University of California
Los Angeles

Improved Partial Instrumentation for Dynamic Taint Analysis in the JVM

A thesis submitted in partial satisfaction
of the requirements for the degree
Master of Science in Computer Science

by

Joseph Cox

2016

Abstract of the Thesis

Improved Partial Instrumentation for Dynamic Taint Analysis in the JVM

by

Joseph Cox

Master of Science in Computer Science

University of California, Los Angeles, 2016

Professor Jens Palsberg, Chair

Dynamic taint tracking is an important field of study with many Java-based tools and systems created to implement it, including Phosphor, a general purpose taint tracking tool designed for commodity JVMs like Oracle and OpenJDK. Phosphor works by instrumenting core Java libraries and the entire application bytecode with operations to accurately propagate taint information. Prior work strived to reduce the performance overhead of Phosphor by doing partial instrumentation. The analysis that determined which parts of the program to instrument was effective but flawed.

This paper aims to improve that analysis and further reduce the performance overhead by instrumenting less of the program. We use the Petablox program analysis tool and custom Datalog rules to perform an information flow analysis that better models Phosphor's behavior, including calls across native library boundaries. We find that we obtain a reduction in the amount of a program that needs to be instrumented by 79.9%.

The thesis of Joseph Cox is approved.

Miryung Kim

Todd D. Millstein

Jens Palsberg, Committee Chair

University of California, Los Angeles

2016

Table of Contents

List of Figures

List of Tables

Acknowledgments

I would like to profusely thank Dr. Jens Palsberg for the incredible patience, kindness, and support he has shown me during my time as a Masters student. Thank you for welcoming me to research as an undergraduate and allowing me to stay on as a Masters student and enjoy the ride. I have learned more about program analysis than I ever thought I would and more about life than I did in my entire undergraduate career.

I would also like to thank Christian, John, and Matt for being such wonderful lab mates and friends. Thank you for helping me understand the week's paper, the day's lecture, or whatever else I would ask about. Thank you for helping me navigate through grad school, being such patient teachers, and always fostering a great environment in lab. Scrubbadoosh!

CHAPTER 1

Introduction

By designing a static information flow analysis to accurately model the dynamic taint tracking performed by PHOSPHOR, a significant reduction in the instrumentation required by PHOSPHOR is achieved.

Dynamic taint tracking, also called dynamic information flow analysis, is the technique of automatically tracking the paths of certain pieces of data through a program during runtime. As such, information about the data can be discovered as it enters and exits the program, is written to and read from, and affects other parts of the program. Taint tracking is useful for both determining the origin of a piece of data as well as knowing information about its state during program execution. Data is tagged as it enters the program through user input or some other method, and tags are propagated as data is operated on or combined with other data. The method of tag propagation may vary but must be done in a way that the origins of a piece of data are discoverable by inspecting the tag.

Plenty of motivation exists for taint tracking, including the canonical example of detecting injection attacks such as SQL injections. Tracking data such as user input can prevent injection of potentially malicious data into critical functions or SQL command processing methods, which is the focus of some previous works [9] [11]. Many solutions and systems exist, all with various limitations. Some systems [8] [9] modify programs at the bytecode level but only track data flow of String-type objects. Another [7] can track data flow of all Java types but only targets an incomplete "research JVM" (read: not Oracle or OpenJDK). Most taint

tracking systems suffer from bad performance, soundness, precision, portability, or some combination of the four.

PHOSPHOR [3] is a taint tracking system for the Java Virtual Machine (JVM). It differs fundamentally from prior work in how it stores and propagates taint tags and is able to be performant, sound, precise, and portable. PHOSPHOR achieves taint tracking through instrumenting program bytecode. Unlike previous approaches, PHOSPHOR stores taint tags as shadow variables, one for each concrete variable in the program. Runtime operations cause modification and combination of taint tags. For example, the resulting taint tag of an arithmetic operation is the bitwise OR of the taint tags of the operands. Aside from its portability, soundness, and precision achievements, PHOSPHOR introduces, on average, a 53% performance overhead. PHOSPHOR's main mode of usage is to provide source and sink files, which enumerate the methods from which data of interest originates and methods into which we want to know if tainted data enters.

For simplicity and completeness, PHOSPHOR instruments the entire application bytecode and assumes that the instrumented program will be run on a JVM with fully instrumented Java libraries. Previous work [12], on which this paper's author was a participant, explores only partially instrumenting the application bytecode, doing so with a new tool called PHOSPHOR-PI. The key insight of this work is that that only a subset of the methods in the program callgraph, and methods that read or write shared data with those methods, need to be instrumented. Except for some special cases, methods that do not lie on any of the paths from any source to any sink are of no interest and can remain uninstrumented. This prior work involved producing PHOSPHORpi by modifying PHOSPHOR to support partial instrumentation and performing static analysis to generate the list of methods to be instrumented. This approach is able to save, on average, 11% of performance overhead compared to PHOSPHOR.

However, this prior work has certain limitations. First, it fails to consider

some program behaviors, such as calls into native libraries. Second, it is imprecise due to a number of special cases which result in more instrumented code. In this paper, we describe a different taint analysis for partial instrumentation that more completely and accurately models the taint tracking done by PHOSPHOR, including calls to native libraries. Our approach also further reduces instrumentation. While PHOSPHOR and PHOSPHOR-PI store taint tags as 32-bit integers, we reduce them to boolean properties, which allows us to remove instrumentation on certain methods in which the taintedness of data is unchanged throughout the method. The analysis described in this paper is designed to integrate with further modifications made to PHOSPHOR by its original author.

