State Consistency Checking for Non-reentrant Function Based on Taint Assisted Symbol Execution

Bo Yu, Qiang Yang(✉), and CongXi Song

College of Computer, National University of Defense Technology, Changsha 410073, China
290149807@qq.com

Abstract. Non-reentrant functions are commonly used in multi-thread programs, such as network services and other event-driven programs, to reserve some global states in a concurrent context. However, calling non-reentrant functions may bring several kinds of dangerous pointer dereference faults, and will lead to serious consequences such as program vulnerabilities. To beat this, this paper presents an approach to check state consistency against non-reentrant functions based on taint analysis and symbol execution technology. The proposed method records the program taint states and traces the data flow during the symbol execution process where some rules are specified to check the state consistency and exceptions such as null pointer reference, pointer double free and pointer use-after-free. We implement a proof-of-concept system SC2NRF based on the symbol execution framework *angr*. Further experiments show that our approach is able to effectively check state consistency of non-reentrant functions in binary programs.

Keywords: Binary program · State consistency · Non-reentrant function · Taint analysis · Symbol execution

1 Introduction

Non-reentrant functions have been widely used in large-scale real-life programs to provide user-friendly and smart functionality. There are usually referenced in a concurrent context a competing code sequences (e.g. threads or signal handlers) that may influence their states [1, 2]. If not carefully designed, the calling of programs with non-reentrant functions will lead to dangerous program state faults, and then introduce serious consequences, such as pointer dereference error. For clarity, we explain this problem using two CVEs with high scores, i.e. CVE-2018-0101 and CVE-2015-0291. In the first example, a buffer address in a non-reentrant function is assigned to a global pointer and a local buffer pointer. The local buffer pointer is freed in the tail of the function, while the global pointer still holds the buffer address. When the function is called again, a double-free vulnerability is triggered and a system crash will happen. In the another example CVE-2015-0291, there is a non-reentrant function in which the address of a local buffer *A* is assigned to another local pointer *B* and the length of buffer

J. Li et al. (Eds.): SPNCE 2019, LNICST 284, pp. 498–508, 2019.
https://doi.org/10.1007/978-3-030-21373-2_40

A is also assigned to another local variable *C*. When the function returned, local pointer *B* is set to NULL, but the local variable *C* is not processed accordingly. When this function is called again in a specified condition, the local pointer *B* remains there, but the value of local variable *C* holds the last time assignment, and the rest codes uses the value of *C* to access *B*, which leads to a null-pointer dereference vulnerability and results in process crash.

Both the two CVEs are exemplars about non-reentrant function problems, however, the issue cannot be avoided or resolved using existing thread safety techniques. The root cause is that the states of different variables are inconsistent in program semantics, and this inconsistency in a defect program implementation will cause serious software faults. Existing vulnerability detection technologies (e.g. double free checking and null-pointer checking techniques based on static and dynamic detection methods) cannot meet the requirements of checking such state inconsistency in non-reentrant function. Additionally, state inconsistencies also exist in block-chain applications such as smart contracts [3, 4]. The core problem of consistency checking is to find the relations of state variables and track the data flow of state variables in different execution paths. Based on the data flows in a non-reentrant function, existing defects in binary programs can be detected for further improvements.

According to above analysis of this inconsistency issue, we model it as a kind of consistency checking of program states between different program paths. The key factors to solve this issue are program state model, state analysis of program executions and dynamic consistency checking, and several techniques can be leveraged for this purpose. In general, both the static analysis and dynamic analysis can be used to analyze non-reentrant function. But static analysis solutions (e.g., [5]) usually have high false positives because many program states are dynamically generated [6], and are only fit for small programs due to intrinsic challenge (i.e., alias analysis). Given a specific execution, the program state and data flow are deterministic, it is easier and more reliable to use dynamic analysis to check state consistency.

