A Critical Review of Dynamic Taint Analysis and Forward Symbolic Execution

Asankhaya Sharma
Department of Computer Science
National University of Singapore
asankhs@comp.nus.edu.sg

ABSTRACT

In this note, we describe a critical review of the paper titled "All you wanted to know about dynamics taint analysis and forward symbolic execution (but may have been afraid to ask)" [1]. We analyze the paper using Paul Elder critical thinking framework [2]. We start with a summary of the paper and motivation behind the research work described in [1]. Then we evaluate the study with respect to the universal intellectual standards of [2]. We find that the paper provides a good survey of the existing techniques and algorithms used for security analysis. It explains them using the theoretical framework of operational runtime semantics. However in some places the paper can do a better job in highlighting what new insights or heuristics can be gained from a runtime semantics formulation. The paper fails to convince the reader how such an intricate understanding of operational semantics of a new generic language SimpIL helps in advancing the state of the art in dynamic taint analysis and forward symbolic execution. We also found that the Paul Elder critical thinking framework is a useful technique to reason about and analyze research papers.

Keywords

Dynamic Taint Analysis, Forward Symbolic Execution, Security Analysis

1. INTRODUCTION

Taint analysis refers to tracking of information flow through the program. It can be used to enforce security policies and detect malicious inputs. Taint analysis can be done using dynamic as well as static techniques. The paper [1] focusses on dynamic taint analysis and forward symbolic execution. The motivation for undertaking such a study is the usefulness of these methods in Unknown Vulnerability Detection, Automatic Input Filter Generation, Malware Analysis and Test Case Generation. The paper introduces a general language SimpIL which is used to describe the dynamic taint analysis and forward symbolic execution algorithms. SimpIL language provides a generic uniform framework to describe and discuss existing dynamic taint analysis and forward symbolic execution algorithms. Building on the operational semantics for this language the paper shows how taint policies can be specified and enforced as runtime semantics. Forward symbolic execution can also be defined for SimpIL in a similar way which can then be used for symbolic reasoning.

Thus the primary motivation behind the paper is to provide a well-defined language and framework to compare and contrast existing techniques of dynamic taint analysis and forward symbolic execution. The paper also describes various challenges and design choices that the user faces while building a taint analysis tool for a particular real world system. Overall we find that the paper is well motivated and explains the state of the art in taint analysis from a theoretical perspective by using operational semantics of a language. In existing work [3, 4] on dynamic taint analysis the underlying language is not formalized and the operational semantics are not given clearly. This leads to ambiguities in interpretation of results and does not explain the design choices. This paper builds on the operational semantics of SimpIL to address these short comings and provides a common way to describe the existing work on dynamic taint analysis.

On the other hand forward symbolic execution has been shown [5] to be useful for malware analysis and finding unknown vulnerability in the code. The operational semantics of SimpIL can also be used to define a forward symbolic execution framework for the language. This helps to understand the current symbolic execution engines (KLEE [7], EXE [8], and Bitblaze [6]) used for security analysis from the perspective of programming language semantics. However the paper does not do a good job of incorporating the static analysis results into a symbolic execution framework as described in [9]. Our main findings of this critical review are the following.

- The paper defines a generic language SimpIL which can explain existing algorithms for dynamic taint analysis and forward symbolic execution as extensions of runtime semantics.
- The paper provides board coverage of existing techniques and does not go in depth.
- The paper is not precise in explaining how to mitigate some of the pitfalls in implementing these algorithms for a real world system.
- The paper does not provide evaluation or evidence to support the claim that operational semantics are a good way to describe these algorithms. In particular it is not clear to the reader what new or novel technique can be supported in this framework which is not present in the literature.
- There is no soundness or completeness result for the operational semantics of SimpIL which leads to under tainting and over tainting.

We have organized the critique as follows. In the next section we will describe the SimpIL language and summarize the critics of the features and semantics of the language. The section 3 will focus on dynamic taint analysis, while section 4 deals with forward symbolic execution. We will review some related work in section 5 and finally we conclude in section 6.

