ISSN 0361-7688, Programming and Computer Software, 2018, Vol. 44, No. 6, pp. 459–466. © Pleiades Publishing, Ltd., 2018.
Original Russian Text © M.V. Belyaev, N.V. Shimchik, V.N. Ignatyev, A.A. Belevantsev, published in Trudy Instituta Sistemnogo Programmirovaniya, 2017, Vol. 29, No. 3.

Comparative Analysis of Two Approaches to Static Taint Analysis

M. V. Belyaev[a],*, N. V. Shimchik[a],, V. N. Ignatyev[a],***, and A. A. Belevantsev[a,b],******

[a] *Ivannikov Institute for System Programming, Russian Academy of Sciences, Moscow, 109004 Russia*

[b] *Moscow State University, Moscow, 119992 Russia*

**e-mail: mbelyaev@ispras.ru*

***e-mail: shimnik@ispras.ru*

****e-mail: valery.ignatyev@ispras.ru*

*****e-mail: abel@ispras.ru*

Received July 19, 2018

Abstract—Currently, one of the most efficient ways to detect software security flaws is taint analysis. It can be based on static code analysis, and it helps detect bugs that lead to vulnerabilities, such as code injection or leaks of private data. Two approaches to the implementation of tainted data propagation over the program intermediate representation are proposed and compared. One of them is based on dataflow analysis (IFDS), and the other is based on symbolic execution. In this paper, the implementation of both approaches in the framework of the existing static analyzer infrastructure for detecting bugs in C# programs are described. These approaches are compared from the viewpoint of the scope of application, quality of results, performance, and resource requirements. Since both approaches use a common infrastructure for accessing information about the program and are implemented by the same team of developers, the results of the comparison are more significant and accurate than usual, and they can be used to select the best option in the context of the specific program and task. Our experiments show that it is possible to achieve the same completeness regardless of the chosen approach. The IFDS-based implementation has higher performance comparing with the symbolic execution for detectors with a small amount of tainted data sources. In the case of multiple detectors and a large number of sources, the scalability of the IFDS approach is worse than the scalability of the symbolic execution.

Keywords: taint analysis, static analysis, IFDS, symbolic execution

DOI: 10.1134/S036176881806004X

1. INTRODUCTION

At present, static analysis is increasingly used to find serious vulnerabilities in software, such as code injection or leaks of private data. In addition to the obvious advantages of the early detection of an issue, the static analysis indicates the exact points that are the sources and sinks of an error, and it also provides a sequence of diagnostic points containing the path of the issue propagation. Using testing, it is difficult to test all nontrivial scenarios of using the software that often contain vulnerabilities. By contrast, the detectors based on static analysis cover almost all possible paths of the program execution, which is a significant advantage, especially in what concerns security.

An effective method of static analysis of programs aimed at finding security issues is taint analysis. The warnings obtained by the analyzer can also be used by automatic verification tools based on dynamic analysis or for generating input data causing the vulnerability.

The taint analysis is based on the idea of propagating taints along all possible paths of "interesting" data in the program, beginning from a source (e.g., reading user data from a form) to a sink (e.g., execution of a SQL query). Thus, it is possible to find out how an intruder can inject his code or which confidential data can be compromised (in the example, this is SQL injection). Taint analysis can be implemented based on static and dynamic analysis. In addition to the advantages of using static analysis listed above, it is possible that the program author intentionally implemented the user data leak but the leak is hidden when the program recognizes that it is being executed within a dynamic analyzer. This problem is especially important for mobile applications, and it must be detected before the malicious program is deployed.

In this paper, we consider two approaches to the static taint analysis. The first approach is based on solving the IFDS-problem—this is called the tabulation algorithm [1]. This method analyzes dataflow by

reducing the analysis to solving the reachability problem on the interprocedural graph; the complexity of this problem is $O(ED^3)$ and only $O(ED)$ for locally separable problems, where E is the number of edges in the (nonextended) interprocedural control flow graph (CFG) and D is the number data analysis facts. This approach is implemented in the well-known tool FlowDroid [2, 3].