Several dynamic analysis techniques are widely adopted for program analysis, e.g., symbol execution [7], taint analysis [8] and information flow analysis [9]. Symbol execution could explore program paths thoroughly and discover program states in a more proactive way, but most solutions of symbol execution could not accurately analyze the relations of program states in different paths. Taint analysis can trace the taint propagation process and record the taint relation of program states, but existing solutions are not fine-grained enough for consistency checking. Current solutions based on information flow analysis include flow-insensitive, path-sensitive, and context-sensitive flow analysis [9–11], can build data flow in multi-grain levels. However, the disadvantages of these approaches are that the inter-process data flows are omitted, including call stack data flow and function return data flow, and data flows constructed are limited. Similar to heap overflow [12] and double-fetch [13, 14], state inconsistencies are root cause of program faults. Hence, a fine-grained state consistency analysis approach is necessary to solve this problem.

To address this issue, the data flow of non-reentrant function need to be constructed, and consistencies of data flow are modeled and checked in case of any non-consistencies in given non-reentrant function. Based on this idea, we present a state consistency checking method, and propose an inter-process data flow model for state

consistency checking. Especially, we leverage symbol execution technology to construct multiple execution traces, together with the data flow constructed with taint analysis technique.

The main contributions of this paper include: (1) a state consistency checking framework (implemented as SC2NRF system) for constructing and analyzing inter-process data flow; (2) a symbol assisted taint analysis approach to trace data flow and build data flow relations; and (3) a rule-based state consistency checking algorithm to find data flow faults in program implementations.

The rest of this paper is organized as follows. In Sect. 2, the state consistency checking framework together with its working process and checking algorithm is discussed in detail. Section 3 describes the implementation and experiments of the SC2NRF system. Discussions and the conclusions are presented in Sect. 4.

2 State Consistency Checking Framework

We aim to discover state inconsistencies in program execution process with the combination of static control analysis and symbol execution. To achieve this goal, we define the data flow models of binary program in advance, and describe how to manually define consistency rules and automatically generate them based on heuristic algorithms.

Furthermore, to make the solution efficient and practical, we propose an approach which uses taint-assisted symbol execution methods to generate full data flow and detect potential inconsistencies of program states. At last we concrete values in path constraints and inconsistency constraints to speed up the checking process.

Figure 1 depicts the working flow of the proposed framework. Firstly, we analyze the target program with static analysis to construct the control flow graph and identify function call attributes. The program is run by taint assisted symbol execution and we will collect taint expressions and path constraints during each block's execution. Our analysis module then tries to build global data flow relations and checks any inconsistencies against given consistency rules. At last, inconsistency results are exported and identified for further analysis.

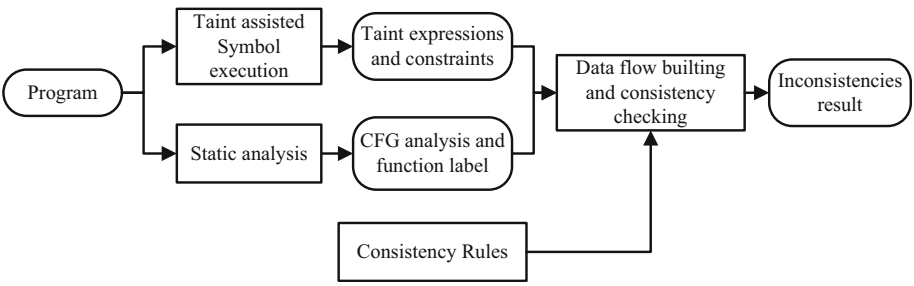


Fig. 1. Framework overview.

