1, X_2, ..., X_n$, where Y is the head of the rule, and $X_1, X_2, ..., X_n$ are literals in the body of the rule. The above rule can be translated into natural language as "If X_1 and X_2 and ... and X_n are true, then Y is true". The rule head can be one or more predicates and the rule body can be predicates, negative predicates, or constraints. A Datalog fact is a special type of datalog rule, without any rule body, but only having a rule head.

Currently, Datalog has many variants, including *Soufflé* [24], *CodeQL*, and so on. *Soufflé* is a new logic programming language developed from Datalog, which overcomes several limitations in classical Datalog and has been adopted by many previous works. For instance, Flores-Montoya et al. [18] designed a binary disassembler based on *Soufflé*. In this paper, we generate Datalog facts and rules for IoT firmware in *Soufflé* syntax.

2.3 Motivation

We discuss the motivation of this paper from three perspectives as follows.

IoT vendor perspective. In addition to using open-source TPCs, IoT vendors may also integrate closed-source TPCs in their IoT firmware. However, even closed-source TPCs can have serious security vulnerabilities resulting from API misuses. For example, the closed-source Qualcomm QCMAP suite, which is widely used in routers and IoT gateways, has been found to have a vulnerability due to the lack of checking the return values of functions, leading to the Denial-of-Service attack [23]. Therefore, it is essential for IoT vendors to thoroughly evaluate and test closed-source TPCs for API misuses to ensure the security of their devices.

IoT security company perspective. Many IoT vendors turn to specialized IoT security companies to perform security assessments of their products. To maintain the privacy and security of their data, IoT vendors typically only provide closed-source firmware to these security firms. To conduct a comprehensive security evaluation of firmware, it is crucial for these security companies to have the ability to detect TPC usage violations within the closed-source firmware.

Consumer perspective. Due to the widespread use of IoT devices and the ever-emerging vulnerabilities, consumers (e.g., researchers and business companies) are becoming increasingly concerned about the security of IoT devices. TPC usage violations in IoT firmware can result in vulnerabilities that may lead to privacy breaches and significant losses for consumers. As consumers usually only have access to the closed-source firmware of their products, it is important for those concerned about IoT product security to have a practical system that can detect TPC usage violations in IoT firmware.

3 UVSCAN Design

In this section, we present the design details of UVSCAN. At a high level, UVSCAN aims to automatically find TPC usage violations in IoT firmware. As shown in Figure 1, UVSCAN mainly has five modules: API specification extraction, programming expression generation, firmware processing, rule-driven analysis engine, and usage violation detection. The workflow is as follows.

First, the API specification extraction module accepts TPC documents as input and extracts API specifications for TPCs based on a customized sentiment-based model and an MRC-driven system. **Then**, the programming expression generation module translates API specifications into programming expressions. **Next**, the firmware processing module extracts objects from firmware and identifies the TPCs used in firmware. The rule-driven analysis engine module accepts the extracted objects as input, encodes them into Datalog facts, and designs four usage violation checkers. **Finally**, the usage violation detection module parses programming expressions into the rule-driven analysis engine and then performs the usage violation check on IoT firmware.

3.1 TPC Usage Violation Category

Before we introduce the design of UVSCAN, we first introduce four representative categories of TPC usage violations, which are summarized by analyzing many TPC usage violations reported in previous works [21, 28, 30, 45] and/or in the real-world applications.

Deprecated API violation. A set of APIs will be deprecated or abandoned during the development of TPCs for various reasons, e.g., security issues, low performance, etc. Using the deprecated APIs may bring serious security problems. For instance, attackers can break the cryptographic protection mechanisms if the target program is still using the deprecated API `RAND_pseudo_bytes()` from OpenSSL [3].

Return value violation. A great number of APIs have return values, which are used for indicating function execution results or status. If the return value of an API is unchecked or incorrectly checked, it may bring unpredictable security problems. For instance, it has been found that several functions for policy enforcement in Apache Accumulo do not properly check the return value will cause authorization bypass [5].

Argument violation. When many APIs are invoked, the corresponding arguments should also be passed into APIs. These arguments often have strict constraints. For instance, the `errbuf` argument in `pcap_open_live()`, which is from libpcap, should be set into a zero-length string before calling the API. Failure to check the argument of an API can have serious consequences. For instance, the Zephyr Project, a scalable real-time operating system, did not validate the argument in some system calls in versions 2.1.0 and later versions prior to 2.2.0, causing the privilege escalation attack [4].

Causality violation. Many APIs may have a strict causal relationship. For instance, lock and unlock, fopen and fclose, as well as malloc and free are the three most common causal relationships. Besides, many APIs also have pre- and post-condition requirements, which we also regard as causal relationships. For instance, `sqlite3_config()`, which is from SQLite, should be called before `sqlite3_initialize()` or after `sqlite3_shutdown()`. Violating the required causal relationship may cause critical consequences, e.g., information leakage and system crash.

3.2 API Specification Extraction

The purpose of API specification extraction is to extract the API specifications from corresponding TPC documents, which will be used in the usage violation detection module.

3.2.1 Document Distillation

TPC documents typically contain a wealth of sentences describing various aspects of the APIs. In this paper, we regard each sentence in the TPC document as an API description. Nevertheless, not all API descriptions are relevant to API specifications, which we regard as irrelevant API descriptions. Intuitively, these irrelevant API descriptions may affect the reliability of the final extracted API specifications. Therefore, to precisely extract API specifications, we should first distill TPC documents by filtering out irrelevant API descriptions and digging for relevant API descriptions. Unfortunately, TPC documents are always loosely organized and do not have consistent formats. It is difficult to recognize the relevant API descriptions automatically and precisely.

Previous research indicates that relevant API descriptions usually have a strong sentiment [30]. For instance, a description in OpenSSL states that "*the initialization vector iv should be a random value*," which has a strong sentiment. Nevertheless, this observation does not apply to all scenarios when analyzing the TPC document and will introduce false positives. For example, the sentence "*additionally it indicates that the session ticket is in a renewal period and should be replaced*" has a strong sentiment word "*should*" but it is not a relevant API description. In this case, the pronoun "*it*" actually represents the return value of the API, and the sentiment word "*should*" refers to "*session ticket*", which is not related to the specification of the API. The main reason behind this problem is that the above observation does not consider that the sentiment is target-dependent. There are two challenges to address this problem. First, we need to resolve the coreference in API descriptions. Second, we need to design a practical method to distinguish the sentiment of different targets and obtain the exact sentiment of the API.

To address the above challenges, we propose a novel sentiment-based document distillation model, leveraging the coreference resolution model, the bidirectional long-short term memory network (BiLSTM) [46] with the multi-head self-attention mechanism [41]. Specifically, the coreference

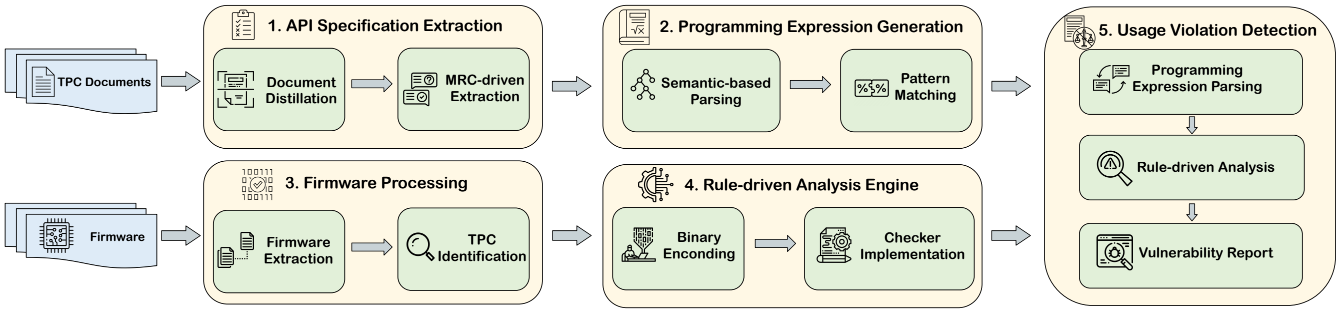


Figure 1: Framework of UVSCAN.

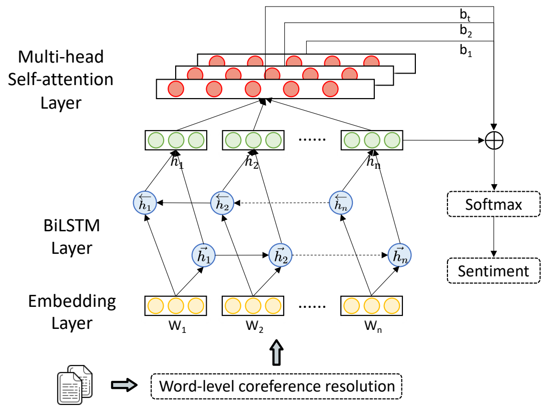

Multi-head Self-attention Layer
BiLSTM Layer
Embedding Layer
Softmax
Sentiment
Word-level coreference resolution

Figure 2: Sentiment-based document distillation model.

resolution model is designed to resolve the coreferences, e.g., pronouns, in TPC documents. BiLSTM performs well in learning the hierarchical information of sentences. The multi-head self-attention mechanism can capture strong sentiment words by leveraging the contextual information as well as the syntactic and semantic features. Since BiLSTM can enhance the semantic abstraction ability of the multi-head self-attention mechanism, we customize a sentiment analysis network by combining these two techniques, which can obtain the sentiment of each target and find the exact sentiment of the API.

