University of California
Los Angeles

Java Defensive Optimization

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

by

Haoqing Wang

2015

Abstract of the Thesis

Java Defensive Optimization

by

Haoqing Wang

Master of Science in Computer Science

University of California, Los Angeles, 2015

Professor Jens Palsberg, Chair

We design and implement two Java optimizations based on JSR308 annotations: pure method memoization and dynamic taint analysis optimization. In pure method memoization, we introduce JavaMem, an automatic method memoization tool for Java. JavaMem memoizes on Java pure or partially pure method at runtime according to offline execution time profiling results and static analysis-based correctness proof. We achieve 3% speedup on average on selected Dacapo benchmarks. For dynamic taint analysis optimization, we design and implement METEOR, a portable dynamic taint checking tool that instruments taint propagation instructions and optimizes the instrumented code by leveraging annotations. We compete our METEOR against Phosphor. With a slightly slower baseline performance on non-annotated code, METEOR still beats Phosphor on fully annotated benchmarks.

The thesis of Haoqing Wang is approved.

Majid Sarrafzadeh

Todd Millstein

Jens Palsberg, Committee Chair

University of California, Los Angeles

2015

Table of Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

Compiler optimization has been an important research topic for years. With numerous new features in programming languages, new optimization opportunities have been discovered. In this paper, we study optimization with the new Java 8 feature: type annotation. Type annotation [12] is an extension of Java type system and can be used with conventional Java type. Developers can define their customized type system by creating new type annotations and defining new type-checking rules. Type annotations are declared in Java source code and can be preserved in Java class files. Storing annotations in a class file allows us to optimize Java programs based on type annotations without accessing source files, which is critical for a programming language with large libraries. Type annotations can be either written by programmer or derived from static analysis and crowdsourcing. Given the broad underlying meanings and affluent resources of annotations, we believe new optimizations are still to be explored.

In this study, we have two categories of annotation-based optimizations:

- Optimization to speed up the program itself
- Optimization to speed up the inserted dynamic checks for ensuring various properties of interest

For the first category, we show an example of *@Pure* annotation which is allowed to be attached to pure method. Pure method has no visible side effects and its execution preserves the previous states of objects. Pure method memoization

is a technique that either stores the return value before returning from the first method call or returns the stored value without executing the method body provided the parameters are same. This technique has been studied in languages with pure methods or partial evaluation in [3],[17] and [37]. However their work limits the memoization to computation-intensive mathematical functions or languages having pure function features such as Haskell. Memoization on Java macro benchmarks faces challenges as memoization rules tend to be over restrictive. In chapter 2, we present JavaMem: a memoization tool targeting Java Dacapo benchmarks. JavaMem allows method parameters to be any reference in addition to primitives or language-embedded immutable object references. In section 2.4, we introduce a partial memoization technique applying to methods with only partial code being pure. We include experimental results on five benchmarks that were already finely designed and optimized in section 2.5.

For the second category, we describe an example of *@Trusted* and *@Untrusted*, a set of security annotations essentially related to information flow analysis. These two annotations have subtype relation such that annotated code can be type checked by annotation checker given security constraints are satisfied. However, with the presence of type downcast, dynamic information flow security can not be guaranteed by the static checker. Another problem can be that static checking targets partial code by design, which results in no runtime soundness. To ensure soundness, we decide to integrate security type annotation into dynamic taint checking. In our research, security annotations have three functionalities:

- To prove partial code security regardless of downcast(by static checking).
- To provide optimization information.
- To serve as a declarative and readable system for defining security policies.

We design and implement our own dynamic taint checker integrated with security annotation features. The implementation of dynamic taint checking is studied

in [5], [4], [22], [7], [8], [35], [33], [31], [29], [27], [32] and [18]. Our taint checking tool uses similar approaches from the previous work and is written on Soot library for being portable, maintainable and extensible. We evaluate our tool METEOR on micro and macro benchmarks in section 3.4.

The remainder of the thesis is as follows. Chapter 2 presents JavaMem including its offline profiling technique, correctness proof static analysis and partial memoization. We include experimental results of both pure and partially pure method memoization in section 2.5. Chapter 3 presents METEOR design and implementation. Chapter 4 compares JavaMem and METEOR to prior works. Chapter 5 concludes this study.

CHAPTER 2

Memoization

In this chapter, we propose JavaMem as a tool to optimize Java bytecode by method memoization. JavaMem contains three main components:

1. An offline profiler to find pure method candidates according to performance profiling information.
2. A static analyzer to identify methods out of candidates to memoize.
3. An instrumentor to rewrite methods from *step 2* at bytecode level.

Memoization in Java has not been explored on macro benchmarks such as Dacapo benchmark suite. Unlike in languages with pure function like Haskell and ML, programmers cannot declare pure functions in Java before Java 8. Memoization also requires identical values on method actual parameters. Memoization for Java faces two main challenges due to these restrictions:

- Pure methods are not explicitly declared.
- Parameter deep comparison incurs high overhead.

The first challenge is addressed by many program analysis and in our case, by adopting JSR308 annotation to indicate pure methods. *@Pure* annotation can be inferred using inference tool in [16]. The second challenge limits the number of methods worth memoization by incurring high parameter comparison overhead. To alleviate parameter comparison overhead, we decide to focus on methods with

only immutable parameters and compare locations of them. We define *immutable parameters* as those that are not mutated after the pure method is called. Note that parameter immutability is at object immutability level which is different from reference immutability in that, it enforces that object is immutable through the whole program after initialization while the later ensures immutability within the pure method.

The process of memoization is as follows. First JavaMem identifies all pure methods and profiles them to select the candidates for memoization. Once candidates are chosen, JavaMem then starts a static analysis to identify those with only immutable parameters and primitives. At last, the instrumentor transforms the bytecode by inserting instructions. In section one, we present our approach for pure method candidate selection. We then show how JavaMem finds pure methods with immutable parameters from the candidates in section two. Section three provides code transformation for optimization. In section four, we investigate further in partial memoization on impure methods. In the last section, we include the experimental results on selected Dacapo benchmarks.

2.1 Pure Method Candidate Selection

Pure method is a method without side effects according to the definition in [30]. We define *@Pure* as the type annotation for declaring method purity. We use the tool *ReImInfer* in [16] to infer pure methods. *ReImInfer* uses a context sensitive constraint solver to generate reference immutability annotations from which method purity is inferred.

We define *candidate methods* as pure methods that are worth memoization in terms of saving execution time. Note that parameters in candidate methods are not enforced to be immutable. We define *target methods* as those selected out of candidates methods with only immutable parameters or primitives.

Benchmarks	Pure method number	Called number	Candidate number
avrora	3698	116	0
xalan	4019	388	5
fop	3355	860	9
luindex	1497	147	0
lusearch	1497	31	1
sunflow	756	181	0

Table 2.1: Pure method number and candidate method number.

