Alberto Lovato

Concurrency and Static Analysis

Ph.D. Thesis

Università degli Studi di Verona
Dipartimento di Informatica

Advisor:
Prof. Fausto Spoto

Università di Verona
Dipartimento di Informatica
Strada le Grazie 15, 37134 Verona
Italy

Contents

1 Introduction

Computers are complex machines, crafted to work reliably in today's busy world. But computers need software to do something more than heating the room they are housed in, and software has to be reliable as well. Building reliable software ultimately reduces to good programming skills.

Programming can be a pleasant activity at times, but it is also undoubtedly difficult. Fortunately, the programmer is not alone in his work. This is a great time to program, with plenty of development tools, methodologies, the Internet and the like. Among these goodies, automatic software verification is emerging as a necessary step in the development life cycle.

The complexity of programming comes from multiple factors, but there is one fundamental reason that drives all the research in software engineering practices and automatic verification: the human factor.

Where It All Begins

It all begins in the brain. Despite the mechanical nature of some tasks, programming is still mainly a human activity. The brain works abstractly like a CPU: it is equipped with a little, super fast *short-term* memory (a cache!), and a relatively big and slow *long-term* memory.[1] Brain's computations work on the short-term memory, that unfortunately is quite small. If something is not in this memory, it has to be recovered from the slow long-term memory. When the brain is forced to focus on different or complex tasks, the "cache" is completely reloaded. Big context switches can last several minutes, and cause performance degradation [66].

This preliminary discussion on how the brain works is related to programming, as a programmer's brain can focus only on a portion of code or

[1] this is a simplification: modern theories about how the mind works attribute to the *working memory* system the duty of assisting thought; for an overview of the working memory, see [24]; for a general description of the mainstream theories regarding memory, see [40]

functionality at a time. This is why software engineering methodologies and paradigms exist, that help programmers to tame software complexity. For instance, applying object oriented programming principles increases modularity (hence reducing coupling), and allows programmers to focus their attention on small pieces of code (classes). With such countermeasures, programmers can cope reasonably well with complex software. But there's another dimension that puts reason at risk: concurrency.

Concurrency

When two entities act at the same time, we say that they are concurrent. In particular, we will consider *threads* acting on *shared memory*, on *multi-core* machines.

Concurrency is too important in computing systems to ignore it. Even when the user is performing a single task, many threads execute transparently to maintain user interface responsiveness, or to avoid blocking waiting for I/O. It became even more important in recent years, as the microprocessor industry pushed *multi-* and *many-core* hardware as the only viable alternative to maintain the performance growth trend expected by customers. Software developers are actively involved in this “multithreaded revolution”. They can no more wait for the next-generation processor to multiply sequential performance: they have to carefully design applications to exploit hardware parallelism.

Consistency

As an example of problems arising in multithreading, a thread can start updating a data structure, and then another thread can sneak in and see the same structure in an inconsistent state. This inconsistency can be solved by imposing that the second thread can act on the structure only when the first finishes. In other words, the set of updates performed by the first thread must be *atomic*. This was a *race condition*: the outcome of the computation depends on non-deterministic events, like thread scheduling. Bugs resulting from race conditions are often difficult, or impossible, to reproduce in a testing environment. Their inclination to disappear when a programmer tries to reproduce them—by inserting logging, or statements changing the timing of events—led to call them *heisenbugs* [63], in honor of the physicist Werner Heisenberg, who asserted the principle that observing a system alters its state.

Somehow related to race conditions are *data races*, happening when more threads access the same memory location concurrently, without synchronization, and at least one of them writes to that location.

While data races are relatively easy to detect syntactically with automatic procedures (accesses to shared locations must be protected by synchronization), race conditions are dependent on the *semantics* of the program: the programmer assumes that certain sequences of operations are not interrupted.

A simple race condition that can go unnoticed is the increment: `i++`. This is a single statement, but it is not atomic, and can be interrupted by a concurrent thread. If the initial value of `i` is 0, two concurrent increments can result in a final value of 1. This is a *time of check to time of use*, or *TOCTOU*, race condition.

Race conditions and data races can both be solved with the use of locks. But locking causes problems as well. Apart from performance issues—excessive locking turns a parallel program into an almost sequential one—*deadlocks* and *livelocks* can prevent a program to progress. Deadlocks can occur when there is a circular dependency on locks, for example when two threads acquire locks in reverse order. Each thread can thus be blocked, waiting to acquire a lock held by the other. Livelocks are different in that threads are not blocked, but still cannot progress, as they execute the same circular code.

Visibility

As if all this complexity were not enough, the reality of processor and compiler technology poses other threats to programmers' sanity. Not only different threads need to consistently manipulate shared data structures, but they can even see and work on different copies of the same structure. And the execution order of instructions can be different from what the programmer expects. Hence, there are also *visibility* problems.

These phenomena are explained by transformations performed by modern processors' architectures and optimizing compilers. In multi-core systems, variables can be stored in registers, or in caches private to a core, and thus become invisible to threads running on different cores. Moreover, independent instructions can be profitably reordered, if this does not invalidate the intra-thread semantics. These optimizations are mandatory, in order to maintain an acceptable performance, but can lead to incorrect semantics for multithreaded programs, that is to say, they break *sequential consistency* [70]. Lamport defined this model of consistency using these words: "The result of any execution is the same as if the operations of all processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.", *i.e.*, according to the *program order*. So, to restore program correctness, we need to ensure that shared memory locations are accessed only with the protection of *memory barrier* instructions.

A *memory model* is a set of rules describing the allowed interactions of threads with the shared memory, and the rules of visibility. The original Java specification included a memory model, that was flawed and then fixed in version 1.5. For C and C++, a memory model was not provided until the 2011 standards.

Reasoning about Concurrency

Multithreaded programs are much more difficult to reason with than simple, sequential ones. When a programmer looks at a program's code, threads are not explicit. He sees program statements or expressions, even those regarding concurrency, but he can hardly tell which thread actually executes a section of code, when it does, and which data structures are used by more threads. All concurrency reasoning typically happens in the programmer's mind, and in any non-trivial multithreaded program there are far too many possible interactions to be manageable.

Practices and abstractions have been developed to reduce the impact of the exploding complexity of concurrency. The already mentioned object oriented principles come to rescue here too: *encapsulation* limits data visibility, and so assists programmers in restricting the focus of reasoning. Another thing that really can make a difference is to share only *immutable* data: a resource that cannot change is safe to use by multiple threads. This even eliminates the need to reason about thread interactions.[2] This paradigm is implemented in so called *shared-nothing* frameworks, such as those based on the *actor model*, like Akka [1], in which concurrency is based on actors operating on immutable messages, or *MapReduce*, like Hadoop [3], used for parallel processing of large amounts of data. Also *functional languages* encourages immutability, with the use of *pure functions*, *i.e.*, functions without side effects. These exhibit the *referential transparency* property: they can be substituted with their evaluation in every occurrence, without changing the program's behavior.

The Java Case

Java is widely used as a general purpose programming language. It is often supported by an execution environment, in order to give developers an abstraction over the physical machine where the code runs. Concurrency in Java was included since the first version, with thread creation and management, mutual exclusion locks, atomic variables and other features. The multi-platform nature of Java entails that the way Java programs behave should be the same on all architectures. This is accomplished by providing high level instructions that hide low level details, like memory barriers, or atomic instructions.

The Java Language Specification [62] describes the *memory model* that can be assumed by programmers. It trades *sequential consistency*—a luxury we can't afford—with weaker rules, imposing visibility only in presence of *synchronization actions*. The weakness of the model makes reasoning about concurrency considerably harder, but there are good news: we don't need to worry about visibility if our programs are free from data races!

[2] on the other hand, enforcing immutability is not trivial, and often using only immutable resources is not feasible

Contributions

This thesis has three main contributions:

- the development of a fast multi-threaded Java library for the manipulation of binary decision diagrams,
- a static analysis for the identification of injection vulnerabilities, whose implementation uses our multi-threaded library to perform several analysis in parallel,
- a static analysis for the verification and inference of the locking discipline of concurrent programs.

Each of them has been published in conference proceedings.

The first one [75] (*SEFM* 2014) describes an early version of the multi-threaded BDD library. It shows details of the implementation, as well as comparisons with other Java and C libraries, in single- and multi-threaded environments, concluding that our library is fast and consumes less memory.

The analysis of injections was published in [47] (*LPAR-20* 2015). It is an example of denotational analysis. The results of the analysis, implemented in the Julia program analyzer, are compared to those produced by three other tools, for a set of well established security benchmarks. In all cases Julia performed reasonably fast, and found all the vulnerabilities exhibited by the benchmark programs, along with some false positives.

The locking inference is described in [48] (*ICSE* 2016). It uses a novel *definite locked expressions analysis*, a constraint-based analysis, in which an *abstract constraint graph* is constructed and then solved to find expressions that are definitely locked at some program point. The paper shows how the locking inference compares with the type checking performed by the *Lock Checker* [7].

1.1 BDD Concurrent Library

Binary decision diagrams, or BDDs, are data structures for the representation of Boolean functions. These functions are of great importance in many fields. Symbolic model checking [35] is a method for the verification of finite state systems that uses Boolean functions. Software static analysis also may use Boolean functions, to represent transition or relations of properties on program variables. All these practical applications benefit from a compact and efficient representation.

It turns out that BDDs are the state-of-the-art representation for Boolean functions, and indeed all real world applications use a BDD library to represent and manipulate Boolean functions. As we will see in subsequent chapters, it can be desirable to perform Boolean operations from different threads at the same time. In order to do this, the BDD library in use must allow threads to access BDD data safely, avoiding race conditions. We developed a Java

BDD library, that is fast in both single and multi-threaded applications, that we use in the Julia static program analyzer.

1.2 Identification of Injection Vulnerabilities

Dynamic web pages and web services react to user input coming from the network, and this introduces the possibility of an attacker *injecting* special text that induces unsafe, unexpected behaviors of the program. Injection attacks are considered the most dangerous software error [79] and can cause free database access and corruption, forging of web pages, loading of classes, denial-of-service, and arbitrary execution of commands. Most analyses to spot such attacks are dynamic and unsound (see Sec. 3.2).

We defined a sound static analysis that identifies if and where a Java bytecode program lets data flow from *tainted* user input (including servlet requests) into critical operations that might give rise to injections. Data flow is a prerequisite to injections, but the user of the analysis must later gage the actual risk of the flow. Namely, analysis approximations might lead to false alarms and proper input validation might make actual flows harmless.

Our analysis works by translating Java bytecode into Boolean formulas that express all possible explicit flows of tainted data. The choice of Java bytecode simplifies the semantics and its abstraction (many high-level constructs must not be explicitly considered) and lets us analyze programs whose source code is not available, as is typically the case in industrial contexts that use software developed by third parties, such as banks.