2. SIMPIL LANGUAGE

In existing work on dynamic taint analysis and forward symbolic execution it has been shown that assembly-like languages can be used to reason about programs written in any language. The language used in the formalization of the paper resembles a simple intermediate language used typically by compilers. The expressions in the language are side-effect free. Following shows a simple sample program written in SimpIL which is used for illustration throughout this note.

```
x := 2 * get_input(.)
y := 5 + x
goto y
```

In reality however the expression in a given assembly language may not be side-effect free and thus have to be translated into a form which makes them side-effect free [6]. The SimpIL language does not have high-level features like functions, buffers and user-level abstractions. These constructs also have to be encoded or translated into the form used by SimpIL. The following example shows how a function can be encoded in the SimpIL language.

```
/* Caller function */
esp := esp + 4
store ( esp , 6 ) /* retaddr is 6 */
goto 9
/* The call will return here */
halt
/* Callee function */
…
goto load ( esp )
```

Nevertheless, the language is powerful enough to describe most dynamic taint analysis algorithms. The authors provide examples of several taint policies which can be implemented as extensions of runtime semantics of SimpIL. Several alternative policy choices can all be incorporated in the same framework of operational semantics. This is claimed by the authors to be the biggest benefit of using such a formulation.

2.1 Operational Semantics

The approach taken in this paper is to introduce an operational semantics for the language SimpIL which is later used to describe dynamic taint analysis and forward symbolic execution. The semantics of SimpIL presented in the paper is actually big step semantics (and not small step semantics). The expressions are evaluated to values in each of the rules. However the authors of the paper do not mention this point and just use the generic term of operational semantics to describe their algorithms. The operational semantics is described clearly in the paper with several examples. These examples help make the notion concrete and precise for the reader.

The authors argue that since dynamic program analyses are defined in terms of actual program executions, operational semantics provide a natural way to define dynamic analysis. This observation is critical to support both dynamic taint analysis and forward symbolic execution as runtime extensions of the operational semantics. This observation is fair from a security analysis perspective however from a programming language perspective this may be stretching the idea of operational semantics a bit too much. In programming language research semantics for languages have been widely studied. Several kinds of semantics have been proposed and used depending on the domain like operational semantics (big step and small step), denotational semantics, axiomatic semantics, algebraic semantics etc.

Usually a language is defined with syntax and semantics while program analyses are understood in form of different frameworks like monotone framework, abstract interpretation, type-effects system etc. The problem with using operational semantics to define dynamic analysis becomes clear when we have to introduce a new semantics of every kind of taint policy. The taint policies have to be baked into the semantics. So depending on the application domain the user have to redefine all the rules of the operational semantics to take into account the new taint policy. This situation is not much different from the existing work where users have to build new algorithms in order to handle different application domains. We talk about it more in the next section.

3. DYNAMIC TAINT ANALYSIS

The purpose of dynamic taint analysis is to track information flow between sources and sinks. A taint policy is used to determine exactly how taint flows as a program executes. Since the operational semantics described in the paper for the language SimpIL is neither sound nor complete, the use of this operational semantics leads to both under tainting and over tainting. This is a common problem for most existing systems as well [3, 4, 6]. The following example shows how the taint is propagated from input in the sample SimpIL program.

```
x := 2 * get_input(.) {x → T}
y := 5 + x {x → T, y → T}
goto y {x → T, y → T}
```

3.1 Dynamic Taint Analysis Semantics

The operational semantics of SimpIL are modified to bake in taint policy rules. All the existing operational semantics rules of the SimpIL language are modified to take into account the propagation of the taint. With evaluation of an expression the taint is also propagated from the premise of the rule to the conclusion. Given a taint policy the propagation of the taint can be tracked through the program as defined by the dynamic taint analysis semantics. The exposition in this section is quite precise and detailed with examples explaining the user of taint policies in deriving the dynamic taint analysis semantics. Given a dynamic taint analysis semantics the taint policy can be applied to do taint checking.