Because of high quantity of pure methods, our profiler instruments all pure methods at once. Call count and execution time are both profiled and used for estimating potential speedup. The execution time is valid only after the method is fully warmed up. We define the general execution time of method as:

$$E = \frac{\sum T_i}{n} + BT$$

In the formula, T_i is the execution time and n is the count of pure method calls. BT is the overhead of executing profiling instructions. Note that pure methods can be recursive, resulting in less accurate execution time estimation. To resolve this, we define the general execution of recursive pure methods as:

$$Er = \frac{2 * E}{n - 1} - \frac{BT}{n * (n - 1)}$$

The following equation holds:

$$ST = \begin{cases} (E - P) * n & non - recursive \\ \frac{(Er - P) * n}{D} & recursive \end{cases}$$

ST is the execution time that can be potentially saved by memoization. D is the average depth of call stack for the recursive call. P is the overhead of

retrieving the memoized value. Memoization targets candidates with ST being big enough($> 1.0\%$) compared to benchmark execution time. Note that P and BT are profiled in an isolated execution. Table 2.1 shows the number of pure methods in some Dacapo benchmarks. Note that it is impossible to cover every execution path during benchmarking. Column *Called number* gives the number of pure methods that are *actually* executed. Column *Candidate Number* is the number of candidate methods. The number of candidates is small due to that P is proportional for most pure methods that run quickly such as getters in class.

2.2 Static Correctness Proof

A method call is allowed to return the memoized value without executing method body if:

1. it is a pure method and
2. it is called with the same parameters between two executions.

Ensuring 2 is straightforward when the method is static and parameter is primitive. However, with reference parameters, deep object comparisons are demanded. Comparison between two objects in language without deep object immutability feature such as in [14] is difficult despite that *equal* method can be used in some cases. But in general, deep object comparison produces unacceptable overhead and *equal* method does not enforce deep object equality. To resolve this, we first define same parameters as follows:

- They are primitives with same values
- They are *1)* references and *2)* every value transitively reachable from them is same

Criteria 2 is proved if a pure method has only same primitives or immutable parameters with same locations. In fact, we could relax criteria 2 by narrowing down the reachable variables to those only read by the method. Figure 2.1 shows an example of parameter immutability. The only value read by the method *foo* is *a.f1*. We consider *a* as an immutable parameter(receiver) for method *foo* even though field *f2* of *a* is mutated after the first call of *foo*.

A key observation is that some objects are deep immutable only after certain execution point. For example, in xalan, the AST nodes are initialized in the first execution phase and then stay unchanged until the end of execution.

The above discussion leads to approaches to identify target methods. We design a static analyzer including a whole program pointer analysis and two whole program analysis to prove parameter immutability.

For the pointer analysis, we first use an inclusion-based context-insensitive analysis similar in [19] to generate the points-to graph. We use Soot as our analysis tool and modify its Spark library. We keep the original context-insensitive analysis and extend the points-to graph to hold extra information in heap object. A data structure *F(p)* is created for each heap object for later analysis.

We have two analysis for candidate method p: field read analysis and parameter immutability analysis. The field read analysis is to identify, for each candidate method parameter, which field is read. It inputs a points-to graph, a call graph and a candidate p and outputs a set of objects and their fields read by p. The parameter immutability analysis is, for each candidate method p, to judge whether or not all its parameters are immutable. It inputs a points-to graph(with added analysis results from field read analysis), a control flow graph and a method p and outputs a boolean indicating target method.

For field read analysis, because we only focus on transitively reachable objects and their fields read by the method, an inter-method analysis is performed to

```
public static void main(){
   A a = new A();
   a.set(1,2);
   a.foo(1); // record
   a.setF2(0); // modify field f2 of object a
   a.foo(1); // return memoized value
}
Class A{
   int f1;
   int f2;
   public A(){
   }
   public void set(int x, int y){
      this.f1 = x;
      this.f2 = y;
   }
   public void setF2(int n){
      this.f2 = n;
   }
   @Pure public int foo(int num){
      return num + this.f1;
   }
}
```

Figure 2.1: Parameter immutability

```
Input : CallGraph, Points-to Graph, p
Output : Set
//initiate:
Set = ∅
M = ∅
add p in M
MethodToAnalyze = all methods reachable from p
//core algorithm:
Repeat
   remove a method p from M and p from MethodToAnalyze
   for each instruction Inst in p
      if Inst has the form v = f.s
      then
         for every object o that f points to
         add o to Set and filed s to F(p) of o
   for every m ∈ callers(p) and in MethodToAnalyze
      add m to M
Until MethodToAnalyze == ∅
```

Figure 2.2: RMR analysis for candidate method p

compute such a set: RMR. For each method p, $RMR(p)$ is defined as a set that contains all reachable objects read by the method on the heap. After the analysis, each object owns a data structure $F(p)$ containing the fields read by method p. The field read analysis is a straightforward context-insensitive, flow-insensitive analysis based on the call graph and points-to graph. The analysis is described through the algorithm in Figure 2.2.

For parameter immutability analysis, we use a backwards dataflow analysis framework in [21] to find target methods. For candidate method p, we com-

pute transitively reachable objects set *INI(p)* from parameters. The rules of the dataflow analysis for proving p as target method are described as follows(we describe the rules ignoring the cases in static field and array without losing generality. Our analysis is implemented correctly for these cases).

1. $out(b) = \cap in(s)$(s is the successor block of b)
2. $in(b) = out(b) - Kill(b)$(*Kill(b)* returns a set containing o in that, b is a block having the form $f.v = t$. Field v is contained in *F(p)* for o in *RMR(p)*, where every o is the object pointed to by reference f.)

in(b) and *out(b)* are the immutable object sets before and after execution respectively. The complete algorithm to prove a method to be a target method is shown in Figure 2.3 in which p is a candidate pure method. The algorithm runs a dataflow analysis and returns *TRUE* if p is a target method. For reflection, we use the embedded TamiFlex in Soot which records information of reflections from pre-executions.

2.3 Memoization Implementation

In this section, we describe JavaMem code transformation. We create a customized HashMap for each method to store parameter values and the return value. Note that for reference parameters, the virtual address is used as Hashmap key. This is safe in our experiment since the object virtual address is unchangeable in a benchmarking cycle. We instrument the targeted methods in that, at method entry, we first inspect the existence of actual parameters in the HashMap. If they are cached in Hashmap, JavaMem retrieves the return value from the map and returns it. Before the original return instruction, JavaMem inserts instructions to hash the return value. Figure 2.4 shows code transformation after instrumentation.

```
Input : p, Control Flow Graph, Points-to Graph
Output : Boolean
//initiate :
E = INI(p)∩RMP(p)
in[Exit] = E;
For every block i
   in[i] = E
ChangeNodes = all blocks
TM =  Ø
//core algorithm:
Repeat
   remove i from changeNodes
   for every successor block s of i
      out[i] ∩= in[s]
   orig = in[i]
   in[i] = out[i] - kill(out[i])
   if orig != in[i]
      for every predecessor p of i, add p to changeNode
Until changeNodes == Ø
if E == In[k], (k is the first block of p) return TRUE
return FALSE
```

Figure 2.3: Pure method parameter immutability analysis algorithm

```
public int foo(A this, List list, int num){
   MenKey key = MemRuntime.getKey(this, list, num);
   if (A.memMap.contains(key)) return A.memMap.get(key);
   // code of pure computation
   . . .
   // end of pure computation
   A.memMap.put(key, v);
   return v;
}
```

Figure 2.4: Memoization code transformation

2.4 Partial Memoization

Pure method memoization demands method purity. In fact, there exists non-pure method that executes impure instructions only after pure computation. An example to illustrate this is that a pure method is inlined at the entry of an impure method. In Figure 2.5, the method runs pure instructions first and then an impure instruction loading the result of pure computation. To discover partial memoization opportunities, we use an intra-method dataflow analyzer in Soot which first splits method into a pure part followed by an impure part. The first impure instruction delimits the two parts. For implementation brevity in partial pure analysis, we assume that any impure instruction can change the pre-states of objects. A list of impure instructions is shown in Table 2.2. After splitting, we compute variable dependent set of the impure part. In Figure 2.5, the method invocation of *bar* is the first impure instruction that sets method body apart.