Figure 2 presents the architecture of our model. We first leverage WL-Coref [16], an off-the-shelf coreference resolution model, to resolve the coreferences in the TPC document. Next, we pass the processed TPC document into our BiLSTM model with the multi-head self-attention mechanism. The sentence will be converted to word vectors in the embedding layer. Then, the BiLSTM layer will analyze the sentence from both forward and backward directions and extract hidden vectors from word vectors. Hidden vectors will be passed into the multi-head self-attention layer. Finally, the output of the multi-head self-attention layer will be fed into a softmax function to obtain the final sentiment of a sentence. If the sentence has a strong sentiment, we regard it as a relevant API description.

In addition, three authors of this paper manually label relevant API descriptions to construct the dataset for training and

Table 1: Question set.

Category	Question
Return Value	What are return values supposed to be? In which condition does the function have a return value?
Causality	What operation is required if the return value is *ReturnValue$_i$*? What operation is required if *Condition$_i$*? Which function should be called before the API? Which function should be called after the API?
Argument	What is the value of the N-th argument supposed to be before the API? What is the value of the N-th argument supposed to be after the API? How to check the N-th argument before the API? How to check the N-th argument after the API?

evaluating our sentiment-based document distillation model. Though strong sentiment words (e.g., "*must*" and "*should*") are good criteria to label relevant API descriptions, we not only rely on them since they may lead to false positives (as discussed in previous paragraphs) and false negatives. An example of the false negative is the sentence "*call SSL_get_error() with the return value ret to find out the reason*," which lacks strong sentiment words but is still considered a relevant API description. This sentence also has a strong sentiment since it is imperative and gives a forceful command. Therefore, we label the relevant API descriptions according to a set of criteria, including strong sentiment words, sentence patterns, negative structures, etc. The results annotated by each author will be cross-checked. We describe the dataset details in Section 4.1.

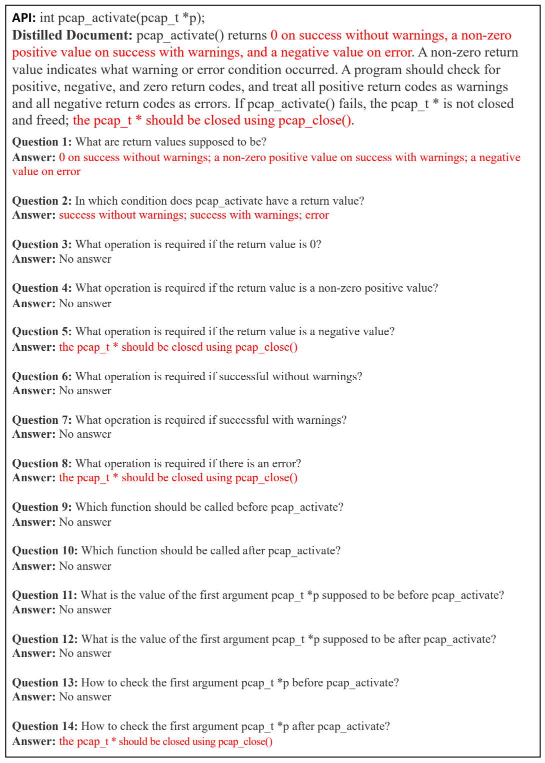
3.2.2 MRC-driven Extraction

In this step, we extract precise API specifications from relevant API descriptions, which will be further converted into programming expressions and used by the rule-driven analysis engine. Nevertheless, it is challenging to extract precise API specifications since it usually involves contextual understanding which cannot be properly handled by the keyword or template matching [32, 33]. For instance, a typical API description could be "*SQLITE_OK be returned by sqlite3_snapshot_recover if successful, or an SQLite error code otherwise*". For this case, we need to understand the context to infer that the latter part of the sentence is equal to "*SQLite error code be returned by sqlite3_snapshot_recover* *if failed*". To address the problem, we propose a novel MRC-driven extraction method by using the MRC system to extract precise API specifications. There are two main challenges to apply the MRC system to our task. First, there are no relevant examples to guide the design of query questions about TPC usage violations, which are an important part of the MRC system. Second, there is no publicly accessible dataset that can be used to train our MRC system. A well-labeled dataset is important for the performance of an MRC system. We elaborate our solutions to these challenges as follows.

Query question design. To solve the challenge of designing query questions, we analyze the possible causes of each kind of TPC usage violation and design the corresponding questions. For instance, the possible causes of the causality violation are lacking a call to the required function before or after the API, and missing or incorrect operations under certain conditions or return values. Therefore, we design query questions for the causality violation based on the above possible causes. We do not design questions for the deprecated API violation since we can obtain the deprecated API list for the TPC from its official website.

Table 1 presents the query questions designed for different kinds of usage violations mentioned in Section 3.1. These questions are well-suited for our MRC system for three key reasons. First, these questions are summarized through a thorough analysis of hundreds of real-world TPC usage violations and the examination of numerous TPC documents. They are all highly relevant to the possible causes of various usage violations and cover the vast majority of cases. Second, these questions contain both extractive and no answer questions, where the answer to each question is either a continuous subsequence extracted from API descriptions or no answer. Developing the ability to answer such questions, especially those with no answer, can enhance the system's ability to understand and interpret the context, further improving its generalization ability when presented with new TPC documents. Finally, all questions are clearly phrased and straightforward, rendering them suitable for further dataset annotation.

Additionally, the answers to two questions related to the return value will become part of the two questions related to the causality ($Condition_i$ and $ReturnValue_i$). $Condition_i$ indicates the prerequisites of an API having the return value, which is often closely linked to $ReturnValue_i$. Typically, TPC documents specify the required operations that correspond to specific return values, and these operations are presented together with the respective return values. Nevertheless, there are cases where the required operations are not associated with the return value, but instead with the condition. For instance, libpcap documentation states that "*pcap_activate returns 0 on success without warnings, a non-zero positive value on success with warnings, and a negative value on error. If pcap_activate fails, the pcap_t* $*$ *is not closed and freed; it should be closed using pcap_close*". In this scenario, the operation (*close pcap_t* $*$ *using pcap_close*) is associated with the condition (*pcap_activate fails*) rather than the return value (*a negative value*). Therefore, in light of the above situation, we design specific query questions to obtain the exact operations under certain conditions.


API: int pcap_activate(pcap_t *p);
Distilled Document: pcap_activate() returns 0 on success without warnings, a non-zero positive value on success with warnings, and a negative value on error. A non-zero return value indicates what warning or error condition occurred. A program should check for positive, negative, and zero return codes, and treat all positive return codes as warnings and all negative return codes as errors. If pcap_activate() fails, the pcap_t * is not closed and freed; the pcap_t * should be closed using pcap_close().

Question 1: What are return values supposed to be?
Answer: 0 on success without warnings; a non-zero positive value on success with warnings; a negative value on error

Question 2: In which condition does pcap_activate have a return value?
Answer: success without warnings; success with warnings; error

Question 3: What operation is required if the return value is 0?
Answer: No answer

Question 4: What operation is required if the return value is a non-zero positive value?
Answer: No answer

Question 5: What operation is required if the return value is a negative value?
Answer: the pcap_t * should be closed using pcap_close()

Question 6: What operation is required if successful without warnings?
Answer: No answer

Question 7: What operation is required if successful with warnings?
Answer: No answer

Question 8: What operation is required if there is an error?
Answer: the pcap_t * should be closed using pcap_close()

Question 9: Which function should be called before pcap_activate?
Answer: No answer

Question 10: Which function should be called after pcap_activate?
Answer: No answer

Question 11: What is the value of the first argument pcap_t *p supposed to be before pcap_activate?
Answer: No answer

Question 12: What is the value of the first argument pcap_t *p supposed to be after pcap_activate?
Answer: No answer

Question 13: How to check the first argument pcap_t *p before pcap_activate?
Answer: No answer

Question 14: How to check the first argument pcap_t *p after pcap_activate?
Answer: the pcap_t * should be closed using pcap_close()

Figure 3: An example of a distilled document with associated query questions and annotations.

Dataset construction. To address the problem of lacking the dataset, we manually review 15,000 API descriptions from TPC documents and annotate the answer to each designed question, which is a one-time task. Specifically, most answers are annotated with the continuous subsequence since we need the auxiliary information to generate the programming expression later. For questions that do not have an answer, such as those where the document does not necessitate further operations under certain return values, they are marked as unanswerable. Additionally, the annotated condition will be slightly rephrased to fill in the stem of the corresponding question ($Condition_i$). The rephrasing is performed automatically by Grammarly, a well-known grammar checker. Take the descriptions of libpcap in the last paragraph for example. We first annotate the "*success with/without warnings*" and "*error*" as conditions. Next, we directly put the condition into the stem of the question and rephrase it by using Grammarly. The final question will be formed as "*What operation is required if successful with/without warnings/there is an error?*" The results annotated by each author will be cross-checked. We describe the dataset details in Section 4.1. Moreover, we provide an example of a distilled document with associated query questions and annotations, as shown in Figure 3.

MRC system implementation. Currently, there are various off-the-shelf MRC systems. By comparing different MRC systems, we finally choose to use *RoBERTa* [29] in our system. *RoBERTa* is optimized from BERT [14] and has been proved efficient in many MRC tasks [35]. We fine tune the MRC system on our ground-truth dataset to obtain a great performance, which achieves an 88.23% F1 score.