The other approach is based on the symbolic execution of the program. It performs a single traverse of the function exploded graph thus simulating the execution of each function occurring in the graph. The functions are traversed in topological order according to the call graph to ensure that the analysis of all callee is completed by the time the caller begins to be analyzed.

In dataflow analysis, in addition to the choice of the scheme for taint propagation, the selection of methods used for modeling the program environment (library functions), for analyzing virtual calls, for analyzing source codes in interface description languages (such as XAML) have a significant impact on the result. Both approaches examined in this paper use a unified analysis infrastructure, including the environment representation. For this reason the issues mentioned above are not considered in this paper.

The paper is organized as follows. In Section 2, we consider the overall structure of the SharpChecker analyzer that is required for the implementation of both methods. Sections 3 and 4 describe the operation model and implementation of taint analysis using IFDS and symbolic execution, respectively. In Section 5, we summarize the results obtained by both algorithms both on the special project WebGoat.NET, which models widespread vulnerabilities, and on a set of open source projects in C# In the last section, we make some generalizations and discuss the main directions of future research.

2. INFRASTRUCTURE OF SharpChecker ANALYZER

We will compare algorithms of taint analysis using SharpChecker [4, 5], which is the static analyzer for detecting bugs in C# programs. It can find more than 100 types of bugs using syntax analysis, dataflow analysis, and context- and path-sensitive symbolic execution. The overall schematic of the analyzer operation is shown in Fig. 1.

First, SharpChecker builds the static call graph of the program to be analyzed, which is then refined by analyzing virtual methods and calls through interfaces and delegates. The subsystem for the analysis of implicit calls makes it possible to consider all methods that can be called in a given instruction thus allowing detectors and other analysis subsystems to select a way of using this data. For example, the null dereferencing detector sequentially applies all summaries of candidates, while other detectors can heuristically select only one candidate based, e.g., on resource intensity of required operations. Since the majority of attacks are performed using user interfaces, which are in turn implemented through interfaces in C#, the quality of devirtualization algorithms is of major importance for taint analysis.

Symbolic execution is based on traversing the call graph in the direction from the callees to the callers. This makes it possible to reduce the interprocedural analysis to the intraprocedural analysis by constructing a summary of each method. The summary contains all the required data about the effects of calling the given function taking into account the calling context. This is achieved due to the preconditions calculated using an SMT solver if required.

During the intraprocedural analysis, the control flow graph for each method is constructed. The vertices of this graph are basic blocks (sequences of adjacent instructions). The instructions are the vertices of the abstract syntax tree, which has instruction semantics. The edges in the control flow graph are directed — they point to the basic block into which the control can be transferred after the current block completes executing. The entry and exit points of the method are represented with a special (empty) basic blocks. The exit block is connected to all basic blocks that can break the execution of the method body (e.g., blocks containing the return instruction or exception). Note that the control flow graph may contain loops; therefore, there may be an infinite number of different execution paths from the entry to the exit point of the method. To limit the number of paths to be considered, the acyclic exploded graph is constructed for the control flow graph. In this exploded graph, the transitions via backward edges are replaced by transitions into special components that copy certain parts of the control flow graph. Since the exploded graph is constructed for a fixed depth and since it does not contain backward edges, the maximal length and the total number of different paths from the entry to the exit points is limited.

The symbolic execution assumes that the formal parameters of the analyzed method, all fields and properties of the base class, and all static fields of other classes have symbolic values. For this reason, the formal parameters and the memory initial state at the entry point of the method are parameterized by a set of symbolic variables the types of which coincide with the types of the original fields and variables and no values of the formal parameters are specified. The instructions of the program are assigned new semantics that enables them to operate with symbolic expressions (formulas over symbolic variables and constants) rather than with concrete values. For example, if two integer symbolic variables $V1$ and $V2$ are added, then the result of the symbolic execution is the formula $V1 + V2$. Similar actions are performed on unary oper-