We adopt the similar approach used in pure method memoization for partial memoization. The only difference is that the return value becomes variables in dependent set. To simplify implementation, we define a partial memoizable method

```
public int foo(int ){
   A r0;
   int i0;
   int i1;
   int i2;
   int i3;
   int i4;
   r0 := @this;
   i0 := @parameter0;
   i1 = r0.val;
   i2 = virtualInvoke r0.getV(); // A.getV(int) is a pure method
   i2 = i1 + i2;
   //this line is delimiter
   i3 = virtualInvoke r0.bar(i2); // A.bar(int) is an impure method
   i4 = i2 + i3;
   return i4;
}
```

Figure 2.5: Method before partial memoization

by two criteria:

- The dependent set contains at most one variable.
- It follows the same criteria as defined in pure method memoization except that the targeted method is impure.

Figure 2.6 shows the code transformation after partial memoization.

instruction	description
local.field = *v*	field write
local[m] = *v*	array write
field = *v*	static field write
invokeExpr	when the method is impure

Table 2.2: List of impure instruction

2.5 Evaluation

We evaluate JavaMem on Dacapo benchmark suite. Memoization on Dacapo is challenging in that every benchmark in Dacapo suite is very well manually tuned by experts. Methods that can be potentially memoized were designed using space-for-time techniques such as hashmap cache. However we are still able to find speedup of two benchmarks through pure method memoization: xalan and fop. And in xalan, luindex and lusearch, we obtain speedup by partial memoization. In Table 2.3, we present the pure method memoization speedup results.

```
public int foo(int ){
   A r0;
   MemMap r1;
   int i0;
   int i1;
   int i2;
   int i3;
   int i4;
   bool b0;
   r0 := @this;
   i0 := @parameter0;
   r1 = r0.memMap;
   b0 = virtualInvoke r1.containsKey(r0,i0);
   if b0 goto lable0;
   i1 = r0.val;
   i2 = virtualInvoke r1.getV(); // A.getV(int) is a pure method
   i2 = i1 + i2;
   r1.put(r0,i0,i2); // memoize value
   goto lable1;
   //this line is dilimiter
label0:
   i2 = virtualInvoke r1.getValue(r0,i0);
label1:
   i3 = virtualInvoke r0.bar(i2); // A.bar(int) is an impure method
   i4 = i2 + i3;
   return i4;
}
```

Figure 2.6: Method after partial memoization

Benchmark	Original	Pure No.	Partial Pure No.	Opt (speedup)
avrora	3021ms	0	0	N/A
xalan	563ms	1	3	541ms(4.1%)
fop	261ms	1	1	254ms(2.8%)
luindex	602ms	0	1	580ms(3.8%)
lusearch	698ms	0	1	683ms(2.2%)
sunflow	1827ms	0	0	N/A

Table 2.3: Memoization experimental results (*Pure No.* is the number of pure methods we memoize and *Opt* is the execution time after memoization.)

CHAPTER 3

Efficient Dynamic Taint Checking

In this chapter, we introduce our tool METEOR for dynamic information flow analysis and optimization. METEOR has two major parts: an bytecode instrumentor that inserts taint propagation instructions and an optimizer that takes advantage of JSR 308 annotation and static checking. The bytecode instrumentor enables taint propagation by instrumenting both system and application libraries. It also reads annotations from class files and optionally from configuration files. The optimizer performs mostly offline to eliminate unnecessary propagation instructions according to annotations. METEOR is implemented on Soot and ASM. It is worth mentioning that our tool makes no modification to JVM which makes it portable.

3.1 Taint Propagation Implementation

Our approach adopts variable-based taint propagation techniques. Tracking taint tags by variable is consistent with the granularity of JSR 308 annotation. In the following subsections, we describe our taint storage and propagation implementation.

3.1.1 Taint Tag Storage

METEOR attaches a 32-bit integer to every variable. Each bit in the 32-bit integer represents a type of annotation and 32 different types of security annotation

can be supported as JSR308 defines 17 types of annotations so far. In this study, we are only concerned with a single type of annotation but could easily extend our work to multiple types. METEOR creates 32-bit taint variables for local variables, method parameters, return values and object fields. For taint storage, [36] shows a centralized storage where all taint tags are stored in a global array, which incurs high overhead. We use de-centralized storage approach in which each taint tag is locally stored with each variable. Ensuring consistency with JSR 308 annotation faces the challenge of storing taint tags for primitive array because JSR 308 annotation can declare annotations on array elements. METEOR tracks each element in primitive array to obtain high precision. For each object, we add a *self-tag* field in the object class indicating the taint value of the object itself. Adding a field to *java.lang.Object* class crashes Hotspot JVM as result of changing the internal class structure. To resolve this, we create an interface with *self-tag* and modify all classes to extend it.

For each local variable or parameter, METEOR creates a taint local variable or parameter if it is a primitive. If the variable is an object or object array, no modification is needed because it has a *self-tag* already. If it is a primitive array, we create a taint integer array of the same length in which each integer is a taint tag of the element in the original primitive array. The only tricky case is that for multi-dimensional primitive array, we use the feature provided by JVMTI to convert every multi-dimensional primitive array to a special object at runtime.

For object fields, we modify the class file to create a taint field for each primitive field. Similarly, METEOR creates an array of taint tags for each primitive array field. Note that for HotSpot 8, adding fields to specific classes such as Java primitive wrapper classes is forbidden. To resolve this, we use the technique in [36] and pre-allocate an external map which stores taint tags for these fields. The tag could be retrieved by using the signature of the object and offset of the field which is computed at compile time.

3.1.2 Taint Propagation

In this subsection, we describe the code transformation for taint propagation. Previous work such as [5] instrumented class files at bytecode level. We choose Soot as our tool library to rewrite bytecode at a higher abstraction level to ensure high readability and extensibility. Soot reads bytecode and tranlates bytecode to Jimple which is a 3-address representation. Jimple instruction is stackless with references to stack translated to local variables. Local variables in Jimple have specific names and are fully typed. This contributes to static analysis design and simple code generation. Next, for each type of Jimple statement or expression, we discuss our code transformation strategy.

Assignment statement: Assignment statement contains a LHS and a RHS where the taint tag always flows from RHS to LHS. The taint tag of the LHS is identical to that of the RHS. Specifically, if LHS or RHS has primitive type, we add an assignment instruction to assign the taint tag of the RHS to the LHS.

Arithmetic expression: Arithmetic statement has the general form of $x+y$. The taint tag of the arithmetic result is the least upper bound of taint tags of operands. We simply refer to the bit-wise OR of the two operands as the taint tag of arithmetic expression.

Class field and method declaration: Our instrumentor iterates through primitives or primitive arrays and adds an extra taint field for each such variable. Method declaration must be transformed to a corresponding instrumented one with a different name to avoid overriding issues. A taint parameter is added for the primitive or primitive array parameter to pass the taint tag into the method body.

Method invocation: METEOR instrumentor modifies each method invocation statement to have the signature of the instrumented method. For each primitive or primitive array formal parameter in the original method, METEOR

places the taint variable of those parameters in right places. After the invocation, we have to retrieve the taint tag of the return variable if the return type of method is not void. To avoid the overhead of pre-allocating a boxed object in [5], we simply pre-allocate a global class to temporarily store the return taint tag and then pass back to the call site. Note that global class is one-to-one mapped to Java thread to preserve thread safety.