The remainder of the thesis is as follows. Chapter 2 presents background information on relevant tools and prior work. Chapter 3 presents the high-level approach taken to achieve partial instrumentation. Chapter 4 describes the implementation details behind achieving partial instrumentation. Chapter 5 shares results of the implemented approach. Chapter 6 concludes the thesis.

CHAPTER 2

Background and Prior Work

This paper builds on a number of existing tools and prior work done by the author and others. In this chapter we describe the existing tools in greater detail and how they were modified or supplemented to support the analysis in the paper. First, we expand on the technical details of Phosphor. Second, we describe details of Petablox, including its architecture and Datalog fact engine. We also describe prior work on partial instrumentation in greater detail to help elucidate the differences between the analysis in that prior work and the analysis in this paper.

2.1 Phosphor

Phosphor [3] is a general purpose taint tracking tool that is capable of performing fast, portable, sound, and precise data flow tracking for the JVM. Phosphor is able to be far more portable than previous methods because it does not require program source code. Instead, it only requires program bytecode and achieves taint tracking by instrumenting that bytecode. Under the hood, Phosphor instruments bytecode using the ASM byte code manipulation library [6]. Phosphor achieves soundness by considering every possible opcode to appear in the bytecode and adding operations to support taint propagation for each opcode. Specific behavior for each opcode is detailed at the end of [3].

Phosphor is meant to be used to track data from user-specified source meth-

ods to user-specified sink methods. A user can specify files detailing source and sink methods as inputs to the PHOSPHOR program. The format of a method description inside those files is:

```
<java.io.InputStream:  int read(byte[],int,int)>
```

The format includes, sequentially, the enclosing class, return type, name, and argument types. All types are fully qualified with the package name. This is the same method description format for all of the analyses presented in this paper.

PHOSPHOR tracks data flow by adding instrumentation to propagate shadow variables representing taint tags for every local variable, argument, field, class instance, array element, and more, in the program. Taint tags are represented as 32-bit integers, so in theory, there are up to 2^{32} possible taint tags. For ease of tag propagation, PHOSPHOR considers there to be 32 possible base taint tags that can be assigned to data entering the program. PHOSPHOR taints data originating from source methods with a different taint tag for each source method. Therefore, there can be up to 32 enumerated source methods for each to have a unique tag. As data moves through the program, taint tags are often modified and combined to accurately show the origin of the data. For example, if a variable x originates from method `foo` and has taint tag `0x00000001`, variable y originates from `bar` and has taint tag `0x00000002`, and $z = x + y$;, the taint tags of x and y are bitwise-ORed, resulting in a tag of `0x00000003`. A simple inspection of the taint tag of z shows that its value originated from both `bar` and `foo`.

PHOSPHOR also tracks data across boundaries into native libraries with a few special cases. For `System.arraycopy`, the taint array of the source array is simply copied as a taint array for the destination array. For other native calls with return values, a bitwise OR of all the taint tags of the arguments is done to generate the taint tag of the return value.

A fully instrumented program requires fully instrumented Java libraries to run

correctly. This contributes to a downside to PHOSPHOR, which is the runtime performance overhead of, on average, 53%. It is hard to know without a study, but it is possible that this is still too high an overhead for widespread adoption in systems.

2.2 Petablox

Petablox [2] is a large-scale program analysis tool developed at Georgia Tech. Petablox is capable of running analyses such as deadlock analysis, datarace analysis, and k-cfa object sensitive and insensitive analyses on very large codebases.

Petablox uses Datalog [10] to generate program analysis facts from other program facts. Datalog is a logic programming language and a strict subset of Prolog. Statements in Datalog take on the form `x(t) :- y(t), z(t).`, where the conjunction of the predicates on the right hand side implies the left hand side.

Internally, Petablox uses the Soot bytecode optimization framework [13] to obtain a high level representation (namely Jimple [14]) of the program bytecode. Petablox then translates that representation into Datalog program facts. Petablox maintains a number of program entity domains, including domains of methods, local variables, fields, and more. These domains are populated with all of the relevant entities generated by Soot. Domains are typically labeled with a single letter in Petablox. For example, domain M represents the domain of all methods in the program.

Petablox also computes a number of relations on top of the domains. These relations come in two basic types:

- Relations which add a level of specificity to a domain and create a subset of the domain. An example would be a relation consisting of all fields that are declared final.

- Relations which relate two or more different domains.

The second kind of relation is the most common. An example would be the `MmethArg` relation, which relates methods, the local variables which represent the arguments to the method, and the position of those variables in the argument list. This relation allows one to query if a local variable is the same variable as a positional argument. The components of the relation are restricted to certain predefined domains. The method must be in the method domain, the local variable must be in the domain of local variables, and the position must appear in the domain of integer argument positions. This gives Petablox relations a certain aspect of typing and prevents the addition of any facts that do not have the correct type or do not appear in the associated domain. The existence of facts in domains cannot be modified by relations, as domains are populated once and are constant thereafter.

Some relations are defined using Java and some are defined directly in Datalog. The relations defined in Java are typically those that are easier to create with access to Soot types and utilities. Relations defined directly in Datalog are typically those that are logically abstract and are components of an analysis. Predicates defined in Java are computed simply by running the definition code after the program domains have been populated. However, computing predicates defined in Datalog requires a Datalog engine to run the Datalog code. Petablox offers compatibility with two such engines: bddbddb [15] and the proprietary Logicblox database [1]. Logicblox runs on a modified form of Datalog called LogiQL, so Petablox includes tools to convert Datalog code to LogiQL code for use with Logicblox.

Unlike other program analysis tools, the scope of what Petablox analyzes is tightly related to the reachability of the analyzed program. Petablox will compute method reachability, either statically or dynamically based on configuration, and only analyze program entities residing within reachable methods. This makes

many analyses much easier because they often are only concerned with behavior of reachable program points, and Petablox gives this for free.