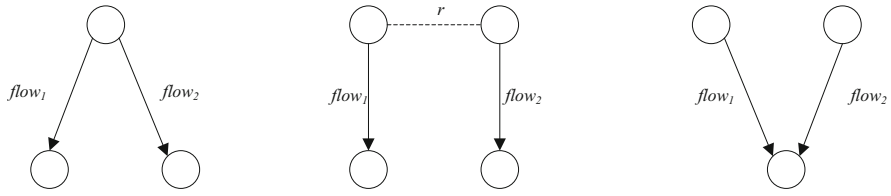
2.1 Information Flow Model for Code Block

A data flow shows how instructions correlate with each other with respect to the production and consumption of data. Efficiently generating a sound data flow for a binary slice has several challenges. First, program slicing requires a flow-sensitive and context-sensitive data flow analysis, which however has a run-time complexity exponential to the number of all possible paths in a program. Second, analyzing the data flow of binary programs poses some unique problems. Hence, we propose a data flow model upon a single code block of binary program to provide fine-grained data flow relations.

Definition 1. Given CFG_f the control flow graph of a non-reentrant function f and c a code block in CFG_f, we model the operation expressions list of block c as $I_c = \{y_i = op(x_i)$, for x_i in block $c\}$, where op is an algorithm operation or logic operation. Given two blocks in CFG_f namely c_1 and c_2, edge $e_{condition} = (c_1, c_2)$ records condition of block c_1 to block c_2 if data I_{c1} of c_1 flows to I_{c2} of c_2 under the *condition*.

Definition 2. We use DFR_c to denotes summary of data flow relation in block c. For each dfr in DFR_c, $dfr(y_i, y_j)$ means that there is a data flow relation between y_i and y_i. For example, if y_i is the buffer pointer of a input string, and yj is the string length of the input string, then $dfr(y_i, y_j)$ equals to $y_j = str\_length(y_i)$, where *str_length* is a typical kind of data flow relations. Other data flow relations include pointer aliasing, pointer repositioning, logic operations and so on.

We also define the typical data flow relations between code blocks as in Fig. 2. In Fig. 2(a), the fork flow model means that two data flows are forked by program execution conditions, and in Fig. 2(b), relation flow means that $flow_1$ and $flow_2$ are flowed to next code block with the relation r, and in Fig. 2(c), flow1 and flow2 are aggregated in one block by algorithm operation or logic operation. In our discussion, the three data flow relations are common in real-programs.



(a) fork flow (b) relation flow (c) aggregate flow

Fig. 2. Three kinds of basic data flow relations.

Based on the data flow relations in Definitions 1 and 2, we model the fine-grained data flows along the execution traces of non-reentrant functions. However, data flow relations in real programs are complex and diverse, and the state consistency rules have to be defined based on data flow relations. To handle the issues mentioned above, our framework follows [15] to analyze data flow, which is an inter-procedural data flow analysis algorithm that uses *def-use* chains.

2.2 Taint-Assisted Symbol Execution Approach

To analyze a given non-reentrant function, we build the constraints between input bytes and the data flow relations of program code blocks by performing symbolic execution. However, existing symbol execution approaches are lack of support for fine-grained data flow analysis. We propose a new solution to mitigation this issue. More specifically, we tag data flows and taint them between code blocks. Based on the state-copy symbol execution techniques, we can calculate the overall data flow relationship in parallel, and check state consistency online. It is worthy that detecting state inconsistency between different program paths no matter there is any broken consistency rules.

The SC2NRF system tracks taint data flows of program paths. In general, it performs a fine-grained taint propagation analysis to track each value's source and data flow relation.

However, previous taint analysis solutions trace only the taint sources. The relation between different data flow and taint propagation process are not recorded. For example, programs may use *strlen* or other custom functions to infer some values (e.g., length) of the inputs, and then use them as size to allocate memory. In this case, operations upon the input buffer are indirectly affected by the inputs. Both the input buffer and variable that record the length of the buffer need to be record.

Classical dynamic taint analysis solutions usually do not propagate taint information for control dependencies [16], due to the concern of taint propagation efficiency. Instead, the SC2NRF system taints the symbol input, the data flow relation and their propagation by extending the symbol execution techniques to support fine-grained data flow taint. More specially, this system leverages the position of input bytes as taint attributes, and propagate these taint expressions along the trace.

