Inferring API Correct Usage Rules: A Tree-based Approach

Majid Zolfaghari, Solmaz Salimi, Mehdi Kharrazi
S4 Laboratory, Department of Computer Engineering
Sharif University of Technology, Tehran, Iran
Email: {m.zolfaghari, s.salimi, kharrazi}@sharif.edu

***Abstract*—The lack of knowledge about API correct usage rules is one of the main reasons that APIs are employed incorrectly by programmers, which in some cases lead to serious security vulnerabilities. However, finding a correct usage rule for an API is a time-consuming and error-prone task, particularly in the absence of an API documentation. Existing approaches to extract correct usage rules are mostly based on majority API usages, assuming the correct usage is prevalent. Although statistically extracting API correct usage rules achieves reasonable accuracy, it cannot work correctly in the absence of a fair amount of sample usages. We propose inferring API correct usage rules independent of the number of sample usages by leveraging an API tree structure. In an API tree, each node is an API, and each node's children are APIs called by the parent API. Starting from lower-level APIs, it is possible to infer the correct usage rules for them by utilizing the available correct usage rules of their children. We developed a tool based on our idea for inferring API correct usages rules hierarchically, and have applied it to the source code of Linux kernel v4.3 drivers and found 24 previously reported bugs.**

***Index Terms*—API Correct Usage Rule, API Misuse, Software Vulnerability.**

I. INTRODUCTION

ONE of the most important causes of software vulnerabilities is programmer's errors ranging from uninitialized values [1], off-by-one [2], and buffer overflow [3] on one side and in-appropriate or at times calls with incorrect formatting to APIs [4]. Many real-world vulnerabilities exist that are caused by API incorrect use. For example, CVE-2014-4113, a vulnerability which results in DoS or privilege escalation attacks in Microsoft Windows, arrives because of an incorrect API error handling. Another example is CVE-2018-0171, which is the cause of widespread attacks on Cisco devices in April of last year because of $memcpy$ incorrect usage[5], which leads to a buffer overflow in a function. In fact, researchers have shown in [6], [7], [8], [9], [10], [11], [12] that API incorrect use is a common cause of software bugs and vulnerabilities which can lead to potentially dangerous consequences, and intractable security problems.

While incorrect use of third-party APIs is more common among programmers, the same problem can happen for all the predefined interfaces even in the programmers' code, including program functions. Thus in this paper, API refers to all interfaces with well-defined input and output, including third-party APIs and functions.

One could argue that the incorrect usage of programming APIs is due to the developers inexperience, but multiple research studies on the topic [13], [14] discuss that the lack of knowledge about API's correct usage rules is one of the leading causes of its incorrect usage. In most cases, developers use third-party API's documentation to understand their proper usage and avoid undesired functionality and in turn security flaws. But complete documentation does not exist for most library codes and APIs. In the absence of documentation, developers can extract correct usage rules by checking the source code. Although that would be a laborious, time-consuming, and error-prone task [15] and may take almost as much time as to re-develop the API from scratch.

Given the significant impact of incorrect API usage and the security consequences it may cause, a number of proposals have been put forward to extract API's correct usage rules automatically. The predominant approach has been to assume that most developers are properly employing the API and therefore infer their proper usage based on the majority usage samples in available code. Such a statistical technique is compelling and is used in the core of most recent works [16], [17], [18], [19]. While statistically extracting API correct usage rules can achieve reasonable accuracy, it cannot work correctly in the absence of a fair amount of sample usages [18]. In fact, if the number of sample usages is small, statistical methods could extract incorrect usage as a rule and therefore need manual effort to refine the results.

While recent research into inferring API correct usage focuses on extracting rules based on the usage samples of each API, in most cases, APIs are developed with the help of other APIs. That means if the correct usage rules of some basic and common APIs are known, it is possible to decide if any API that uses them is employed correctly or not. For example, Figure 1 shows part of the $libc$ library in a tree structure, representing the relation between APIs in the library. As shown in the figure, there are some APIs in this tree that call each other in a hierarchy.