Originally, Petablox used a different Datalog schema to represent programs. Petablox used the Doop pointer analysis framework [5], which came with its own set of Datalog predicates to represent a Java program. During the development effort for this paper, Petablox switched to the new Datalog schema which has been discussed up to this point. This posed a significant hurdle, given that the two schema were built upon completely different architectures. Doop did not have any concept of domains and did not consider reachability, instead including all entities in a program.

However, the new schema certainly has limitations compared to Doop. First, the new schema does not include primitive-typed variables in the domain of local variables. As such, none of the relations using local variables include any primitives. There is also no domain of constant values, so those program entities are not represented. The new schema also does not have a relation listing native methods in the program. Instead, they choose a select few native methods and simulate the bodies of those methods so that analyses can model the behavior of calls to native code. However, all other native methods are anonymized and simply become stubbed methods without any concrete body.

2.3 Phosphor-PI

A key insight of the previous work with partial instrumentation, Phosphor-PI [12], is that some methods of the target program do not have to be instrumented if they do not interact with any data of interest. The data of interest is data that is originally returned from sink methods and eventually enters source methods. There is some subset of all the methods in the program that must be instrumented. To discover this subset, Phosphor-PI first looks at the methods which reside on

the callgraph from source to sink and then add some special cases.

The analysis for PHOSPHOR-PI begins with a callgraph in which nodes represent methods and edges represent calls from callers to callees. The callgraph is generated by Petablox. The analysis marks nodes that represent source and sink methods with a special label. Then, it generates the forward callgraph from the source methods. That is, it first marks nodes that call the source nodes, then recursively marks all the nodes that are callees of other marked nodes. Then, it generates the backwards callgraph to the sink methods. That is, it starts with a fresh graph, marks the sink nodes, and then recursively marks nodes that are callers of marked nodes. After computing the intersection of these two sets, PHOSPHOR-PI has all the methods that reside on all the possible paths from any of the source methods to any of the sink methods.

There are a number of special cases in which extra methods have to be added to the set of methods to be instrumented, for reasons related to analysis correctness and bytecode correctness. As an example of analysis correctness, some methods not in the intersection set may store data originating from a source method into an instance field. The taint tag of the field must be updated else the tag will be incorrect when the data reaches a sink. PHOSPHOR-PI solves this case very imprecisely by adding any method which stores to an instance field to the instrumentation set. The entire method gets instrumented by PHOSPHOR and the field's taint tag is updated correctly.

Another type of special case is concerned with bytecode correctness. As an example of such as special case, PHOSPHOR boxes certain types, including multidimensional arrays, with a new boxed type that contains both the underlying values and taint tags. For example, the type `int[][]` becomes `TaintedIntArr[]`, where each `TaintedIntArr` contains both the integer array value and its associated array of taint tags. In cases where instrumented code calls uninstrumented code, the uninstrumented code may have a multidimensional array as a return

type. The instrumented caller code will expect a boxed value as a return but the uninstrumented callee code will return an unboxed value. To solve this, the uninstrumented callee is simply added to the intersection set and instrumented, which changes its return type to the boxed type. There are additional special cases of both kinds.

This prior work did have a number of limitations. First, it did not consider calls into native libraries as part of its analysis. Second, the special cases tended to add a large number of extra methods and could have been done more precisely. The approach to be presented in this paper attempts to solve these issues.

CHAPTER 3

Approach and Implementation

In this chapter we describe the approach taken for our analysis. We show how the analysis propagates taint through the program and how this more accurately models the behavior of PHOSPHOR than previous work. The analysis is constructed as a series of Datalog rules. Each rule is implied by a logical conjunction of one or more rules. The rules are composed of base program facts generated by Petablox, which are facts that describe the structure of the program, such as the type of a local variable. We begin with the facts that describe taintedness in local variables and fields. We then describe facts that are concerned with how taint is propagated through native library calls. We then describe facts that determine which methods need to be tracked. We finally describe facts about which call sites of tracked methods will call instrumented versions of the callee and which will call uninstrumented versions. We also detail additions and modifications to Petablox that were required to enable the analysis.

To elucidate the how the Datalog rules apply to a Java program and how data flow facts are deduced, a code example is provided in section 3.4, along with a program fact base, relevant rules from the following sections, and a solution to the rules.

Underscore characters in the Datalog rules represent arguments which may resolve to any entity in the relevant entity domain. For example, an underscore character given as an argument for a type domain can resolve to `int`, `java.lang.String`, or any other type.

3.1 Tainted and Untainted Objects

3.1.1 Taint Predicates

We begin by describing the predicates that represent tainted and untainted values. `taintedX` and `untaintedX` represent tainted and untainted local variables, respectively. The local variables can be of primitive or reference type. If a local variable appears in `taintedX`, then the variable contained tainted data at some point. The same is true of a variable appearing in `untaintedX`. We are not interested in whether a variable *ever* contains tainted or untainted data, but rather if it *always* contains tainted or untainted data. If the latter is true, then we do not have to track the data through a method, since we know that it will be tainted at the end if it is tainted at the beginning.

For this insight to work, we must make a fundamental simplification of how taint is tagged and tracked through the program. Originally, PHOSPHOR maintained a 32-bit taint tag which described exactly how a variable had been tainted and the origin of its taint. If we do not plan to track taint propagation through some methods in which a variable is always tainted or always untainted, we must simplify the taint tag to be only a boolean value. The reason for this is that a variable may be always tainted in a method, but the exact details of its taint may change through that method. The 32-bit taint tag may change from one non-zero taint tag to another. Without tracking taint propagation through that method, we are not able to track that change. The most we can know is if a variable is tainted at the beginning of a method and it is always tainted during the method, it will be tainted at the end of the method.