For the generic instrumentation, we adopt a taint propagation policy based on the notion of data dependency [17–19]. If the output is a direct copy or transformation of the input, then it will be tainted if the input is tainted. This gives us a good coverage over all instructions and allows us to implement taint propagation without a special handler for each type of instruction in VEX IR instruction set, which is an architecture-agnostic, side-effects-free representation of a number of target machine languages. Based on VEX IR, the SC2NRF system implements taint propagation with operation expressions for tracing fine-grained data flow relations. To achieve this goal, we have identified several kinds of operation expressions in Table 1. These operation expressions account for the exact semantics of the data flow. They are used both as a performance improvement for commonly-used instructions, and an accuracy improvement for instructions with certain modes of operation that violates the basic data dependency rules.

To record the taint attributes and taint expression in symbol execution process, we have defined several kinds of set to aid analysis process. These recording set include store expression table $STORE_{REC}$, register expression table REG_{REC}, temp expression table TMP_{REC} and global taint label table $GLOBALTAINT_{REC}$, to record the memory store records, register writing records, local variables and immediate variables and global taint information in the symbol execution process.

Furthermore, the traces we collected only include user space instructions. Then some data flows will be missing when the system library functions are called. The

Table 1. Operation expression cases to generic data dependency propagation.

Instruction type	Reason and specific expression
Algorithm, logic and BITS wide transformation instructions	These instructions are recorded with the form of value = *op*(EXP_{A1}, EXP_{A2},…), and new expression is built and inserted into TMP_{REC}, where *op* is a concrete instruction type and A_i is one of the arguments
System library call	These instructions are identified by function call graphs generated by static analysis, and are recorded with the form of value = *call*(EXP_{A1}, EXP_{A2},…), and a new expression is built and inserted into $TMP_{REC,}$ where *call* is a concrete instruction type and A_i is one of the arguments
Memory store instructions	To identify the operation semantics of these instructions, both the target memory *address* and the *data* need to be checked before generating an expression. These writing data could be one of the memory store types, include arguments preparing for next function call with the form of *BP+positive_immediate_value*, local variable saving for temp results with the form of *BP-positive_immediate_value or BP-negative_immediate_value*, reading return results from sub function call *SP-positive_immediate*. Hence, the corresponding expression is inserted into $STORE_{REC}$(*address*)
Memory loading instructions	Similarly, the address of memory loading instructions could be one of the following types, namely loading immediate value with the form of *immediate_value*, loading argument values from program stack with the form of *add*(*sp*, *positive_immediate*), loading temp results from program stack with the form of *add*(*bp*, *negative_immediate*), and so on, then an expression with the form is inserted into TMP_{REC}
Register writing instructions	In these register writing instructions, the written data should be one of the following types, immediate value with the form of *immediate*, or another temp result in TMP_{REC}. In all cases, the target register ID is checked to analyze the program semantic. For example, if the register ID is BP or SP, the instruction is going to preparing program stack for function call entry or function call exit. Otherwise, an expression is inserted into REG_{REC} to save the temp result
Register reading instructions	At first, if the register ID is in REG_{REC}, then get the operation expression from REG_{REC} and insert it into TMP_{REC}. Otherwise, we use the lazy initialization approach proposed in previous work to initialize the register and generate a temp expression and insert it into TMP_{REC}

SC2NRF system will check values of registers before and after the system call instructions (e.g., malloc). If the value of any register other than the destination operand has changed, a potential data flow missing is found. In this case, we will clean the taint attributes of the register to avoid false positives. Another choice is using summary information of syscalls to propagate the taint attributes for kernel execution.

The SC2NRF system leverages a lazy initialization policy to initialize registers in REG_{REC} and global memory in TMP_{REC}. As non-reentrant functions with initialization code that is responsible for setting various memory locations to initial values, setting up request handlers, and performing other housekeeping tasks cannot execute before analysis if some data structures are not initialized, superfluous paths based on normally infeasible environment conditions are introduced into the analysis. To mitigate this, SC2NRF system adopts a lazy approach to memory and register initialization. When the symbol execution engine encounters a memory read from uninitialized memory, it identifies other procedures that contain direct memory writes to that location, and labels them as initialization procedures. If an initialization procedure is identified, the state is duplicated: one state continues execution without modification, while the other one runs the initialization procedure before resuming execution. This allows the SC2NRF system to safely execute initialization code without the risk of breaking the analysis.