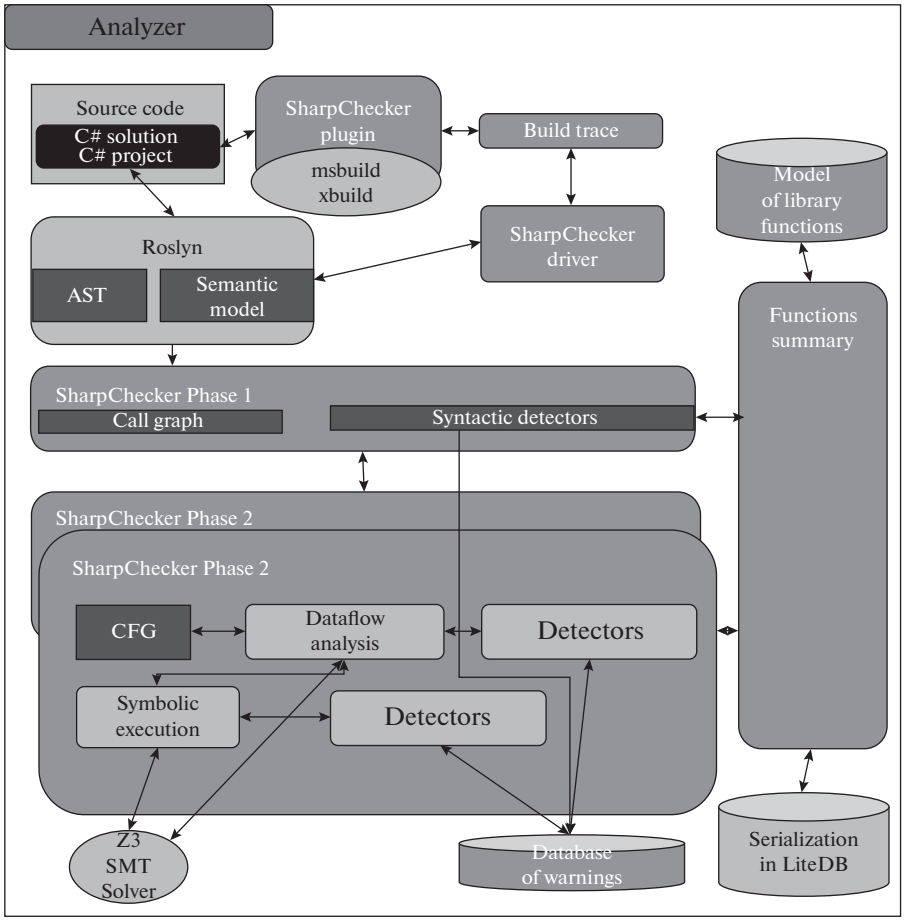

Fig. 1. Schematic of SharpChecker operation.

ators and conditional instructions. The values that are not constant and cannot be expressed in terms of the existing symbolic variables are modeled by introducing new symbolic variables. The method calls are modeled using the data stored in the callee summary. To reduce the number of paths to be analyzed, the state merging is also used. At the point where the paths in the control flow graph merge, a new analysis context is created, which is constructed by merging (using the rules specific for each type) the corresponding pairs of facts arriving on the incoming edges.

For modeling the environment, the analyzer implements a number of subsystems. The first subsystem is based on storing "interesting" properties of library functions, e.g., that the function creates a resource that must be later disposed or that the first parameter cannot be null. For taint analysis, this subsystem can be extended to enable it to support library transfer functions and sets of sources and sinks.

Another method for modeling the environment allows one to specify the model of library functions in C#. This is especially important for implementing collections that have an internal state that must be taken into account for a high quality analysis. This mechanism was elaborated to support the propagation of taints by calling library functions.

Thus, the existing infrastructure of SharpChecker makes it possible to model the environment, construct the internal representation of data, and collect data needed for the implementation of both approaches to taint analysis in a unified way.

3. TAINT ANALYSIS ALGORITHM BASED ON IFDS

This algorithm is based on propagating the information about tainted data through the basic blocks and edges of the interprocedural control flow graph. The taints can be propagated from sources to sinks and the other way around. The *forward* analysis begins from the sources, and the *backward* analysis begins from the sinks. The rules governing the propagation of tainted data are specified by *transfer functions*. The analysis of each source (sink in the backward analysis) is performed independently.