The predicates `taintedF` and `untaintedF` represent tainted and untainted fields in the same manner as local variables.

3.1.2 TaintSrc and TaintSink Predicates

The predicates `taintSrc` and `taintSink` represent the methods that appear in the source and sink files passed to PHOSPHOR. These are the methods that PHOSPHOR tracks data from and to.

The predicates are implemented in Java as an addition the Petablox tool. The source and sink files can be passed to Petablox with a new configuration option and are read and filled into the `taintSrc` and `taintSink` predicates.

3.1.3 Assignment Rules

The following are the Datalog rules to handle propagation of taint by way of assignment:

$$\texttt{taintedX(x)} \text{ :- } \texttt{taintedX(y)},\ \texttt{MXVarAsgnInst(\_, x, y)}. \tag{3.1}$$

$$\texttt{untaintedX(x)} \text{ :- } \texttt{untaintedX(y)},\ \texttt{MXVarAsgnInst(\_, x, y)}. \tag{3.2}$$

$$\texttt{taintedF(f)} \text{ :- } \texttt{MputFldInstX(f, x)},\ \texttt{taintedX(x)}. \tag{3.3}$$

$$\texttt{untaintedF(f)} \text{ :- } \texttt{MputFldInstX(f, x)},\ \texttt{untaintedX(x)}. \tag{3.4}$$

$$\texttt{taintedX(x)} \text{ :- } \texttt{MgetFldInstX(x, f)},\ \texttt{taintedF(f)}. \tag{3.5}$$

$$\texttt{untaintedX(x)} \text{ :- } \texttt{MgetFldInstX(x, f)},\ \texttt{untaintedF(f)}. \tag{3.6}$$

Figure 3.1: Rules for tainting assignment statements

The `MXVarAsgnInst` predicate is part of Petablox's program representation schema. `MXVarAsgnInst(m,x,y)` represents a statement of the form $x = y$ within the scope of method m. Rules 3.1 and 3.2 represent the propagation of taint from a local variable into another by way of assignment. Rule 3.1 states that if the right hand side is tainted, the left hand side will be tainted as well. Rule 3.2 states the same for untainted propagation.

Rules 3.3 and 3.4 represent taint propagation of local variables being assigned to fields. The `MputFldInstX` predicate is an abstraction over both instance and static field stores and represents statements of either the form $b.x = y$ or $ClassName.x = y$. Rule 3.3 states that if the right hand side is tainted, the field will be tainted as well. Rule 3.4 states the same but for untainted propagation.

Rules 3.5 and 3.6 represent taint propagation of fields being assigned to local variables. The `MgetFldInstX` predicate is an abstraction over instance and static field loads, similar to `MputFldInstX` from before, and represents statements of either the form $x = b.y$ or $x = ClassName.y$. Rule 3.5 states that if the field is tainted, the local variable will also be tainted. Rule 3.6 states the same but for untainted propagation.

Statements of the form $b.x = c.y$ where a field is loaded and then stored into a field do not need to be explicitly modeled, since under the covers the field is first loaded into a local variable, which is handled by Rules 3.5 and 3.6, and then the local variable is stored into the left hand side field, which is handled by Rules 3.3 and 3.4.

3.1.4 Return Rules

The following are the Datalog rules to handle propagation of taint through returns.

```
taintedX(x) :- IinvkXRet(i,z,x), IM(i,m), MmethXRet(m,z,y), taintedX(y).   (3.7)

untaintedX(x) :- IinvkXRet(i,z,x), IM(i,m), MmethXRet(m,z,y),
                 untaintedX(y), !taintSrc(m).                              (3.8)

taintedX(x) :- taintSrc(m), IinvkXRet(i,_,x), IM(i,m).                     (3.9)
```

Figure 3.2: Rules for tainting returns

Rules 3.7 and 3.8 model the taint propagation of the return of local variables in callee methods into local variables of caller methods.

The `IinvkXRet` predicate relates invocation sites to local variables the invocation returns into. `IinvkXRet(i,z,x)` means that the zth return variable of call site i is stored into local variable x. The `IM` relation relates invocations to their resolved methods. `IM(i,m)` means that method m is the resolved method of call site i. The `MmethXRet` predicate relates methods and their return variables. `MmethXRet(m,z,x)` means that local variable x is the zth return of method m.

Rule 3.7 states that when the zth return variable of the method m is tainted, the local variable x receiving the zth return of invocation site i will be tainted.

Rule 3.8 states the same as 3.7 but for the propagation of untainted data, with one exception. If m is a source method, then the return variable should be tainted regardless of the content of m. Therefore we add an extra check to Rule 3.8 that m is not a source method.

Rule 3.9 represents the flow of tainted data from source methods into local variables. It states that if method m is a source method, and invocation site i resolves to m, then any local variable being assigned a return value from m is tainted.

3.1.5 Argument Rules

The following are the Datalog rules to handle propagation of taint through variables passed as arguments.

$$\texttt{taintedX(x)} \texttt{ :- } \texttt{MmethXArg(m,z,x)},\ \texttt{IM(i,m)},\ \texttt{IinvkXArg(i,z,y)},\ \texttt{taintedX(y)}. \tag{3.10}$$

$$\texttt{untaintedX(x)} \texttt{ :- } \texttt{MmethXArg(m,z,x)},\ \texttt{IM(i,m)},\ \texttt{IinvkXArg(i,z,y)},\ \texttt{untaintedX(y)}. \tag{3.11}$$

Figure 3.3: Rules for tainting arguments

Rules 3.10 and 3.11 model the taint propagation of arguments passed into invocation sites to the local variables representing the parameters of a method.