With the taint propagation process defined in Sect. 2.2, the SC2NRF system can build global information flow relations along the program traces in symbol execution. Let uses T_G for a global variable set and T_L as a local variable set, then the information flow of a non-reentrant function T_F can be built. Based on the taint expression and symbol execution, the information flow building process is shown as follows.

Let uses f as a non-reentrant function to be analyzed, CFG_f is the inter-process control flow graph of function f, c_i is a basic block in CFG_f's node set. Let uses $T_I = T_G \cup T_L \cup T_F$ to denote the variable set to be analyzed in code block $_c$, where $T_G = \{T_{g1}, T_{g2},\ldots, T_{gm}\}$, $T_L = \{T_{l1}, T_{l2},\ldots, T_{lm}\}$ and $T_F = \{T_{f1}, T_{f2},\ldots, T_{fm}\}$. For each step in symbol execution, let uses c to the denote current code block executed. The following steps S1–S5 illustrate how to build the global information flow.

S1. Sets the arguments and global variables in function f as tainted variables;
S2. For each element $y_i = op(x_i)$ in I_c, The SC2NRF system uses the operation expression case to propagate the tainted variables and record the taint expression simultaneously.
S3. If y_i is an global variable in T_G, then we can search $STORE_{REC}$ using the key y_i and get $exp = STORE_{REC}[y_i]$, which is the information flow of y_i. Meanwhile, the taint label of y_i can be obtained by expression $GLOBALTAINT_{REC}[exp]$;
S4. If y_i is an local variable in T_I, then we can search TMP_{REC} using the key y_i and denote it as $exp = TMP_{REC}[y_i]$, which is the information flow of y_i. Also, the taint label of y_i can be obtained by expression $GLOBALTAINT_{REC}[exp]$;
S5. If y_i is an argument to a sub function in T_F, then it will be disposed according to the type. If y_i is a register argument, we search the REG_{REC} with expression y_i and denote it as $exp = REG_{REC}[y_i]$, which is the information flow of y_i, and the taint label of y_i is $GLOBALTAINT_{REC}[exp]$. Otherwise, the argument y_i is a stack argument, then we acquire the temp expression of y_i from $TMP_{REC}[y_i]$ and search the $STORE_{REC}$ with keyword $exp = TMP_{REC}[y_i]$, that is, we get $exp_1 = STORE_{REC}[exp]$,

which is the information flow of y_i. At last, the taint label of y_i can be expressed as $GLOBALTAINT_{REC}[exp_1]$.

2.3 State Consistency Checking Algorithm

To provide automated support for state consistency checking requirements, we have design an algorithm for analyzing the program states. The SC2NRF system checks the requirements of state consistency against a given consistency rules. Because the consistency rules describe the properties that non-reentrant functions must satisfy, the consistency rules by our algorithm are independent of a particular application.

Consistency rules defined in the SC2NRF system are expressed with condition lists. Meanwhile, the condition types include operation timing condition, space condition, other attribute condition, which usually are defined on function calls, value assignments, and memory positions and so on. For example, given a consistency rule of the case that a global buffer pointer and a local buffer pointer pointing to the same location, it follows a common operation list with the form of R = <*malloc, free, set_to_NULL*> (otherwise double-free or use-after-free vulnerability will happen on this buffer).

Checking the consistency of program states is usually quite complex. We must follow different data flows of this buffer and check whether the common operation list is preserved during the execution process. To solve this issue, the SC2NRF system leverages a consistency checker to incrementally check program state when a new code block is executed. We design a checking algorithm to validate operation expressions of each code block's execution against current program states, which is show in Algorithm 1. To simplify the analysis process, we use function *get_exp*(x) to denote the retrieving of taint expression of variable x, *update_state* to denote the taint propagation and taint expressions computing.