Additionally, it is important to note that we will check for overlapping answers to different questions in order to identify any shared preconditions and avoid false positives. For instance, as illustrated in Figure 3, Questions 5, 8, and 14 share the same answer and have a common precondition, which is "*pcap_activate() returns a negative value on error*." If we fail to identify this precondition for Question 14, the system may erroneously check `pcap_t *` when `pcap_activate()` succeeds, resulting in a false positive.

3.3 Programming Expression Generation

Since API specifications are in natural language, they cannot be directly understood by our analysis engine without some form of processing or translation. Therefore, it is necessary to create a structured, machine-readable representation of each API specification, which is nevertheless very challenging. The main problem here is that API specifications do not have consistent formats and thus cannot be simply mapped by keywords or templates.

To address the problem, we design a novel NLP-guided approach by breaking the API specification into small phrases according to their POS, constructing the corresponding dependency tree, and then mapping phrases into programming expressions. These programming expressions are structured, machine-readable representations that follow a specific format, which we define as *Operation(argument1, argument2, ...)*. More specifically, **first**, we leverage the POS tagging [31] to annotate each word in an API specification and identify the relation between the words. **Second**, we create a dependency tree by combining words with a close relationship. The root node describes the relation between the two leaf nodes. Some words, e.g., determiners and adjectives, will be removed from the dependency tree. **Third**, we maintain a list that includes patterns to map common phrases into programming expressions, e.g., map "greater than or equal to" into ">=". The patterns consist of three parts. The first part is the patterns collected from previous works [10, 30]. The second part is generated from the first part by using synonym replacement. We leverage WordNet [8], a large lexical database of English, to find the synonym of a pattern. For instance, "*must be freed*" can be replaced as "*must be released*". The third part is generated from the first and second parts by changing the voice of the patterns, e.g., from the passive to the active voice and vice versa, which will not change the meaning of patterns. **Finally**, we traverse the dependency tree in post-order, and

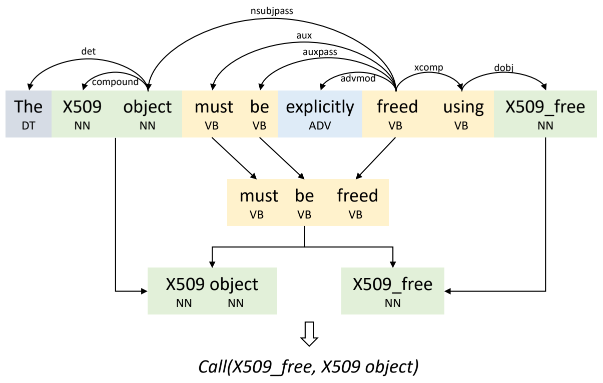


Figure 4: An example of programming expression generation.

map phrases in root nodes into programming expressions according to the patterns. In addition, to provide a clearer understanding of the process, we use the specification "*the X509 object must be explicitly freed using X509_free*" as an example, as shown in Figure 4. Initially, we annotate the POS for each word in the specification and identify their relationships. Next, we construct a dependency tree that represents the structure of the specification. The root node "*must be freed*" indicates the relationship between the two leaf nodes "*X509 object*" and "*X509_free*." Finally, the phrase "*must be freed*" is matched by our patterns and translated into the predefined "*CALL*" operation type. The two leaf nodes are used as arguments for the "*CALL*" operation.

3.4 Firmware Processing

The goal of firmware processing is to extract the contained objects and identify the target TPC in firmware. We mainly process firmware by two steps: firmware extraction and TPC identification.

3.4.1 Firmware Extraction

The first step is to unpack firmware and extract the contained objects, including the linked libraries, filesystems, and so on. These objects are vital for later TPC identification and binary encoding. To achieve this goal, we mainly leverage the extraction module of *FirmSec* [48], which is implemented based on *binwalk* [1] and provides extra support for three filesystems: SquashFS, JFFS2, and YAFFS. During our analysis, we find the original extraction module cannot deal with the UBI filesystem. To address this issue, we make an improvement to the extraction module by implementing a plugin that is specifically designed to extract objects from the UBI filesystem.

3.4.2 TPC Identification

The second step is to identify the target TPC in firmware and recover the unknown TPC-related APIs. Locating the TPC and its APIs in firmware is a basic requirement for further usage violation analysis. Generally, source-level API misuse detectors can easily find the target APIs by searching the source code. Nevertheless, it is challenging to locate the target APIs at the binary-level since firmware may be stripped and the symbols are removed. The main problem here is to identify the TPC and its APIs in firmware with limited information. To solve this problem, we mainly leverage *FirmSec*, a state-of-the-art tool for TPC identification, with two enhancements. *FirmSec* can recognize the TPCs used in firmware at TPC-level and version-level with high precision by using syntactical features (e.g., function names) and control-flow graph (CFG) features from TPCs and firmware. Specifically, *FirmSec* first leverages the edit distance and ratio-based matching to determine the similarity of unstripped syntactical features. Next, *FirmSec* uses a customized neural network to compare the CFG features. Finally, *FirmSec* combines the results of the syntactical feature matching and CFG feature matching to identify TPCs in IoT firmware.

Our enhancements are as follows. **First**, except for the original syntactical features (e.g., string literals and function names) used by *FirmSec*, we adopt two extra syntactical features. *B2SFinder* [44] indicates that complex branch sequences (e.g., switch/case and if/else structures) are hardly changed during the compilation. The constant conditions used in switch/case and if/else structures will not be removed in the stripped firmware while string literals and function names may be removed. Therefore, we extract the constant conditions in switch/case and if/else structures as new syntactical features. **Second**, we simplify the CFG feature matching process by eliminating redundant CFG feature matching computations. We filter out some obvious irrelevant functions, such as having completely different function names and no corresponding special strings, in advance based on the syntactical features mentioned above and only perform the CFG feature matching on the remaining functions.

Additionally, though syntactical features are important to *FirmSec*, it can still perform TPC identification when all syntactical features are stripped. Specifically, *FirmSec* achieves a precision of around 90% and a recall of 80% on TPC identification by solely relying on CFG features [48]. Therefore, the existence of CFG features is the minimum necessity to enable our TPC identification. Moreover, *FirmSec* can recover the stripped function names according to CFG feature matching results. These recovered function names will be further used to locate APIs in IoT firmware.

3.5 Rule-driven Analysis Engine

Though we have extracted API specifications from TPC documents and generated the corresponding programming expressions, we still lack a practical method to leverage them for usage violation detection. Existing API misuse detectors do not consider to use API specifications too much. This is because they mainly perform the detection at the source-level, and thus it is possible and much easier for them to rely on existing tools for the API misuse detection. For instance, *Advance* uses a sophisticated system *CodeQL* [2] to conduct API misuse detection. Nevertheless, detecting the usage violation at the binary-level is much more complicated than that at the source-level. It is challenging to obtain the exact operations related to API usage at the binary-level due to the complex logical relationships between assembly instructions. Currently, there is no solution that could properly handle this problem. Therefore, to address the challenge, we introduce Datalog to our task, which is powerful in logical inference. Nevertheless, there are also two challenges to apply Datalog to our task. First, it is challenging to extract the information in firmware about API usage and encode it into facts. Currently, there is no uniform standard for encoding firmware into facts. Second, it is difficult to design Datalog rules to perform the usage violation check on the generated facts since the logical inference for different usage violations varies widely. To solve the above challenges, we implement a rule-driven analysis engine with the following designs.

3.5.1 Binary Encoding

To address the challenge of encoding the binary into Datalog facts, we develop a binary-to-Datalog encoding tool that takes advantage of *Ddisasm* [18], along with our customized facts that mainly focus on the register usage of an API. The tool accepts the extracted binaries, with the identified TPC-related information, from IoT firmware as input, and encodes them into Datalog facts.

Ddisasm, a binary analysis and rewriting tool, has implemented a method to encode the binary into Datalog facts. Figure 5 presents the standard used by *Ddisasm* of encoding each instruction in binary into a set of initial Datalog facts. Based on initial Datalog facts, *Ddisasm* further generates more than 300 kinds of facts through various built-in rules. These facts represent various information in the binary. Nevertheless, we find only a few facts related to the API usage, hindering the TPC usage violation analysis. To solve this problem, we customize rules to generate facts that correspond to the usage of APIs, which are related to return values, arguments, and adjacent calling functions. Specifically, **first**, we generate facts describing the register that holds the return value or the argument of an API. We declare relations related to the register usage and then customize rules for each relation based on existing facts. The output of the relation is the generated fact. For instance, we first declare a relation `function_r0_usage` to describe the R0 register usage, which holds the return value or the argument of a function in ARM32. Next, we implement the rules based on the existing facts, e.g., `instruction_get_op` and `op_regdirect_contains_reg` to obtain the corresponding facts of `function_r0_usage`. As shown in Figure 6, we present an example of a rule that verifies whether the return value is checked using the condition $<= 0$. The rule is based on three existing facts and the output of the rule is the fact

$$\begin{array}{rl}
\text{(address)} & A \\
\text{(size of the instruction)} & S_{\text{instr}} \\
\text{(size of the data element)} & S_{de} \\
\text{(register)} & R \\
\text{(segment register)} & R_{\text{seg}} \\
\text{(base register)} & R_{\text{base}} \\
\text{(index register)} & R_{idx} \\
\text{(instruction code)} & I \\
\text{(unique identifier of the i-th operand)} & O_i \\
\text{(immediate)} & IM \\
\text{(multiplier)} & M \\
\text{(displacement)} & D
\end{array}$$

$$\begin{array}{lll}
Predicate & ::= & instruction(A, S_{instr}, P, I, O_1, O_2, O_3, O_4) \\
& | & invalid(A) \\
& | & op\_regdirect(O_i, R) \\
& | & op\_immediate(O_i, IM) \\
& | & op\_indirect(O_i, R_{seg}, R_{base}, R_{idx}, M, D, S_{de})
\end{array}$$

Figure 5: Initial datalog facts used by *Ddisasm*.

```
.decl function_r0_usage_blez0(EA:address)
.output function_r0_usage_blez0

function_r0_usage_blez0(EA:address) :-
        instruction(EA,_,_,"BLEZ",_,_,_,_,_,_),
        instruction_get_op(EA,_,Op),
        op_regdirect_contains_reg(Op,"R0").
```

Figure 6: An example of a rule that checks the return value.