The `MmethXArg` predicate relates methods and their parameters. `MmethXArg(m,z,x)` means that local variable x is the zth parameter of method m. The `IinvkXArg` predicate relates local variable arguments and method invocation sites. `IinvkXArg(i,z,x)` means that local variable x is the zth argument passed into the method represented by invocation i at invocation site i.

Rule 3.10 states that when the zth argument is tainted, the zth parameter of method m will be tainted. Rule 3.11 states the same but for the propagation of untainted data.

3.1.6 Constants and New Object Rules

Constants and new object allocation statements are untainted by definition, given that they did not originate from source methods. The following are the Datalog rules to ensure that local variables and fields that have been assigned constant values and newly allocated objects are marked as untainted.

```
untaintedX(x) :- MXAsgnConst(_, x, _).                                  (3.12)

untaintedF(f) :- MputFldInstConst(f, _).                                (3.13)

untaintedX(x) :- XV(x, v), MobjValAsgnInst(_, v, _).                    (3.14)

untaintedX(x) :- MmethXArg(m, z, x), IM(i, m), IinvkConstArg(i, z, _).  (3.15)
```

Figure 3.4: Rules for constants and allocations

Rule 3.12 marks local variables on the left hand sides of statements like $x = 5$ as untainted. The `MXAsgnConst` predicate represents those statements. `MXAsgnConst(m,x,c)` means that constant c is assigned to local variable x in method m. We want these local variables to be marked as untainted regardless of the constant or method, so we use underscores as the method and constant arguments.

Rule 3.13 does the same as 3.12 but for fields. The `MputFldInstConst` predicate models stores of constants into fields, such as statements of the form $b.x = 5$. Again we mark the field f as untainted regardless of the constant.

Rule 3.14 marks local variables that are assigned objects from object allocation statements such as the form $x = newA()$ as untainted. The `MobjValAsgnInst` predicate represents such object allocation statements. `MobjValAsgnInst(m,v,h)` means that object allocation statement h assigns to local variable v inside method m. In this predicate, the local variable belongs to a different entity domain that previous local variables. Petablox includes a domain V which represents all local variables of reference type. Because all object allocation statements create objects of reference type, the predicate uses domain V. To add the correct local variable in the domain of all local variables to the `untaintedX` predicate, we use the `XV` predicate, which relates identical objects between the reference type variable and all variable domains.

Rule 3.15 marks local variable parameters of methods as untainted if a constant was passed into the invocation. The predicate `IinvkConstArg` represents any constant passed into a method invocation.

No rule is required to model the passing of an object allocation statement into a method invocation. The object allocation is first placed into a local variable, which is modeled by Rule 3.14, and then passes the local variable into the invocation, which is modeled by Rules 3.10 and 3.11.

3.2 Native Code

The following are the Datalog rules to model taint propagation of taint across native call boundaries.

$$\texttt{taintedX(x)} \texttt{ :- } \texttt{MmethXRet(m, z, x), nativeM(m), IM(i, m),} \\ \texttt{IinvkXArg(i, z, y), taintedX(y).} \tag{3.16}$$

$$\texttt{untaintedX(x)} \texttt{ :- } \texttt{MmethXRet(m, \_, x), nativeM(m), !taintedX(x).} \tag{3.17}$$

Figure 3.5: Rules for tainting across native calls

Rules 3.16 and 3.17 represent the taint propagation across native call boundaries for native calls which return a value. The `nativeM` predicate is a subset of the method domain and represents all methods that have the native modifier. Rule 3.16 states that the return of a native method is tainted if there exists any argument passed to the method that is tainted. Rule 3.17 states that any returns from native methods that have not been marked tainted are untainted. This captures the situations where none of the arguments to a native method are tainted.

$$\texttt{taintedX}(\texttt{x}) \texttt{ :- } \texttt{arraycopyM}(\texttt{m}),\ \texttt{IM}(\texttt{i},\texttt{m}),\ \texttt{IinvkXArg}(\texttt{i},1,\texttt{y}),\ \texttt{IinvkXArg}(\texttt{i},3,\texttt{x}),\ \texttt{taintedX}(\texttt{y}). \tag{3.18}$$

$$\texttt{untaintedX}(\texttt{x}) \texttt{ :- } \texttt{arraycopyM}(\texttt{m}),\ \texttt{IM}(\texttt{i},\texttt{m}),\ \texttt{IinvkXArg}(\texttt{i},1,\texttt{y}),\ \texttt{IinvkXArg}(\texttt{i},3,\texttt{x}),\ \texttt{untaintedX}(\texttt{y}). \tag{3.19}$$

Figure 3.6: Rules for tainting across `System.arraycopy`

Rules 3.18 and 3.19 represent the taint propagation across native call boundaries specifically for `System.arraycopy`. This is a void native method that performs a shallow copy of its `src` argument into the object provided as the `dest` argument. The `arraycopyM` predicate represents only one method, `System.arraycopy`. Rule 3.18 states that if the argument at index 1 (the `this` object is at index 0), which is the source array, is tainted, then the argument at index 3, which is the destination array, is also tainted. Rule 3.19 states the same but for propagation of untainted data.

3.3 Tracked and Untracked Methods

The following are the Datalog rules that govern which methods are tracked and which are untracked.

$$\texttt{tracked}(\texttt{m}) \texttt{ :- } \texttt{MM}(\texttt{m},\texttt{n}),\ \texttt{taintSink}(\texttt{n}). \tag{3.20}$$

$$\texttt{tracked}(\texttt{m}) \texttt{ :- } \texttt{MX}(\texttt{m},\texttt{x}),\ \texttt{taintedX}(\texttt{x}),\ \texttt{untaintedX}(\texttt{x}). \tag{3.21}$$

$$\texttt{untracked}(\texttt{m}) \texttt{ :- } !\texttt{tracked}(\texttt{m}). \tag{3.22}$$

Figure 3.7: Rules for tracking methods

Rules 3.20 and 3.21 model which methods should be tracked and Rule 3.22 models which methods should be untracked. Rule 3.20 states that any method m which calls a method n should be tracked if n is a sink method. Rule 3.21 states that if there is any local variable in method m which is both tainted and untainted at some point in m, then m should be tracked. Rule 3.22 states that all methods not tracked should be in the `untracked` predicate.