In this thesis we describe:

- an object-sensitive formalization of taintedness for reference types, based on reachability of tainted information in memory;
- a flow-, context- and field-sensitive static analysis for explicit flows of tainted information based on that notion of taintedness, which is able to deal with data dynamically allocated in the heap (not just primitive values);
- its implementation inside the Julia analyzer, through binary decision diagrams, and its experimental evaluation.

Sec. 8.4 shows that our analysis can analyze large real Java software. Compared to other tools available on the market, ours is the only one that is sound, yet precise and efficient. Our analysis is limited to explicit flows [93]; as is common in the literature, it does not yet consider implicit flows (arising from conditional tests) nor hidden flows (such as timing channels).

1.3 Locking Discipline Inference

The standard approach to prevent data races is to follow a locking discipline while accessing shared data: always hold a given lock when accessing a given shared datum. It is all too easy for a programmer to violate the locking discipline. Therefore, tools are desirable for formally expressing the locking discipline and for verifying adherence to it [38, 74].

The book *Java Concurrency in Practice* [59] (JCIP) proposed the `@Guard-edBy` annotation to express a locking discipline. This annotation has been widely adopted; for example, GitHub contains about 35,000 uses of the annotation in 7,000 files (`https://github.com/search?l=java&q=GuardedBy&type=Code`).

`@GuardedBy` takes an argument indicating which lock should be held, which can be

- *this*, the object containing the field
- *itself*, the object referenced by the field
- the name of a field referencing the lock object (prefixed by the class name if static)
- the name of a nullary method returning the lock object
- the name of a class (like "`String.class`") specifying the `Class` object to use as the lock object.

A class name can be used as a prefix to disambiguate *this* for inner classes.

The original `@GuardedBy` annotation was designed for simple intra-class synchronization policy declaration. `@GuardedBy` fields and methods are supposed to be accessed only when holding the appropriate lock, referenced by another field, in the body of the class (or *this*). In simple cases, a quick visual inspection of the class code performed by the programmer is sufficient to verify the synchronization policy correctness. However, when we think deeper about the meaning of this annotation, and when we try to check and infer it, some ambiguities rise.

In Java, a field can be of a primitive type—like `int`, `boolean`, ...—or of a reference type. In the latter case, the field is a simple pointer to the real object. This leads to *aliasing* phenomena: the same object can be accessed from different, unrelated places, such as through fields or variables belonging to different objects. This can happen if the field was assigned an object from the outside, or if the initialized field value *escapes* from the object.

So in Java we really have two kinds of *accesses*:

- accesses to the variable content, that can be a primitive value or a reference, in the form of *variable reading and writing*
- accesses to the referenced object, in the form of *field dereferencing* ("dot" accesses)

This raises the question: when we say that a variable is `@GuardedBy`, which accesses need to be guarded with synchronization? If we limit our reasoning to private field reading and writing, synchronization is local to the object (or class, for static fields). This is the case of the `AtomicCounter` class in Figure 1.1. Here, the value of the primitive field `count` can only be read or written in the three *synchronized* methods—in other words, only when holding

```
public class AtomicCounter {
    @GuardedBy("this")
    private int count;

    public synchronized void increment() {
        count++;
    }

    public synchronized void decrement() {
        count--;
    }

    public synchronized int getValue() {
        return count;
    }
}
```

Fig. 1.1: A thread-safe counter

the lock of the instance. Checking the annotation correctness is very simple in this case: every occurrence of the field `count` needs to be in a critical section with the "`this`" guard. The same simple reasoning applies to "dot" accesses to private fields—and we mean *really* private: no alias.

Things get complicated when we lose this synchronization locality. Let's make the field `count` *public*. Now there is no access control on reading and writing, and the synchronization policy can be circumvented. Synchronization checking is no more intra-class: we need to check that every access to the field in the whole program is guarded. These accesses can be performed through multiple aliases of the container object. The other case in which we lose locality is when we have external aliases of objects referenced by the field, and we consider *dot* accesses, that cause dereferencing.

And there is an additional problem: `@GuardedBy` annotations tie the guarded field to its lock object, and so, whenever an external synchronization is needed, also the lock object must be accessible from the outside. This sort of checking requires alias analysis, to find all aliases of the container object, of objects referenced by guarded fields, and objects used as guards.

`@GuardedBy` is applicable also to methods, meaning that when a method is called, the lock specified by the annotation should be held. The *Julia Analyzer* [13] also recognizes the `@Holding` annotation, which has the same meaning of `@GuardedBy`, but clearly states its applicability to methods. It also allows `@GuardedBy` to be applied to method parameters, local variables, return values, and other types.

The New Definition

Given this ambiguity of the specification for `@GuardedBy`, different tools interpret it in different ways [82, 89]. Moreover, it does not prevent data races, thus not satisfying its design goals. We provide a formal specification that satisfies its design goals and prevents data races. We have also implemented our specification in the Julia analyzer, that uses abstract interpretation to infer valid `@GuardedBy` annotations for unannotated programs. Our technique is not specific to Java and generalizes to other languages. It is not the goal of this implementation to detect data races or give a guarantee that they do not exist. Julia determines what locking discipline a program uses, without judging whether the discipline is too strict or too lax for some particular purpose.[3]

In an experimental evaluation, we compared this tool to programmer-written annotations. Our evaluation shows that programmers who use the `@GuardedBy` annotation do not necessarily do so consistently with JCIP's rules, and even when they do, their programs still suffer data races.

The most important problem with JCIP's definition is that it provides name protection rather than value protection [37]. Name protection is fine for primitive values, which cannot be aliased in Java. Value protection is needed in order to prevent data races on reference values, due to aliasing and because the Java Language Specification defines locking in terms of values rather than names [62]. Unfortunately, most tools that check `@GuardedBy` annotations use JCIP's inadequate definition and therefore permit data races. Our definition prevents data races by providing value protection: if a reference r is guarded by E, then for any value v stored in r, v's fields are only accessed while the lock E is held. (At run time, a lock expression `E` is held by a given thread at a given time if `java.lang.Thread.holdsLock(E)` evaluates to true on that thread at that time.) Inference of this definition requires tracking values v as they flow through the program, because the value may be used through other variables and fields, not necessarily r. Since this is relevant for reference values only, we consider value protection for reference variables and fields only.

[3] The desired locking discipline is unknowable: some data races are benign, a programmer may intend locking to be managed by a library or by clients, locking may not be necessary for objects that do not escape their thread, etc.

2

Background

This chapter provides the theoretical background for the rest of the thesis.

2.1 Boolean Functions

A *Boolean algebra* [78] is composed by a set, two binary operations, a unary operation and two neutral elements, $B = (A, +, \cdot, ^-, 0, 1)$, that satisfy the laws of

- Commutativity $a + b = b + a$, $a \cdot b = b \cdot a$
- Distributivity $a \cdot (b + c) = (a \cdot b) + (a \cdot c)$, $a + (b \cdot c) = (a + b) \cdot (a + c)$
- Identity $a + 0 = a$, $a \cdot 1 = a$
- Complement $a + \overline{a} = 0$, $a \cdot \overline{a} = 1$

A *Boolean function* is a function that can be represented by a *Boolean formula*, which is an expression over a Boolean algebra. Many Boolean formulas can represent the same Boolean function. There are many examples of Boolean algebras—such as the structure $(2^S, \cup, \cap, ^-, \emptyset, S)$, that represents the *set algebra* for a set S. But the most interesting cases are Boolean algebras whose set contains two elements. In these cases, the set is commonly written as $\mathbb{B} = \{0, 1\}$, and the algebra can be denoted by $\{\mathbb{B}, +, \cdot, ^-, 0, 1\}$. The $+$ and $\cdot$ operators represent the familiar "or" (disjunction) and "and" (conjunction) Boolean operations, $^-$ is the logical negation, and 0, 1 are the "false" and "true" constant values. A *switching function* f is a Boolean function defined as $f : \mathbb{B}^n \to \mathbb{B}$. Hence, a switching function maps bit vectors to bits. There are 2^{2^n} switching functions with n inputs. In the following, the terms switching function and Boolean function will be used interchangeably.

2.1.1 Shannon Expansion

Shannon expansion, also known as Boole's expansion theorem, allows the decomposition of a switching function f in terms of the two subfunctions

$g = f_{|x_i=0}$ and $h = f_{|x_i=1}$—meaning that g is the function obtained from f by fixing the value of x_i to 0, and h is the function obtained from f by fixing the value of x_i to 1. The Shannon expansion of the function f is thus

$$f = \overline{x_i} \cdot g + x_i \cdot h$$

g is called the *negative cofactor* of f, and h is the *positive cofactor*. Shannon expansion will be used in Section 4.1 to define binary decision diagrams.

2.1.2 Representations of Boolean Functions

The way we represent switching functions has a big impact on the memory footprint and on the efficiency of algorithms operating on such representations. In this section we describe various representation types, each with its own strengths and weaknesses. Among all, binary decision diagrams offer a compact and efficient representation, suitable for high performance computations.

Truth Tables

A *truth table* is an enumeration of the value of a function for every assignment to the input variables. This representation can be effortlessly evaluated, and two truth tables can be easily combined with Boolean operations. The problem with this representation type is the always exponential size—in the arity of the function. An example of a truth table for the conjunction of two variables is shown in Figure 2.1.

x	y	$x \cdot y$
0	0	0
0	1	0
1	0	0
1	1	1

Fig. 2.1: The truth table for the function $x \cdot y$

Normal Forms

Normal forms represent switching functions in terms of expressions constructed from literals with Boolean operators. Normal forms are written in *levels*, where a single operator is used in each level. Not all switching functions can be represented by one-level normal forms. An example of a normal form with one level is the *parity function*, $x_1 \oplus \ldots \oplus x_n$, where $\oplus$ denotes the *exclusive or* operation. Two-level normal forms can represent all of the switching functions. A *monomial* is the combination of literals by means of a single operator (it's a one-level normal form). So, for example, a $\cdot$ -monomial can be $x_1\overline{x_2}$, a $+$ -monomial $x_3 + \overline{x_5} + x_6$, and a $\oplus$ -monomial $x_0 \oplus x_1 \oplus \overline{x_2}$. $\cdot$ -monomials are called simply *monomials*, whereas $+$ -monomials are called *clauses*.

Disjunctive normal forms, or DNF, are $+$ -products of $\cdot$ -monomials. Similarly, *conjunctive normal forms*, or CNF, can be defined as $\cdot$ -products of $+$ -monomials, and *parity normal forms*, or PNF, as $\oplus$ -products of $\cdot$ -monomials. Given an assignment $a = (a_1, \ldots, a_n) \in \mathbb{B}^n$ for the Boolean variables $x = (x_1, \ldots, x_n)$, the *minterm* of a is

$$m_a(x) = x_1^{a_1} \cdot \ldots \cdot x_n^{a_n}$$

and the *maxterm* of a is

$$s_a(x) = x_1^{\overline{a_1}} + \ldots + x_n^{\overline{a_n}}$$

Here, $x^1 = x$ and $x^0 = \overline{x}$.