`function_r0_usage_blez0`. This rule can be interpreted as first finding the instructions that have the instruction code BLEZ and then further identifying those instructions that include the R0 register. **Second**, we generate facts describing the adjacent function calls before or after an API. The function calls can be identified by focusing on certain instructions, e.g., branch instructions. Therefore, similar to the first step, we define rules to extract the function names in certain instructions. **Finally**, we generate facts describing the operations under certain return values, including function calls, value assignments, etc. In this step, we define rules to extract the function names in certain instructions and the register usage after the return value is checked. The facts generated by the second and third steps will be further used to identify the causality violation of an API.

3.5.2 Checker Implementation

To address the challenge of performing the usage violation check on the generated facts, we design four checkers based on the features of different TPC usage violations.

Deprecated API violation checker. We maintain a list of deprecated APIs for each TPC. Therefore, comparing the function names extracted from firmware with the deprecated APIs in the list is straightforward. Nevertheless, only comparing the function name may result in false positives. For instance, some firmware images may contain a self-implemented function with the same name as a deprecated API. Also, the different TPCs used in firmware may have functions with the same name. To solve this problem, we additionally check three features of the function when the function name matches the list, including the number of arguments, the argument types, and the possible function calling sequence.

Return value violation checker. In this checker, we mainly focus on the operation of the registers that hold the return value of the function, e.g., the R0 register in ARM32. The checker accepts constraints on API return values from programming expressions as input. More specifically, we focus on two kinds of return value violations: lack of checking the return value and incorrect check of the return value. In the former case, we check whether the specific register involves in compare instructions (e.g., CMP) or branch instructions (e.g., BEQ). In the latter case, we further check whether the compared value (e.g., 0) and the corresponding compare operation (e.g., BGE) are consistent with the constraints (e.g., greater than or equal to 0). It is important to mention that we do not simply restrict the scope of return value detection to the current caller function. Though the API is called in the current caller function, its return value may be checked in other functions. For instance, function A calls function B and checks its return value, while function B in turn calls the API and returns the return value of the API. Lack of consideration for this situation could result in false positives. We set the depth of the indirection return value check at most one function call.

Argument violation checker. Similar to the return value violation checker, we focus on the operation of the registers that hold the arguments of the function, e.g., the R0-R3 registers in ARM32. The checker accepts constraints on API arguments from programming expressions as input. Most constraints are related to the value of the arguments (e.g., the argument should be set to 0 before the call) and the handling of pointer arguments (e.g., the pointer argument should be freed after the call). Therefore, we first analyze the value of each argument and the operations of the pointer argument, before and after the call. Next, we check whether the values or operations are consistent with the constraints. In addition, similar to the indirection return value check, we also consider the argument that may be processed indirectly, e.g., free a pointer argument in a different function, to avoid false positives.

Causality violation checker. In this checker, we mainly focus on the functions that are called before or after the API, and the operations under certain return values. The checker accepts constraints on API causality from programming expressions as input. More specifically, we pay special attention to two kinds of causality violations: lack of calling the required function before or after the API (e.g., lock should be called before unlock), and incorrect or without operations under certain return values (e.g., call `SSL_get_error()` if `SSL_do_handshake()` returns 0). In the former case, we define rules to obtain the five closest functions that are called before or after the API respectively. Our analysis indicates that the checker achieves optimal precision when the limit is set to five. We then check whether the required function is in the list of collected functions. In the latter case, we focus on the operations after the return value is checked, including function calls, value assignments, and so on. Similar to the return value violation checker, we do not restrict the scope of causality violation detection to the current caller function to avoid false positives.

The four checkers have multiple built-in rules that are plug-and-play and can be easily extended to new TPCs without requiring additional development efforts. This adaptability stems from the fact that the root causes for usage violations persist consistently across different TPCs, and all checkers are tailored in accordance with the specific root causes of usage violations. In addition, although the API specifications of various TPCs may differ, they will be standardized into a uniform format through programming expression generation.

3.6 Usage Violation Detection

The usage violation detection module aims to perform the Datalog checkers on the Datalog facts of binaries to recognize the potential TPC usage violations. In this module, we first implement a simple parser to pass programming expressions into Datalog checkers according to the operation type of programming expressions. Next, we leverage *Soufflé*, an off-the-shelf Datalog engine, to execute the Datalog checkers. Then, we set *GHIDRA* as our front-end and present the visualizing results on it according to *d3re* [40]. According to the results, we finally generate a vulnerability report that indicates the recognized TPC usage violations in IoT firmware.

4 System Evaluation

In this section, we first introduce the datasets used for evaluation. Next, we evaluate the performance of key components in UVSCAN and the overall performance of UVSCAN. We also compare UVSCAN with multiple state-of-the-arts.

4.1 Dataset

To enable our evaluation, we create the following six datasets.

TPC documents dataset ($D_{TPC-DOC}$) includes the latest documents from four TPCs: OpenSSL, SQLite, libpcap, and libxml2. More specifically, OpenSSL has 1,554 API calls, SQLite has 221 API calls, libpcap has 69 API calls, and libxml2 has 1,497 API calls. We choose these four TPCs for concept validation for the following reasons. (1) They are widely used in IoT firmware according to the results presented in [48]. (2) They have the corresponding ground-truth usage violation dataset created by Lv et al. [30], and Gu et al. [21].

API description dataset (D_{DESC}) is used for training and evaluating our sentiment-based document distillation model. It includes the randomly selected API descriptions from the aforementioned TPC documents. Specifically, D_{DESC} contains $4,086$ API descriptions from OpenSSL and SQLite, including 927 relevant API descriptions and $3,159$ irrelevant API descriptions. We split D_{DESC} into three sets, training set ($D_{DESC_{train}}$), development set ($D_{DESC_{dev}}$), and testing set ($D_{DESC_{test}}$) respectively, according to the ratio of 6:2:2. In addition, to evaluate the generalization ability of our model, we construct an extra dataset ($D_{DESC_{new}}$) consisting of 733 API descriptions from libpcap and libxml2. $D_{DESC_{new}}$ includes 141 relevant API descriptions and 592 irrelevant API descriptions.

MRC dataset (D_{MRC}) is used for training and evaluating the MRC system of UVSCAN. It includes the manually labeled question-answer pairs for each API from OpenSSL and SQLite. More specifically, D_{MRC} contains 19,536 question-answer pairs of 1,775 API calls. We split D_{MRC} into three sets, training set ($D_{MRC_{train}}$), development set ($D_{MRC_{dev}}$), and testing set ($D_{MRC_{test}}$) respectively, according to the ratio of 6:2:2. Besides, we create an additional dataset ($D_{MRC_{new}}$), which includes 1,058 question-answer pairs of 63 API calls from libpcap and libxml2. The APIs in $D_{MRC_{new}}$ have the corresponding manually distilled documents in $D_{DESC_{new}}$. We use $D_{MRC_{new}}$ to evaluate the generalization ability of the MRC system and conduct the error propagation analysis.

Programming expression dataset (D_{PE}) contains the ground-truth programming expressions. We randomly select relevant API descriptions from the aforementioned four TPC documents and manually label the corresponding programming expressions. More specifically, D_{PE} includes 600 programming expressions.

Real-world usage violation dataset ($D_{Real-UV}$) includes the known usage violations, which are in popular C programs, that correspond to the aforementioned four TPCs. The purpose of $D_{Real-UV}$ is to evaluate the usage violation detection accuracy of UVSCAN. As shown in Table 2, $D_{Real-UV}$ includes 77 known usage violations collected from two ground-truth datasets: 59 usage violations in 24 programs from *Advance* [30], and 18 usage violations in 2 programs from *APIMU4C* [21] (the gray lines in Table 2).

Artificial usage violation dataset ($D_{Artif-UV}$) includes the manually created usage violations. The purpose of this dataset is to enlarge $D_{Real-UV}$ and provide a comprehensive evaluation of UVSCAN. We create $D_{Artif-UV}$ by manually inserting the incorrect code into the open-source IoT firmware, which includes 69 manually created usage violations from 23 firmware images, as shown in Table 2.

4.2 Evaluation of UVSCAN

4.2.1 API Description Extraction Accuracy

In this step, we evaluate the API description extraction accuracy of UVSCAN with two metrics: accuracy and F1 score. We also compare UVSCAN with three off-the-shelf tools: *Advance* [30], *RCNN* [27], and *ALICS* [33]. *Advance* is the first one that adopts semantic analysis to find API descriptions in TPC documents. It is important to mention that *Advance* treats the API descriptions as API specifications directly. *RCNN* is a popular classifier that can also be applied to sentiment analysis tasks. *ALICS* leverages semantic templates with shallow parsing to recognize API descriptions.

We train UVSCAN from scratch on $D_{DESC_{train}}$ for 100 epochs with a batch size of 128. We set the learning rate

Table 2: Ground-truth usage violation dataset.