Based on this observation, we present a non-statistical automated system for extracting API usage rules. Unlike previous approaches which require some usage samples, we infer correct usage rules of an interior API without any need for its usage samples. The main idea is that we can arrange APIs of a library in one or more tree that every node is an API, each interior API is developed based on some other APIs, therefore to extract correct usage rules for internal nodes, we need to know only the leaves correct usage rules.

APISan [18] places correct usage rules into three categories:

978-1-7281-4374-3/19/$31.00

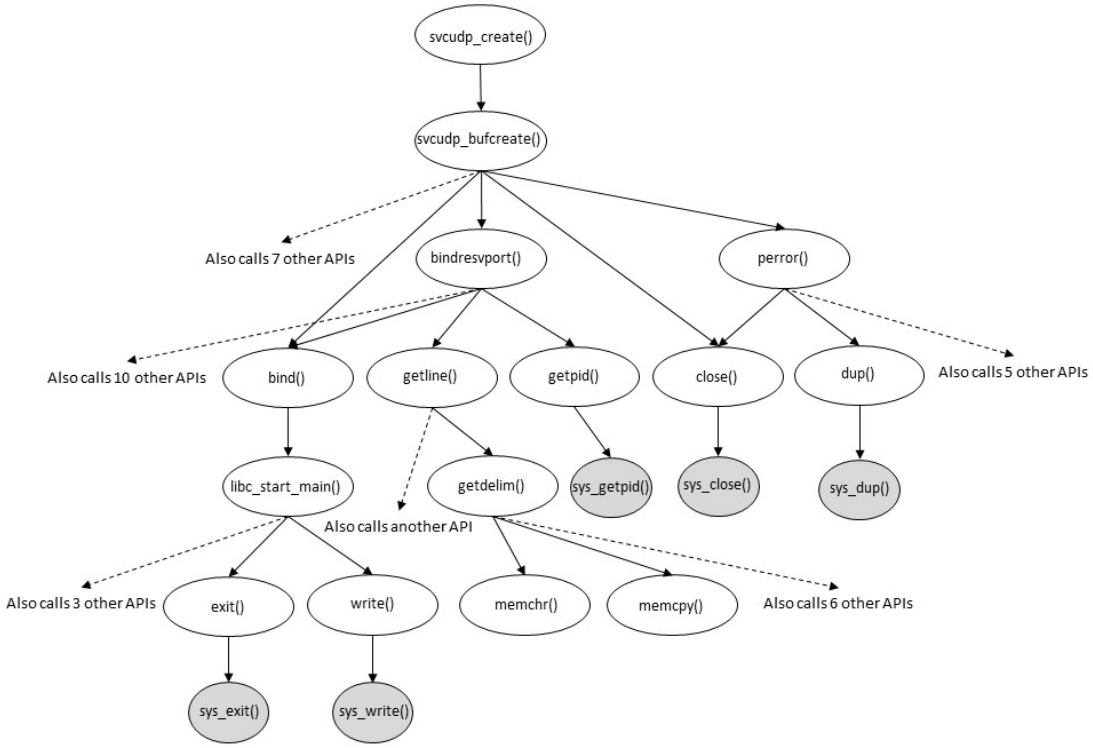


Fig. 1: Arranging some APIs in $libc$ in a tree. White nodes are APIs and gray nodes are system calls. By having correct usage rules of leaves, we can infer correct usage rules of higher level APIs.

(i) API return values, (ii) relations between API arguments, and (iii) causal relationships between APIs. We have opted to choose the same rules; given input's source code and proper usage rules of only leaf APIs, our proposed method starts from the lowest level of trees, and extract rules based on the rules it has or extracted previously. Then it infers correct usage rules of that API based on correct usage rules of APIs it calls and finally, translates the rules into a human-readable format and provides them to the user.