Algorithm 1. Consistency checking algorithm for program state.

Inputs	Consistency rules R; Code block c; Operation list I_c={y_i=op(x_i), for x_i in block c}; Current program state set S;
Outputs	Inconsistency results IR;

Step S0	Set IR={}
	For each element y_i=op(x_i) in I_c:
	  exp=*get_exp*(y_i)
	  S=*update_state*(S, exp)
	  For each state s in S:
	  For each r in R:

Step S1	If s_1.flow ∩ r.conditions ≠ ∅ and p s_2.flow ∩ r.conditions ≠ ∅ :
	  ir=<s_1, s_2, *r.conditions*>
	  IR.add(ir)
	  End for
	  End for
	End for

Consistency rules are broken once their conditions are meet in two data flow, which is shown in Algorithm 1. Thus, this algorithm checks the program state from the angle of data flow relations, consistency conditions and new attributes in current code block c. It is easy to know that the computational complexity of this algorithm is $O(max(|R|) \times max(|S|) \times max(|I|))$, where the upper bound of $|I|$ is the max size of a code block size, and the upper bound of $|S|$ is the max number of paths in target non-reentrant function.

3 Implementation and Evaluations

In this section, we present the implementations and evaluation of the SC2NRF system. Our prototype implementation leverages *angr* as the symbol execution engine, which provides the state-copy path discovery and is suitable for our state management and consistency checking. In addition, the taint propagation and data flow relations are expressed in VEX immediate representation language. The taint analysis components takes about 1100 LOC of python codes, and this component is integrated with *angr*'s main code. Meanwhile, the *state* class of *angr* is also patched to support the taint expression computing.

The analysis environment is a 64-bit Ubuntu 16.04 system running on a computer of Intel Xeon (R) CPU E5-2630 v4 2.20 GHz and 64G RAM. As show in Table 2, 4 known vulnerabilities in 4 applications are validated in our experiment. During this test, three kinds of data were collected, including the number of code blocks, consumed time and the length of tainted instructions. The total number of code blocks in a tested function represents the workload of the analysis process, and the consumed time of each test is increased as the number of code blocks increases. The max length of tainted expressions in each function varies according to the computed data flow. Finally, the SC2NRF is able to discover all of them successfully. It is worth noting that, additional python codes are needed to analyze each function.

Table 2. Known state inconsistencies validated by SC2NRF system.

ID	Function	Code blocks	Consumed time (s)	Max length of tainted Exp.
CVE-2018-0101	sub_8079B40	895	4.12	29
CVE-2015-0291	sub_4406C0	1335	6.43	8
CVE-2015-8651	sub_996EC80	2006	8.02	24
CVE-2011-0073	sub_1046620E	720	3.94	76

For vulnerability CVE-2015-0327 in Openssl 1.0.2, it requires to invoke the *clinethello* message twice. During the execution process, different paths will check a

common global pointer for reading and writing. Our solution can detect the inconsistencies in the two paths. Other samples also illustrate the effectiveness of our approach.

Actually, this solution in general will not generate false negatives since the rules are defined manually. The SC2NRF system can validate inconsistencies existing in known vulnerabilities but not all are not exploitable. Another disadvantage of SC2NRF is that the non-reentrant functions in binary program need to be identified manually by program analysis, which is a time-consume job.

4 Conclusions

State inconsistencies are the root cause of memory corruptions in non-reentrant functions. In this paper, we have proposed an approach for state consistency checking which based on consistency rules, and implemented a system called SC2NRF using famous symbol execution *angr*. The key components of SC2NRF system include taint assisted symbol execution module and consistency checking module. It is able to explore each program path in depth to find inconsistencies that are hard to detect and prone to miss by existing solutions. We also evaluated the SC2NRF system with several known CVEs, making our solution more practical. The experiment results show that this solution is effective.