With all these definitions in place, we can then define for every Boolean function f the *canonical disjunctive normal form*, cDNF, as

$$f(x) = \sum_{a \in \mathrm{on}(f)} m_a(x)$$

and the *canonical conjunctive normal form*, cCNF, as

$$f(x) = \prod_{a \in \mathrm{off}(f)} s_a(x)$$

where on(f) is the *on-set* of f—the set of all satisfying assignments of f—and off(f) is the *off-set* of f—the set of all non-satisfying assignments of f.

Truth tables, cDNF and cCNF are examples of canonical representation types. A representation type for switching functions is called *canonical* if every function has exactly one representation of this type. This entails that two equivalent functions are described by the same representation, and so equivalence test can be performed easily: it suffices to compare two representations for equality. In the absence of canonicity—for instance for generic DNF and CNF—equivalence testing is hard. In fact, given two representations in disjunctive (conjunctive) normal form, testing for their equivalence is a *co-NP*-complete problem. In other terms, testing if two DNF (CNF) represent *different* functions is *NP*-complete [78]. Another canonical form is the *ring sum expansion* (RSE), a particular PNF where every literal is positive. A drawback of RSE is the exponential size of the representation even for simple functions like the disjunction of variables.

Circuits and Formulas

Augmenting the number of levels of normal forms, the compactness of the representation increases as well. Given a set Ω of basic operations (a *basis*), an *Ω-circuit* S is an acyclic graph with its nodes categorized as

- input nodes, labeled with a variable or a constant, 0, 1,
- function nodes, labeled with a basic operation in Ω,
- output nodes, representing functions of interest.

Each node of the circuit represents a switching function. Figure 2.2 shows the circuit representing a *full adder*, a switching function that computes the sum of the bits x, y and c (the *carry bit*). The *depth* of a circuit, the number

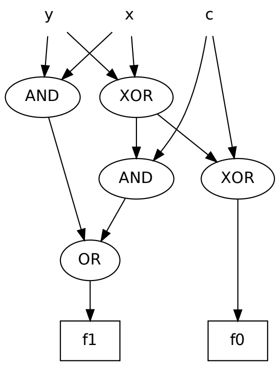


Fig. 2.2: Circuit of a full adder

of levels, corresponds roughly to its execution time. A basis Ω is *complete* if Ω-circuits are a universal representation type, *i.e.*, every switching function can be represented as an Ω-circuit. For example, the *standard basis* $\{+, \cdot, ^{-}\}$ is complete, as it can be used to construct a three-level circuit corresponding to a DNF. As another example, the basis $\{\oplus, \cdot\}$ is complete, as each switching function can be represented by a RSE. Even if circuits are not a canonical representation type, they are in most cases substantially more compact than normal forms. This compactness is due to the ability of circuits to use a function as the input to more than one node—that is to say, the outdegree of a node can be greater than one. The fact that in circuits an intermediate function can be used more than once complicates the algorithmic handling, and so it may be undesirable. A circuit whose nodes have outdegree 1 is called a *formula*. As such, a formula consists of a collection of trees, each rooted in an output node. Figure 2.3 presents a formula for the full adder. Checking for

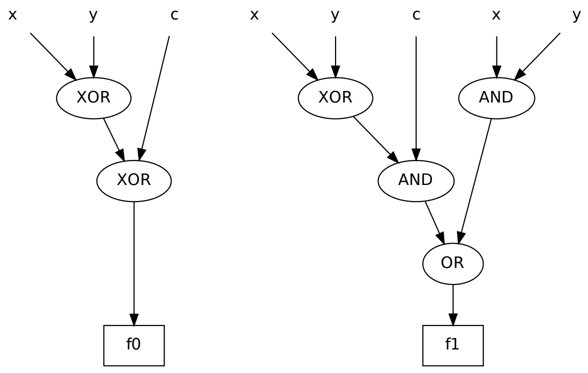


Fig. 2.3: A full adder represented as a formula

equivalence is hard (*co-NP*-complete) for circuits and formulas, too.

2.2 Concurrency in Java

Since the beginning, the Java language included facilities for cross-platform concurrency, in the form of abstractions modeling certain desired behaviours.

First of all, *threads* are taken into account at the language level and supported as first-class entities. In other languages, C for example, multi-threading is an additional feature, delivered by external, implementation-specific libraries (e.g. POSIX Threads). Java threads can be created as instances of the `Thread` class, or used indirectly via other library classes like `java.util.Timer`, or by means of new abstractions introduced starting from the version 5 of the language, like *executors*. Moreover, frameworks such as Spring [16], or Android [2], create threads behind the scenes.

This makes it relatively easy to start new threads and take advantage of today's ubiquitous multicore hardware. The main challenge is to manage interactions between threads, as these can introduce bugs that are hard to find. More precisely, what can cause problems is *mutable state* of objects, *i.e.*, when the values of fields can be modified after object creation. On the other way, immutable objects can be safely accessed by different threads.

The Java language provides two main keywords for dealing with mutable state: `synchronized`, to delimit blocks of code or entire methods as critical sections, and `volatile`, to denote fields that need to be shared among threads.

Synchronized blocks and methods define mutual exclusion sections of code, that can be accessed by only one thread at a time. Every Java object has an *intrinsic lock*, that can be used by synchronized blocks as a monitor to allow execution by only one thread at a time.

Volatile fields can be accessed by multiple threads safely, meaning that their value is consistent from one thread to another.

Mutual exclusion is not sufficient to ensure thread-safety [59]. Modern multicore processors employ optimizations that can invalidate standard assumptions of correctness of sequential code. They can for example cache variables in registers or in cache memory local to a core, making their value invisible to other cores—and to threads running on them. Moreover, instructions can be reordered, providing that single threaded semantics is still valid.

Hence, there is the additional problem of the *visibility* of shared data. Volatile variables are not cached in registers or local core memory, they are shared between threads, and so they always contain the most recently written value. Synchronized blocks provide mutual exclusion and visibility, volatile variables only visibility.

The programmer developing concurrent software on modern processors needs to be aware of these hardware features to produce correct code. Java cross-platform nature led to insert in the language specification rules dictating particular ordering in operations regarding shared memory. The so called Java Memory Model (JMM) describes the interaction of threads with the main memory.

2.2.1 The Java Memory Model

The Java Language Specification [62] defines the relations *synchronizes-with* and *happens-before* between actions:

- *happens-before* is an ordering between sections of code, imposing restrictions on when a section can start executing.
- *synchronizes-with*—that implies *happens-before*—refers to synchronization with main memory; *lock* and *unlock* operations are ordered according to this relation (an *unlock* on a monitor comes before a *lock* on the same monitor).

For example a write to a volatile variable *synchronizes-with* all subsequent reads. This means that accesses to a volatile variable act on its most recent version.

So, exiting from a synchronized block (*unlock*) or writing to a volatile variable have the additional effect of restoring visibility of all variables visible up to that point. Entering in a synchronized block (*lock*) is like reading a volatile variable—they come after unlock or write.

In addition to an intrinsic lock, every object has an associated wait set of threads. When in a thread the method `wait()` is invoked, the thread blocks, and may wake up when another thread calls `notify()` (wake up a thread) or `notifyAll()` (wake up all threads in the set).

2.3 Abstract Interpretation

Abstract interpretation is a theory of sound approximation of the semantics of programs based on monotonic functions over posets, in particular lattices. This theory is mostly used in static analysis, offering compile-time techniques for producing safe and computable approximations to the set of values or behaviors arising dynamically at run-time when executing a program. Starting from a concrete domain C (the domain used by the conventional semantics) we can obtain, through an abstraction, an abstract domain A, capturing some (but not all) properties of the concrete objects.

Definition 2.1. *A partial order $\leq$ on a set X is a binary relation $\leq$ on a set X which is*

- *reflexive:* $\forall x \in X.\, x \leq x$
- *antisymmetric:* $\forall x, y \in X.\, (x \leq y \wedge y \leq x) \Rightarrow x = y$
- *transitive:* $\forall x, y, z \in X.\, (x \leq y \wedge y \leq z) \Rightarrow x \leq z$

Definition 2.2. *A partially ordered set, or poset, $\langle X, \leq \rangle$ is a set X with a partial order $\leq$ on it.*

Definition 2.3. *A chain of a poset $\langle X, \leq \rangle$ is a totally ordered subset of X,* i.e.*, a set $C \subseteq X$, such that $\forall x, y \in C.\, x \leq y \vee y \leq x$*

Definition 2.4. *A poset $\langle X, \leq \rangle$ satisfies the ascending chain condition, or ACC, iff any infinite sequence $x_0 \leq x_1 \leq \ldots \leq x_n \leq \ldots$ is not strictly increasing,* i.e.*, $\exists k \geq 0.\, \forall j \geq k.\, x_j = x_k$.*

Definition 2.5. *A poset $\langle X, \leq \rangle$ satisfies the descending chain condition, or DCC, iff any infinite sequence $x_0 \geq x_1 \geq \ldots \geq x_n \geq \ldots$ is not strictly decreasing,* i.e.*, $\exists k \geq 0.\, \forall j \geq k.\, x_j = x_k$.*

Definition 2.6. *A poset $\langle X, \leq \rangle$ can have*

- *a top or maximum element $\top \in X$ such that $\forall x \in X.\, x \leq \top$*
- *a bottom or minimum element $\bot \in X$ such that $\forall x \in X.\, \bot \leq x$*

Definition 2.7. *Let $\langle X, \leq \rangle$ be a poset and $S \subseteq X$.*

- *$M \in X$ is an upper bound of S iff $\forall x \in S.\, x \leq M$.*
  The least upper bound of $\langle X, \leq \rangle$ is denoted by $lubX, supX, \bigvee X, \bigsqcup X$.
  $x \sqcup y \triangleq \bigsqcup\{x, y\}$.
- *$m \in X$ is a lower bound of S iff $\forall x \in S.\, m \leq x$.*
  The greatest lower bound of $\langle X, \leq \rangle$ is denoted by $glbX, infX, \bigwedge X, \bigsqcap X$.
  $x \sqcap y \triangleq \bigsqcap\{x, y\}$.