Our evaluation shows that our approach is scalable and effective in extracting API correct usage rules. We analyzed the Linux kernel drivers as a test case and extracted correct usage rules for its APIs. Also to show the usefulness of the extracted rules, we applied our tool to source code of drivers in Linux kernel v4.3 and found 24 already confirmed bugs which were caused due to the incorrect usages of extracted rules. In short, our paper makes the following contributions:

- To the best of our knowledge, this is the first work that leverages the fact APIs are developed in a hierarchical way, and such structure could be used to extract API usage rules based on lower level APIs.
- As part of this work, we have developed a tool for extracting correct usage rules for the desired API.

The rest of this paper is organized as follows. Section II compares our work to previous works. Section III explains our approach. Section IV describes our work's implementation. Section V evaluates our approach. Section VI discuss the limitations of our approach and section VII concludes.

II. Related Work

In this section, we survey related work in the area of inferring API specifications. We categorize proposed methods based on the type of specification they infer into multiple categories:

A set of techniques infer the proper order of the API call from the source code in order to guide users to call APIs in the correct order or to detect violations in software codes. One of these approaches that infer implicit causal relations is PRMiner [20] which uses data mining techniques. Another method is APIMiner [21] that concludes API usage patterns as partial orders. MAPO [16] infers API usage patterns by clustering the API call sequences and recommends correct usage patterns as a plugin for Eclipse IDE. Other approaches that we can mention in inferring API usage pattern are UP-Miner [22] that indicates sequences of functions using two-level clustering, MLUP [17] which infers multi-level API usage patterns and Buse et al. [23] that presents how these patterns are used in real codes with some examples. Above approaches define API usage pattern as sequences of API calls, and all of them infer these patterns statistically.

Another group of technique infer proper API error handling rules. For example, Acharya et al. [24] infer API error-handling specifications from C source code using data mining techniques and without any user input. Another approaches are APEx [19] which infers error specifications for C APIs by extracting majority constraints on error paths and ErrDoc [25] that concludes error-handling specifications to find and repair error handling bugs in C source code. All of these approaches

also infer the API error handling rules statistically.

Alternatively a number of approaches infer other API specifications automatically. For example, MERLIN [26] checks information flow specifications by creating information flow graphs, PRIME [27] deals with temporal specifications of APIs, and Nguyen et al. [28] checks APIs preconditions. These technique, similar to those noted above, infer the API specifications statistically.

Lastly, and most importantly, there have been a newer proposal with which different aspects of the usage pattern are considered. More specifically, APISan [18] extracts usage patterns classified into four categories (i) return values, (ii) relations among arguments, (ii) causality between APIs, and (iv) implicit pre- and post-conditions for the APIs. Nevertheless and similar to previous approaches, APISan infers these rules statistically and based on available usage patterns, assuming that the correct usage is prevalent. Table I compares proposed approaches for inferring APIs correct usage rules.

III. Proposed Approach

Extracting a correct usage rule for an API involves numerous considerations, especially without having access to expert knowledge or documentation and specially when an API is defined in a large library. While in most cases majority usage is considered as expert knowledge, this approach is not reliable if there is no way to show that majority usage is correct, mainly when most of the API usages are codes reused within or between software systems. We could undoubtedly winnow down duplicate usages, and use the remaining unique usages to overcome this problem. However, this approach may need algorithms to detect code clones, where in most cases syntax-based code clone approaches are considered as fragile. Furthermore, the number of remaining appropriate sample usages might be insufficient to make a valid decision.

Based on this knowledge and trade-offs between accuracy and efficiency, our goal is to extract correct usage rules independent of majority usage for most of the APIs. The main idea of our approach is arranging APIs of a source code in one or more tree and inferring correct usage rules of each API based on correct usage rules of its children. To make our approach more clear, suppose we want to infer correct usage rules of *myAPI* which its definition is shown in Figure 2a. Based on its definition, we can create its call tree like Figure 2b. Now, suppose the available correct usage rules of children APIs are:

- The return value of *malloc* should be checked whether is null or not to find whether the memory is allocated or not.
- After calling *malloc*, *free* should be called, because the are causally related.
- First and third arguments of memcpy should be related. In other words, the amount of third argument should be less or equal than amount of first argument.