3.2 Dynamic Taint Checking

This section introduces several dynamic taint policies in the paper taken from [6, 3, 2]. Taint can be introduced, propagated and checked using the dynamic taint analysis based operational semantics of SimpIL. Some application domains may have specific requirements like – memory address should be tainted. Users may also wish to define their own taint policy. These application and user specific requirements can lead to different

taint policies which can be handled by modifying the semantics to suit the policy.

There are two short comings of the approach mentioned in the paper – one it still leads to under tainting and over tainting, as the operational semantics is neither sound nor complete with respect to the taint policy. Under tainting refers to the case when the dynamic taint analysis does not introduce taint when it should. It can happen because of many reasons – unknown code, insufficient instrumentation, language features etc. Over tainting refers to the case when the analysis introduces taint when it is not necessary. Over tainting may be considered a sound and conservative approximation of the analysis. In practice it may lead to a large number of false positives.

In addition, there is no way in the semantics of SimpIL described in the paper to remove taint once it is added in the system. This is called the sanitization problem. In certain cases users may write routines to perform sanitization or to detect and handle malicious inputs. Existing systems like Temu [6] and TaintCheck [4], allow some simple cases to handle sanitization problem. They check for well-known constant functions to eliminate taint. It is not clear as to how the dynamic taint checking described in this paper can handle the sanitization routines. The language SimpIL does not have provision for functions and higher level constructs. It is not possible to add sanitization checks directly as extension of the runtime semantics. This leads us back to the situation in prior work where ad hoc heuristics are used to handle such corner cases. There are many challenges when implementing a taint analysis for a real system. They are discussed in the next section in detail.

3.3 Challenges

Many of the common problems with existing system are still there with the operational semantics of SimpIL. The time of detection vs. time of attack problem shows that a dynamic taint analysis may raise an alert too late. This paper does not address that issue. Some form of static checking may be helpful in these situations. Another aspect of the dynamic taint analysis semantics is that it cannot detect taint based on control flow. In [9], symbolic jumps are handled in the control flow by using a combination of static and dynamic techniques for taint analysis. This paper leaves out any static methods from its scope and thus is a bit narrow when addressing the challenges faced by practitioners in the field.

Existing systems for dynamic taint analysis use several heuristics to make them work in practice. The paper does not mention how these heuristics can be incorporated in the operational semantics of SimpIL. The description of dynamic taint analysis as an extension to runtime semantics of SimpIL does make the formulation easy to understand and read. The real world practical problems and pitfalls are mentioned in the paper although no solution is proposed. The paper also recognizes the limitations of doing a dynamic analysis, we cannot reason about multiple paths. The next section describes how they address the problem using forward symbolic execution.

4. FORWARD SYMBOLIC EXECUTION

Forward symbolic execution allows to reason about behavior of program on multiple paths by using logical formula to represent the program execution. This paper describes the semantics of forward symbolic execution for SimpIL and shows how that can be used to reason about security of the program. The description in the paper about this section is brief and only shows the rules for a subset of the SimpIL language. However it is clear to the reader how the other rules will look like as there are many examples. The following example shows how forward symbolic execution builds the path condition formula for the sample SimpIL program.

```
x := 2 * get_input(.) [true]
if x – 5 == 14 goto 3 else goto 4 [(2 * s) – 5 == 14]
if x – 5 == 14 goto 3 else goto 4 ![(2 * s) – 5 == 14]
```

4.1 Semantics of Forward Symbolic Execution

Similar to section 3.1 the forward symbolic execution rules are given as extensions to runtime semantics of the language SimpIL. For each of the operational semantics rule there is now a symbolic counterpart. It evaluates the premise symbolically and builds a formula called the path condition. The process of building a symbolic execution though conceptually simple has many practical problems. Some of them are mentioned in the paper.