Definition 2.8. *A join semi lattice $\langle X, \leq, \sqcup \rangle$ is a poset such that for any $x, y \in X$ there exists $x \sqcup y$.*

Definition 2.9. *A meet semi lattice $\langle X, \leq, \sqcap \rangle$ is a poset such that for any $x, y \in X$ there exists $x \sqcap y$.*

Definition 2.10. *A lattice $\langle X, \leq, \sqcup, \sqcap \rangle$ is both a join and meet semi lattice.*

Definition 2.11. *A complete lattice is a poset $\langle X, \sqsubseteq \rangle$ such that for every $S \subseteq X$ its lub $\bigsqcup S \in X$.*

Definition 2.12. *Let $f\colon X \to X$ be an operator on a poset $\langle X, \sqsubseteq \rangle$. $x \in X$ is a fixpoint of f if $f(x) = x$. The least fixpoint of f is denoted by $lfp(f)$. The greatest fixpoint of f is denoted by $gfp(f)$.*

With *abstract interpretation* it is called a theory by which a concrete semantics of a program is *abstracted* into a simpler one, that doesn't exhibit the original behavior, but represents instead only the property of interest. So a *concrete domain* C, used by the conventional semantics, is transformed into an *abstract domain* A through an abstraction function.

Definition 2.13. *Let C be the concrete domain and A the abstract one. A function $\alpha\colon C \to A$ is an abstraction function. A function $\gamma\colon A \to C$ is a concretization function.*

Definition 2.14. *Given two posets $\langle C, \leq_C \rangle$ and $\langle A, \leq_A \rangle$, and given two functions $\alpha\colon C \to A$ and $\gamma\colon A \to C$, $\langle C, \alpha, \gamma, A \rangle$ is a Galois connection (GC), and we denote it with $C \underset{\alpha}{\overset{\gamma}{\rightleftarrows}} A$, if*

- *α and γ are monotone*
- *$\forall c \in C.\, c \leq_C \gamma(\alpha(c))$*
- *$\forall a \in A.\, \alpha(\gamma(a)) \leq_A a$*

Alternatively, $\langle C, \alpha, \gamma, A \rangle$ is a Galois connection if

- *α and γ are monotone*
- *it is an adjunction: $\forall a \in A. \forall c \in C.\, \alpha(c) \leq_A a \Leftrightarrow c \leq_C \gamma(a)$*

Definition 2.15. *If $\alpha \circ \gamma = \iota_A$, we call the GC a Galois insertion, and we denote it as $C \underset{\alpha}{\overset{\gamma}{\twoheadleftarrow\!\!\rightarrow}} A$*

3

State of the Art

3.1 Binary Decision Diagrams

3.1.1 Existing Implementations

There are many alternatives for BDD manipulation in the Java world. First of all, native code libraries written in C—such as BuDDy [5], CUDD [8] and CAL [6]—can be used by invoking their functions via the Java Native Interface (JNI). This approach has a series of shortcomings: native libraries are not cross platform, and so a version for every execution environment has to be provided; they cannot be used concurrently, nor multiple factories can be created; then it may be difficult to adapt their C programming interface to a friendly object oriented one—even if, as explained below, JavaBDD already includes interfaces to these native libraries. Then there are pure Java libraries, like JavaBDD [11], JDD [12] or SableJBDD [14]. JavaBDD seems to be the library of choice for the Java world. It is heavily inspired by BuDDy, and includes a factory implementation that is a direct translation of BuDDy in Java. It offers also other factories that are interfaces to BuDDy, CAL, CUDD and JDD. JDD performs well with problems involving only simple Boolean operations, like in the n-queens problem, but its performance drops considerably when it comes to other operations like variable replacement or quantification. Moreover it exhibits weird behaviors, such as exiting the JVM instead of throwing exceptions in some operations, like `replace`, `exist` and `forall`. This makes it unsuitable for production environments. SableJBDD is in very early stages of development and as such it exhibits poor performance and very high memory consumption.

3.1.2 The Concurrent Library Sylvan

Sylvan [108] is a concurrent C++ library for the manipulation of decision diagrams. It employs an unique table and an operation cache that can be safely accessed by multiple threads of execution, like does our implementation.

Parallelism in BDD operations is exploited by spawning two concurrent task when recursing down the low and high children of the BDD. Load balancing between threads is done with their own implementation of *work-stealing*. In work-stealing, there are a pool of workers to which tasks are submitted for execution. Each worker has a queue of tasks assigned to it. When a worker's queue is empty, it can steal a task from other queues, thus ensuring that no worker sits idle, as long as there are tasks to execute. When a task submit a new subtask, as is the case with recursive computations like those on BDDs, the subtask is inserted in the same queue of the main task, and can then be stolen by other workers.

3.2 Identification of Injection Vulnerabilities

The identification of possible injections and the inference of information flows are well-studied topics. Our injection identification technique is scalable to real world Java code, unlike many others. Most injection identification techniques are dynamic and/or unsound. Existing static information-flow analyses are not satisfactory for languages with reference types.

Identification of Injections. Data injections are security risks, so there is high industrial and academic interest in their automatic identification. Almost all techniques aim at the dynamic identification of the injection when it occurs [69, 67, 77, 110, 99, 44, 100, 97] or at the generation of test cases of attacks [22, 73] or at the specification of good coding practices [98].

By contrast, static analysis has the advantage of finding the vulnerabilities before running the code, and a sound static analysis *proves* that injections *only* occur where it issues a warning. A static analysis is *sound* or *correct* if it finds all places where an injection might occur (for instance, it must spot line 17 in Fig. 8.1); it is *precise* if it minimizes the number of false alarms (for instance, it should not issue a warning at line 22 in Fig. 8.1).

Beyond Julia, static analyzers that identify injections in Java are FindBugs (`http://findbugs.sourceforge.net`), Google's CodePro Analytix (`https://developers.google.com/java-dev-tools/codepro`), and HP Fortify SCA (on-demand web interface at `https://trial.hpfod.com/Login`). These tools do not formalize the notion of *taintedness* (as we do in Def. 8.8). For the example in Fig. 8.1, Julia is correct and precise: it warns at lines 15, 17, and 20 but not at 22; FindBugs incorrectly warns at line 17 only; Fortify SCA incorrectly warns at lines 15 and 17 only; CodePro Analytix warns at lines 15, 17, 20, and also, imprecisely, at the harmless line 22. Sec. 8.4 compares those tools with Julia in more detail.

We also cite FlowDroid [23], that however works for Android packages, not on Java bytecode, and TAJ [107], that is part of a commercial product. Neither comes with a soundness proof nor a definition of taintedness for variables of reference type.

Modelling of Information Flow. Many static analyses model explicit and often also implicit information flows [93] in Java-like or Java bytecode programs. There are data/control-flow analyses [36, 71, 94, 80]; type-based analyses [101, 109, 26, 27, 68, 55] and analyses based on abstract interpretation [56]. They are satisfactory for variables of primitive type but impractical for heap-allocated data of reference type, such as strings. Most analyses [27, 36, 68, 55, 71, 80, 94, 109] assume that the language has only primitive types; others [26, 56] are object-insensitive, *i.e.*, for each field f, assume that $a.f$ and $b.f$ are both tainted or both untainted, regardless of the container objects a and b. Even if a user specifies, by hand, which f is tainted (unrealistic for thousands of fields, including those used in the libraries), object-insensitivity leads to a very coarse abstraction that is industrially useless. Consider the `String` class, which holds its contents inside a `private final char[] value` field. If any string's `value` field is tainted, then every string's `value` field must be tainted, and this leads to an alarm at every use of strings in a sensitive context in the program, many of which may be false alarms. The problem applies to any data structure that can carry tainted data, not just strings. Our analysis uses an object-sensitive and *deep* notion of taintedness, that fits for heap-allocated data of reference type. It can be considered as data-flow, formalized through abstract interpretation. This has the advantage of providing its correctness proof in a formal and standard way.

3.3 Locking Discipline Inference

Despite the need for a formal specification for reasoning about Java's concurrency and for building verification tools [38, 74, 28], we are not aware of any previous tool built upon a formalization of the semantics of Java's concurrency annotations [59]. The JML (Java Modeling Language) `monitors_for` statement [91, 10] corresponds to the JCIP `@GuardedBy` annotation [59], together with its limitations: name protection and semantic ambiguities. Currently, [91] requires to write such annotation, manually, in source code, together with other, non-obvious annotations about the effects of each method. Once that hard manual task is done, the JML annotations can be model-checked, which is only proved to work on small code.

Warlock [105] was an early tool that checked user-written specifications of a locking discipline, including annotations for variable guards and locks held on entry to functions. ESC/Java [52] provided similar syntax and checked them via verification conditions and automated theorem-proving, an approach also applied to other concurrency problems [51]. All these tools are unsound and do checking rather than inference. Similarly to our inference, [81] infers locking specifications by generating the set of locks which must be held at a given program location and then checking the lockset intersection of aliasing accesses. It is based on possible rather than definite aliasing and hence is unsound.

Most approaches, including ours, explicitly associate each variable with a lock that guards access to it. An alternative is to use ownership types and make each field protected by its owner, which is not necessarily the object that contains it [31, 41]. This approach is somewhat less flexible, but it can leverage existing object encapsulation specifications and can be extended to prevent deadlocks [30].

These concepts can also be expressed using fractional permissions [111]. Grossman [64] extended type-checking for data races to Cyclone, a lower-level language, but did not implement or experimentally evaluate it.

Previous inference techniques include unsound dynamic inference of lock types [92] and sound inference via translation to propositional satisfiability, for most of Java [50]. In [65], a trace of execution events is recorded at runtime, then, offline, permutations of these events are generated under a certain causal model of scheduling constraints. This leads to a fast, but unsound, bug-finding technique for concurrency problems. By contrast, our approach is sound, more precise, and more scalable. Improving our aliasing analysis [18] would improve the recall of our implementations.

JCIP [59] does not mention aliasing, but it does mention instance confinement. JCIP notes that instance confinement only works with an “appropriate locking discipline”, but does not define the latter term. Our use of aliasing is less restrictive and more flexible, and our analysis is effective without a separate instance confinement analysis.

3.4 Static and Dynamic Analysis

Static analysis concerns the extraction of properties of a program by means of examining its code. It builds a model of the program state, that reacts to the program’s behavior. A model considering every possible runtime behavior of the program might be too complex to be manipulated effectively, and so some information needs to be abstracted away, producing a simpler model that represents only properties of interest. It is usually a requirement that this abstraction must maintain soundness, *i.e.*, the derived property must be present also in the complete, original, model. This conservativeness might lead to loss of precision, if a property valid in the complete model is not present in the abstracted one.

Dynamic analysis operates by executing the program, collecting results, and verifying that these match the desired behavior. Given the absence of abstraction, dynamic analyses are always precise. On the other side, they are typically unsound, as they do not cover every execution paths for every possible input.

The analyses described in this thesis need to consider all execution paths, in order to maintain soundness. For example, an untrusted value might flow into a trusted sink via a path that was not considered by a dynamic analysis. Or, a field might be accessed in such a path. So, dynamic analyses are not suitable to catch all synchronization errors in a program, nor the injection of untrusted data into trusted places.