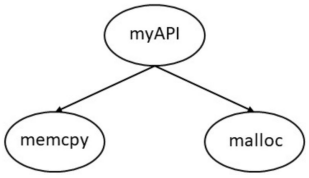
So, using proposed approach we can infer correct usage rules

TABLE I: Comparison of Proposed Approaches for Inferring APIs Correct Usage Rules

Approach	**Specifications Type**			**Application**		**Based on Majority Usage**
	Call Order	Error Handling	Other	Find Violations	Guide Users	
PRMiner [20]	✓	✗	✗	✓	✗	✓
APIMiner [21]	✓	✗	✗	✗	✓	✓
MAPO [16]	✓	✗	✗	✗	✓	✓
UP-Miner [22]	✓	✗	✗	✗	✓	✓
Buse et al. [23]	✓	✗	✗	✗	✓	✓
MLUP [17]	✓	✗	✗	✗	✓	✓
Acharya et al. [24]	✗	✓	✗	✓	✓	✓
APEx [19]	✗	✓	✗	✗	✓	✓
ErrDoc[25]	✗	✓	✗	✓	✗	✓
MERLIN [26]	✗	✗	Information flow	✗	✓	✓
PRIME [27]	✗	✗	Temporal	✗	✓	✓
Nguyen et al. [28]	✗	✗	Preconditions	✓	✓	✓
APISan [18]	✗	✗	Return values, Arguments' relation, APIs causality, Conditions	✓	✗	✓
Our Approach	✗	✗	Return values, Arguments' relation, APIs causality	✓	✓	✗

```
char* myAPI(char *src,char *dst,int n)
{
        int size = n*8;
        memcpy(dst,src,size);
        src = malloc(size);
        return src;
}
```

(a) Definition of *myAPI*. This API calls *memcpy* and *malloc*.



(b) Created tree for *myAPI* based on its definition

Fig. 2: Definition of an API and created tree for it

of *myAPI* based on call tree in Figure 2b and above rules:

- The return value of *myAPI* should be checked to see whether it is null or not.
- After calling *myAPI*, *free* should be called.
- The second argument of *myAPI* should be related to its third argument times eight.

That way, we can infer correct usage rules of each API based on correct usage rules of APIs it calls.

Figure 3 illustrates the core system and overall work-flow of the proposed system, where the main steps are:

1) **Feeding correct usage rules**: As we have discussed, we leverage the fact that most of APIs developed hierarchically and used other APIs in their source code. We want to traverse the tree from leaves to higher level APIs. Hence we need some leaf APIs as starting nodes that we know the correct usage rules for them. In our proposed method, APIs that do not call other APIs are considered as leaf APIs. Users can feed any correct rule as input, in most cases, some of ubiquitous APIs' correct rules are enough to start the traversing.
2) **Preprocessing the input source code**: To obtain the tree and API levels, after receiving the input source code, our method automatically pre-processes all APIs defined in the code to obtain which lower level APIs are called in them and also derives some other features that show how these called APIs are employed. In our method, we obtain these data by a symbolic execution [29] based approach.
3) **Inferring the correct usage rules**: In this step, we first generate an API tree, where its root is the target API, its children are APIs called in it, and leaf nodes are APIs that do not call any other APIs, where their correct rules must be provided as inputs. Then we traverse the tree from bottom to top (from leaves to root). In each level of the API tree, we infer correct usage rules of APIs in that level, using correct usage rules of their children and add them to rules database. When we get to the root, we obtain correct usage rules of target API.
4) **Translating extracted rules**: Our goal is to provide extracted rules to users (e.g., a programmer) so they can understand them and accurately use them in their code and avoid potential software vulnerabilities. Hence, in this step, our method automatically translates the extracted rules to a human-readable format.