Method return: Before return statement, our instrumentor inserts an instruction to pass the taint tag of return primitive or primitive array to the global class.

NewArray: Jimple has a *newarray* expression which initiates a new array. For each *newarray(Primitive Type)* expression, a *newarray(int)* instruction is inserted to create an array containing taint tags of the original array elements.

Native method call: Although our instrumentation tool executes on whole application and system libraries, methods not written in Java bytecode are not applicable for instrumentation. These methods are native methods in machine code and can be called from bytecode call cites. We adopt the same strategy used in [5]. Before native method calls, we pre-compute the bit-wise OR of all inputs reachable from every actual parameter. To make implementation simpler without losing soundness, we take into account only the taint tags of primitives and references.

Reflective method call: JVM supports reflection which dynamically retrieves methods from a class and invokes the method. At call site, the instrumentor collects taint tags for parameters and intercepts the reflective call. A reflective call to the instrumented method is therefore called by the interceptor instead.

3.2 Sinks and Sources

JSR 308 security annotation introduces *@Trusted* and *@Untrusted* annotations. By declaring variable as *@Trusted*, we mean it can only be a trusted variable while *@Untrusted* variable could be either trusted or untrusted. Subtype relation holds: $@Trusted <: @Untrusted$. In dynamic taint analysis, sensitive values are denoted by *@Untrusted* in our experiments. As we mentioned, JSR308 security annotations have various types such as *@Trusted*, *@Random @Readonly*, *etc.* They can appear in any place of type-use. This leads to a broader and more fine-grained definition of sources and sinks.

Sinks: Sinks are places where only the non-least restrictive taint tag is acceptable. In our system, they are places where non-sensitive(*@Trusted*) values are expected.

Sources: Sources are places where the non-most restrictive values are created. In our system, they are referred to places where sensitive(*@Untrusted*) values are created.

In phosphor[5], sinks and sources are declared at method level, which tends to be restrictive in industry-level development. For example, Phosphor can only indicate return value of a method to be a source and method parameter as a sink. This leads to a problem of over-constraining the way in which application is written. Figure 3.1 shows a case where method *write* is treated as a sink regardless of context. However in some call sites, the developer knows the method will not incur a security problem, so sensitive values are allowed. Similarly, the constructor of a class could return either a sensitive or non-sensitive object. Sinks and sources in METEOR are defined at type-use level which means whenever a type use appears we could treat the correlated variable as a source or sink.

Adding sources and sinks for simple application is a trivial task. In [36], they insert assertions in code to define sources and sinks. Prior definition of sources

```
//call site 1 :
   write(s); // s should be non-sensitive value
   @Trusted A v = new @Trusted A();
//call site 2:
   write(p); // p is allowed to be sensitive
   @Untrusted A  v  = new @Unstrusted A();
```

Figure 3.1: Type-use level sources and sinks

and sinks is restricted in a way that sinks and sources appear in small numbers. For example, in SQL injection detection, sources are hard-coded strings and sinks are functions executing sql commands. But when it comes to JSR 308 annotation with broad meanings, adding sources and sinks could be untraceable and error prone because the assumption of small number does not hold. Static information flow type system such as JSR308 annotation checking tool is developed to enforce information flow safety while its soundness is not preserved due to partial checking and downcast, which will be explained later.

In dynamic taint analysis, with less readable sources and sinks declared in form of assertion or declaration in the configuration file, extending the code or changing the security policy is also error prone. We solve this by taking advantage of JSR308 annotations. As we described in chapter 2, the JSR308 annotation provides a statically checkable and highly readable security type system. We integrate the JSR308 security annotations into our dynamic information flow analysis. The annotation serves three purposes :

1. Enabling partial program security by static checking.
2. Directing compiler optimizations
3. Providing a formal declarative approach to define flexible and easy-to-change security policies.

Combining JSR308 annotations with dynamic information flow analysis faces some challenges, in that the static annotation checker provides no guarantee that the whole system is safe at runtime. In fact, it is infeasible using static checker to check the whole code in most cases[12]. We refer to JSR 308 static checking as partial checking when it checks partial code leaving some library code unchecked. In addition, annotation type downcast exists for the convenience of type checking but requires runtime checking on it. With only partial checking, we next describe some key rules of annotation-to-instruction transformation to preserve soundness.

Annotation on Typecast: JSR308 annotation could be attached to a typecast. Typecast is categorized as follows.

1. Typecast as a source on an object initiation
2. Typecast as a sink
3. Typecast as a source on any instruction except object initiation

For category 1, METEOR inserts instructions right after the *newRef* or *newArray* instruction to initialize the taint tag to be a source(*@Untrusted*) value. For category 2, the instrumentor inserts check instructions based on the type of the variable. For primitives, it checks the taint variables against the static taint annotations while for objects it checks the *self-tags*. Note that if the typecast annotation is a source(*@Untrusted*) annotation, which means a upcast, no instruction is required to be inserted. For category 3, we define a special annotation *@Init* to mark the typecast as a source. Introducing category 3 is important because it adds flexibility for developers to define flexible security rules without modifying the source code. Figure 3.2 illustrates code transformation on typecast as a source and sink respectively. The method *fooCastSink* is transformed regarding category 2 and *fooCastSource* is transformed according to category 3.

Sink annotation on parameter: Without the annotation static checker, every sink variable needs to be dynamically checked. With whole program static

```
public void foo(){
   @Trusted A v = (@Trusted A) bar();
   //code
}
public void fooCastSink(){
   A v = bar();
   if (v.self_tag | 0x00000001 != 0x00000000) throw new
       TaintExeception();
   //code
}
public void fooCastSource(){
   A v = bar();
   v.self_tag |= 0x00000001
   //code
}
```

Figure 3.2: Typecast code transformation

checking, we could simply ignore the sink annotations on parameters at runtime and only insert dynamic checks on downcasts. But with partial checking, we conservatively insert check instructions for each sink parameter because callbacks from unchecked code could propagate sensitive(*@Untrusted*) values into the method.

Source annotation in configuration file: A configuration file contains annotation declarations defined by developers at method level. It is also the responsibility of programmer to ensure the correctness of these method declarations[12]. In practice, these methods are from unchecked code and their method bodies are not statically checked. METEOR assumes these method interfaces are correct and do not violate soundness in terms of return taint tag. For example, *@Trusted foo()* means that this method return value is always set to *@Trusted*. Figure 3.3 shows a configuration file that declares *exec* and *load* as sinks. Note that the default annotation is *@Untrusted* in static annotation checker(*exec* returns *@Untrusted* value) while the default taint tag in METEOR dynamic analysis is mapped to *@Trusted*.

We implement annotation transformation on our modified ASM tool. Because JSR308 is only partially supported by ASM5.0 in that annotations with bytecode offset are not supported, we modify the ASM library to precisely insert instructions by reading the bytecode offset of JSR308 annotations.