3.4 Code Example

```
public class TaintTest {
  public int f;
  public static void main(String[] args) {
    TaintTest t = new TaintTest();
    t.run();
  }

  public void run() {
    String s = taintedString();
    foo(s);
  }

  public void foo(String s) {
    System.out.println(s);
  }

  public String taintedString() {
    return new String();
  }

  public void unreachable() {
    String s = taintedString();
    System.out.println(s);
  }
}
```

Figure 3.8: Java code example

The Java program in 3.8 demonstrates how some of the aforementioned Datalog rules apply to compute tainted data and tracked methods. We will show the domain of methods filled by Petablox, a relevant set of program facts generated by Petablox, and relevant facts deduced by our rules.

```
"<java.io.PrintStream: void println(java.lang.String)>"
"<TaintTest: java.lang.String taintedString()>"
"<TaintTest: void main(java.lang.String[])>"
"<TaintTest: void <init>()>"
"<TaintTest: void run()>"
"<TaintTest: void foo(java.lang.String)>"
```

Figure 3.9: Domain of methods for code in fig. 3.8

Figure 3.9 shows the domain of methods created by Petablox. This domain contains all the methods reachable from the program's main method. As such, the `unreachable` method is absent from this domain and will not appear in any base program facts or deduced facts. Petablox can be configured to compute reachability either statically or dynamically.

```
taintSink("<java.io.PrintStream: void println(java.lang.String)>").
taintSrc("<TaintTest: java.lang.String taintedString()>").

IM("!0<TaintTest: void run()>",
   "<TaintTest: java.lang.String taintedString()>").
IM("!1<TaintTest: void run()>",
   "<TaintTest: void foo(java.lang.String)>").

IinvkXRet("!0<TaintTest: void run()>", 0,
          "java.lang.String s in <TaintTest: void run()>").
IinvkXArg("!1<TaintTest: void run()>", 1,
          "java.lang.String s in <TaintTest: void run()>").

MmethXArg("<TaintTest: void foo(java.lang.String)>", 1,
          "java.lang.String s in <TaintTest: void
             foo(java.lang.String)>").

MM("<TaintTest: void foo(java.lang.String()>",
   "<java.io.PrintStream: void println(java.lang.String)>").
```

Figure 3.10: Relevant program fact base for code in fig. 3.8

Figure 3.10 shows relevant program facts for the code in figure 3.8. The `taintSink` and `taintSrc` facts show that specific methods are sink and source methods, respectively. The `IM` facts show that the 0th and 1st call sites in the `run` method resolve to the methods `taintedString` and `foo`, respectively. The `IinvkXRet` and `IinvkXArg` facts show that the return variable of the 0th call site and the argument passed to the 1st call site, respectively, are both the same local variable `s` in the `run` method. The `MmethXArg` fact shows that the parameter of the `foo` method is the local variable in `foo` named `s`. Finally, the `MM` fact shows that the `foo` method calls the `println` method.

Now we will show certain computed facts and explain how they were computed.

```
taintedX("java.lang.String s in <TaintTest: void run()>").
taintedX("java.lang.String s in <TaintTest: void foo()>").
tracked("<TaintTest: void foo(java.lang.String)>").
untracked("<TaintTest: void run()>").
```

Figure 3.11: Sample of computed facts for code in fig. 3.8

The first computed fact states that the local variable named `s` in the `run` method is tainted. This results from applying Rule 3.9. The `taintSrc`, `IinvkXRet`, and `IM` facts from figure 3.10 imply that the variable is tainted. The second computed fact is computed similarly, applying Rule 3.10.

The `tracked` fact is computed by applying Rule 3.20. The method `foo` is tracked because it calls a sink method, namely `println`. The method `run` is untracked due to the application of Rule 3.22. Rule 3.21 does not apply to `run` because its only local variable is only tainted and never untainted.

3.5 Instrumented and Uninstrumented Call Sites

In addition to considering some methods to be tracked and some untracked, sometimes invocations to tracked methods do not pass in any tainted data and do not need to call instrumented code. As such, we can maintain an instrumented and an uninstrumented version of each tracked method. When no arguments passed to an invocation are tainted (that is, all of the arguments belong only to the `taintedX` predicate and not the `untaintedX` predicate), the called method can be the uninstrumented version.

The following are the Datalog rules that determine which method call sites will call the instrumented version of a method and which will call the uninstrumented version.

$$\texttt{inst(m,n)} \texttt{ :- } \texttt{MI(m,i), IM(i,n), IinvkXArg(i, \_, x), taintedX(x), tracked(n).} \tag{3.23}$$

$$\begin{aligned}\texttt{inst(m,n)} \texttt{ :- } &\texttt{MgetInstFldInstX(n, \_, \_, f), taintedF(f),} \\ &\texttt{MI(m,i), IM(i,n), tracked(n).}\end{aligned} \tag{3.24}$$

$$\begin{aligned}\texttt{inst(m,n)} \texttt{ :- } &\texttt{MgetStatFldInstX(n, \_, f), taintedF(f),} \\ &\texttt{MI(m,i), IM(i,n), tracked(n).}\end{aligned} \tag{3.25}$$

$$\texttt{uninst(m,n)} \texttt{ :- } \texttt{!inst(m,n).} \tag{3.26}$$

Figure 3.12: Rules for instrumenting call sites

Rule 3.23 states that the invocation entity i corresponding to the call of method n inside method m will be the instrumented version if any of the arguments passed to the invocation are tainted at any point. Rule 3.24 states that the call site of n in m is instrumented if n loads from a tainted instance field. Rule 3.25 states the same as 3.24 but for static fields.