3.1. Taint Analysis as an IFDS Problem

Let us give the necessary definitions.

The *access path* is the list consisting of the initial object and fields. The initial object of the access path may be a literal, local variable, function parameter, return value of a method, reference `this`, static field of a class, expression (e.g., call of a method or operator), or a temporary initial object used in the implementation of transfer functions that does not represent any actual object of the program. Here are examples of access paths: `x`, `x.y.z`, `x.[...].w`, `this.f`, `a.f()`, `"password"`, `return`, `(a + b).p`.

The *length* of the nonempty access path is the number of fields in it plus one. Access paths are stored in the form of a prefix tree the root of which is the special empty access path {root} of length zero.

The *data analysis fact* is the information that the access path is tainted. The fact contains the tainted access path and the preceding fact used to restore the propagation path of tainted data.

Data analysis point is the information that the access path contained in the fact is tainted at the given point of the program. The data analysis point contains a fact and the basic block in which this fact is examined.

The analysis algorithm works with data analysis points, and the transfer functions work directly with data analysis facts and access paths. Different security defects have different relations between the number of possible sources and sinks; for this reason, both the forward (from source to sink) and backward (from sink to source) analysis were implemented to enable the implementation of defects detectors.

Each fact propagates independently of the others. This enables us to create summaries containing the analysis results of a method and use them if the entry point is encountered in the program once more. The summary of an entry point contains the output points obtained by the analysis, including data propagation paths. The use of such summaries significantly speeds up the analysis because there is no need to repeatedly examine not only the method for which the summary was created but also all methods called by it (including the transitively called methods).

The main function (`Solve`, Listing 1) processes the majority of instructions of the basic block; it is called for each entry point of the analysis (source points for the forward analysis and sink points for the backward analysis) and for each instruction within the method being analyzed. This function controls the point processing queue and the application of summaries. If a summary for an entry point exists, then the points from this summary are used and the result is returned. The application of a point is to replace the original input point fact from the summary by the fact of the current input point of `Solve` so as to restore the data propagation path. If no summary exists, then the entry point is added to the queue, and the main processing loop is started. In this loop, points are sequentially extracted from the queue, the function `SolveOnePoint`, which propagates the tainted data, is called for them, and the result is placed into the queue. The loop is executed until the queue becomes empty. The points that reached the method exit point are written to the result.

```
1: function Solve(point) → {Point}
2:   if ∃ a summary for point
3:     return Apply(summary points for point)
4:   worklist ← {point}, outpoints ← ∅
5:    while worklist ≠ ∅
6:      pt ← worklist.Get()
7:    if pt ∈ visited
8:     continue
9:    visited ← visited ∪ {pt}
10: worklist.PutAll(SolveOnePoint(pt))
11: if BasicBlock(pt) is exit block
12:    outpoints ← outpoints ∪ {pt}
13: store outpoints as a summary for point
14: return outpoints
```

Listing 1. Pseudocode of the function Solve for forward analysis.

The function `SolveOnePoint` propagates the tainted data using the transfer functions `NormalTF`, `CallTF`, `RetTF`, and `Call2RetTF`. In lines 10–12, the original algorithm is extended to support unbalanced returns, which allows us to start the analysis beginning from sources rather than from the method **main**, which can be unavailable; moreover, we can examine in the process of analysis only the methods containing tainted data. The return from a method is said to be balanced if the method was reached by a chain of calls during analysis. For the balanced return, the specific point of the program, which is the target of return from the function call, is known. When the analysis process returns into the method in which it started, the return from this method is unbalanced; therefore, all points at which this method is called must be examined to continue the analysis. Examples of balanced and unbalanced returns are shown in Fig. 2.