3.3 Optimization

With annotations available in bytecode, we are able to design and implement several optimization techniques. Our optimizations take advantage of static analysis from annotation inference tool which automatically generates annotations in bytecode. There are four optimization phases focusing on different optimization aspects. In the following subsections, we show the details of our optimizations.

```
class Runtime {
   public Process exec(@Trusted String command);\\sink command
   public Process exec(@Trusted String[] cmdarray);
   public Process exec(@Trusted String[] cmdarray, String[] envp);
   public Process exec(@Trusted String[] cmdarray, String[] envp,
      Filedir);
   public Process exec(@Trusted String command, String[] envp);
   public Process exec(@Trusted String command, String[] envp,
      Filedir);
   public void load(@Trusted String filename);
   public void loadLibrary(@Trusted String libname);
}
```

Figure 3.3: Configuration file

3.3.1 Non-sensitive Method Propagation Elimination

If no non-sensitive(*@Untrusted*) taint tag flows out of method, we do not have to instrument the method body with taint propagation instructions. We define the default taint tag of a variable as the value of sink annotation(*@Trusted*). We define a method as a non-sensitive method if it contains only non-sensitive(*@Trusted*) annotation on escaping instruction. Escaping instruction is defined as the instruction that propagates values to fields, array elements, method parameters or return variables. Table 3.1 shows the list of escaping Jimple instructions. If the method is a non-sensitive method, it does not propagate non-sensitive values. We mark these methods and eliminate every propagation instruction.

Instruction	Description
specialinvoke local.m(. . .)	special method call
interfaceinvoke local.m(. . .)	interface method call
virtualinvoke local.m(imm , ..., imm)	virtual method call
staticinvoke m(imm , ..., imm)	static method call
new refType	object initialization
newarray(type) [m]	one-dimensional array initialization
*newmultiarray(type)[m]**	multi-dimensional array initialization
return v	return
local.field = v	object field write
local[imm] = v	array write
field = v	static field write
throw ex	exception throw

Table 3.1: List of escaping instruction in Jimple

3.3.2 Parameter Checks Elimination

As we mentioned above, methods having sink parameters have to inspect taint tags of sink parameters at runtime before entering the method body because of callbacks from unchecked code. Parameter checking instructions are eliminated if the method is private. We also identify methods that are called only from checked code. For reflections, we use the same tool in chapter 2 to obtain runtime reflection information. Table 3.2 shows the number of such methods.

3.3.3 Sensitive Branching

For methods that are not marked as non-sensitive methods, we perform optimization based on the observation that taint propagation does not need to start until an untrusted taint tag is found. More specifically, taint tracking needs to start only

Benchmark	Method-total	Method-to-optimize
avrora	439	361
batik	N/A	N/A
eclipse	N/A	N/A
fop	790	604
h2	344	3
jython	131	15
luindex	235	165
lusearch	262	176
pmd	143	15
sunflow	83	25
Tomcat	2342	1590
tradebeans	537	331
tradesoap	572	306
xalan	1863	974

Table 3.2: Parameter elimination methods(*Method-total* is the total number of method with sink parameter. *Method-to-optimize* is the number of chosen method out of method-total to eliminate parameter check.)

after the first non-sensitive(*@Untrusted*) taint tag leaks out of the method local scope. Our optimization is inspired by [25]. Inside a sensitive method, our instrumentor performs a special transformation: it keeps the original bytecode method body(untracked version) and appends the instrumented method body(tracked version) to the original one. Before each escaping instruction, a check instruction *CK* is inserted and a corresponding label *L-CK* is created pointing to a corresponding bytecode in the instrumented code. At runtime, PC jumps to the label if an untrusted taint tag is detected. We show the example of sensitive branching transformation in Figure 3.4 and Figure 3.5. Note that the first integer parameter

```
public void foo(int, int){
   A r0;
   int i0;
   int i1;
   r0 := @this;
   i0 := @parameter0;
   i1 := @parameter1;
   r0.value_tag = i1;
   r0.value = i0;
}
```

Figure 3.4: Code before sensitive branching

is the value and the second is the taint tag.

METEOR eliminates unnecessary branching checks by two criteria:

1. The checking target is non-primitive or primitive array and
2. It has a *@Trusted* annotation.

Suppose we have *@Trusted* annotation on *i0*, branching checks and corresponding propagation instruction in tracked version can be eliminated. Figure 3.6 illustrates the details of code transformation.

3.3.4 Non-escaping Propagation Elimination

In our experiments, there are some methods that we can not optimize using *non-sensitive method propagation elimination* or *sensitive branching* because of instrumentation issues such as method body size exceeding 64KB limit. As we discussed above, an instruction does not propagate sensitive information if no sensitive(*@Untrusted*) value escapes through escaping instruction. METEOR eliminates propagation for such instructions in these methods.

```
public void foo(int, int){
      A r0;
      int i0;
      int i1;
      r0 := @this;
      i0 := @parameter0;
      i1 := @parameter1;
      if i1 != 0x00000000 goto label0;
      r0.value = i0;
      return;
   label0:
      r0.value_tag = i1;
      r0.value = i0;
}
```

Figure 3.5: Code after sensitive branching

```
public void foo(int, int){
      A r0;
      int i0;
      int i1;
      r0 := @this;
      i0 := @parameter0;
      i1 := @parameter1;
      r0.value = i0;
      return;
   label0:
      r0.value = i0;
}
```

Figure 3.6: Code after annotation optimization

3.4 Evaluation

In this section, we evaluate the performance of METEOR. We first evaluate our dynamic information flow analysis tool with the optimizations off. This serves as a baseline of taint propagation performance. For baseline performance, we compare METEOR analysis with Phosphor analysis in both micro and macro benchmarks. We choose Phosphor because it is the only portable analyzer that works on Dacapo benchmarks. We then evaluate our optimizer on benchmarks with only simple sources and sinks. In the simple source-sink evaluation, we show how our general optimization without the aids of annotations could speedup the execution. In the last subsection, we evaluate our tool in macro benchmarks with annotations and compare the results with Phosphor.

3.4.1 Experiment Setup

Fully evaluating METEOR faces challenges in that there exists no dynamic information flow analyzer integrated with JSR308 annotations. To show the overall effects of our defensive optimization and our instrumentation tool, we decide to hack Phosphor and add the annotation features to it. Phosphor is a dynamic taint analyzer written on ASM library. We choose Phosphor because we believe it is the only tool available providing both soundness and good performance on Dacapo benchmarks and we are familiar with ASM. We are able to get Phosphor source code and integrate our modified ASM tool into it. We keep all phosphor original propagation instrumentation and optimization unchanged and only add features of reading JSR308 annotations and transforming them to the corresponding bytecode instructions. Note that we do not add annotation-based optimization to phosphor. (In fact, we fail to add our optimization to phosphor because the way that phosphor is written involves too many interactions between different instrumentation and optimization phases, which is very error prone although we are able to get help from the author).

For all benchmarks, we have to recompile the source code with Java 8 compiler to support JSR308 annotations. We manually change class files in xalan, lucene, luindex, tomcat and eclipse to make the compiling succeed. We add key annotations to places where we believe there should be sinks and sources. And we also partially annotate the key components of those benchmarks to serve as the main frames of annotation sets. We then use the *uw* annotation inference tool to generate fully-annotated benchmarks. The *uw* tool automatically inserts downcasts to ensure that the static annotation checker checks without error. Such automatic downcast insertion is included in their newest version of unpublished inference tool. Because the way how the inference tool and programmers put annotation is not always run-time secure with the presence of downcast, we encounter numerous security exceptions forcing programs to halt. To resolve this, we carefully

tune the benchmarks and change some security checks to non-sensitive(*@Trusted*) initializations. For most benchmarks, we find such manual work is done within an hour based on our understanding of the benchmarks by repeating execution. For our experimental purpose, we assume these changes are safe although further expert advice might be needed to prove the correctness of these changes.
All experiments are done in Macbook Pro 13 with 4GB RAM and 2.5GHz CPU with power on and the execution time is the average of 10 best runs of 10 runs after the coefficient variant is small enough. We use HotSpot 8 as our JVM.