- **Symbolic Memory.** All the memory references in the formula may not be concrete. Thus when taint is propagated it is not clear how to handle the variable which corresponds to symbolic memory.
- **System Calls.** The analysis may not have access to all the source code. System calls (such as IO), libraries and unknown procedures may lead to loss in precision of the analysis.
- **Path Selection.** During forward symbolic execution it is often not clear which path to choose. In general the number of paths in a program may be unbounded so some heuristic is needed in practice.

The paper does a good job of describing the problems but again falls short of giving solutions to some of these problems. The paper cites relevant existing systems and mention how they handle these issues. The semantics of forward symbolic execution are described only briefly.

In particular it is not clear how the paper adds to the understanding of the use of forward symbolic execution for security analysis. The formulation of this section in the paper is weaker when compared with dynamic taint analysis. There is already a huge amount of work in forward symbolic execution for test case generation and debugging from the software engineering community. This paper does not do a good job to showing how it builds on that body of work or how security analysis lead to some unique challenges in this area. In the next section we review some of the challenges mentioned in the paper on forward symbolic execution.

4.2 Challenges

The common challenges encountered while building a symbolic execution framework as described in the paper are already listed in section 4.1. The paper describes the symbolic memory problem in detail and shows how using a SMT Solver [11, 12] may be useful in such a scenario. By using satisfying solutions to the path condition formula containing the symbolic address we can generate new inputs for the program. This is already used in tools which do automated random testing like CUTE [10] and DART [13].

Another big problem with symbolic execution is deciding which path to choose. This is referred as the path selection problem in the paper. Path exploration has been well studied in the literature

on testing and debugging. This paper recalls some of the existing approaches.

- **Depth-First Search.** Exploring the paths in a depth first manner as used in KLEE [7] and EXE [8]. This approach can get stuck in an unwinding loop.
- **Concolic Testing.** This refers to using concrete execution to produce a trace of a program execution and then building the formula to follow the same path. This search strategy is also called generational search and is used in [10, 14, 15].
- **Random Paths.** KLEE [7] has support for choosing the path randomly with weights that are assigned to path based on depth.
- **Heuristics.** Most of the real systems use many heuristics like distance between instructions, states etc.

The path selection problem is a common challenge for symbolic execution engines and the forward symbolic execution based operational semantics of SimpIL also suffers from these shortcomings.

Another difficult aspect of forward symbolic execution is how to handle system/library calls or unknown code. One approach is to create summaries of their side effects [16, 7, 8]. Another approach can be to use concolic execution as in [10]. The paper mentions some existing work on this but cannot show the usefulness of the operational semantics based formulation of the forward symbolic execution of SimpIL.

The performance of forward symbolic execution is usually exponential in the number of program branches due to the path explosion problem. Existing systems perform heuristics which use caching of formulas [7, 8], elimination of redundant terms [15, 17] and weakest preconditions [18-21]. The following trivial example in form of a SimpIL program shows how program branches can quickly lead to exponential blowup. Even though all the assignments are same but the path conditions for these assignments can blow up if substitution is used.

```
x := get_input(.)
x := x + x
x := x + x
x := x + x
if e1 then S1 else S2
if e2 then S3 else S4
if e3 then S5 else S6
assert(x < 10)
```

5. RELATED WORK

The focus of the paper is on the use of operational semantics to define dynamic security mechanisms like taint analysis and forward symbolic execution. However there are other approaches which provide a similar framework for this [22, 23]. The prior work doesn't focus so much on dynamic taint analysis and taint policy checking.

The analysis descriptions in [13, 14, 24] use an informal semantics which can lead to ambiguities and errors in implementation. The paper provides a good comparison with existing related work and delineates the contribution of the authors well. The paper also lists some of the open problems in the area in form of challenges and opportunities. The work described in the paper can be applied to several different domains. Some of the applications provided in related work of the paper are as follows.