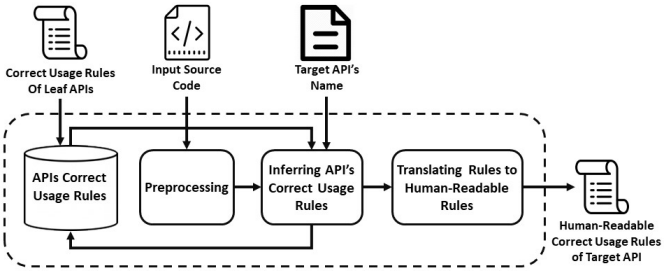


Fig. 3: Overview of our tool's architecture and work-flow.

The implementation details will be discussed in the following section.

IV. Implementation

To show the usefulness of our proposed method, we developed a prototype which is implemented in 1.1K lines of code, as shown in Table II. To extract function information, instead of using a known compiler's preprocess engine, we developed a robust grammar-based code parser utilizing ANTLR [30], to run symbolic execution on input source code, we applied APISan's symbolic execution engine, and finally to generate and translate extracted rules, we developed an engine based on Python scripts. In the following, we explain the main component in details.

A. Generating output of symbolic execution

A robust parser, developed in Java and ANTLR framework, extracts function information in Java. This parser explores the source to find the name of functions defined in a file, line numbers of start and end of an API and input arguments of each function.

To symbolically execute a target API, we need to perform symbolic execution on input source code. Since we want to check each API's definition independently, we don't need inter-procedural analysis. So, we applied APISan's relaxed

TABLE II: Components and lines of code of our prototype

Component	Programming Language	LoC
Parser	Java	460
Preprocess	Python	170
Rules Inference	Python	390
Translating Rules	Python	60
Total		**1080**

TABLE III: Types of APIs Usage Rules

Index	Rule Type	Format	Description	Example	Example's Meaning
1	Return Value Rule	$APIname(start, end)$	The return value of an API should be in a specific range.	$open(0, 2147483648)$	The return value of $open()$ should be greater than or equal to zero.
2	Causality Relation Rule	$APIname1, APIname2$	After or before calling an API, another API should be called.	$malloc, free$	$malloc()$ and $free()$ should be called together and respectively.
3	Input Arguments Rule	$APIname(f1, f2)$	Some input arguments of an API are inter-related.	$strncpy(@0, @2)$	The first and third arguments of $strncpy()$ should be related.

symbolic execution as our symbolic execution engine without losing accuracy. APISan's relaxed symbolic execution engine which is based on Clang 3.6 [31], overcomes the path-explosion problem [32] by ignoring inter-procedural analysis and unrolls each loop only once. It can provide the output of symbolic execution for APIs in input source code and accomplish our requirements. In the proposed method, the output of symbolic execution phase for any target API is instant to be the input for the next phase.

B. Inferring Rules

The principal goal of our proposed method is to extract a set of correct usage rules of APIs. Therefore, there is a necessity to determine the API usage rules format to represent and interpret, which we leverage the same categories of APISan's [18] rules: **(1)** Function return values, **(2)** Causal relationships between functions and **(3)** Relationships between function arguments. Table III summarizes the format and description of these rules, as well as an example that explicates the rule usage.

V. Evaluation

A set of rules that constituted of API correct usage can be evaluated with two separate viewpoints. The first is measuring the soundness and completeness of inference rules. This kind of evaluation is extremely time-consuming and error-prone because there is a need for expert knowledge that decides the correctness of rules. The second method to evaluate the set of correct API usage rules is using them to find vulnerabilities in the source code of well-known and public open-source libraries and projects. This kind of evaluation is based on the assumption that by extracting API usage pattern from a target program, it is possible to decide if the pattern matches the correct rule or not. Any different pattern, previously reported as a vulnerability or bug, is a good indication that correct API usage rule is performing accurately.

We selected the second evaluation method to asses our method's ability to define correct API usage rules and how it is beneficial to confirm vulnerabilities in real-world applications. In the following, first, we particularly articulate the evaluation method and then employ the Linux kernel drivers source code as a test case to check its API usage patterns and match them with a set of correct API usage rules.

A. Evaluation Method

In order to asses our method, in the step1, the method infers correct usage rules of APIs defined in the source code of a target program; In this step, a set of correct usage rules of the target program's lower-level APIs that place as a leaf in the API tree is provided as an input.