3.4.2 Baseline Performance

In this evaluation, we compare taint propagation performance. We instrument the whole JRE8 libraries and benchmarks with METEOR instrumentor and Phosphor instrumentor respectively. In Table 3.3, we show the size of the benchmarks and type and quantity of annotations inserted.

Benchmarks	Lines of Code	Number of @Random	Number of @Trusted
JLV	1210	11	0
Jsortsql	483	0	9
JRQsortsql	849	6	9
Benchmarksql	3984	7	69
MonteCarlo	14092	186	0

Table 3.3: Micro Benchmark size and annotation details.

JLV is a benchmark running Las Vegas algorithms. We annotate it with *@Random* annotation set as JLV generates and accepts random values. Jsortsql is a micro benchmark that executes SQL scripts. We use @Trusted annotation set to ensure that each method *executeQuery* allows only trusted commands. JRQsortsql is a modified version of Jsortsql which executes quick sort to rank the query results. We annotate it with both *@Trusted* and *@Random* annotation sets. Benmarksql

is a commonly used mid-sized benchmark annotated by @Trusted annotation set. MonteCarlo is another widely used mid-sized benchmark that runs randomized algorithms. We annotate its algorithm core libraries and recompile the benchmarks with Java 8 compiler. Dacapo benchmark suite contains 14 benchmarks for measuring JVM performance. With METEOR, we are able to instrument 12 out of them and run them to completion. Batik fails to instrument because it uses special compiler library that forbids METEOR from analyzing. Eclipse fails because of library compiling issues. With Phosphor, we are able to instrument 9 of them and execute them without exception. Though we can run through all 14 benchmarks in the virtual machine provided by the author, we believe there are some special modifications made on either the JRE libraries and the benchmarks or its instrumentation requires special manipulation. For the purpose of propagation-only performance comparison, we refer to the results in [5]. Table 3.4 shows the results of overhead in both METEOR and Phosphor. In Table 3.5, we include the results of Dacapo benchmarks. The *Original* column shows the execution time without taint propagation.

Benchmarks	Original	Phosphor	overhead	METEOR	overhead
JLV	158ms	280ms	77.85%	312ms	97.47%
Jsortsql	823ms	1249ms	51.76%	1312ms	59.42%
JRQsortsql	911ms	1312ms	44.02%	1428ms	56.75%
Benchmarksql	2091ms	2139ms	22.96%	2180ms	42.56%
MonteCarlo	2448ms	4376ms	78.76%	5120ms	91.50%

Table 3.4: Propagation-only baseline performance and overhead.

We notice that the propagation overhead of METEOR is higher than that of phosphor. We observe that, in phosphor implementation, there already exists some optimizations based on simple static analysis at bytecode level. Although there is a chance JIT would do the same optimization in some cases, such opti-

Benchmarks	Original	MET	MEOH	PHOH
avrora	3021ms	3145ms	4.1%	3.3%
batik	1253ms	N/A	N/A	13.5%
eclipse	17932ms	N/A	N/A	219.6%
fop	261ms	462ms	77.0%	57.7%
h2	4043ms	5967ms	47.6%	38.2%
jython	1623ms	2739ms	68.8%	56.9%
luindex	602ms	1112ms	84.7%	41.6%
lusearch	698ms	1521ms	117%	92.8%
pmd	1572ms	1698ms	8%	27.6%
sunflow	1827ms	2570ms	40.7%	35.0%
Tomcat	1602ms	1981ms	23.7%	38.2%
tradebeans	3217ms	4268ms	32.7%	31.9%
tradesoap	15823ms	18976ms	19.9%	20.6%
xalan	563ms	971ms	72.5%	50.2%

Table 3.5: Propagation-only performance in Dacapo.(*MET* is the execution time of METEOR. *MEOH* is the overhead of METEOR. *PHOH* is phosphors overhead)

mization at instrumentation phase still generates execution speedup.

3.4.3 Simple Sources and Sinks Benchmark Performance

We declare simple sources and sinks in the benchmarks. In case we intend to perform dynamic taint checking with minimum efforts, we insert sources and sinks into the program at method level. Note that we do not use static checker here and this evaluation is intended to show the maximum speedup without annotations. In Table 3.6 and 3.7, we present the list of source and sink methods. Table 3.8 shows the execution time in METEOR and Phosphor.

Class	Name&Descriptor
java.io.InputStream	int read(byte[])
java.io.InputStream	int read(byte[],int,int)
java.io.BufferedReader	int read()
java.io.BufferedReader	int read(char[],int,int)
java.io.BufferedReader	int readLine()
java.io.Reader	int read(char[])
java.io.Reader	int read(char[],int,int)
java.io.Reader	int read(CharBuffer)
java.io.Reader	int read()

Table 3.6: Method as a source in simple source-sink experiment

3.4.4 Fully-Annotated Benchmark Performance

With fully-annotated benchmarks, METEOR is able to take advantage of annotations. We instrument JRE8 libraries and benchmarks with METEOR optimizations on. On Phosphor side, we use our modified version of Phosphor to perform annotation transformation. Table 3.9 shows the fully-annotated performance comparison between METEOR and Phosphor. As we could see, with annotations available, the speedup due to METEOR optimizaitons is obvious. In our experiment, sunflow delivers the highest speed up. By our observation, it is because that sunflow tends to have a large number of non-sensitive methods.

Class	Name&Descriptor
java.io.OutputStream	void write(int)
java.io.OutputStream	void write(byte[])
java.io.OutputStream	void write(byte[],int,int)
java.io.PrintStream	void println(java.lang.String)
java.io.PrintStream	void println(java.lang.Object)
java.io.BufferedWriter	void write(char[],int,int)
java.io.BufferedWriter	void write(int)
java.io.BufferedWriter	void write(java.lang.String,int,int)
java.lang.Process	exec(java.lang.String[])

Table 3.7: Method as a sink in Simple source-sink experiment

Benchmarks	METEOR Ori.	METEOR Sim.	Phosphor
avrora	3145ms	3270ms	3124ms
batik	N/A	N/A	N/A
eclipse	N/A	N/A	N/A
fop	462ms	342ms	416ms
h2	5967ms	5201ms	5632ms
jython	2739ms	2394ms	N/A
luindex	1112ms	963ms	941ms
lusearch	1521ms	1491ms	1444ms
pmd	1698ms	1683ms	1543ms
sunflow	2570ms	2331ms	2579ms
Tomcat	1981ms	1782ms	N/A
tradebeans	4268ms	3821ms	N/A
tradesoap	18976ms	17988ms	N/A
xalan	971ms	794ms	862ms

Table 3.8: Comparison in simple source-sink case.(*METEOR Ori.* is the propagation only time. *METEOR Sim* is the time with simple source and sink.)