References

1. Agre, J.R., Tripathi, S.K.: Modeling reentrant and nonreentrant software. In: Proceedings of the 1982 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems, pp. 163–178 (1982)
2. Wang, K., Chen, J.: Symmetry Detection for incompletely specified functions. In DAC Conference, pp. 434–437 (2004)
3. Kiffer, L., Rajaraman, R.: A better method to analyze blockchain consistency. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, pp. 729–744 (2018)
4. Tsankov, P., Dan, A., Drachsler Cohen, D., Gervais, A., Buenzli, F., Vechev, M.: Securify: practical security analysis of smart contracts. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, pp. 1–15 (2018)
5. Feist, J., Laurent, M., Potet, M.L.: Statically detecting use after free on binary code. J. Comput. Virol. Hacking Tech. **10**(3), 211–217 (2014)
6. Long, F., Sidiroglou-Douskos, S., Kim, D., Rinard, M.: Sound input filter generation for integer overflow errors. In: Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 439–452 (2014)
7. Do, T., Fong, A.C.M., Pears, R.: Dynamic symbolic execution guided by data dependency analysis for high structural coverage. In: Maciaszek, L.A., Filipe, J. (eds.) ENASE 2012. CCIS, vol. 410, pp. 3–15. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-45422-6_1
8. Corin, R., Manzano, F.A.: Taint analysis of security code in the KLEE symbolic execution engine. In: Chim, T.W., Yuen, T.H. (eds.) ICICS 2012. LNCS, vol. 7618, pp. 264–275. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-34129-8_23

9. Bai, J.J., Wang, Y.-P., Lawall, J., Hu, S.: DSAC : effective static analysis of sleep-in-atomic-context bugs in kernel modules bugs in kernel modules. In: Proceedings of the 2018 USENIX Annual Technical Conference, pp. 587–600 (2018)
10. Yamaguchi, F., Maier, A., Gascon, H., Rieck, K.: Automatic inference of search patterns for taint-style vulnerabilities. In: Proceedings of the 2015 IEEE Symposium on Security and Privacy, pp. 797–812 (2015)
11. Xue, L., et al.: NDroid: towards tracking information flows across multiple android contexts. IEEE Trans. Inf. Forensics Secur. **14**(3), 814–828 (2019)
12. Jia, X., Su, P., Yang, Y., Huang, H., Feng, D.: Towards efficient heap overflow discovery. In: Proceedings of the 26th USENIX Security Symposium, pp. 989–1007 (2017)
13. Lu, K., Wang, P.-F., Li, G., Zhou, X.: Untrusted hardware causes double-fetch problems in the I/O memory. J. Comput. Sci. Technol. **33**(3), 587–602 (2018)
14. Wang, P., Lu, K., Li, G., Zhou, X.: A survey of the double-fetch vulnerabilities. Concurr. Comput. Pract. Exp. **30**(6), 1–20 (2018)
15. Tok, T.B., Guyer, Samuel Z., Lin, C.: Efficient flow-sensitive interprocedural data-flow analysis in the presence of pointers. In: Mycroft, A., Zeller, A. (eds.) CC 2006. LNCS, vol. 3923, pp. 17–31. Springer, Heidelberg (2006). https://doi.org/10.1007/11688839_3
16. Schwartz, E.J., Avgerinos, T., Brumley, D.: All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In: IEEE Symposium on Security and Privacy, pp. 317–331 (2010)
17. Daniel, M., Honoroff, J., Daniel, M., Charlie, M.: Engineering heap overflow exploits with Javascript. In: Proceedings of WOOT, pp. 1–6 (2008)
18. Duck, G.J., Yap, R.H., Carvallaro, L.: Stack bounds protection with low fat pointers. In: Network and Distributed System Security Symposium, pp. 1–20 (2017)
19. Neugschwandtner, M., Milani Comparetti, P., Haller, I., Bos, H.: The borg: nanoprobing binaries for buffer overreads. In: Proceedings of the 5th ACM Conference on Data and Application Security and Privacy, pp. 1–19 (2015)