TPC	Program	Version	# Usage Violation
		$D_{Real-UV}$	
OpenSSL	dovecot	0eaf77d	1
		394391e	1
	mutt	101e05d6	6
	ntp	2383333	1
		c70fc4b	1
	openfortivpn	07946c1	1
		f755c99	1
		0007b2d	3
	ovs	9da8b2f	1
	PHP	7a4584d	1
	SPICE	ef9a8bf	1
	unbound	ffed368	1
	httpd	2.4.37	11
	curl	7.63.0	7
SQLite	anope	2a5e782	1
		aeefe16	1
	darktable	70820b1	1
	librdf	5d074c1	1
libpcap	arp-scan	f013b45	1
	arping	b37fb24	1
	ettercap	89b5542	5
		dfcabfc	2
		891a281	1
	freeradius	57fbb95	1
	knock	4b8ad4d	1
	libnet	008c994	1
	ntop	66f6f48	1
	tcpdump	39be365	1
		224b073	2
	wireshark	51a99ca	1
libxml2	abiword	80fee4c	2
		ebcc445	4
		$D_{Artif-UV}$	
OpenSSL	OpenWrt	-	6
	Tomato-shibby	-	6
	AsusWrt	-	6
SQLite	OpenWrt	-	6
	Tomato-shibby	-	6
	AsusWrt	-	5
libpcap	OpenWrt	-	6
	Tomato-shibby	-	6
	AsusWrt	-	5
libxml2	OpenWrt	-	6
	Tomato-shibby	-	6
	AsusWrt	-	5

Table 3: API description extraction accuracy.

Tool	$D_{DESC_{test}}$	
	Accuracy	F1
UVSCAN	92.41%	85.24%
Advance [30]	85.07%	74.37%
RCNN [27]	76.25%	61.50%
ALICS [33]	38.43%	29.02%

Table 4: API specification inference accuracy.

Tool	$D_{MRC_{dev}}$		$D_{MRC_{test}}$	
	EM	F1	EM	F1
UVSCAN (*RoBERTa*) + Distilled Documents	87.48%	90.17%	86.52%	88.23%
RoBERTa + Original Documents	79.34%	81.06%	76.91%	78.60%

Table 5: Generalization ability and error propagation analysis on the MRC system of UVSCAN.

Tool	$D_{MRC_{new}}$	
	EM	F1
UVSCAN (*RoBERTa*) + Manually Distilled Documents	88.19%	90.85%
UVSCAN (*RoBERTa*) + Automatically Distilled Documents	84.40%	85.89%

to 1e-3 and the dropout rate to 0.25. We save the model when it achieves the best accuracy on $D_{DESC_{dev}}$ during the 100 epochs. We also set up *Advance*, *RCNN*, and *ALICS* according to their instructions respectively. Both *Advance* and *RCNN* are trained on $D_{DESC_{train}}$. As shown in Table 3, our system achieves 92.41% accuracy and 85.24% F1 score on $D_{DESC_{test}}$, which are higher than the other three methods. The main reason for our excellent performance is that UVSCAN solves the coreference in TPC documents and makes good use of contextual information to identify the sentiment of APIs. Nevertheless, *Advance* and *RCNN* do not solve the coreference first and fail to fully leverage the contextual information. Besides, *RCNN* does not leverage the attention mechanism, hindering its performance on the sentiment analysis task. Besides, the template-based matching method adopted by *ALICS* cannot correctly handle different TPC documents since they do not have a consistent format. In addition, we evaluate the generalization ability of UVSCAN on extracting API descriptions from new TPC documents. Specifically, UVSCAN achieves 89.05% accuracy and 74.86% F1 score on $D_{DESC_{new}}$, indicating its great ability to generalize when presented with new TPC documents.

We further explore the possible reasons for false positives and false negatives of UVSCAN as follows. First, our coreference resolution model cannot handle a part of coreferences in the TPC document, making it difficult to identify the exact sentiment of several API descriptions, causing false positives. Second, our sentiment analysis model cannot recognize some relevant API descriptions that do not exhibit a clearly expressed sentiment. For example, the description "*bindings are not cleared by the sqlite3_reset() routine*" cannot be recognized by the model due to the lack of an apparent strong sentiment.

4.2.2 API Specification Inference Accuracy

In this step, we evaluate the API specification inference accuracy of UVSCAN with two metrics: Exact Match (EM) and F1 score. EM represents the percentages of predicted answers that match any of the ground-truth answers. EM and F1 score are two major metrics used in SQuAD [35, 36], which is a well-known MRC dataset, to evaluate the performance of MRC systems. In this paper, we also adopt these two metrics to evaluate UVSCAN.

We set the learning rate to 1e-5 and train UVSCAN (*RoBERTa*) for 500K steps with a batch size of 256 sequences on $D_{MRC_{train}}$. As shown in Table 4, the EM and F1 score of UVSCAN are 87.48% and 90.17% respectively on the development set, and 86.52% and 88.23% respectively on the testing set. To further evaluate the importance of document distillation, we conduct an ablation experiment by directly comparing UVSCAN with *RoBERTa*. More specifically, *RoBERTa* is trained on $D_{MRC_{train}}$ with the same parameters as UVSCAN but without eliminating irrelevant API descriptions. Since our MRC system is developed from *RoBERTa* as well, this ablation experiment can be regarded as we perform *RoBERTa* on distilled documents and original documents, respectively. The distilled documents are obtained based on our document distillation model. The results show that UVSCAN outperforms *RoBERTa* on both $D_{MRC_{dev}}$ and $D_{MRC_{test}}$, highlighting the significance of document distillation. During the analysis, we notice that *RoBERTa* extracts incorrect answers from irrelevant API descriptions, resulting in false positives.

Besides, although the MRC system of UVSCAN is developed from *RoBERTa*, which has already demonstrated great generalization capabilities, we still evaluate its generalization ability on $D_{MRC_{new}}$. In addition, we analyze the error propagation between the API description extraction module and the API specification inference module by performing UVSCAN on $D_{MRC_{new}}$ with manually distilled and automatically distilled documents, respectively. Notably, as shown in Table 5, UVSCAN achieves 84.40% EM and 85.89% F1 score on $D_{MRC_{new}}$ with automatically distilled documents, highlighting its great capacity to generalize when faced with new TPC documents. Moreover, both the EM and F1 score on manually distilled documents are higher than those on automatically distilled documents, demonstrating error propagation between the two modules. The false positives and false negatives caused by the API description extraction module have an influence of about 5% on the result of the API specification inference module.

We further analyze the possible reasons for false positives and false negatives caused by the MRC system of UVSCAN as follows. First, UVSCAN wrongly extracts answers from TPC documents for unanswerable questions, leading to false positives. Addressing the unanswerable questions is still an open research question for the MRC task. Second, complex or ambiguous API descriptions cannot be properly handled by UVSCAN, which may also cause false positives and false negatives. For instance, the description "*the return values of the SSL*_ctrl() functions depend on the command supplied via the cmd parameter*" lacks clarity in stating the return values of `SSL*_ctrl()`, and this information is also not provided in other related descriptions. Besides, the function name `SSL*_ctrl()` has a wildcard character and is not a standard one, making it challenging for our MRC system to comprehend its meaning.

Table 6: Programming expression generation accuracy.

Tool	D_{PE} Recall	D_{PE} FNR
UVSCAN	82.17%	17.83%
Jdoctor [10]	30.50%	69.50%
DRONE [50]	50.33%	49.67%
Advance [30]	72.50%	27.50%

4.2.3 Programming Expression Generation Accuracy

In this step, we evaluate the programming expression generation accuracy of UVSCAN with two metrics: recall and false negative rate. We also compare UVSCAN with three state-of-the-arts: *Jdoctor* [10], *DRONE* [50], and *Advance*. *Jdoctor* leverages the pattern matching and lexical matching to translate specifications into expressions. *DRONE* uses more than 60 heuristics to translate Java API specifications to expressions. We follow the instructions declared in their papers to set up them and perform them on D_{PE}.

As shown in Table 6, UVSCAN achieves a recall of 82.17% and a false negative rate of 17.83%, which is the highest recall and the lowest false positive rate among all methods. The improvement from *Advance* to UVSCAN shows the effectiveness of extracting more precise API specifications before generating programming expressions. As for another two methods, first, *Jdoctor* performs well in translating API specifications with simple arithmetic and logical operations but fails to deal with complicated API specifications. Second, for *DRONE*, we find its more than 60 heuristics are well suitable for Java API documents but do not perform well on C API documents.

Our further analysis shows that the false negatives of UVSCAN are mainly due to the incorrect API specifications inferred from several API descriptions, leading us to generate the erroneous programming expressions. Besides, though we maintain lots of patterns to map phrases into programming expressions, they cannot cover several uncommon phrases, leading to false negatives. For instance, we lack a pattern to match the uncommon phrase "*must be an index*," which appears only within the following API description: "*the second argument must be an index into the aConstraint[] array*."

4.2.4 Usage Violation Detection Accuracy

In this step, we leverage two metrics, precision and recall, to evaluate the usage violation detection accuracy of UVS-

Table 7: Usage violation detection accuracy.