Rule 3.26 states that any invocation not instrumented should be uninstrumented.

3.6 Additions to Petablox

Many modifications and additions to Petablox were necessary to enable the above analysis. These modifications came in the form of new entity domains and new relations, both written in Java. Some new relations were created by using an existing relation as a template and modifying necessary types.

The first large addition was a new domain for local variables that included primitive typed locals. The existing domain for local variables, domain V, only included reference type local variables. We decided to add a domain that included

primitive local variables instead of modifying domain V to include primitives due to issues with backwards compatibility. In addition to adding this new entity domain, many existing relations had to be copied and modified to use the new domain instead of the old reference-only domain. Many of the predicates in the above rules that contain the character X as part of the name are examples of these new relations.

Also, Petablox previously handled native methods in an unexpected way. Petablox first removed the native modifier from all native methods and turned them into abstract methods without bodies, indistinguishable from other abstract methods. Then Petablox took a few specifically enumerated native methods, including `System.arraycopy` and added fake bodies to simulate the behavior of these methods. PHOSPHOR did not model taint propagation across native call boundaries in the same way the simulated bodies created by Petablox did. Therefore we had to modify how Petablox looked at native methods to accurately model PHOSPHOR's behavior. Petablox was made to not remove the native modifier from native methods and native methods were added to a new `nativeM` predicate.

3.7 Modifications to PHOSPHOR

Modifications to PHOSPHOR were required to allow it to make use of the information generated by the analysis in this paper. We coordinated with the original author of PHOSPHOR to implement these modifications. A feedback loop was established where the original author would make changes to PHOSPHOR and we would generate analysis results or modify the analysis if necessary and send those results back. The details of modifications made to PHOSPHOR are not described in this paper.

CHAPTER 4

Results

We evaluate our approach by comparing the number of instrumented methods produced by the analysis of PHOSPHOR-PI to the number of tracked methods produced by the analysis in this paper.

4.1 Benchmark set

While PHOSPHOR was evaluated on performance using all 14 benchmarks of the DaCapo benchmark suite [4] included in the 9.12-bach release, PHOSPHOR-PI was evaluated on performance using only 7 of the 14 benchmarks: avrora, batik, h2, pmd, sunflow, tomcat, and xalan, due to numerous problems with bytecode verification for the remaining 7. For PHOSPHOR-PI, lists of tracked methods were generated for all 14 of the benchmarks except for tradesoap, so we can compare tracked method counts between that project and this paper. The lists of tracked methods produced for PHOSPHOR-PI are probably not completely accurate because more code may have had to be instrumented to solve the bytecode verification issues. However, the comparison can still be made because, if anything, the method lists for PHOSPHOR-PI would only have been larger. If the tracked method counts from this paper's analysis are favorable compared to those for PHOSPHOR-PI, then they would only be more favorable if the method lists for PHOSPHOR-PI were larger.

Tracked method counts for our approach could only be generated on 10 of the

14 Dacapo benchmarks. For the remaining four, eclipse, tomcat, tradebeans, and tradesoap, Petablox raised errors when trying to compute method reachability as described in section 3.4.

4.2 Method counts

Benchmark	[12]	tracked	tracked/[12]	reachable	reachable/[12]	tracked/reachable
avrora	46948	6724	14.3 %	16384	34.9 %	41.0 %
batik	50625	8643	17.1 %	19066	37.7 %	45.3 %
fop	52183	8253	15.8 %	23568	45.2 %	35.0 %
h2	49765	15232	30.6 %	42455	83.5 %	35.9 %
jython	55125	11512	20.9 %	32653	59.2 %	35.3 %
luindex	21397	4135	19.3 %	11287	52.7 %	36.6 %
lusearch	21524	5135	23.9 %	12256	56.9 %	41.9 %
pmd	47148	14124	30.0 %	31277	66.3 %	45.2 %
sunflow	45700	9251	20.2 %	21077	46.1 %	43.9 %
xalan	48437	5352	11.1 %	14476	29.9 %	37.0 %

Table 4.1: Tracked method counts for [12] and this paper's analysis

Column 2 of Table 4.1 shows the tracked method counts produced by the previous analysis for PHOSPHOR-PI for each of the Dacapo benchmarks. Column 3 shows the tracked method counts produced by the analysis in this paper. The tracked methods are the methods that will become instrumented by PHOSPHOR to track taint propagation through the method. Remaining untracked methods only contain local variables which remain always tainted or untainted for the duration of the method and as such we do not need to track taint. Because instrumentation of methods is the primary contributer to the performance overhead of PHOSPHOR and PHOSPHOR-PI, comparing the tracked method counts of the previous analysis and the analysis in this paper is a good proxy for comparing performance overhead.

On average, our approach reduces the number of tracked methods by 79.9%. The individual benchmark percentages are shown in Column 4.

However, this number is somewhat misleading. PHOSPHOR-PI analyzed all of the methods appearing in the code for each benchmark and computed tracked methods from that set. For our approach, Petablox gives us reachability for free - Petablox only adds methods to the method entity domain that are reachable from the main method of the program. Our Datalog analysis is then applied to only those methods in the entity domain. Because the Dacapo benchmarks contain reflective calls, we configure Petablox to use its built-in dynamic reachability tool to compute reachable methods.