Here, Method 1 is the method from which the analysis begins, the solid arrows denote calls, the dot-and-dash arrows show balance returns, and the dotted arrows show unbalanced returns.

```
1: function SolveOnePoint(point) → {Point}
2:   points ← NormalTF(point)
3:   if last statement of BasicBlock(point) is a call
4:     points_old ← points, points ← ∅
5:     foreach p in points_old
```

```
6:      points_call ← CallTF(p, call)
7:      points_callee ← ∪_{p'∈pointscall-Solve(p', call)}
8:      points_ret ← ∪_{p''∈pointscallee}RetTF(p'', call)
9:      points ← points ∪ points_ret ∪ Call2RetTF(p, call)
10:   if BasicBlock(point) is exit block ∧ point wasn't created by processing a call
11:     foreach call site call of current function
12:       points ← points (∪_{p'∈RetTF(point, call)}Solve(p'))
13:   return points
```

Listing 2. Pseudocode of the function SolveOnePoint for forward analysis.

In the backward analysis, the control is often transferred to the middle of a function after the return from a call; for parts of such methods, summary is also built, and it can be repeatedly used to speed up the analysis. This is the most important distinctive feature of the backward analysis implementation.

The actual implementations of `Solve`, `SolveOnePoint`, and transfer functions contain extensions for the computation of not only resulting points but also diagnostic points — the data analysis points at which defects were detected.

3.2. Transfer Functions for Taint Analysis

The analysis uses four types of transfer functions — intraprocedural transfer function, call transfer function, return transfer function, and call to return transfer function. Each of them takes at its input a data analysis point and the required additional data and returns a set of data analysis points. The internal implementations of these functions operate on facts rather than on data analysis points.

To define transfer functions, we introduce the following notation: T is the set of tainted access paths; x, y, and z are access paths, $\diamond$ is a unary operator, $\circ$ is a binary operator; $x.[f]$ is an arbitrary access path obtained from the access path x by adding the list of fields $[f] = .f_1.f_2 \ldots f_p$, including a empty field; $\$"t_0\{a_1\}\ldots\{a_n\}t_n"$ is the string interpolation in C#, where t_i are constant string parts ($0 \le i \le n$) and a_i are arguments ($1 \le i \le n$).

Transfer functions for the forward and backward analysis are clearly different. However, to grasp the idea, it is sufficient to consider only the forward analysis.

Normal transfer function `NormalTF`

This function performs the intraprocedural propagation of tainted data within the basic block by sequentially processing its instructions in direct order. The output points are built based on the set T using the control flow graph to determine the next basic blocks.

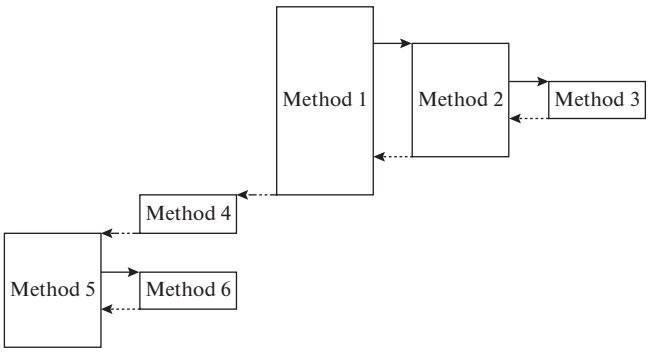

Fig. 2. Balanced and unbalanced returns.

$\texttt{var x = y}: T \to T \cup \{x.[f] : y.[f] \in T\};$

$\texttt{x = y}: T \to T \cup \{x.[f] : y.[f] \in T\} \setminus \{x.[f] : y.[f] \notin T\};$

$\texttt{◇x}: T \to T \cup \{(\diamond x).[f] : x.[f] \in T\};$

$\texttt{x ∘ y}: T \to T \cup \{(x \circ y).[f] : x.[f] \in T\};$

$\texttt{x ∘= y}: T \to T \cup \{x.[f] : y.[f] \in T\};$

$\$"t_0\{a_1\}\ldots\{a_n\}t_n": T \to T \cup (\cup\{\$"t_0\{a_1\}\ldots\{a_n\}t_n".[f] : \exists i; 1 \le i \le n \wedge a_i.[F] \in T\};$

$\texttt{return x}: T \cup \{\mathit{return}.[f] : x.[f] \in T\}.$

Call transfer function `CallTF`

This function propagates he tainted data from the current method to the method being called by matching the actual and formal parameters, including the implicit parameter *this*, and preserves the values of the static fields. Everything else is inaccessible in the called method.