In step 2, all other API usage patterns are obtained, to find violations of inferred rules in the target program. We name the violation usages as violation set, which are API usages that do not match with inferred rules.

In step 3, the violation set is examined to find a real vulnerability. A real vulnerability should be a zero-day or previously reported vulnerability.

B. Test case: Vulnerabilities in Linux Kernel Drivers

We applied our evaluation method to source code of drivers in Linux kernel v4.3. At first, we need to initialize our rules' database with correct usage rules of low-level APIs. Thus, we extracted proper usage rules for 20 low-level APIs manually, such as $kmalloc$, $kfree$, etc. Then, we ran our tool to extract correct usage rules for APIs in higher levels. These rules can help programmers to apply higher level APIs accurately.

The evaluation resulted in finding 24 real vulnerabilities which they have been already confirmed. Found bugs are shown in Table IV.

Figure 4 shows some APIs which are defined and used in Linux kernel drivers. These APIs are arranged in a tree based on their callings which means that higher level APIs call lower

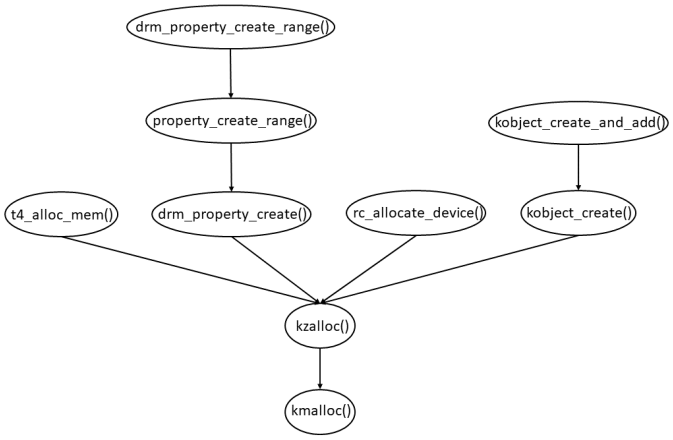


Fig. 4: A part of call tree for Linux Kernel drivers APIs

TABLE IV: Vulnerabilities found in Linux Kernel v4.3 drivers source code

Index	Vulnerable Code	API name	Number of Vulnerabilities
1	linux-4.3/drivers/media/usb/usbtv/usbtv-video.c	kzalloc()	1
2	linux-4.3/drivers/ata/sata_sx4.c	kzalloc()	1
3	linux-4.3/drivers/net/wireless/mwifiex/sdio.c	kzalloc()	2
4	linux-4.3/drivers/hsi/hsi.c	kzalloc()	2
5	linux-4.3/drivers/net/ethernet/chelsio/cxgb4/clip_tbl.c	t4_alloc_mem()	1
6	linux-4.3/drivers/gpu/drm/drm_crtc.c	drm_proprty_create()	1
7	linux-4.3/drivers/media/rc/igorplugusb.c	rc_allocate_device()	1
8	linux-4.3/drivers/acpi/sysfs.c	kobject_create_and_add()	1
9	linux-4.3/drivers/gpu/drm/i2c/ch7006_drv.c	drm_property_create_range()	1
10	linux-4.3/drivers/gpu/drm/gma500/framebuffer.c	drm_property_create_range()	1
11	linux-4.3/drivers/gpu/drm/drm_crtc.c	drm_property_create_range()	12
	Total		**24**

level APIs and the only leaf we have is $kmalloc$. As it reveals, with the help of only one API's correct usage rule as input, we could infer correct usage rules for eight APIs and discover 24 bugs. All of the bugs we found can lead to critical security problems: e.g., man-in-the-middle, system crash, etc.

According to Figure 4, we only utilized the correct usage rule of $kmalloc$. $kmalloc$ is an API for allocating memory in the kernel, it gets the size of required memory and some flags, and if it can allocate memory successfully, returns a pointer to the newly allocated memory, and if it faces an error, it returns null. If the return value is null and we put some data in it, the system will crash, or some other security policies may fail.