Performance	UVSCAN x86	UVSCAN ARM	UVSCAN MIPS	*Advance*	*APISAN*	*APEx*
$D_{Real-UV}$						
Precision	72.84%	74.70%	77.03%	80.72%	17.31%	23.07%
Recall	76.62%	80.52%	74.03%	87.01%	11.69%	7.79%
$D_{Artif-UV}$						
Precision	68.92%	74.32%	76.47%	77.33%	25.49%	34.62%
Recall	73.91%	79.71%	75.36%	84.06%	18.84%	13.04%

CAN. Since we are the first to address the TPC usage violation problem in binary IoT firmware, we cannot conduct the exact same comparison with previous works. We finally choose to compare with three source-level API misuse detectors: *Advance*, *APISAN* [45], and *APEx* [26]. *APISAN* first infers API specifications by analyzing the source code of various usage examples of the API in C programs, and then implements a series of checkers for different misuse cases. *APEx* focuses on API error specifications. It first analyzes the error-handling code in multiple API call sites to get error constraints, and then performs under-constrained symbolic execution to find the misuses. Besides, since we only compare with fully automated API misuse detectors, we do not compare with *ARBITRAR* [28], which requires human interaction when executing the system. In summary, three API misuse detectors perform on the source-level while UVSCAN performs on the binary-level. We perform UVSCAN on $D_{Real-UV}$ and $D_{Artfi-UV}$ respectively. The programs in these two datasets are compiled into three different architectures (x86, MIPS, and ARM) along with two different optimization levels (O0 and O3) under GCC 5.4.0.

For UVSCAN on $D_{Real-UV}$, as shown in Table 7, it achieves 72.84% precision, 76.62% recall on x86, 74.70% precision and 80.52% recall on ARM, and 77.03% precision and 74.03% recall on MIPS. For UVSCAN on $D_{Artfif-UV}$, it achieves 68.92% precision and 73.91% recall on x86, 74.32% precision and 79.71% recall on ARM, and 76.47% precision and 75.36% recall on MIPS. Though $D_{Real-UV}$ and $D_{Artfi-UV}$ have the ground-truth, we still manually check the reported results to avoid the potential corner cases. The results show that UVSCAN has great performance across different architectures. Besides, we notice that though UVSCAN is performed on binary-level, it still has much higher precision and recall than *APISAN* and *APEx*, and is very close to the performance of *Advance*. This is because we infer precise API specifications from TPC documents and design a rule-driven analysis engine that can handle complex logical inference. We further analyze the false positives and false negatives of these API misuse detectors. For *APISAN* and *APEx*, first, we find both of them cannot detect deprecated APIs. Second, most of their inferred API specifications are incomplete or incorrect, leading to false positives and false negatives. The main reason is that *APISAN* and *APEx* heavily rely on analyzing the source-level C programs, which may contain the API usage examples, to infer the API specifications. Nevertheless, the C programs may not use all APIs in a TPC and even cannot cover all possible usage examples for an API, resulting in incomplete or incorrect API specification inference. For *Advance*, though its inferred API specifications are more accurate than *APISAN* and *APEx*, the incorrect verification code and the limited performance of *CodeQL* cause the false positives and false negatives. For instance, *CodeQL* fails to perform a dataflow analysis in some complicated cases, leading to false positives.

In addition, to find the causes of false positives and false negatives of UVSCAN, we further explore them as follows. First, incorrect and imprecise API specifications are the main reason for the false positives and false negatives. Currently, we cannot extract the API specifications without errors. Second, incorrect programming expressions may also lead to multiple false positives and false negatives.

5 Large-scale Analysis on IoT Firmware

To obtain an in-depth status quo understanding of the TPC usage violation problem in IoT firmware, we further leverage UVSCAN to conduct a large-scale analysis on 4,545 firmware images. In this section, we aim to answer the following research questions.

- **RQ1:** Which are the most prevalent TPC usage violations in IoT firmware?
- **RQ2:** What are the practical impacts of TPC usage violations on IoT firmware?

5.1 Experimental Setup

Before conducting the analysis, we need to first construct a large-scale firmware dataset. To achieve this goal, we obtain 34,136 firmware images from *FirmSec* [48], containing 35 different kinds of firmware from hundreds of vendors. These firmware images use a total of 584 TPCs. Since we study four TPCs for concept validation, we do not perform the analysis on all firmware images. Therefore, we first use the firmware processing module of UVSCAN to identify the firmware images that employ the four TPCs we studied. Additionally, we cross-check our results by using the original analysis results obtained from the authors of *FirmSec*. As shown in Table 8, we find 4,545 firmware images from 9 vendors using at least one of the four TPCs. More specifically, 4,126 firmware images adopt OpenSSL, 2,253 firmware images contain SQLite, 3,820 firmware images use libpcap, and 1,229 firmware images utilize libxml2.

5.2 Usage Violation Distribution

This subsection answers RQ1. As shown in Table 9, we present the large-scale usage violation detection results on our

Table 8: Firmware dataset composition.

Vendor	Category	# Firmware	TPC			
			OpenSSL	SQLite	libpcap	libxml2
D-Link	IP Camera	156	156	44	151	0
	Router	437	424	224	254	0
	Switch	49	49	0	49	0
	Smart Home	21	21	16	12	0
TP-Link	IP Camera	268	268	87	226	0
	Router	757	725	297	711	0
TRENDnet	IP Camera	225	155	40	170	0
	Router	255	216	64	163	0
	Switch	112	112	10	112	0
DAHUA	IP Camera	207	88	10	204	0
Xiongmai	IP Camera	105	38	4	104	0
Fastcom	Router	103	48	16	98	0
Xiaomi	Router	20	20	20	20	0
TSmart	Smart Home	344	336	61	263	0
OpenWrt	Router	1,486	1,470	1,360	1,283	1,229
Overall		4,545	4,126	2,253	3,820	1,229

firmware dataset. We find the firmware dataset consists of consecutive firmware sets, involving at least 937 firmware images. Each consecutive firmware set includes a series of historical firmware images belonging to the same device. Therefore, it is possible that the same usage violation exists in multiple historical firmware versions associated with the same device. In our analysis, we treat each historical firmware image as a distinct entity. The same violation that exists in different historical firmware will be counted cumulatively. Besides, to get more accurate results of the deprecated API violation, we maintain version-specific deprecated API lists for each TPC in the large-scale evaluation. According to the results, we have the following findings.

First, TPC usage violations are widespread in IoT firmware. More specifically, we detect $27,621$ usage violations of the four TPCs in the 4,545 firmware images, including 11,348 deprecated API violations, 8,304 causality violations, 5,891 return value violations, and 2,078 argument violations. **Second**, the deprecated API violation takes the majority of the overall usage violations, accounting for 41%. **Third**, the causality violation and return value violation are also widespread in IoT firmware, accounting for 30% and 21%, respectively. For the causality violation, most usage violations are due to lacking a call for some APIs. **Finally**, more than 40% usage violations are inherited from other TPCs adopted by firmware. For instance, the latest version (2.22.19) of Xiaomi Router Mini is still using cURL 7.33.0, which contains a number of OpenSSL usage violations. Besides, the four TPCs also have some internal usage violations. For example, we find that the OpenSSL used in many TP-Link does not check the return value of `BN_CTX_get`, which may cause the Denial-of-Service attack.

We have reported all the usage violations to the corresponding vendors. Up to now, D-Link, TP-Link, Xiaomi, and TSmart have responded to us and 206 usage violations have been

Table 9: Usage violation distribution.

TPC	# Deprecated API Violation	#Causality Violation	# Return Value Violation	# Argument Violation
OpenSSL	4,831	3,679	3,521	1,073
SQLite	2,740	1,996	931	112
libpcap	3,359	2,515	1,364	857
libxml2	418	114	75	36
Overall	11,348	8,304	5,891	2,078

confirmed by vendors as vulnerabilities, and seven of them have been assigned CVE IDs with high severity. In addition, at least 117 of the 206 confirmed violations have been fixed.

5.3 Practical Impacts

This subsection answers RQ2. Though we have identified many usage violations in IoT firmware, we still lack an understanding of the practical impacts of them. Therefore, we conduct further analysis to explore the potential consequences of these usage violations and gain deeper insights into their practical impacts. We manually analyze hundreds of usage violations and collect feedback from vendors about our reported usage violations. Based on the analysis results, we classify the possible impacts of TPC usage violations into three categories: security vulnerabilities, ordinary bugs, and no impacts. **First**, our analysis identifies a set of usage violations that could be exploited to perform attacks, e.g., the Man-In-The-Middle attack, and we regard these types of violations as security vulnerabilities. To provide real-world examples of the attacks that could be carried out using these violations, we conduct two case studies, which are discussed in the following subsections. **Second**, our analysis reveals that many usage violations are ordinary bugs that may result in the malfunctioning of firmware but cannot be leveraged for attacks. One such example is found in certain TP-Link firmware images that exhibit an incorrect verification of the return value of `SSL_write()` as < 0 instead of $<= 0$, causing the usage violation. TP-Link regards this type of violation as a conventional bug rather than a vulnerability as it merely causes functional errors without posing any threat to potential attacks. **Finally**, we discover that some usage violations have no impact on firmware. For instance, we detect some firmware images that do not call `SSL_get_error()` when `SSL_do_handshake()` returns 0, though it is a requirement stated in the OpenSSL document. Nevertheless, since `SSL_get_error()` only returns the error code for diagnosis purposes, this particular violation does not impact the firmware.