Because the Dacapo benchmarks are designed to be performance benchmarks, each benchmark has three or four execution sizes: `small`, `default`, `large`, and sometimes `huge`. When running the benchmark, the size is selected by passing it as a parameter to the benchmark. This parameter determines what code the benchmark runs and how much it stresses the system. Because we have configured Petablox to compute reachability dynamically, we must provide Petablox with inputs to run the benchmarks. We have configured Petablox to compute reachability by running the benchmarks at their `default` sizes, which is provided as an input to each benchmark. It is likely that if we had ran the benchmarks on their `large` size, more methods would be reachable and our approach would produce more tracked methods than for the `default` benchmark size.

Column 5 reports the total number of reachable methods as computed by Petablox for each benchmark to show how much of the reduction in tracked methods can be attributed to Petablox's reachability analysis. On average, the reachability analysis reduces the number of tracked methods by 48.8%. The individual percentages are shown in Column 6.

Column 7 shows the ratio of tracked methods in our approach to the number of reachable methods computed by Petablox. This ratio shows how much further

reduction in instrumentation our analysis produced past Petablox's reachability analysis. Our approach was responsible for a 60.6% reduction from the number of reachable methods to the number of tracked methods, and 31% out of the total 79.9% reduction in instrumentation in PHOSPHOR-PI Therefore, the reachability analysis contributes to the reduction in tracked methods more strongly than our analysis, though our analysis does result in a significant level of reduction.

4.3 Benchmarking PHOSPHOR

While the above measurements of tracked method counts are good proxies for comparing performance, they do not directly measure real-world performance of the modified PHOSPHOR tool described in section 3.6. Additional evaluation of the analysis approach in this paper can be done by measuring runtime performance of PHOSPHOR on the Dacapo benchmarks given the information on which methods to track and which call sites to instrument. These performance results were not obtained due to time restrictions and other confounding factors.

CHAPTER 5

Conclusion

We were able to write an improved information flow analysis, which compared favorably to previous work. It enabled us to reduce the amount of instrumentation required to track data with Phosphor by an average of 79.9%. This improvement came with a trade-off in functionality, in which we restricted Phosphor to only track taint as a boolean property instead of with a 32-bit integer as it originally did. We believe the trade-off is worthwhile given the potentially significant reduction in performance overhead.

We were not successful in completely evaluating our approach, as we were not able to measure performance overhead by directly running the updated version of Phosphor on the Dacapo benchmarks. Also, our analysis was not alone in contributing to the reduction in instrumentation. The built-in reachability analysis performed by Petablox alone produced a 48.8% reduction in instrumentation. However, our analysis also made a significant contribution, being responsible for about 31% out of the total 79.9% of the overall reduction. We think that there is great potential for future work in dynamic taint analysis by partial instrumentation, both to further reduce performance overhead and to relax the taint tracking restriction we placed on Phosphor without sacrificing performance.

References

[1] Logicblox. `http://www.logicblox.com`. Accessed May 2016.

[2] Petablox: Large-scale software analysis and analytics using Datalog. `http://www.cc.gatech.edu/~naik/petablox.html`. Accessed May 2016.

[3] Bell, J. and Kaiser, G. E. Phosphor: Illuminating Dynamic Data Flow in Commododity JVMs. In *OOPSLA '14: Proceedings of the 2014 ACM International Conference on Object Oriented Programming Languages and Systems Languages & Applications*, pp. 83–101, New York, NY, USA, 2014. ACM.

[4] Blackburn, S. M., Garner, R., Hoffman, C., Khan, A. M., McKinley, K. S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S. Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J. E. B., Phansalkar, A., Stefanović, D., VanDrunen, T., von Dincklage, D., and Wiedermann, B. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*, pp. 169–190, New York, NY, USA, October 2006. ACM Press.

[5] Bravenboer, M. and Smaragdakis, Y. Strictly Declarative Specification of Sophisticated Points-to Analyses. In *OOPSLA '09: 24th annual ACM SIGPLAN conference on Object Oriented Programming, Systems, Languages, and Applications*, New York, NY, USA, Oct 2009. ACM.

[6] Bruneton, E., Lenglet, R., and Coupaye, T. ASM: a code manipulation tool to implement adaptable systems. *Adaptable and extensible component systems*, **30**:19, 2002.

[7] Chandra, D. and Franz, M. Fine-grained information flow analysis and enforcement in a java virtual machine. In *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*, pp. 463–475, Dec 2007.

[8] Chin, E. and Wagner, D. Efficient character-level taint tracking for java. In *Proceedings of the 2009 ACM Workshop on Secure Web Services*. ACM, 2009.

[9] Halfond, W. G. J., Orso, A., and Manolios, P. Using positive tainting and syntax-aware evaluation to counter sql injection attacks. In *SIGSOFT '06/FSE-14*, pp. 175–185, New York, NY, USA, 2006. ACM.

[10] Stefano, C., Georg, G., and Letizia, T. What you always wanted to know about Datalog (and never dared to ask). *IEEE Transactions on Knowledge and Data Engineering*, **1**(1):146–166, 1989.

[11] Suh, G. E., Lee, J. W., Zhang, D., and Devadas, S. Secure program execution via dynamic information flow tracking. In *ASPLOS XI*, pp. 85–96, New York, NY, USA, 2004. ACM.

[12] Thakur, M. Dynamic Taint Tracking using Partial Instrumentation for Java Applications. Master's project, University of California, Los Angeles, Los Angeles, CA, USA, 2015.

[13] Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., and Sundaresan, V. Soot - a Java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, p. 13. IBM Press, 1999.

[14] Vallee-Rai, R. and Hendren, L. J. Jimple: Simplifying Java bytecode for analyses and transformations. 1998.

[15] Whaley, J. and Lam, M. S. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *ACM SIGPLAN Notices*, volume 39, pp. 131–144. ACM, 2004.