Benchmark	Anno-METEOR	Anno-Opt.	Anno-Phos	Speedup
avrora	3616ms	3209ms(12.6%)	3522ms	2.7%
batik	N/A	N/A	N/A	N/A
eclipse	N/A	N/A	N/A	N/A
fop	497ms	395ms(25.8%)	455ms	9.2%
h2	6580ms	5380ms(22.3%)	6007ms	11.6%
jython	3102ms	2852ms(8.8%)	N/A	N/A
luindex	1149ms	1193ms(-3.8%)	1008ms	-18.4%
lusearch	1510ms	1484ms(1.8%)	1532ms	3.2%
pmd	1973ms	1888ms(4.5%)	1858ms	-1.6%
sunflow	5039ms	3490ms(44.3%)	4878ms	39.8%
Tomcat	3085ms	2643ms(16.7%)	N/A	N/A
tradebeans	4403ms	3981ms(10.6%)	N/A	N/A
tradesoap	20874ms	18211ms(14.6%)	N/A	N/A
xalan	1019ms	827ms(23.2%)	920ms	5.5%

Table 3.9: Fully annotated benchmarking.(*Anno-METEOR.* is the time with optimizer off. *Anno-Opt.* is the time after optimization. *Anno-phos* is the time using Phosphor. *Speedup* is the final comparison between METEOR and Phosphor)

CHAPTER 4

Related Work

4.1 Memoization

4.1.1 Candidate Selection

Deciding that a pure method is worth memoizing is a difficult task. The cost of parameter comparison, result caching and retrieval, cache hit rate and method execution time contribute to the complexity. In [2], they calculate the execution time saved by memoization given the target method has a very specific functionality such as quick sort. But the overall performance is undecided due to the lack of estimation on other costs. Our work uses an once-and-for-all offline profiler to accurately estimate the potential speedup for each method.

4.1.2 Parameter Comparison

Parameter deep comparison is critical for retrieving the hashed result from hashmap. A typical deep comparison would require testing equality of transitively reachable variables between two objects. The cost of such deep comparison in Java could negate the potential memoization speedup. Several techniques have been proposed to solve this. Simply comparing object locations is the simplest one but lacks the soundness. Hash-Consing[13][28] is a technique that ensures there only exists one copy of the value such that object comparison is simplified to location comparison. While hash-consing alleviates comparison overhead, the cost it incurs is shown to be unacceptable in [23][24][10] due to garbage collection overhead and

large memory usage. In JavaMem, we ensure parameter immutability by static analysis. As a result, the cost of parameter comparison is constant without losing the soundness of memoization.

4.1.3 Partial Parameter Comparison

For a pure method, actual parameters are partially compared if and only if the subset of all transitively reachable values read by the method are compared. Such partial parameter comparison is sufficient for the purpose of memoization. Conventional parameter comparison technique would compare all transitive values. Such technique is over restrictive as a method usually depends on only a subset of those values. Dynamic analysis to compute the method-read subset is explored in [15][1] where a dependence set is computed dynamically and comparison is done by comparing only the values in those set. While their technique appears to be effective, the cost of dynamic analysis is hard to estimate. Additionally, there can be still a high cost if the subset is large. The work in [26] forces programmers to manually identify the subset, which is infeasible in practical use. Our approach takes advantage of static analysis to conservatively obtain a list of subset values.

4.1.4 Object Immutability Inference

Immutability has been studied in terms of both reference immutability and object immutability. Reference immutability is proposed in [6]. And the tools to infer reference immutability are developed in [9]. Reference immutability differs from object immutability in that it enforces transitively reachable values from a reference to be immutable with only the usage of that reference. For example, an object could be mutated in the first phase and then assigned to an immutable reference by which the object can not be mutated. Note that the object could be mutated if there are other mutable references. Even though reference immutability

provides useful information for program analysis such as pure method inference, it is insufficient to be used directly to serve our purpose. For example, an immutable reference as a parameter can be mutated if other mutable reference to the same object exists. Object immutability guarantees all transitively reachable values from the object is immutable after initialization phase. [20] presents a static analysis to infer immutable objects. Immutability in their work is defined in class context and any field write after constructor invocations renders the object immutable. This is restrictive in our use case when the object has a long initialization phase after constructor invocations. The work in [34] defines stationary fields whose all reads happen after all writes, which enables long initialization spanning multiple methods. They define that an object initialization ends when the object is stored into heap reference. As a result, their analysis does not track objects once objects are stored in heap. Although their analysis is highly scalable in large benchmarks, it loses analysis accuracy and analyzes only class field immutability. Our analysis is directed by pure method information and infers object field immutability with long initialization. Scalability is not an issue in our case since the pure method candidates are small in number.

4.2 Dynamic Taint Checking

Many promising dynamic taint checking systems have been designed and implemented in various ways. Among them, Dytan [8] provides both data-flow and control-flow dynamic taint analysis in x86 code. Their tool instruments all system-wide x86 executables without relying on customized runtimes. In their experiment, sources are non-hard-coded strings and sinks are database access functions. All sources are directly specified in binary code, which makes specifying taint policy difficult. They show up to 5.53x overhead on data-flow only taint analysis.

Phosphor [5] is closest to our work in that it is a portable taint analysis tool

that tracks data-flow taint in Java. METEOR differs from phosphor in three aspects: First, Phosphor transforms multi-dimensional array to a special wrapper class, which is problematic when the multi-dimensional array is passed into native code. Although in Dacapo benchmarks, no such case exists, we find Phosphor fail to instrument Hadoop. Second, Phosphor is implemented in ASM which is a bytecode rewriting library, while METEOR is implemented in Soot which takes advantage of Jimple instruction set. In fact, we find implementing propagation and optimization in METEOR is easier and less error-prone due to the small number of Jimple instructions compared to bytecode instructions. The soundness of dynamic taint analysis tool is rarely proved. Phosphor uses unit test to prove the correctness of taint propagation. We believe a static program prover should exist to formally prove the soundness of the tool and Jimple instruction set is better for such static proof. Third, Phosphor does not perform unnecessary taint tracking elimination optimization while METEOR provides several optimization techniques based on annotations.

Lift [25] is also a x86 dynamic taint tracking tool which adopts redundant propagation elimination optimizations. The most important optimization is Fast-Path optimization which generates a tracked version and untracked version in a single method body and switches between them. Our sensitive branching is similar to Fast-Path optimization and differs in two ways. First in Lift, branching checks happen together before a code segment(a block) and if one of the tag is unsafe, it jumps to the tracked version and will switch back in next segment. In METEOR, each check happens before every escape instruction, which allows more untracked instructions to execute. Once it jumps to the tracked version, it does not jump back. This is based on our experiments that 90.4% jump-backs would jump to tracked version again within the next two checks. Second, METEOR eliminates unnecessary branching checks in untracked version based on annotations. METEOR also eliminates propagation instructions if the method is

non-sensitive. Lift generates overhead from 1.06x to 3.60x.

TaintDroid [11] is an interpreter-based system-wide taint tracking system on DVM. It requires the modification of VM interpreter and like Phosphor, it does not perform optimizations on eliminating unnecessary propagation. It tracks array elements in a single tag, which incurs loss of precision. While it has an average overhead of 1.14x, the tracking accuracy and portability could be issues in general use.

CHAPTER 5

Conclusion

We design and implement JavaMem and METEOR to utilize JSR308 annotations for optimizations and improve performance on both micro and macro benchmarks. To the best of our knowledge, We are the first to combine Java 8 annotations with dynamic taint analysis to bring dynamic information flow analysis closer to practical use. And we are the first to explore memoization in macro benchmarks in Java and show execution speedup.

References

[1] Martín Abadi, Butler Lampson, and Jean-Jacques Lévy. Analysis and caching of dependencies. *SIGPLAN Not.*, 31(6):83–91, June 1996.