Consequently, the return value of $kmalloc$ needs to be checked to decide if it is null or not. $kzalloc$ is an API that uses $kmalloc$ and returns its return value. Therefore our tool infers that return value of $kzalloc$ should be checked whether it is null or not. We found 6 cases which violate this rule and are reported as bugs. $t4\_alloc\_mem$, $drm\_proprty\_create$, $rc\_allocate\_device$ and $kobject\_create$ use $kzalloc$ in and pass this API's return value. Hence, our tool infers that these APIs' return value and also the return value of APIs that pass them should be checked to find null pointers. We found 18 similar cases which violate these rules in Linux kernel v4.3 drivers, as well. Other open source projects analysis and providing some run-time analysis is considered as complementary evaluation for future work.

VI. Discussion

In this section, we discuss the limitations of our approach and state potential future works to mitigate these limitations. We should consider that the result's accuracy is entirely dependent on the input's accuracy. Therefore, any incorrect item in the input set will propagate in the API tree and affects the resulting rules.

Based on this point, we need accurate inputs to avoid errors in the resulting output, which means we need accurate usage rules of lower-level APIs to infer correct usage rules of higher-level APIs. Such input rules can be obtained from existing documents (it is more probable that we can find complete documents of leaf APIs, because they are more useful and more famous), generated manually based on their source code (since leaf APIs are more limited than higher-level APIs, this work is not so time-consuming) or obtained by available tools.

In order to simplify our implementation, we did not consider running paths in our processing. This does not affect argument relation and returns value rules, but it can affect causality rules, which causes the generation of false-positive rules of this type (rules that should not be generated). In order to avoid generating these false-positive rules, running paths should be considered, and the processing should be performed based on paths. This could be done as future work to reduce false-positive rules.

VII. Conclusion

We proposed an automated system for inferring API correct usage rule from its source code. To extract rules, we leveraged the fact that most APIs are developed hierarchically, and therefore it is possible to arrange APIs in some trees where each node is an API, and interior nodes call lower-level APIs in their source code. By the help of API trees, our proposed method inferred correct usage rules of each API based on correct usage rules of APIs it calls. For evaluation, we applied the proposed technique to the Linux kernel drivers and showed that it successfully extracted correct usage rules for APIs with very few inputs.

Although one of the limitations of our proposed method is the requirement of accessing correct usage rules for the lowest level APIs, in most cases, utilizing the proper usage rules of few APIs as input is sufficient for the method. Our results show that API usage rules extracted by our tool are correct, and API usages that do not match with these rules are reported as bugs previously.

References

[1] "CWE-457: Use of uninitialized variable." [Online]. Available: https://cwe.mitre.org/data/definitions/457.html

[2] "CWE-193: Off-by-one error." [Online]. Available: https://cwe.mitre.org/data/definitions/193.html

[3] "CWE-120: Classic buffer overflow." [Online]. Available: https://cwe.mitre.org/data/definitions/120.html

[4] "CWE-20: Improper input validation." [Online]. Available: https://cwe.mitre.org/data/definitions/20.html

[5] R. Bazhin. (2018) Cisco smart install remote code execution. [Online]. Available: https://embedi.com/blog/cisco-smart-install-remote-code-execution

[6] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov, "The most dangerous code in the world: validating ssl certificates in non-browser software," in *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 2012, pp. 38–49.

[7] S. Amann, S. Nadi, H. A. Nguyen, T. N. Nguyen, and M. Mezini, "Mubench: A benchmark for api-misuse detectors," in *Mining Software Repositories (MSR), 2016 IEEE/ACM 13th Working Conference on*. IEEE, 2016, pp. 464–467.

[8] S. Amann, H. A. Nguyen, S. Nadi, T. N. Nguyen, and M. Mezini, "A systematic evaluation of api-misuse detectors," *arXiv preprint arXiv:1712.00242*, 2017.

[9] J. Sushine, J. D. Herbsleb, and J. Aldrich, "Searching the state space: A qualitative study of api protocol usability," in *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension*. IEEE Press, 2015, pp. 82–93.