Method declaration:

`ReturnType f(param1, param2, …, paramN).`

Method call: `a.f(arg1, arg2, …, argN).`

In the resulting points, the basic block *Entry* of the called method is indicated.

$$T \to \{\mathit{paramI}.[f] : \mathit{argI}.[f] \in T\} \cup \{\mathit{this}.[f] : a.[f] \in T\} \cup \{x.[f] : \mathit{IsStstic}(x) \wedge x.[f] \in T\}.$$

Return transfer function `RetTF`

This function intraprocedurally propagates the tainted data from the current method to the calling method by matching the actual and formal parameters, including the implicit parameter *this*, and preserves the values of the static fields. Everything else is inaccessible to the calling method. Note that the access paths of length one corresponding to formal parameters are not propagated because their changes in C# are local for the current method if no keyword

Table 1. Results produced by the IFDS analysis method

Defect type	Project	SNAP	WebGoat		OmniDB + Spartacus		netMQ	ShareX	Shadowsoc	OpenRA	Banshee	Cassandra	In total
LDAP INJECTION	TP	8											8
	FP												
SQL INJECTION	TP		15	(29)	35	(630)							50
	By-design				2	(36)							2
	FP												
COMMAND INJECTION	TP												
	FP				1								1
CONSTANT CREDENTIALS	TP				75		1	1					77
	Tests	1							6		1	10	18
	Initialization				4								4
	Empty password							3	1	12			16
	FP				1								1
INFORMATION_ EXPOSURE	TP												
	Test												
	FP							64					
RESOURCE INJECTION	TP		1										1
	FP				2								2

ref or **out** is specified. The return is into the basic block that follows the calling basic block.

$$T \to \{argI.x.[f] : paramI.x.[f] \in T\}$$
$$\cup \{argI : IsOutParameter(paramI)$$
$$\wedge\ paramI \in T\} \cup \{a.x.[f] : this.x.[f] \in T\}$$
$$\cup \{x.[f] : IsStatic(x) \wedge x.[f] \in T\}$$
$$\cup \{a.f(arg1, arg2, \ldots, argN).[f] : return.[f] \in T\}.$$

Call to return transfer function `Call2RetTF`:

This transfer function performs the intraprocedural propagation of taints from the caller basic block into the return block for the access paths that are inaccessible in the method being called. Such access paths are nonstatic fields, variables, and parameters that were not passed into the called function as arguments. As in `RetTF`, the return basic block is specified at the resulting points.

$$T \to \{x.[f] : \neg IsStatic(x) \wedge a \neq x \wedge !\exists I : argI$$
$$= x \wedge x.[f] \in T\} \cup \{x : \neg IsStatic(x)$$
$$\wedge\ (a = x \vee \exists I : argI = x) \wedge x \in T\}.$$

3.3. Testing Results

Testing of security issues in intensively used software is difficult because the majority of vulnerabilities are already eliminated, and the number of analyzer messages is very small. For this reason, it is convenient to compare approaches using a special information security project WebGoat.NET, which intentionally contains vulnerabilities. However, we also provide results obtained for other open source projects containing vulnerabilities.

Testing was performed on two identical computers with an Intel Core i7-6700 processor, 32 GB RAM under Windows 10 x64. The results are shown in Table 1.

The results produced by the analyzer were examined by hand to estimate the true positives ratio. It was found that some bugs were detected in tests and at variable initialization; certain warnings turned out to be false positives because they pointed at normal scenarios of the program usage.

4. TAINT ANALYSIS BASED ON SYMBOLIC EXECUTION

In distinction from the IFDS approach, in this method symbolic values are tainted. The sets of sources, sinks, and transfer functions can consist of

- method calls with tainted input and (or) output parameters;
- reads or writes of class fields;
- formal parameters of the method.