- **Automatic Test-case Generation.** Forward symbolic execution has been widely used in test-case generation to achieve high code coverage [7, 8, 13-15, 25]. However the paper does not provide an evaluation on how good the operational semantics of SimpIL is when compared to the rest.
- **Automatic Filter Generation.** Input filters can detect and block malicious inputs from the input stream [16-18]. No such filter for the SimpIL has been shown in the paper nor is it clear how such a filter can be derived from the operational semantics.
- **Automatic Network Protocol Understanding.** Dynamic taint analysis can help in understanding behavior of network protocols [26, 27]. The dynamic taint analysis semantics of the paper can help in formalization of some of the techniques present in papers on network protocol understanding.
- **Malware Analysis.** Dynamic taint analysis and forward symbolic execution can be used to analyze the malware behavior [5]. The paper presents a good framework for these analyses to be expressed as extensions of the runtime semantics.
- **Web Applications.** Taint analysis has also been used to detect attacks like SQL injection in web applications. However the SimpIL language is focused at assembly level and binary attacks, it does not directly correspond to such use cases.
- **Taint Performance & Frameworks.** The paper mentions techniques used in literature to improve the performance of taint analysis but does not offer any new insight in this aspect. The framework proposed in the paper is the key contribution of the paper. The operational semantics of SimpIL language can be extended to incorporate most of the existing taint analysis algorithms
- **Extensions to Taint Analysis.** The rules proposed in the paper assume data to be either tainted or not. Recent work [28] has proposed a generalization of taint analysis based on channel capacity which can quantify the amount of influence an input has on a particular program statement.

6. CONCLUSION

This note described a critical review of the paper titled "All you wanted to know about dynamics taint analysis and forward symbolic execution (but may have been afraid to ask)". We found the paper to be well written, clear and easy to read. Judging by using Paul Elder critical thinking framework we make the following observations. The paper has more breadth than depth. The paper is precise at most of the places. The paper is not very accurate as the operational semantics is not sound and complete. The paper is clear and follows a logical progression.

Starting from basics and definitions it builds on to describe the techniques and framework. The contribution of the paper is not significant as it does not report any new or novel results but helps in understanding existing systems. The paper is also not fair in its use of operational semantics as it requires defining a new semantics for each kind of taint policy. The paper is highly relevant to the research at that time. It provides a framework in which readers can describe problems and consider challenges.

We find the paper interesting and joy to read. The reader learns a lot about the field of dynamic taint analysis and forward symbolic execution. The paper also exposes to some unsolved problems and challenges in the area which are ripe targets for future work. Overall from programming language perspective the paper may not contribute much to the state of the art but it is very useful survey for the practitioners in security analysis. The Paul Elder critical thinking framework is a good way to analyze a research paper. It helped us reason about the paper from several different aspects and universal intellectual standards.

7. REFERENCES

[1] Edward J. Schwartz , Thanassis Avgerinos , David Brumley, All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask), Proceedings of the 2010 IEEE Symposium on Security and Privacy, p.317-331, May 16-19, 2010.

[2] Paul Elder Critical Thinking Framework. http://louisville.edu/ideastoaction/what/critical-thinking/paul-elder-framework

[3] James Clause, Wanchun Li, Alessandro Orso, Dytan: a generic dynamic taint analysis framework, Proceedings of the 2007 international symposium on Software testing and analysis, July 09-12, 2007.

[4] James Newsome and Dawn Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In Proceedings of the Network and Distributed System Security Symposium, February 2005.

[5] Andreas Moser, Christopher Kruegel, Engin Kirda, Exploring Multiple Execution Paths for Malware Analysis, Proceedings of the 2007 IEEE Symposium on Security and Privacy, p.231-245, May 20-23, 2007.

[6] Bitblaze binary analysis project. http://bitblaze.cs.berkeley.edu, 2007.

[7] Cristian Cadar, Daniel Dunbar, Dawson Engler, KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs, Proceedings of the 8th USENIX conference on Operating systems design and implementation, p.209-224, December 08-10, 2008.