Part I

Binary Decision Diagrams

4

Boolean Function Manipulation with BDDs

The representation type plays a fundamental role in the tractability of Boolean functions: a function with an exponential size in one kind of representation can be represented with few elements in another. BDDs are rather compact representations, and so are used in many fields, when efficient manipulation of Boolean functions is required.

In this chapter we will dive into the representation of choice for Boolean functions, Binary Decision Diagrams. We will describe the main theoretical concepts, along with implementation features exhibited by many BDD *libraries* or *packages*. Finally we will describe our own implementation, that is thread-safe and has particular features.

4.1 Binary Decision Diagrams

In decision diagrams, a series of choices determines an evaluation of the represented function. A choice in the context of binary decision diagrams consists of a test on a Boolean variable: if a variable is assigned a value, true or false, the corresponding path is taken.

A switching function can be represented by a *binary decision tree*, in which

- internal nodes represent Boolean tests, are labeled by a Boolean variable and have two outgoing edges, the *0-edge* and the *1-edge*
- leaves (or sinks) are labeled with the constants 0 or 1, meaning that the described function evaluates to false or true, respectively.

Figure 4.1 shows an example of a binary decision tree. We can assume that each variable is evaluated at most once on each path. A 1-edge, taken if the node variable evaluates to true, is usually drawn as a solid line, whereas a 0-edge, taken if the node variable evaluates to false, is usually drawn as a dashed line. Given a node n, the successor reached with its 1-edge is called $high(n)$, while the successor reached with its 0-edge is called $low(n)$.

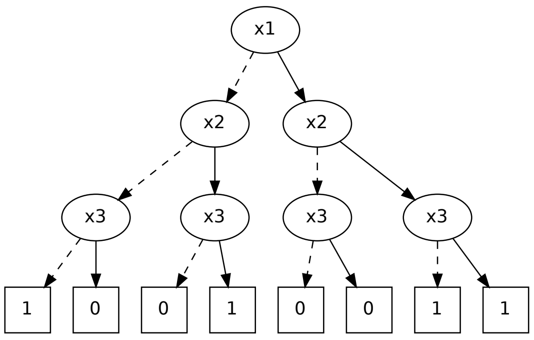


Fig. 4.1: Binary decision tree for the function $\overline{x_1 x_2 x_3} + x_1 x_2 + x_2 x_3$

The binary decision trees described above can be generalized to structures called *branching programs* or *binary decision diagrams*, directed acyclic graphs with exactly one root and the same kind of nodes as binary decision trees. Figure 4.2 shows one of all the possible branching programs for the parity function $x_1 \oplus x_2 \oplus x_3$. Differently from decision trees, for branching programs

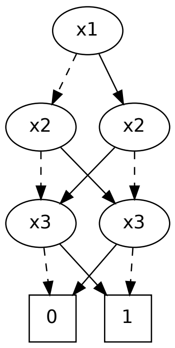


Fig. 4.2: A branching program for $x_1 \oplus x_2 \oplus x_3$

a variable can be evaluated more than once on a path. A node can have more than one predecessor, and so there may be several paths leading from the root to the node, on which the variable may be tested in different manners. Hence, even if a variable has already been tested on a path from the root to a node labeled by the same variable, this test cannot be eliminated in general.

This freedom of general branching programs can be a limitation when it comes to the efficiency of algorithms for the manipulation of these structures. Efficient algorithms can only be employed if none of the variables is read several times on a path. A branching program with this property is called a *read-once branching program.* This implies that a binary decision tree is also a read-once branching program. In a read-once branching program each path is a *computation path*, that is, given an input $a = (a_1, \dots, a_n)$, at the node x_i the path follows the edge with label a_i—a computation path cannot contain a 0-edge and a 1-edge for the same variable. For read-once branching programs satisfiability can be solved in polynomial time, but computing conjunction, disjunction and exclusive or of two read-once branching programs is *NP*-hard [78].

4.1.1 Ordered Binary Decision Diagrams

Although branching programs were known since the 1950s, it was Randal E. Bryant [33] who improved these structures by imposing ordering and reduction restrictions, achieving canonicity and compactness. An *Ordered Binary Decision Diagram* (OBDD) is a read-once branching program in which variables on a path from the root to a sink are ordered according to a fixed total order. More precisely, given a total order π defined on the set of variables $x_1, \dots, x_n$, for each edge leading from a node labeled with x_i to a node labeled with x_j, it holds that $x_i <_\pi x_j$. Every node in an OBDD defines a Shannon expansion of the function represented by the diagram rooted in that node. Let f be the function represented by the node n labeled with x_i. Referring to Section 2.1.1, let's call the negative cofactor of f $f_{\overline{x_i}}$, and the positive cofactor f_{x_i}. The function f can then be written as

$$f = x_i f_{x_i} + \overline{x_i} f_{\overline{x_i}}$$

This means that $high(n)$ represents the function f_{x_i} and $low(n)$ represents the function $f_{\overline{x_i}}$.

4.1.2 Reduced OBDDs

There can be two types of redundancy in an OBDD:

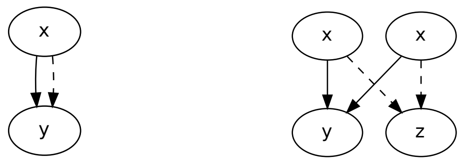
- if $low(n) = high(n)$, then the test in n is useless
- if certain subgraphs are *isomorphic* the same information is represented more than once in the OBDD

Two OBDDs are isomorphic if there exists a bijection ϕ between the two sets of nodes such that for each node n either

- $var(n) = var(\phi(n))$, $\phi(high(n)) = high(\phi(n))$, $\phi(low(n)) = low(\phi(n))$, or

- both n and $\phi(n)$ are sinks with the same label (0 or 1).

Figure 4.3 shows how redundant nodes in an OBDD look like. If an OBDD



(a) the test in x is useless (b) the two x nodes are isomorphic

Fig. 4.3: Types of redundancy in an OBDD

has no redundant nodes,it's called *reduced* (ROBDD). The above conditions suggest two reduction rules:

Elimination rule if $low(n) = high(n) = m$ then redirect all incoming edges in n to m and eliminate n.

Merging rule if $var(n) = var(m)$, $low(n) = low(m)$ and $high(n) = high(m)$ (m and n are isomorphic) then redirect all incoming edges in n to m and eliminate n.

An OBDD is reduced if and only if the reduction rules cannot be applied. Figure 4.4 shows an example of a ROBDD. Compared to the binary decision tree for the same function in Figure 4.1, this representation uses far fewer nodes. An OBDD of a switching function f with respect to the variable order π is reduced if and only if it is isomorphic to the minimal OBDD of f with respect to π. So, for each variable order π, the ROBDD representation of a switching function with respect to π is uniquely determined, *i.e.*, ROBDDs form a canonical representation type for switching functions. Canonicity of ROBDDs implies that there is exactly one ROBDD for the constant *true* and one for the constant *false*. Hence, tautology (*i.e.*, being equivalent to the constant *true*) and satisfiability (*i.e.*, not being equivalent to the constant *false*) can be tested in constant time, comparing the canonical representations of the functions. In the general case of Boolean expressions these problems are *NP*-complete. It turns out that ROBDDs, beside providing a canonical representation of switching functions, can be manipulated efficiently, and for many practically important functions the ROBDD representations are quite small.

To construct a ROBDD the above reduction rules can be applied to an already existing OBDD. Another possibility consists on preventing the pro-

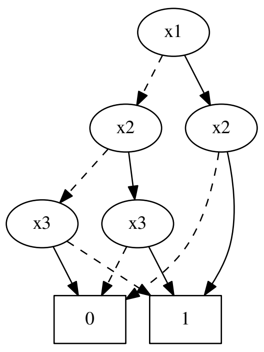


Fig. 4.4: ROBDD for the function $\overline{x_1 x_2 x_3} + x_1 x_2 + x_2 x_3$

duction of redundant nodes during the OBDD construction. The construction starts at the root, in a top-down manner: the root is labeled with the first variable in the order, x_i, and recursively constructing the *high* and *low* nodes as ROBDD for the subfunctions f_{x_i} and $f_{\overline{x_i}}$, respectively. If for each node the reduction rules are satisfied, the final diagram is a ROBDD. The elimination rule is implemented by checking if the two successors for the node are the same, while the merging rule is implemented by taking care that only a copy of a particular BDD is present in memory. Section 4.2 explains how this can be implemented in software.

In practice, BDD is used as a synonym for ROBDD, and so it will be in the following.

4.2 General Implementation Features

The advantages of BDDs outlined in Section 4.1.2, like the equivalence testing in linear time and the compact representation, can be magnified by a clever implementation. For example, equivalence testing can be performed in constant time, just by comparing pointers or integer indexes pointing to uniquely represented structures. The first efficient BDD package was developed by Brace, Rudell and Bryant in 1990 [32]. The implementation ideas employed in this early accomplishment served as inspiration for all successive packages.

The basic data element for storing a BDD is the node. A structure representing a BDD node must store at least the variable index and pointers to low and high children. Adding additional information can improve the performance of algorithms, but it increases the memory consumption as well, and possibly worsens data locality, so it becomes necessary to find a compromise that maximizes performance. The general implementation choices presented below are implemented in existing BDD packages, although with modifications and adaptations.

Shared BDD Different BDDs can share the same subgraphs, and so several functions can be represented as a single directed acyclic graph, a *shared BDD*, in which each subgraph (and hence each node) is represented exactly once. This memory representation achieves *strong canonicity*: not only two equivalent functions are represented by identical graphs, but actually by the *same* graph. Employing this optimization allows one to perform equivalence testing in constant time, comparing the root nodes of the graph (or pointers to them).

Unique table Strong canonicity is implemented by means of a *unique table*. To ensure that each subfunction is represented exactly once in the shared BDD, before adding a new node, a query is raised to the table. If a node with the desired variable index and *low* and *high* pointers already exists, a pointer to that node is returned instead. This operation is performed quickly if the unique table is implemented as a hash table, using a suitable hash function. The hash table can contain pointers to actual node data, or the data themselves. The BDD is kept in reduced form by checking if a node already exists (thus adhering to the *merging rule* of Section 4.1.2) and by testing if the *low* and *high* pointers coincide (*elimination rule*), This reduction is generally done through a function called `MK`:

```
Function MK(v,l,h)
  if $l = h$ then
    return l
  if a node with (v,l,h) can be found in the table then
    return it.
  add a node $n$ to the table
  return n.
```

The first `if` corresponds to the elimination rule, and the second to the merging rule.