The basic algorithm can be described as follows. Let the program state during symbolic execution be specified by the current instruction I, current path predicate P, the set of symbolic values V, and the map-

ping of the set of variables onto the set of symbolic expressions.

We complete this state by the set of tainted values $T \in V$.

In the course of symbolic execution, the following checks are made for each state:

• if I belongs to the set of sources, then the symbolic values corresponding to the output parameters of this instruction are added to the set T (i.e., they become tainted);

• if I belongs to the set of transfer methods and at least one of its parameters is tainted, then all output parameters also become tainted;

• if I belongs to the set of sinks and at least one of its parameters is tainted, then the algorithm has detected an error in the program.

If symbolic execution with state merging is used, then during merging the set of tainted symbolic values is obtained by the union of two original sets: $T = T_1 \cup T_2$.

The proposed algorithm can ascertain the fact that the tainted data reached the sink; however, to use this algorithm in practice, e.g., to check the validity of a warning, the set of diagnostic points must be constructed that shows to the user the data flow path in the program from the source to the sink. To this end, each element in the set T should be associated with a data structure TInfo that accumulates and stores information about the point where the value was tainted and about the executed transfer methods. The taint propagation path is called trace. Note that this structure can store more than one trace because there are transfer methods that have more than one input parameters (e.g., the string concatenation operation stores the taint points of both its arguments); another situation is when symbolic states are merged (when the same variable was tainted using different paths from different sources).

In addition, the proposed algorithm is intraprocedural. To make it applicable for detecting interprocedural paths, method summaries (the data structure that stores information about the method properties revealed as a result of the intraprocedural analysis) can be used. Note that in this case the order of method analysis becomes important: the methods must be analyzed in reverse topological order (beginning from leaf vertices) in the call graph to provide the calling methods with the information about the properties of the called methods.

To the symbolic state of the program, we must add the set PT of potentially tainted values; this set is similar to the set T of tainted values defined above with the following exceptions:

• Initially, the symbolic values corresponding to the formal parameters of the examined method are added to the set PT, and the corresponding T Info stores information about the method entry point;

• instructions from the set of sinks do not add elements into PT;

• when a potentially tainted value reaches an instruction in the set of sinks, information about the fact that this method becomes an interprocedural sink, the method entry point, and the trace segment are added to the summary;

• similarly, at the exit of the method body, information about the fact that this method becomes an interprocedural source (for the values in T) and (or) a transfer method (for the values in PT), the output symbolic values, and the trace segment are added to the summary.

Interprocedural sources and sinks are processed in the same way as the ordinary ones except for the fact that the trace segment associated with such a source or sink is added to the intraprocedural trace thus forming the interprocedural propagation trace of the tainted data.

To detect different types of bugs, different taint categories must be used. For this reason, some of these categories must be analyzed individually, thus repeatedly analyzing the same propagation paths of tainted data (because the sets of transfer methods are typically identical). A way to resolve this problem is to introduce the concept of a tag. The tag is an identifier inherited from the source that is assigned to the entry point in the structure TInfo. Tags are transparent for the transfer methods, and they are checked when the sink is reached—if the tags of the tainted data and the sink are different, no actions are performed.

To fight the false positives that occur due to the existence of unreachable code in the program, the data structure storing the taint information also stores the path predicates, and the following rules for merging the states A with the path predicate P_A and B with the predicate P_B are used:

• if the value of the variable v is tainted in state A and not tainted in state B, then the value of v is considered tainted with the predicate P_A;

• if the value of the variable v is tainted in both states A and B and they have the common source, then the value of v is considered tainted with the predicate $(P_A \wedge P_B)$;

• if the value of the variable v is tainted in both states A and B but they have different sources, then both traces (each with its own predicate) are stored in TInfo.

When a tainted value with the predicate P_t reaches a sink, the formula corresponding to the predicate $P_t \wedge P$ is constructed in the state with the path predicate P, and this formula is passed to the SMT solver. If the solver decides that the formula is not satisfiable, then no warning is reported.