5.3.1 Denial-of-Service Attack

Based on our further analysis, we find some firmware images from TSmart do not check the return value of many APIs from OpenSSL, including `SSL_renegotiate`, `SSL_do_handshake`, `SSL_peek`, and `SSL_set_session_id_context`. Lacking a check for the return value of the above APIs will result in serious consequences, e.g., the Denial-of-Service attack. For instance, `SSL_renegotiate` is an essential API to handle the SSL renegotiation process between the client and the server. It has two return values that indicate the success or error status respectively. The SSL renegotiation process usually consumes many computing resources since it involves complicated calculation. If the server does not check the return value of `SSL_renegotiate`, the server will mistakenly enter the next step, e.g., `SSL_do_handshake`, leading to more waste of resources. Therefore, if some low-power devices, e.g., IoT devices, allow insecure renegotiation and fail to impose restrictions on renegotiation requests, an attacker could initiate many renegotiation requests to exhaust their computing resources, causing a Denial-of-Service attack.

To exploit this vulnerability, we mainly employ emulating-based dynamic analysis. We here adopt *FIRMADYNE* [11] to conduct dynamic analysis on the firmware. *FIRMADYNE* supports emulating the Linux-based firmware on the desktop. It overcomes the general challenges in emulating firmware, such as the presence of hardware-specific peripherals. After emulating the firmware, we create many malicious clients to send renegotiation requests to the firmware. Finally, we successfully perform the Denial-of-Service attack on the vulnerable firmware.

```
const SSL_METHOD *method;
SSL_CTX *ctx;
SSL *ssl;
...
/* Set SSLv2 client hello */
method = SSLv23_client_method();
...
/* Create a new SSL context */
ctx = SSL_CTX_new(method);
...
/* Create new SSL connection state object */
ssl = SSL_new(ctx);
...
/* Set the verification flags for ssl */
SSL_CTX_set_verify(ctx, SSL_VERIFY_PEER,...);
...
SSL_connect(ssl);
```

Figure 7: SSL certificate validation process.

5.3.2 Man-In-The-Middle Attack

Typically, firmware adopts OpenSSL to perform certificate validation during SSL/TLS handshake. Figure 7 presents a standard SSL certificate validation process. Nevertheless, we find a set of firmware images from D-Link lack a call to the function `SSL_CTX_set_verify`, which is an essential function for SSL certificate validation. Without calling `SSL_CTX_set_verify`, attackers could forge a certificate to deceive devices and then capture sensitive information or compromise devices, leading to Man-In-The-Middle attacks.

To exploit this vulnerability, we also adopt *FIRMADYNE* to conduct dynamic analysis on the firmware. After emulating the firmware, we create a host with a forged certificate in the same local area network. We leverage ARP spoofing, also known as ARP cache poisoning, to attack the emulated firmware and the gateway, causing any traffic meant for the targeted host to be sent to our host instead. We then redirect the traffic to the target host and successfully establish the connection with the emulated firmware and the target host separately. Finally, we successfully obtain the plaintext transferred between the firmware and the server, which includes sensitive information, e.g., password.

We have contacted D-Link to report the vulnerability, and they tell us they have fixed this vulnerability in the past [6]. It is reasonable since most firmware images in the dataset were collected by Zhao et al. [48] several years ago. Nevertheless, we find this vulnerability still poses a threat to a significant number of D-Link devices. We leverage *Shodan* [7] to search affected D-Link devices and discover more than $10,000$ vulnerable devices are still publicly available on the Internet.

6 Discussion

Ethics. We pay special attention to the potential ethical issues in this work. First, we obtain all the used firmware from legitimate sources. For the firmware images in $D_{Artif-UV}$, we collect them from the corresponding official websites. For the firmware images used in the large-scale analysis, we collect them from *FirmSec* and strictly follow the research-only agreement. Second, we have done responsible disclosure to report all the detected usage violations to vendors.

Limitations and future work. Though UVSCAN achieves a great performance in detecting the TPC usage violation in IoT firmware, it still has several limitations for future research as follows. **First**, UVSCAN has false positives and false negatives. In our experience, the false positives and false negatives are mainly introduced by incorrect API specifications and incorrect rule-driven analysis. Though our NLP model achieves high accuracy in API specification inference, it still cannot process some complicated cases. Meanwhile, our rule-driven analysis mainly targets four kinds of usage violations, which may not cover all possible cases. Besides, some usage violations are extremely complicated and cannot be appropriately handled by UVSCAN. In the future, we will improve the performance of our NLP model and enhance UVSCAN to cover more complicated cases. **Second**, UVSCAN currently is limited to analyzing firmware of three architectures: x86, ARM, and MIPS. In the future, we will extend the analysis to more architectures, e.g., RISC-V, by adding new disassemblers on top of our binary-to-Datalog encoding tool. **Third**, UVSCAN does not consider code obfuscation when analyzing firmware. In this paper, we leverage *binwalk* to unpack and extract firmware, but it only supports analyzing the firmware without code obfuscation. Unpacking firmware with code obfuscation is still an open research question and is out of our research scope. Our paper aims to fill the gap in mapping the high-level specifications from TPC documents to the binary-level violation analysis. Besides, based on our analysis, we find most firmware images are not obfuscated. Therefore, we do not consider code obfuscation in this paper. It will be interesting future work to enhance the ability of UVSCAN to analyze obfuscated firmware.

7 Related work

Firmware analysis. Many works have studied the security of IoT firmware through static or dynamic analysis. Currently, machine learning techniques are widely adopted by static analysis methods. For instance, Xu et al. [43] utilized the neural network to detect similar vulnerable codes in different binaries. Ding et al. [15] adopted learned representation to conduct assembly clone detection in binaries. Zhao et al. [48] conducted the first large-scale analysis of the vulnerabilities introduced by TPCs in IoT firmware based on a customized neural network. Dynamic analysis methods usually perform testing on the emulated firmware. For example, Chen et al. [11] proposed *FIRMADYNE* to emulate IoT firmware on the desktop and conducted a dynamic analysis to find the vulnerabilities in IoT firmware. Feng et al. designed P^2IM [17], a scalable and hardware-independent method, to conduct MCU firmware fuzzing. In addition, a set of works have explored firmware re-hosting techniques, which are essential for dynamic firmware analysis. Clements et al. designed *HALucinator* [12], a high-level emulation system that can re-host firmware through abstraction layer modeling. Scharnowski et al. implemented *FuzzWare* [37] which is a highly efficient monolithic firmware fuzzing system that employs a novel re-hosting technique to avoid path elimination. Nevertheless, applying these methods in our task is not trivial since they are not designed to detect TPC usage violations in IoT firmware.

API misuse detection. A series of works have been proposed to detect API misuses at the source-level. These works can be divided into three categories according to their different API specification inference strategies. The first kind of work obtains API specifications by analyzing many API usage examples. For example, Yun et al. [45] inferred correct API usage by analyzing the symbolic contexts of many API usage examples. Kang et al. [26] obtained error specifications for API functions by analyzing a wealth of corresponding usage examples across multiple C programs. The second kind of methods is to obtain API specifications from generated API usage examples. Wen et al. [42] adopted mutation analysis to generate API usage examples from the correct ones. Nevertheless, the above two methods usually result in low precision since obtaining comprehensive correct usage examples of an API is hard. The third kind of methods is to obtain API specifications from the document. For instance, Lv et al. [30] utilized multiple NLP techniques to infer API specifications from the document. Except for the above three strategies, more recently, Li et al. [28] designed a novel method using active learning to interact with users to detect API misuses. However, this method requires multiple rounds of user interaction which cannot be performed automatically. In this paper, we infer API specifications from TPC documents, as this approach has higher accuracy than other strategies. Compared with similar works that adopt the same strategy, our method heuristically solves the coreference in TPC documents and makes good use of contextual information to infer API specifications, leading to high precision and recall. Moreover, though the above source-level detectors can obtain API specifications, they cannot be applied to our task. This is because their static analysis tools, e.g., *CodeQL*, are designed for the source-level analysis, and IoT firmware images are mostly closed-source binaries.

Datalog-based static analysis. Currently, several works have applied Datalog to static analysis. Grech et al. [20] proposed *Gigahorse* to decompile Ethereum smart contracts by using Datalog. Besides, the development of *Soufflé* [24], a Datalog variant, also inspires many tools to leverage Datalog for static analysis. For instance, Flores-Montoya et al. [18] designed *Ddisasm* to disassemble the binary based on *Soufflé*. These works prove the efficiency of Datalog in static analysis and inspire us to adopt Datalog to explore the TPC usage violation problem in IoT firmware.

8 Conclusion

In this paper, we present UVSCAN, the first automated and practical system to detect TPC usage violations in binary IoT firmware, which fills the gap in mapping the high-level specifications from TPC documents to the binary-level violation analysis. UVSCAN achieves more than 70% precision and recall in detecting TPC usage violations in binary IoT firmware, which has an excellent performance improvement even compared to state-of-the-art source-level works. Moreover, we conduct the first large-scale analysis of the TPC usage violation problem in IoT firmware. We identify 27,621 usage violations caused by four popular TPCs in 4,545 firmware images. Up to now, 206 usage violations have been confirmed by vendors as vulnerabilities, and seven of them have been assigned CVE IDs with high severity.

Acknowledgments

We sincerely appreciate the guidance from the shepherd. We would also like to thank the anonymous reviewers for their valuable comments to improve our paper. This work was partly supported by NSFC under No. U1936215, the State Key Laboratory of Computer Architecture (ICT, CAS) under Grant No. CARCHA202001, the Fundamental Research Funds for the Central Universities (Zhejiang University NG-ICS Platform), NSF under No. 1943100 and 1920462, and Meta Faculty Award.

References

[1] Binwalk. `https://github.com/ReFirmLabs/binwalk`.

[2] Codeql. https://codeql.github.com/.