Computed table Executing BDD operations can be expensive, and so it is reasonable to save the results in a data structure for later retrieval, should an operation be performed again. For example, suppose we have a table C of all Boolean operations computed so far. Boolean operations are typically performed by an `APPLY` function, taking as arguments two BDDs (root nodes) and a binary operator:

```
Function APPLY(op,n1,n2)
    if C contains the result of (n1 op n2) then
        return it.
    if n1, n2 are constants then
        return op applied to n1, n2.

    if var(n1) = var(n2) then
        result = MK(var(n1),
                APPLY(op, low(n1), low(n2)), APPLY(op, high(n1),
                 high(n2)))
    else if var(n1) < var(n2) then
        result = MK(var(n1),
                APPLY(op, low(n1), n2), APPLY(op, high(n1), n2))
    else
        result = MK(var(n2),
                APPLY(op, n1, low(n2)), APPLY(op, n1, high(n2)))

    save result in C
    return result
```

The `APPLY` function produces the resulting BDD using Shannon expansion: when the *low* and *high* children BDDs are constructed, they are combined by the function `MK`. This ensures that the resulting BDD is reduced. To make the result ordered, sub-diagrams are composed such that the lower variable is kept above in the partial BDD. The *computed table* C of all previous results can be realized by a hash table with collision resolution. We can therefore assume that insertion and lookup in C can be performed in constant time. Not only an external invocation of `APPLY` can benefit from already computed results, but also recursive calls in the same invocation. This fixes a bound to the number of recursive calls to $| n1 || n2 |$, where $| n |$ denotes the number of nodes in the BDD rooted at node n. The expected time complexity of `APPLY` is thus $O(| n1 || n2 |)$ [33]. However, remembering all the results computed so far requires a lot of memory. Additionally, algorithms take advantage of time locality in referencing previous results: the longer the time after the computation of a specific result, the smaller becomes the probability that exactly this result is needed again. For this reason, in practice a small hash-based cache performs better and wastes much less memory. In such a structure, only the most recent result is stored (or the few most recent results). Nevertheless, the previous considerations on the time complexity of the `APPLY` algorithm hold only if a full non-forgetting computed table is implemented; if this is not the case, at worst the time behavior is exponential. In practice, extreme cases exhibiting such a worst case behavior appear very seldom.

Similar caches can be implemented for other algorithms as well, such as those for quantification, restriction, simplification and replacement.

- existential or universal quantification are the usual logical operations
- restriction computes the BDD obtained by fixing the value of some of the input variables
- simplification refers to node removal based on a domain of interest d: the resulting BDD represents a function that is evaluated to 1 only for those assignments that satisfy d
- replacement substitutes some variables of the input BDD with other variables; this is not simply a node renaming operation, as the ordering restriction can require a restructuring of the diagram

Garbage collection Boolean operations cause the creation of many temporary BDD nodes. For example, in the construction of the BDD representing a combinational circuit a BDD for each gate is created, that is necessary only for the construction of successor gates' BDD. Which nodes should be considered dead at a given point in an application? Those that do not belong to "interesting" BDDs, from the user's standpoint—such as the whole circuit BDD described above. Unused nodes can be deleted from the table; however, dead nodes can be useful, should the application need a result from the compute cache rooted at a dead node. Moreover, deleting nodes and restructuring tables can be expensive. So, dead nodes should not be removed immediately, but only when it is necessary to gain memory. This *garbage collection* reduces the need to resize the node table. In many packages—such as BuDDy, JavaBDD, JDD—a *reference counter* is maintained for each node. This counter tracks how many incoming edges there are for the node. Every client handle counts as 1 for the root node of the BDD. The counter is incremented whenever a new predecessor node is added and decremented when a predecessor's counter reaches 0 (this means that the node is dead). Garbage collection can be triggered when a sufficient number of nodes are dead. Typically only one or two bytes are reserved for the reference counter: if the number of references for a node exceeds the maximum representable value with the allocated bits, the counter "overflows", and the node becomes immortal.
Instead of keeping a counter of the existing references to a node, another choice could be to implement a two-stage garbage collection: a first stage to *mark* active nodes, and a second (*sweep*) to erase dead nodes (those not reached in the first stage). This leads to a somewhat simpler and less error-prone code.

4.3 Our new Thread-Safe Package

Concurrency problems arise when mutable state is shared. In a BDD package, state is represented essentially by the node table and the operation caches. To safely use these structures from different threads, access has to be carefully synchronized. Our library, that we called *BeeDeeDee*, implements this synchronization in accessing shared data, while maintaining a speed comparable to other libraries. Before delving into the synchronization details, we start with analyzing how the library is structured.

4.4 Library Structure

Information about BDD nodes are stored in a unique table, implemented with a simple integer array `ut`. This choice was mandatory to achieve acceptable performance and memory consumption. Indeed, storing node data in a Java object occupies more memory and produces many dead objects during the life of the application, leading to greater pressure on the Java garbage collector. Additionally, it destroys memory locality, since data for a node can be anywhere in the heap, instead of in contiguous cells of an array. Good memory locality is necessary to exploit high-speed processor caches, and therefore to achieve good performance. These issues, together with indirection and object creation overhead, would produce poor performance compared to *state-of-the-art* BDD packages.

Currently in our implementation a node in the table spans over 5 cells, a total of 20 bytes of memory. An auxiliary array `H` contains pointers (integer indexes) to nodes having a specific hash value. Together, `ut` and `H` form an hash table. Collision resolution is achieved through chaining in the unique table itself, via an integer component pointing to the next node in the table. The hash value is constructed using the variable number and the indexes of the *low* and *high* branches. This increases the size to 24 bytes per node. The first few lines of the `ut` table look like these:

Index	variable number	low	high	next	hcAux
0:	2147483646	-1	-1	1	0
1:	2147483647	-1	-1	2	1
2:	0	0	1	3	2
3:	1	0	1	4	3
4:	2	0	1	5	4
5:	3	0	1	6	5
...					

The variable numbers of the terminals need to be larger than any other variable number, because some library algorithms rely on this for their correctness. *Low* and *high* children are identified by their index on the table; terminal nodes have no children, and so the fields are set to -1. The *next* field points to the next node in the collision chain. The last field, *hcAux*, is a unique identifier of the node, that acts as the hash code for a BDD object. It remains the same during the life of the application, even after a garbage collection deletes nodes and compacts the table (see 4.4.1). This is to ensure that operations on hash based structures (e.g. a `HashMap` or `HashSet`) are consistent in time.

As already said, uniqueness in the node table is implemented by accessing a node in expected constant time via hashing. Due to the fact that BDDs are canonical forms of Boolean formulas, a node represents a class of equivalent formulas. So a BDD is univocally identified by its index in the table. Clients of the library do not see this internal representation, but rather obtain from the factory a `BDD` java object pointing to the right starting node of the diagram in the table.

The `BDD` interface and the `Factory` abstract class form the main public interface of the package. Clients can create a factory of `BDD` objects by invoking the static method `Factory.mk(utSize, cacheSize)`, that takes as parameters the initial size of the unique table, in number of nodes, and the size of the operation caches, in number of entries—refer to Section 4.4.2. References to basic `BDD` objects created by the `Factory` can be obtained by calling the methods

- `makeZero()` and `makeOne()`—to get a reference to the terminal BDDs 0 and 1
- `makeVar(int)` and `makeNotVar(int)`—that, given the desired variable index, return a BDD representing a single variable, respectively in positive and negative form, creating the corresponding single node if necessary.

4.4.1 Internal Garbage Collection

Having all of the BDD nodes stored in a single array prevents the Java Garbage Collector to free memory no more used for a BDD object. So we implemented a garbage collection procedure that compacts the node table, overwriting dead nodes. To find live nodes, instead of counting references to every node, we implemented a *mark and sweep* collection. Every user `BDD` object is considered live until its `free()` method gets called by the programmer. Every node reachable recursively from these high level objects is then considered live as well. Nodes excluded from this pass (*mark*) are thus dead, and so their memory can be reclaimed (*sweep*). Compacting the node table during the garbage collection yields the additional benefit of improving memory locality. Garbage collection is started either when the occupation of the table exceeds 98%, or on user's request.

4.4.2 Operation Caches

Operations such as APPLY can be very expensive. The traditional approach to get better performance is to maintain all operation results computed so far in a hash table with collision resolution. A small hash-based cache, keeping only the last result for each position in the table, has proven to be the fastest solution, due to the cleaner implementation and the fact that it exploits the *temporal* locality of operations, as well as the *spatial* locality of processor caches. Currently our package implements caches for APPLY, QUANTIFICATION, REPLACE and RESTRICT.

4.5 Concurrency

The unique table cannot be modified nor queried when a garbage collection or a resize are being performed, and so special care has been taken to avoid concurrency issues without excessively degrading performance. Different strategies have been applied to isolate garbage collection and resizing of the node table from operations on the table itself.

4.5.1 Synchronizing with Garbage Collection

In a concurrent setup, BDD operations may be initiated in the middle of a garbage collection, and so a mean to synchronize their execution had to be implemented. Using a single monitor for this purpose has the disadvantage that two operations cannot start at the same time. So, a set of locks, implemented by the array `gcLocks`, has been used instead. Every operation obtains a lock from the table with the `getGCLock()` method, via the wrapper class `GCLock`, that implements the interface `Closeable`, thus allowing code to release the lock by using the automatic closing of resources of Java 7:

```java
protected class GCLock implements Closeable {
   private final ReentrantLock lock;

   public GCLock() {
      this.lock = ut.getGCLock();
      this.lock.lock ();
   }

   @Override
   public void close() {
      lock.unlock();
   }
}
```

This frees the programmer from the need to release the lock in a `finally` clause, and leads to simpler code. For example, this is how the method implementing an `and` acquires the lock before performing the operation:

```java
 try (GCLock lock = new GCLock()) {
     return new BDDImpl(innerAnd(id, ((BDDImpl) other).id));
 }
```

If an operation tries to acquire a lock whose reference is stored on a cell of the array, the next operation uses the lock in the cell following the latter[1], in a circular way. In other words, at every call to the `getGCLock()` method, the array index is incremented *modulo* the array length.

[1] we assume that the array has a length of at least 2

```
public ReentrantLock getGCLock() {
    return gcLocks[nextGCLocks = (nextGCLocks + 1)
        % gcLocks.length];
}
```

Hence, when the GC is not executing and no lock is held, a number of BDD operations equal to the length of the `gcLocks` array can start without blocking. When a garbage collection starts, it tries to acquire the lock on all the monitors in the array, and then, when no operation is executing, performs the actual collection.

```
private boolean getAllLocksAndGC() {
    for (ReentrantLock lock: gcLocks)
        lock.lock();
       ...
```

Figure 4.5 illustrates various states of the locks in `gcLocks`, in the case of an array of length 5.