Table 2. Comparison of performance of the analysis methods

	WebGoat.NET	OmniDB	CodeContracts
IFDS	10 s	117 s	377 s
Symbolic execution	7 s	182 s	573 s
Symbolic execution without building the summary of transfer functions	7 s	129 s	512 s

5. TESTING RESULTS

Both methods were tested on identical machines on the same set of open source projects. The analysis methods under examination have only one common detector—finding SQL injection—and both detectors produce the same set of warnings on the entire set of tests. Thus, independently of the implementation, the analysis can be made fairly complete.

Testing the performance of the methods yields the following results.

These results correspond to running the analyzer with only one active detector SQL_INJECTION. The resulting set of warnings is the same for all runs. We conclude that Performances of the methods are comparable, but the method based on IFDS is faster if only one detector is active. In the case of symbolic execution, a larger amount of data needed for other detectors must be computed, and these computations cannot be skipped. In addition, this method computes information about possible propagation of taints for all functions even if this information is not needed for the further analysis. Testing showed that this information is irrelevant, and if its computation is skipped, the symbolic method becomes significantly faster. This is shown in the third row of Table 2.

The main differences in the performance of the compared approaches are detected for scalability. Testing showed that the significant increase of the number of sources has inhomogeneous effect. The number of sources increases when new detectors are added. For example, if the frequently used function ToString() is declared a source, then the execution time of the IFDS method on OmniDB is 278 s rather than 117 s (deceleration is by a factor of 2.4); for the other method, the execution time is 223 s rather than 182 s (deceleration is only by a factor of 1.22). Thus, in the development of new detectors one should take into account the number of sources of tainted data.

6. CONCLUSIONS

Two approaches to the implementation of taint analysis aimed at detecting errors in C# programs were studied; these approaches are based on the IFDS and on symbolic execution. Both methods were implemented within the unified infrastructure of the static analyzer SharpChecker and tested on the same set of projects on the same computers. The examination of results and performance showed that the comparable completeness of the analysis can be reached in both approaches. The method IFDS has significant advantages if there is a small amount of sources; however, its scalability is poor as the number of sources increases; however, this is admissible for the implementation of the majority of detectors.

The practice of developing detectors of security flaws based on taint analysis demonstrates the importance of the completeness of the set of sources, sinks, transfer functions, and the accuracy of modeling the environment (library functions). In addition, the availability of two methods significantly improves the quality and performance of detectors due to comparison of their results.

The further research will be aimed at the investigation of environment modeling methods, including the construction of the program operation model. This construction assumes that the call graph is augmented by edges that make it possible to simulate the user behavior. For the IFDS-based approach, the alias analysis can probably improve the completeness of results.

REFERENCES

1. T. Reps, T., Horwitz, S., and Sagiv, M., Precise interprocedural dataflow analysis via graph reachability, *Proc. of the 22nd ACM SIGPLANSIGACT Symposium on Principles of Programming Languages,* San Francisco, Calif., ACM, 1995, pp. 49–61. http://doi.acm.org/10.1145/199448.199462
2. Arzt, S. et al., FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps, *Proc. of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation,* Edinburgh, United Kingdom, ACM, 2014, pp. 259–269. http://doi.acm.org/10.1145/2594291.2594299
3. Fritz, C. et al., Highly precise taint analysis for Android applications, *Techn., Rep.,* No. TUD-CS-2013-0113, EC SPRIDE, 2013. http://www.bodden.de/pubs/TUD-CS-2013-0113.pdf.
4. Koshelev, V.K., Ignatyev, V.N., and Borzilov, A.I., C# static analysis framework, *Trudy Inst. Sist. Program. Ross. Akad. Nauk,* 2016, vol. 28, no. 1, pp. 21–40.
5. Koshelev, V., Dudina, I., Ignatyev, V., and Borzilov, A., Path-sensitive bug detection analysis of C# program illustrated by null pointer dereference, *Trudy Inst. Sist. Program. Ross. Akad. Nauk,* 2015, vol. 27, no. 5, pp. 59–86.

Translated by A. Klimontovich