The Propagation Strategy Model of Taint Analysis

Yuzhu Ren*, Jiangtao Zhao1 and Cao Zhang1

1State Key Laboratory of Mathematical Engineering and Advanced Computing, Zhengzhou, Henan, 450000, China
*Corresponding author’s e-mail: 11732304@qq.com

Abstract. The taint propagation strategy is the core of the taint analysis technology. When the taint analysis tool analyses the target program, it needs to mark the target data according to the formulated taint propagation strategy. Formulating a reasonable strategy for taint propagation can effectively improve the accuracy of taint analysis. There are two difficulties in developing the taint propagation strategy, namely, alias analysis in static taint analysis and indirect jump in dynamic taint analysis. In our paper, we propose a taint propagation strategy at the intermediate representation language level, which can effectively improve the accuracy of taint propagation.

1. Introduction

Taint analysis is a technique for adding taintlabels to data accessed in a target program and analysing the data propagation process to test program’s security. This technology was proposed in 1998, and has received the attention of researchers since then. It has been widely used in information leakage detection, vulnerability detection, reverse engineering and so on.

The process of the taint analysis is generally divided into three steps [1], taint source identification, taint propagation analysis, and taint convergence point detection. Among them, the taint propagation analysis is the key content of the taint analysis technology. And the taint propagation strategy model determines the accuracy of the taint propagation analysis.

The process of taint propagation is primarily done by dependencies between the program’s variables. According to the dependence, the taint analysis technology can be divided into explicit-flow analysis and implicit-flow analysis.

The explicit-flow analysis mainly includes the data-dependency between the program’s variables, that is, the variable’s taint information an is directly transmitted to the variable b through operations such as assignment or arithmetic operation. There are two problems with taint analysis techniques, namely overtainting and undertainting. Overtainting means that data variables with no dependence on the taint source are marked as taint in the process of taint propagation, that is, false positives are generated. Undertainting is that data variables with dependence on the taint source aren’t marked as taint in the process of taint propagation, that is, false negatives are generated.

In the paper, we summarized the function of the instructions and formulated the corresponding taint propagation strategy based on the application experience of the taint analysis technology.

2. Related work

When researchers analyse the data-dependency, first restore the Call Graph, and then perform specific taint propagation analysis within the function or between functions according to different program characteristics. Common ways of taint propagation include direct assignment propagation, function
call propagation, and alias propagation. Direct assignment propagation and function call propagation have been thoroughly studied in many papers [2-4]. However, there is still room for further improvement in the analysis techniques for alias propagation. Highly accurate static taint analysis techniques tend to have large time and space overhead, so researchers have designed a variety of methods for alias propagation analysis. For example, Livshits and Lam [5] use the context-sensitive alias analysis method to detect Java application vulnerabilities with the PQL [6] (program query language) taint check strategy. Tripp [7] et al. implemented the TAJ tool to perform taint analysis on Java Web applications by means of hybrid slicing combined with object-sensitive alias analysis. The Andromeda [8] tool uses object-sensitive alias analysis to solve the object's access path problem.

When researchers analyse the control-dependency, we need to restore the Control-Flow Graph. Control-flow recovery has been widely discussed in many papers [9-13]. The recovery control-flow graph uses a recursive algorithm to reverse and analyse the basic block $B_a$, identify its possible exits (e.g., subsequent basic blocks $B_b$ and $B_c$) and add them to the control-flow graph, and then repeat recursive analysis of $B_b$ and $B_c$ until no new exits are identified and all basic blocks are added to the control-flow graph. Indirect Jump is a difficulty in control-flow graph recovery. An indirect jump occurs when a binary file transfers control-flow to a target represented by a value in a register or memory. The target with direct jump is encoded into the instruction itself, which is different from normal parsing. The target of indirect jump is affected by multiple factors. Specifically, indirect jump can be divided into the following categories:

**Computed jump.** The goal of computed jump is determined by the application by performing the calculations specified by the code. This calculation can be further dependent on values in other registers or memory. A common example is a jump-table: an application uses registers or memory values to determine an index in a jump-table stored in memory, reads the target address from the index, and then jumps to this address.

**Context-sensitive jump.** Indirect jumps may also depend on the context of the application. A common example is `qsort` in the standard C library, which performs a callback operation to compare given values. Therefore, some jump targets of the basic block in `qsort()` depend on its caller because the caller provides the callback function.