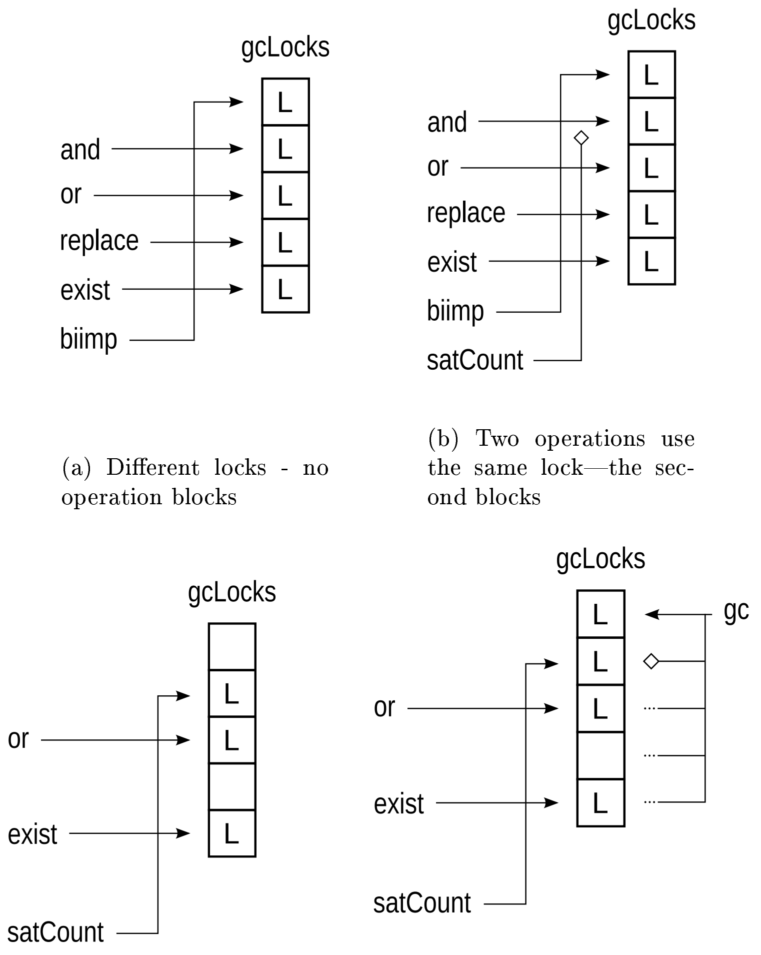
4.5.2 Synchronizing with Resize

Whenever it needs a particular BDD node, an operation calls the `get` method of the unique table. Given the desired variable number and *low* and *high* children, the `get` method searches in the table for a node with these fields, by calculating a hash code and going through the corresponding collision chain. If it finds the node it returns its index, otherwise a new table entry needs to be created.

Resizing is done by a user thread competing for the table, when a particular node is not present in the table and there is no space left for its creation. Here too an array of monitors, `getLocks`, was used to reduce the probability of two operations locking on the same monitor. Differently from the synchronization for the garbage collection, locks are not accessed in a circular manner, but instead the index of the lock in the array is the hash code just computed, *modulo* the array length.

```
public final int get(int var, int low, int high) {
    do {
        int size = this.size;
        int pos = hash(var, low, high, size);
        int result = getOptimistic(var, low, high, pos);
        if (result >= 0)
            return result;

        Object myLock;
        synchronized (myLock = getLocks[pos % getLocks.length]) {
            if (size == this.size
                    pos == hash(var, low, high, this.size))
```



(a) Different locks - no operation blocks

(b) Two operations use the same lock—the second blocks

(c) The second operation acquires the lock

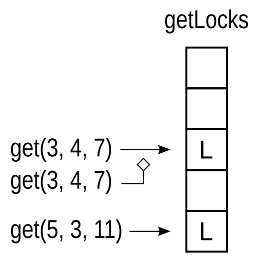
(d) GC starts and blocks on the second lock until all operations have finished

Fig. 4.5: Locking on the `gcLocks` monitors

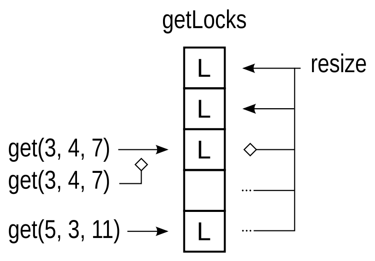
```
                return expandTable(var, low, high, myLock, pos);
            }
        }
        while (true);
    }
```

In the above code, first the current size of the table and the position in the `H` array (that depends on it) is saved, then `getOptimistic` is called, to verify if the desired node is already present. The `getOptimistic` method is not synchronized, since the only operation that can change the position of an existing node in the table is the garbage collection—when it compacts the table—and `get` runs only if there is no collection in progress. In this way `get` invocations for already existing nodes can run in parallel. Conversely, if the desired node is not present in the table, a new node must be created. This time, synchronization is necessary to avoid data races. `get` first checks if the size changed, meaning that a resize occurred while it was trying to access the critical section: in that case it has to recompute the hashcode, since it might have changed, so `get` restarts again. The `expandTable` method adds a node to the table. If there is no room for the node, the table is resized, unless another thread is already resizing it. In this case, the current thread *waits* on `myLock`.

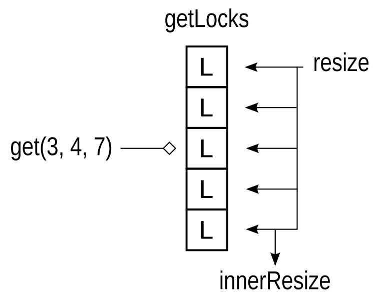
A thread initiating a resizing orderly tries to acquire all of the locks in the array, and then starts the actual resizing. When the resizing thread ends its work, it notifies all the waiting threads by calling `notifyAll` on all the locks in the `getLocks` array. In Figure 4.6 are captured four instants in the life of the monitor array, with a size of five.



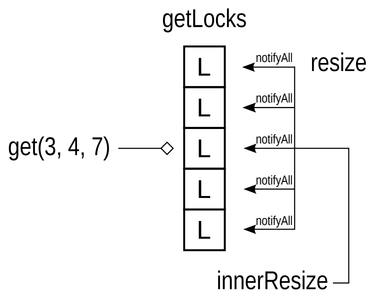
(a) Two `get` for the same node



(b) Resize starts locking



(c) Locks acquired, actual resizing starts



(d) End of resizing, notify waiting threads

Fig. 4.6: The `getLocks` mechanism

4.5.3 Synchronizing Hash Table Updating

After garbage collection or resizing occurs, the hash table has to be updated, as the hash value for a node and the collision chain may have changed. In fact, garbage collection compacts the table, by overwriting dead nodes, and resize, as the name suggests, changes the size of the table. In both cases, the hash value might have to be recomputed. If the size of the table crosses a certain threshold, updating is performed by several threads in parallel. The method `parallelUpdateHashTable` rebuilds the hash table by using a number of threads equal to the number of processors. Each thread updates the positions of the hash table congruent with its index, *modulo* the number of threads. For example, in Figure 4.7, four threads updates the hash table positions *modulo* 4. For each node in the unique table, starting from the last,

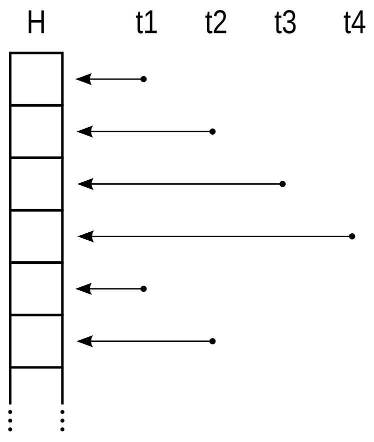


Fig. 4.7: Four threads updating the hash table

collision chain pointers are updated. The code fragment below shows the updating for a node.

```
...
int pos = hash(ut[index], ut[index + 1], ut[index + 2]);

synchronized (updateLocks[pos % updateLocks.length]) {
    setNext(i, H[pos]);
    H[pos] = i;
}
...
```

The parallel updater uses an array of locks called `updateLocks`. After the new position in the hash table has been computed (in `pos`), threads synchronize on locks depending on `pos` to actually update the pointers.

4.5.4 Synchronizing Caches

Our library implements caches for various BDD operations. Accesses to caches need to be synchronized. All the caches use a similar locking scheme: an array of lock objects, selected depending on hash values. We will consider as an example the `ComputationCache`, that stores the results of Apply. The `get` method takes as parameters a Boolean operation and two BDD indexes, and returns the stored result, or -1 if no such result exists yet.

```
int get(Operator op, int bdd1, int bdd2) {
    ...
    if (cache[pos1] == bdd1 && cache[pos2] == bdd2
            && cache[pos] == op)
        synchronized (locks[pos % locks.length]) {
            return (cache[pos1] == bdd1 && cache[pos2] == bdd2
                && cache[pos] == op) ? cache[pos + 3] : −1;
        }

    return −1;
}
```

The `put` method stores the given result on the cache.

```
void put(Operator op, int bdd1, int bdd2, int result) {
    ...
    if (cache[pos1] != bdd1  cache[pos2] != bdd2  cache[pos] != op)
        synchronized (locks[pos % locks.length]) {
            cache[pos] = op;
            cache[pos1] = bdd1;
            cache[pos2] = bdd2;
            cache[pos + 3] = result;
        }
}
```

In both methods, an object contained in the `locks` array is used to lock the position being read or written. The `if` statements surrounding the `synchronized` blocks avoid synchronization when it's not necessary.

5 Experiments and Comparisons

In this chapter we describe some experiments showing the performance of our library in solving several problems. First of all with the construction of the transition relation for sequential circuits. Then with the *n-queens* and *knight's tour* problems. Section 8.6 describes how well our library performs when used in flow analysis.

5.1 Circuits

Binary Decision Diagrams are widely used in hardware verification, both for combinational and sequential circuits. An integral part of the verification process is the representation of the circuit by means of some data structure. The memory consumption and the running time of algorithms depend strongly on the chosen representation. The use of BDDs for the verification of sequential circuits allowed to represent orders of magnitude more reachable states than before [35, 34].

5.1.1 Combinational Circuits

Combinational circuits consist simply of gates, with no memory, and can be represented by simply combining the gates' functionality: every gate is represented by a Boolean function, which is then combined with the others to produce a single function, describing the functional behavior of the circuit. In logic synthesis, a circuit can be transformed from its original incarnation—for example optimized by reducing the number of gates. Hence it is important to check that the transformation process has not introduced errors in the design, that is to say, that the transformed circuit is *functionally equivalent* to the original. Using BDDs as data structures makes equivalence testing very cheap.