[8] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski , David L. Dill , Dawson R. Engler, EXE: automatically generating inputs of death, Proceedings of the 13th ACM conference on Computer and communications security, October 30-November 03, 2006.

[9] Gogul Balakrishnan, Thomas Reps, WYSINWYX: What you see is not what you eXecute, ACM Transactions on Programming Languages and Systems (TOPLAS), v.32 n.6, p.1-84, August 2010.

[10] Koushik Sen, Darko Marinov , Gul Agha, CUTE: a concolic unit testing engine for C, Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering, September 05-09, 2005.

[11] Clark Barrett and Sergey Berezin. CVC Lite: A New Implementation of the Cooperating Validity Checker. In Proceedings of the 16th International Conference on Computer Aided Verification (CAV '04), volume 3114 of Lecture Notes in Computer Science, pages 515-518. Springer, July 2004.

[12] Vijay Ganesh, David L. Dill, A decision procedure for bit-vectors and arrays, Proceedings of the 19th international conference on Computer aided verification, July 03-07, 2007.

[13] Patrice Godefroid, Nils Klarlund, Koushik Sen, DART: directed automated random testing, Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation, June 12-15, 2005.

[14] Michael Emmi, Rupak Majumdar, Koushik Sen, Dynamic test input generation for database applications, Proceedings of the 2007 international symposium on Software testing and analysis, July 09-12, 2007.

[15] Patrice Godefroid, Michael Levin, and David Molnar. Automated whitebox fuzz testing. In Proceedings of the Network and Distributed System Security Symposium, February 2008.

[16] David Brumley, James Newsome, Dawn Song, Hao Wang, and Somesh Jha. Towards automatic generation of vulnerability-based signatures. In Proceedings of the IEEE Symposium on Security and Privacy, pages 2–16, 2006.

[17] Manuel Costa, Miguel Castro, Lidong Zhou, Lintao Zhang, and Marcus Peinado. Bouncer: Securing software by blocking bad input. In Proceedings of the ACM Symposium on Operating System Principles, October 2007.

[18] David Brumley, Hao Wang, Somesh Jha, and Dawn Song. Creating vulnerability signatures using weakest preconditions. In Proceedings of the IEEE Computer Security Foundations Symposium, 2007.

[19] C. Flanagan and J.B. Saxe. Avoiding exponential explosion: Generating compact verification conditions. In Proceedings of the Symposium on Principles of Programming Languages, 2001.

[20] K. Rustan M. Leino. Efficient weakest preconditions. Information Processing Letters, 93(6):281–288, 2005.

[21] William G. J. Halfond, Ro Orso, and Panagiotis Manolios. Using positive tainting and syntax-aware evaluation to counter SQL injection attacks. In Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering. ACM Press, 2006.

[22] Patrice Godefroid, Michael Levin, and David A. Molnar. Active property checking. In Proceedings of the ACM international conference on Embedded software, 2008.

[23] George C. Necula, Scott McPeak, and Westley Weimer. CCured: type-safe retrofitting of legacy code. In Proceedings of the Symposium on Principles of Programming Languages, 2002.

[24] Wei Xu, Eep Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In Proceedings of the USENIX Security Symposium, 2006.

[25] Cristian Cadar and Dawson Engler. Execution generated test cases: How to make systems code crash itself. In Proceedings of the International SPIN Workshop on Model Checking of Software, 2005.

[26] Juan Caballero, Heng Yin, Zhenkai Liang, and Dawn Song. Polyglot: Automatic extraction of protocol message format using dynamic binary analysis. In Proceedings of the ACM Conference on Computer and Communications Security, October 2007.

[27] Gilbert Wondracek, Paolo Milani Comparetti, Christopher Kruegel, and Engin Kirda. Automatic network protocol analysis. In Proceedings of the Network and Distributed System Security Symposium, 2008.

[28] James Newsome, Stephen McCamant, and Dawn Song. Measuring channel capacity to distinguish undue influence. In Proceedings of the ACM Workshop on Programming Languages and Analysis for Security, 2009.