**Object-sensitive jump.** Context jump is a special case of the object jump. Object polymorphism requires the use of virtual functions, usually in the form of function pointer virtual tables, which are queried at runtime to determine the jump target. Therefore, the jump target depends on the type of object that the caller passes to the function.

In addition, there are stream-sensitive, domain-sensitive, and path-sensitive concepts.

Kinder and Veith [14] extended the use of iterative disassembly based on linear scan and recursive descent, and used the data stream analysis to perform multiple disassembly iterations. The static binary analysis platform Jakstab was implemented with disassembly and static analysis. Function, you can establish a control-flow graph during the disassembly iteration to get more accurate results.

### 3. Taint propagation strategy

The taint propagation requires the precise strategies. To ensure portability, we formulate taint propagation at the level of intermediate representation. According to the characteristics of the intermediate representation statement, the taint propagation strategy can be divided into 7 categories.

#### 3.1 Single-data-dependency

As shown in Figure 1, single-data-dependency instructions such as constants, temporary variables, registers, memory values, whose taint labels can be directly propagated to the target data.
3.2 Multi-data-dependency
Multi-data-dependency mainly includes multi-parameter expressions such as bit-operation and arithmetic-operation. The taint propagation strategy of the bit-operation executed the And operation with the multiple parameters’ taint labels and then passed the result to the target data, and the arithmetic-operation propagates the taint labels to the target, as shown in Figure 2.

![Figure 2. Taintlabels propagation diagram of multi-data-dependency](image)

3.3 Data-extension-dependency
Data-extension-dependency mainly includes some data-length-extension expressions, such as 8Uto32, 8Uto64, 32Sto64, and so on. The data-extension-dependency can be divided into zero-extension and sign-extension. The zero-extension is for unsigned numbers and signed non-negative extensions. The taint state of the extended part is marked as 0. The sign-extension is to extend the signed negative number with sign bit of 1. The highest bit taint state of the extended portion is 1, and the taint state of the other extended portion is 0, as shown in Figure 3.

![Figure 3. Taintlabels propagation diagram of Data-extension-dependency](image)

3.4 ITE-dependency
The ITE-expression has three parameters. First, it determines whether the first parameter is TRUE or FALSE. If it is TRUE, it returns the second parameter, or returns the third. The propagation strategy is that if the first parameter is taint data, the return label is taint. If the first parameter is un-taint, the expression is marked with the second or third parameter taint label according to the condition. The result labels are shown in Figure 4.
3.5 The call of the libc library function
Since the taint analysis simulation also involves the call to the libc library function, the taint propagation strategy should also include the processing of the libc library function. As mentioned before, according to the function of the libc library to develop taint propagation strategy, can improve the efficiency of the taint analysis. For example, the memcpy function writes the data stored in the source memory to the target memory, so its taint propagation strategy is spreads the taint label of the source memory to the target memory.

3.6 Control dependency
The condition judgment of the VEX module is if-jump. The taint propagation strategy records the address of the condition judgment statement affected by the taint variable, and then needs to analyze the control dependency more deeply manual analysis.

3.7 Harmless treatment
Harmless processing is the case where the original taint label is cleared after the data has passed through the intermediate representation statement or library function. For example, the return values of statements such as Xor(t1, t1), Sub(t1, t1), and Mul(t1, 0) are fixed values, and library functions such as memset and free. The significance of harmless treatment is to reduce the problem of overtaining and improve the accuracy of taint analysis.
4. Implement
In this paper, we used python3.x to develop a taint propagation strategy and apply it to our prototype. Through verification, our prototype can effectively detect vulnerabilities such as stack overflow and heap overflow in target program.