[10] M. Monperrus and M. Mezini, "Detecting missing method calls as violations of the majority rule," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 22, no. 1, p. 7, 2013.

[11] S. Nadi, S. Krüger, M. Mezini, and E. Bodden, "Jumping through hoops: why do java developers struggle with cryptography apis?" in *Proceedings of the 38th International Conference on Software Engineering*. ACM, 2016, pp. 935–946.

[12] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel, "An empirical study of cryptographic misuse in android applications," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 2013, pp. 73–84.

[13] J. Stylos, B. A. Myers, and Z. Yang, "Jadeite: improving api documentation using usage information," in *CHI'09 Extended Abstracts on Human Factors in Computing Systems*. ACM, 2009, pp. 4429–4434.

[14] M. P. Robillard, "What makes apis hard to learn? answers from developers," *IEEE software*, vol. 26, no. 6, 2009.

[15] J. Bloch, "How to design a good api and why it matters," in *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*. ACM, 2006, pp. 506–507.

[16] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei, "Mapo: Mining and recommending api usage patterns," *ECOOP 2009–Object-Oriented Programming*, pp. 318–343, 2009.

[17] M. A. Saied, O. Benomar, H. Abdeen, and H. Sahraoui, "Mining multi-level api usage patterns," in *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*. IEEE, 2015, pp. 23–32.

[18] I. Yun, C. Min, X. Si, Y. Jang, T. Kim, and M. Naik, "Apisan: Sanitizing api usages through semantic cross-checking." in *USENIX Security Symposium*, 2016, pp. 363–378.

[19] Y. Kang, B. Ray, and S. Jana, "Apex: Automated inference of error specifications for c apis," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2016, pp. 472–482.

[20] Z. Li and Y. Zhou, "Pr-miner: automatically extracting implicit programming rules and detecting violations in large software code," in *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5. ACM, 2005, pp. 306–315.

[21] M. Acharya, T. Xie, J. Pei, and J. Xu, "Mining api patterns as partial orders from source code: from usage scenarios to specifications," in *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ACM, 2007, pp. 25–34.

[22] J. Wang, Y. Dang, H. Zhang, K. Chen, T. Xie, and D. Zhang, "Mining succinct and high-coverage api usage patterns from source code," in *Proceedings of the 10th Working Conference on Mining Software Repositories*. IEEE Press, 2013, pp. 319–328.

[23] R. P. Buse and W. Weimer, "Synthesizing api usage examples," in *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 2012, pp. 782–792.

[24] M. Acharya and T. Xie, "Mining api error-handling specifications from source code," in *International Conference on Fundamental Approaches to Software Engineering*. Springer, 2009, pp. 370–384.

[25] Y. Tian and B. Ray, "Automatically diagnosing and repairing error handling bugs in c," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 2017, pp. 752–762.

[26] B. Livshits, A. V. Nori, S. K. Rajamani, and A. Banerjee, "Merlin: specification inference for explicit information flow problems," *ACM Sigplan Notices*, vol. 44, no. 6, pp. 75–86, 2009.

[27] A. Mishne, S. Shoham, and E. Yahav, "Typestate-based semantic code search over partial programs," in *Acm Sigplan Notices*, vol. 47, no. 10. ACM, 2012, pp. 997–1016.

[28] H. A. Nguyen, R. Dyer, T. N. Nguyen, and H. Rajan, "Mining preconditions of apis in large-scale code corpus," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 166–177.

[29] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[30] T. J. Parr and R. W. Quong, "Antlr: A predicated-ll (k) parser generator," *Software: Practice and Experience*, vol. 25, no. 7, pp. 789–810, 1995.

[31] C. Lattner, "Llvm and clang: Next generation compiler technology," in *The BSD Conference*, 2008, pp. 1–2.

[32] P. Boonstoppel, C. Cadar, and D. Engler, "Rwset: Attacking path explosion in constraint-based test generation," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 351–366.