[2] Umut A. Acar, Guy E. Blelloch, and Robert Harper. Selective memoization. *SIGPLAN Not.*, 38(1):14–25, January 2003.

[3] Giovanni Agosta, Marco Bessi, Eugenio Capra, and Chiara Francalanci. Dynamic memoization for energy efficiency in financial applications. In *2011 International Green Computing Conference and Workshops, IGCC 2012, Orlando, FL, USA, July 25-28, 2011*, pages 1–8, 2011.

[4] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *SIGPLAN Not.*, 49(6):259–269, June 2014.

[5] Jonathan Bell and Gail Kaiser. Phosphor: Illuminating dynamic data flow in commodity jvms. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 83–101, New York, NY, USA, 2014. ACM.

[6] Adrian Birka and Michael D. Ernst. A practical type system and language for reference immutability. *SIGPLAN Not.*, 39(10):35–49, October 2004.

[7] Erika Chin and David Wagner. Efficient character-level taint tracking for java. In *Proceedings of the 2009 ACM Workshop on Secure Web Services*, SWS '09, pages 3–12, New York, NY, USA, 2009. ACM.

[8] James Clause, Wanchun Li, and Alessandro Orso. Dytan: A generic dynamic taint analysis framework. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ISSTA '07, pages 196–206, New York, NY, USA, 2007. ACM.

[9] Telmo Luis Correa, Jr., Jaime Quinonez, and Michael D. Ernst. Tools for enforcing and inferring reference immutability in java. In *Companion to the 22Nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion*, OOPSLA '07, pages 866–867, New York, NY, USA, 2007. ACM.

[10] Pascal Cuoq and Damien Doligez. Hashconsing in an incrementally garbage-collected system: A story of weak pointers and hashconsing in ocaml 3.10.2. In *Proceedings of the 2008 ACM SIGPLAN Workshop on ML*, ML '08, pages 13–22, New York, NY, USA, 2008. ACM.

[11] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association.

[12] Michael D. Ernst. Type Annotations specification (JSR 308). October 2011.

[13] Eiichi Goto and Yasumasa Kanada. Hashing lemmas on time complexities with applications to formula manipulation. In *Proceedings of the Third ACM Symposium on Symbolic and Algebraic Computation*, SYMSAC '76, pages 154–158, New York, NY, USA, 1976. ACM.

[14] Christian Haack and Erik Poll. Type-based object immutability with flexible initialization. In *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, Genoa, pages 520–545, Berlin, Heidelberg, 2009. Springer-Verlag.

[15] Allan Heydon, Roy Levin, and Yuan Yu. Caching function calls using precise dependencies. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pages 311–320, New York, NY, USA, 2000. ACM.

[16] Wei Huang, Ana Milanova, Werner Dietl, and Michael D. Ernst. Reim & reiminfer: Checking and inference of reference immutability and method purity. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 879–896, New York, NY, USA, 2012. ACM.

[17] Joxan Jaffar, Andrew E. Santosa, and Razvan Voicu. Efficient memoization for dynamic programming with ad-hoc constraints. In Dieter Fox and Carla P. Gomes, editors, *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, AAAI 2008, Chicago, Illinois, USA, July 13-17, 2008*, pages 297–303. AAAI Press, 2008.

[18] Vasileios P. Kemerlis, Georgios Portokalidis, Kangkook Jee, and Angelos D. Keromytis. Libdft: Practical dynamic data flow tracking for commodity systems. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*, VEE '12, pages 121–132, New York, NY, USA, 2012. ACM.

[19] Fredrik Kjolstad, Danny Dig, Gabriel Acevedo, and Marc Snir. Transformation for class immutability. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 61–70, New York, NY, USA, 2011. ACM.

[20] Yin Liu and Ana Milanova. Ownership and immutability inference for uml-based object access control. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 323–332, Washington, DC, USA, 2007. IEEE Computer Society.

[21] Rohan Padhye and Uday P. Khedker. Interprocedural data flow analysis in soot using value contexts. In *Proceedings of the 2Nd ACM SIGPLAN International Workshop on State Of the Art in Java Program Analysis*, SOAP '13, pages 31–36, New York, NY, USA, 2013. ACM.

[22] Donald E. Porter, Michael D. Bond, Indrajit Roy, Kathryn S. Mckinley, and Emmett Witchel. Practical fine-grained information flow control using laminar. *ACM Trans. Program. Lang. Syst.*, 37(1):4:1–4:51, November 2014.

[23] W. Pugh and T. Teitelbaum. Incremental computation via function caching. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, pages 315–328, New York, NY, USA, 1989. ACM.

[24] William Pugh. An improved replacement strategy for function caching. In *Proceedings of the 1988 ACM Conference on LISP and Functional Programming*, LFP '88, pages 269–276, New York, NY, USA, 1988. ACM.

[25] Feng Qin, Cheng Wang, Zhenmin Li, Ho-seop Kim, Yuanyuan Zhou, and Youfeng Wu. Lift: A low-overhead practical information flow tracking system for detecting security attacks. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pages 135–148, Washington, DC, USA, 2006. IEEE Computer Society.

[26] Hugo Rito and João Cachopo. Memoization of methods using software transactional memory to track internal state dependencies. In *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java*, PPPJ '10, pages 89–98, New York, NY, USA, 2010. ACM.

[27] Sooel Son, Kathryn S. McKinley, and Vitaly Shmatikov. Diglossia: detecting code injection attacks with precision and efficiency. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, CCS '13, pages 1181–1192, New York, NY, USA, 2013. ACM.

[28] Jay M. Spitzen, Karl N. Levitt, and Lawrence Robinson. An example of hierarchical design and proof. *Commun. ACM*, 21(12):1064–1075, December 1978.

[29] Manu Sridharan, Shay Artzi, Marco Pistoia, Salvatore Guarnieri, Omer Tripp, and Ryan Berg. F4f: Taint analysis of framework-based web applications. In *Proceedings of the 2011 ACM International Conference on Object*

Oriented Programming Systems Languages and Applications, OOPSLA '11, pages 1053–1068, New York, NY, USA, 2011. ACM.

[30] Alexandru Sălcianu and Martin Rinard. Purity and side effect analysis for java programs. In *Proceedings of the 6th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI'05, pages 199–215, Berlin, Heidelberg, 2005. Springer-Verlag.

[31] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. *SIGARCH Comput. Archit. News*, 32(5):85–96, October 2004.

[32] Mohit Tiwari, Xun Li, Hassan M. G. Wassel, Frederic T. Chong, and Timothy Sherwood. Execution leases: A hardware-supported mechanism for enforcing strong non-interference. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 493–504, New York, NY, USA, 2009. ACM.

[33] Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. Taj: Effective taint analysis of web applications. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 87–97, New York, NY, USA, 2009. ACM.

[34] Christopher Unkel and Monica S. Lam. Automatic inference of stationary fields: A generalization of java's final fields. *SIGPLAN Not.*, 43(1):183–195, January 2008.

[35] Shiyi Wei and Barbara G. Ryder. Practical blended taint analysis for javascript. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, pages 336–346, New York, NY, USA, 2013. ACM.

[36] Alexander Yip, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. Improving application security with data flow assertions. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 291–304, New York, NY, USA, 2009. ACM.

[37] Lukasz Ziarek, KC Sivaramakrishnan, and Suresh Jagannathan. Partial memoization of concurrency and communication. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, ICFP '09, pages 161–172, New York, NY, USA, 2009. ACM.