In our prototype, the granularity is marked by byte, and the bitmap information is used (that is, each byte uses one bit of '0' or '1' to indicate the taint state of the byte) to store the taint information. The taint propagation diagram is shown in Figure 5.

```
----- IMark(0x400010, 1, 0)-----
 t0: GET:164(offset=36)
 SimlRepr_Get: offset=56 size=64 value=0x7fffffffdd30 tmark=00000000
 SimlRstmt_Wrtmp: tmp=0 value=0x7fffffffdd30
 PUT(offset=48) = t0
 SimlRepr_Rdtmp: tmp=0 value=0x7fffffffdd30 tmark=00000000
 SimlRstmt_Put: offset=48 size=64 value=0x7fffffffdd30
 t1: ADD:164(t0, 0, 0x0000000000000008)
 SimlRepr_Rdtmp: tmp=0 value=0x7fffffffdd30 tmark=00000000
 SimlRepr_Const: type=ity_164 value=0x8 tmark=00000000
 _sim_add: arg0=0x7fffffffdd30 arg1=0x8 res=0x7fffffffdd30 tmark=00000000
 SimlRstmt_Wrtmp: tmp=5 value=0x7fffffffdd38
 PUT(offset=48) = t5
 SimlRepr_Rdtmp: tmp=5 value=0x7fffffffdd38 tmark=00000000
 SimlRstmt_Put: offset=48 size=64 value=0x7fffffffdd38
 PUT(offset=184) = 0x00000000000040001f
 SimlRepr_Const: type=ity_164 value=0x40001f tmark=00000000
 SimlRstmt_Put: offset=184 size=64 value=0x40001f
```

Figure 5. Taint propagation diagram

5. Conclusion
In this paper, we introduced the related work of taint analysis technology. And based on this, we summarize the seven types of taintlabel propagation strategy in taint analysis. By formulating a taint propagation strategy, the accuracy of taint analysis can be improved, and the problem of overtaining and undertaining can be reduced.

With the development of technology, researchers use machine learning methods to formulates the propagation strategy of the control-dependency, and have implemented some tools [15]. This shows that the accuracy of the taint analysis technology can be further improved, which is the direction of our next work.

References
[1] REN Yuzhu, ZHANG Youwei, AI Chengwei. Survey on taint analysis technology. Journal of Computer Applications, 2019, 39(8): 2302-2309.
[2] Corin, R, & Felipe Andrés Manzano. (2012). Taint Analysis of Security Code in the KLEE Symbolic Execution Engine. In: Proceedings of the 14th international conference on Information and Communications Security. Springer, Berlin, Heidelberg.
[3] Jin-Xin, M. A., Zhou-Jun, L. I., Tao, Z., Dong, S., & Zhang-Kai, Z. (2017). Taint analysis method based on offline indices of instruction trace. Journal of Software.
[4] Ma, J., Zhang, P., Dong, G., Shao, S., & Zhang, J. (2015). TWalker: An efficient taint analysis tool. 2014 10th International Conference on Information Assurance and Security. IEEE.
[5] Livshits V B, Lam M S. (2005). Finding Security Vulnerabilities in Java Applications with Static Analysis. In: Conference on Usenix Security Symposium. USENIX Association.

[6] Martin, M. C, Livshits, V. B., & Lam, M. S. (2005). Finding Application Errors and Security Flaws Using PQL: A Program Query Language. In: Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, San Diego, CA, USA.

[7] Tripp, O., Pistoia, M., Fink, S. J., Sridharan, M., & Weisman, O. (2009). Taj: effective taint analysis of web applications. ACM Sigplan Notices, 44(6), 87-97.

[8] Tripp, O., Pistoia, M., Cousot, P., Cousot, R., & Guarnieri, S. (2013). ANDROMEDA: accurate and scalable security analysis of web applications. In: Proceedings of the 16th International Conference on Fundamental Approaches to Software Engineering.

[9] Jian, C., Xiaojun, S., Yiyang, Y., Yafei, X., & Xiaoming, J. U. (2018). Error detection algorithm for pure software signature based on table-driven. Computer Engineering.

[10] Singh, Anishka & Arora, Rohit & Pareek, Himanshu. (2017). Malware Analysis using Multiple API Sequence Mining Control Flow Graph.

[11] Jain, S., Pal, R., & Srivastava, A. (2014). Control flow graph based framework for effective evaluation of source code. International Journal of Computer Applications, 82(16), 20-27.

[12] Nerina Bermudo, Andreas Krall, & R. Nigel Horspool. (2005). Control Flow Graph Reconstruction for Assembly Language Programs with Delayed Instructions. In: 5th IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2005), 30 September - 1 October 2005, Budapest, Hungary. IEEE.

[13] Dietrich, C., Hoffmann, M., & Lohmann, D. (2015). Cross-kernel control-flow-graph analysis for event-driven real-time systems. , 50(5), 1-10.

[14] Kinder, J., & Veith, H. (2008). Jakstab: A Static Analysis Platform for Binaries. In: Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, NJ, USA, July 7-14, 2008, Proceedings. Springer-Verlag.

[15] Rathi, D., & Jindal, R. (2018). Droidmark: a tool for android malware detection using taint analysis and bayesian network.