[3] Cve-2015-8867. https://nvd.nist.gov/vuln/detail/CVE-2015-8867.

[4] Cve-2020-10058. https://nvd.nist.gov/vuln/detail/CVE-2020-10058.

[5] Cve-2020-17533. https://nvd.nist.gov/vuln/detail/CVE-2020-17533.

[6] D-link vulnerability. https://firmalyzer.com/posts/dlink_ssl.html/.

[7] Shodan. https://www.shodan.io/.

[8] Wordnet. https://wordnet.princeton.edu/.

[9] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J Alex Halderman, Luca Invernizzi, Michalis Kallitsis, et al. Understanding the mirai botnet. In *26th USENIX security symposium*, pages 1093–1110, 2017.

[10] Arianna Blasi, Alberto Goffi, Konstantin Kuznetsov, Alessandra Gorla, Michael D Ernst, Mauro Pezzè, and Sergio Delgado Castellanos. Translating code comments to procedure specifications. In *ISSTA*, pages 242–253, 2018.

[11] Daming D. Chen, Maverick Woo, David Brumley, and Manuel Egele. Towards automated dynamic analysis for linux-based embedded firmware. In *NDSS*, 2016.

[12] Abraham Clements, Eric Gustafson, Tobias Scharnowski, Paul Grosen, David Fritz, Christopher Kruegel, Giovanni Vigna, Saurabh Bagchi, and Mathias Payer. Halucinator: Firmware re-hosting through abstraction layer emulation. In *29th USENIX Security Symposium*, pages 1201–1218, 2020.

[13] S Dbiteboul, R Hull, and V Vianu. Foundations of databases: the logical level, 1994.

[14] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. In *NAACL-HLT*, pages 4171–4186, 2019.

[15] Steven H. H. Ding, Benjamin C. M. Fung, and Philippe Charland. Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In *IEEE S&P*, pages 472–489, 2019.

[16] Vladimir Dobrovolskii. Word-level coreference resolution. In *EMNLP*, pages 7670–7675, 2021.

[17] Bo Feng, Alejandro Mera, and Long Lu. P2im: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling. In *29th USENIX Conference on Security Symposium*, pages 1237–1254, 2020.

[18] Antonio Flores-Montoya and Eric M. Schulte. Datalog disassembly. In *29th USENIX Security Symposium*, pages 1075–1092, 2020.

[19] Martin Georgiev, Subodh Iyengar, Suman Jana, Rishita Anubhai, Dan Boneh, and Vitaly Shmatikov. The most dangerous code in the world: validating ssl certificates in non-browser software. In *ACM CCS*, pages 38–49, 2012.

[20] Neville Grech, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. Gigahorse: thorough, declarative decompilation of smart contracts. In *ICSE*, pages 1176–1186, 2019.

[21] Zuxing Gu, Jiecheng Wu, Jiaxiang Liu, Min Zhou, and Ming Gu. An empirical study on api-misuse bugs in open-source c programs. In *2019 IEEE 43rd annual computer software and applications conference (COMPSAC)*, pages 11–20, 2019.

[22] Boyuan He, Vaibhav Rastogi, Yinzhi Cao, Yan Chen, VN Venkatakrishnan, Runqing Yang, and Zhenrui Zhang. Vetting ssl usage in applications with sslint. In *IEEE S&P*, pages 519–534, 2015.

[23] Ori Hollander and Asaf Karas. Major vulnerabilities discovered in qualcomm qcmap. https://jfrog.com/blog/major-vulnerabilities-discovered-in-qualcomm-qcmap/?vr=1, 2020.

[24] Herbert Jordan, Bernhard Scholz, and Pavle Subotic. Soufflé: On synthesis of program analyzers. In *Computer Aided Verification: 28th International Conference*, pages 422–430, 2016.

[25] Bojan Jovanovic. Internet of things statistics for 2023 - taking things apart. https://dataprot.net/statistics/iot-statistics/, 2023.

[26] Yuan Jochen Kang, Baishakhi Ray, and Suman Jana. Apex: automated inference of error specifications for C apis. In *ASE*, pages 472–482, 2016.

[27] Siwei Lai, Liheng Xu, Kang Liu, and Jun Zhao. Recurrent convolutional neural networks for text classification. In *AAAI*, pages 2267–2273, 2015.

[28] Ziyang Li, Aravind Machiry, Binghong Chen, Mayur Naik, Ke Wang, and Le Song. ARBITRAR: user-guided API misuse detection. In *IEEE S&P*, pages 1400–1415, 2021.

[29] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.

[30] Tao Lv, Ruishi Li, Yi Yang, Kai Chen, Xiaojing Liao, XiaoFeng Wang, Peiwei Hu, and Luyi Xing. Rtfm! automatic assumption discovery and verification derivation from library document for api misuse detection. In *ACM CCS*, pages 1837–1852, 2020.

[31] Christopher D Manning, Mihai Surdeanu, John Bauer, Jenny Rose Finkel, Steven Bethard, and David McClosky. The stanford corenlp natural language processing toolkit. In *ACL: system demonstrations*, pages 55–60, 2014.

[32] Rahul Pandita, Kunal Taneja, Laurie A. Williams, and Teresa Tung. ICON: inferring temporal constraints from natural language API descriptions. In *ICSME*, pages 378–388, 2016.

[33] Rahul Pandita, Xusheng Xiao, Hao Zhong, Tao Xie, Stephen Oney, and Amit Paradkar. Inferring method specifications from natural language api descriptions. In *ICSE*, pages 815–825, 2012.

[34] Rahul Pandita, Xusheng Xiao, Hao Zhong, Tao Xie, Stephen Oney, and Amit M. Paradkar. Inferring method specifications from natural language API descriptions. In *ICSE*, pages 815–825, 2012.

[35] Pranav Rajpurkar, Robin Jia, and Percy Liang. Know what you don't know: Unanswerable questions for squad. In *ACL*, pages 784–789, 2018.

[36] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. Squad: 100, 000+ questions for machine comprehension of text. In *EMNLP*, pages 2383–2392, 2016.

[37] Tobias Scharnowski, Nils Bars, Moritz Schloegel, Eric Gustafson, Marius Muench, Giovanni Vigna, Christopher Kruegel, Thorsten Holz, and Ali Abbasi. Fuzzware: Using precise mmio modeling for effective firmware fuzzing. In *31st USENIX Security Symposium*, pages 1239–1256, 2022.

[38] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Firmalice - automatic detection of authentication bypass vulnerabilities in binary firmware. In *NDSS*, 2015.

[39] Yannis Smaragdakis and Martin Bravenboer. Using datalog for fast and easy program analysis. In *Proceedings of the First International Conference on Datalog Reloaded*, pages 245–251, 2010.

[40] Yihao Sun, Jeffrey Ching, and Kristopher Micinski. Declarative demand-driven reverse engineering. *arXiv preprint arXiv:2101.04718*, 2021.

[41] Elena Voita, David Talbot, Fedor Moiseev, Rico Sennrich, and Ivan Titov. Analyzing multi-head self-attention: Specialized heads do the heavy lifting, the rest can be pruned. In *ACL*, pages 5797–5808, 2019.

[42] Ming Wen, Yepang Liu, Rongxin Wu, Xuan Xie, Shing-Chi Cheung, and Zhendong Su. Exposing library API misuses via mutation analysis. In *ICSE*, pages 866–877, 2019.

[43] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. Neural network-based graph embedding for cross-platform binary code similarity detection. In *ACM CCS*, pages 363–376, 2017.

[44] Zimu Yuan, Muyue Feng, Feng Li, Gu Ban, Yang Xiao, Shiyang Wang, Qian Tang, He Su, Chendong Yu, Jiahuan Xu, et al. B2sfinder: Detecting open-source software reuse in cots software. In *ASE*, pages 1038–1049, 2019.

[45] Insu Yun, Changwoo Min, Xujie Si, Yeongjin Jang, Taesoo Kim, and Mayur Naik. Apisan: Sanitizing API usages through semantic cross-checking. In *25th USENIX Security Symposium*, pages 363–378, 2016.

[46] Shu Zhang, Dequan Zheng, Xinchen Hu, and Ming Yang. Bidirectional long short-term memory networks for relation classification. In *PACLIC*, 2015.

[47] Binbin Zhao, Shouling Ji, Wei-Han Lee, Changting Lin, Haiqin Weng, Jingzheng Wu, Pan Zhou, Liming Fang, and Raheem Beyah. A large-scale empirical study on the vulnerability of deployed iot devices. *IEEE TDSC*, 19(3):1826–1840, 2022.

[48] Binbin Zhao, Shouling Ji, Jiacheng Xu, Yuan Tian, Qiuyang Wei, Qinying Wang, Chenyang Lyu, Xuhong Zhang, Changting Lin, Jingzheng Wu, and Raheem Beyah. A large-scale empirical analysis of the vulnerabilities introduced by third-party components in iot firmware. In *ISSTA*, 2022.

[49] Binbin Zhao, Shouling Ji, Jiacheng Xu, Yuan Tian, Qiuyang Wei, Qinying Wang, Chenyang Lyu, Xuhong Zhang, Changting Lin, Jingzheng Wu, and Raheem Beyah. One bad apple spoils the barrel: Understanding the security risks introduced by third-party components in iot firmware. *IEEE TDSC*, 2023.

[50] Yu Zhou, Changzhi Wang, Xin Yan, Taolue Chen, Sebastiano Panichella, and Harald C. Gall. Automatic detection and repair recommendation of directive defects in java API documentation. *IEEE TSE*, 46(9):1004–1023, 2020.