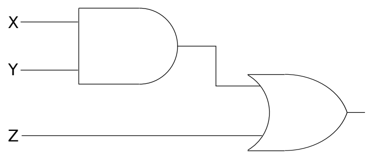


Fig. 5.1: Scheme of the combinational circuit for the function $xy + z$

5.1.2 Sequential Circuits

Sequential circuits include combinational parts and memory elements, so that outputs depend not only on inputs, but also on previous state of the circuit (Figure 5.2). To model the enumeration of states reachable for a sequential

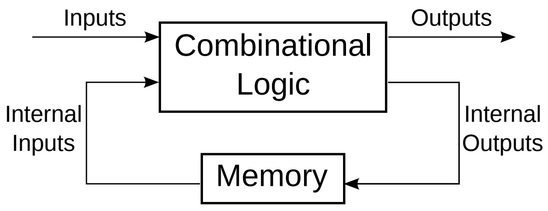


Fig. 5.2: Scheme of a sequential circuit

circuit, a *transition relation* has to be constructed. The transition relation binds the *current state* Boolean variables, V, with the *next state* variables, V'. Every "new" variable v'_i is produced from the current value of the state variables by a given function f_i, $v'_i = f_i(V)$. We can then define a set of relations $N_i(V, V') = (v'_i \Leftrightarrow f_i(V))$ binding old variables with each new variable v'_i. The whole transition relation is the conjunction of these relations:

$$N(V, V') = N_0(V, V') \wedge \ldots \wedge N_{n-1}(V, V')$$

Building the transition relation for a circuit involves a big number of simple Boolean operations. We exercised the relation construction for three circuits taken from the ITC99 benchmark set [9], comparing our library with JavaBDD and JDD. Results are shown in Figure 5.3. Also in this case, BeeDeeDee outperforms the other Java libraries.

5.2 *N*-queens

The n-queens problem consists in placing n queens over an $n \times n$ chessboard so that no one attacks another. Figure 5.4 shows one of the 92 solutions of the 8-queens problem. It translates naturally into the construction of a logical function whose solutions are all possible placements of queens. Figure 5.5

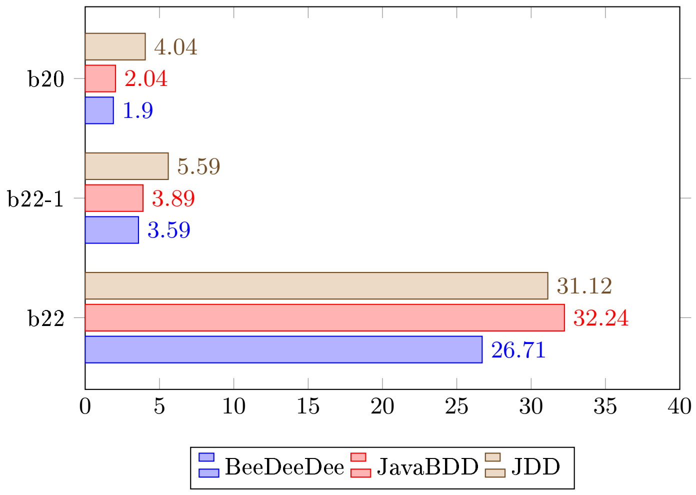


Fig. 5.3: Time (in seconds) for the construction of the transition relation for three circuits from ITC99

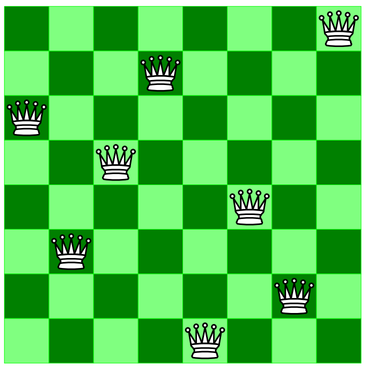


Fig. 5.4: A solution of the 8-queens problem

shows the execution time for constructing the function for the 12-queens problem with BeeDeeDee, compared to the time needed by using the Java libraries JavaBDD and JDD, and the C libraries CUDD and BuDDy. BeeDeeDee is here the fastest Java library and is comparable to BuDDy, the best C library.

Figure 5.6 shows the time for the construction from 1 to 4 BDDs representing the function associated to the same 12-queens problem. Such a construction was performed in parallel on a quad-core processor, by sharing unique table and caches with BeeDeeDee. We see here that we manage to achieve a high degree of real parallelism, since four BDDs are built in 33.6 seconds while a single BDD is built in 22.4 seconds. Some degradation exists, due to synchronization, but the parallel cost is much lower than the theoretical

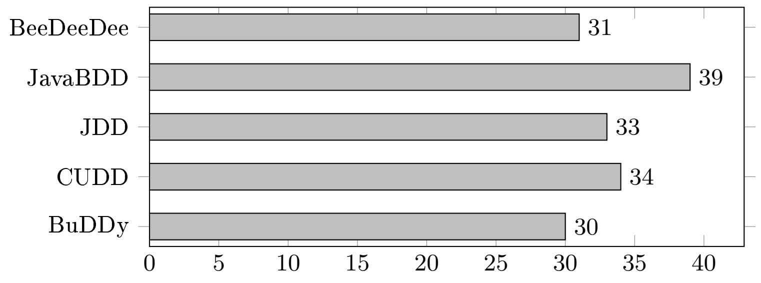


Fig. 5.5: Time (in seconds) for the solution of the 12-queens problem

sequential cost of $4 \times 22.4 = 89.6$ seconds. This example shows that the overhead of synchronization is well acceptable for parallel computations through BeeDeeDee.

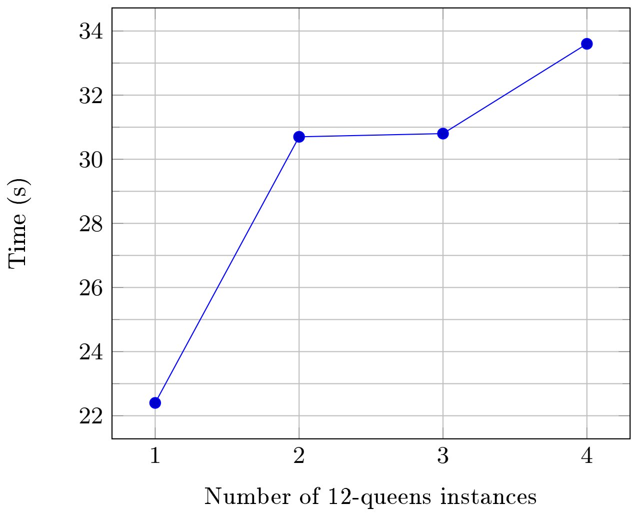


Fig. 5.6: Parallel 12-queens BDDs construction

5.3 Parallel Problems

We developed parallel versions of two problems: *knight's tour* and *transition relation construction* for sequential circuits.

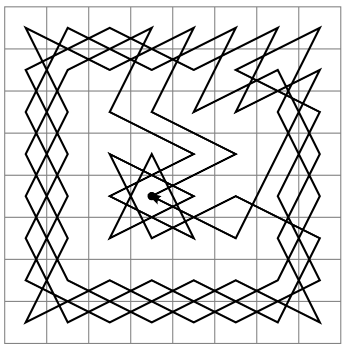


Fig. 5.7: A solution of the Knight's tour problem of size 8

5.3.1 Knight's Tour

In this problem, we try to find if a knight can complete a tour in an $n \times n$ chessboard in which every square is visited once and only once. For our purposes, we model the problem as reachability in a state transition system. We first build Boolean functions over variables associated to squares in the chessboard. These functions represent the initial state and transitions from a state to the next. Then the tour exists if and only if all the squares are reachable. We use the following sequential algorithm:

```
Function KnightsTourExists(N)
  /* initialize variables                                                    */
  for $i = 0 \to N$, $j = 0 \to N$ do
      $x_{ij} = v_{2*(i*N+j)}$ // even index, current state vars
      $x^{next}_{ij} = v_{2*(i*N+j)+1}$ // odd index, next state vars

  /* initial state function, true when no square is reached
     */
  $I = true$
  for $i = 0 \to N$, $j = 0 \to N$ do
      $I = I \wedge \neg x_{ij}$

  /* transition function                                                     */
```

$$T = \bigvee_{0 \le i,j < N, k=i-1,i+1, l=j-2,j+2, 0 \le k,l < N} S(i,j,k,l)$$
$$\bigvee_{0 \le i,j < N, k=i-2,i+2, l=j-1,j+1, 0 \le k,l < N} S(i,j,k,l)$$

```
  /* build reachable states function                                         */
  $R = false$
  do
      $R' = R$
      $exist = \exists x. T \wedge R$
      $replace = exist\{x \mapsto x^{next}\}$
      $R = I \vee replace$
  while $R' \neq R$;
  /* if all squares are reachable, then the tour exists                      */
  for $i = 0 \to N$, $j = 0 \to N$ do
      $R = R \wedge x^{next}_{ij}$
  $reachable = R.satCount$
  if $reachable = N \cdot N$ then
      return true
  return false
```

```
Function S(i,j,k,l)
  /* allowed moves for the knight at $(i,j)$                                 */
  /* $k = i \pm 1, i \pm 2$,  $l = j \pm 1, j \pm 2$                          */
  return $x_{ij} \wedge \neg x_{kl} \wedge \neg x^{next}_{ij} \wedge x^{next}_{kl} \bigwedge_{0 \le i',j' < N, i' \neq i,k, i' \neq j,l} x_{i'j'} \leftrightarrow x^{next}_{i'j'}$
```

The parallel version splits the computation of the T function among different threads. Figure 5.8 shows the execution time when dividing work in different number of threads.

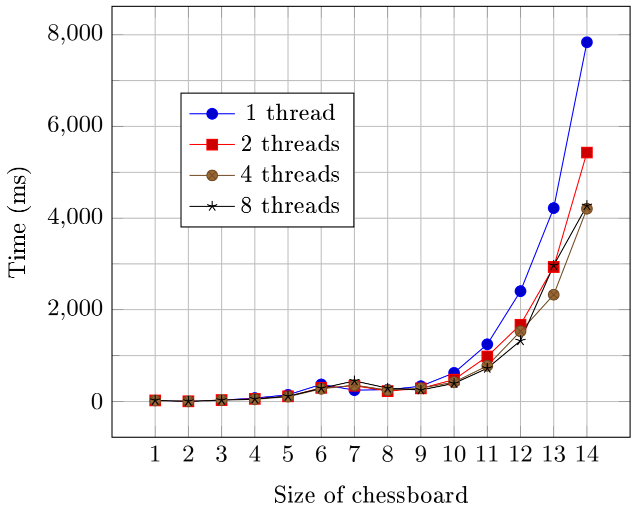


Fig. 5.8: Parallel Knight's Tour computation

5.3.2 Transition Relation Construction

As we said above, the transition relation for a sequential circuit is the conjunction of n different functions $v'_i \Leftrightarrow f_i(V)$:

$$N(V, V') = (v'_0 \Leftrightarrow f_0(V)) \wedge \ldots \wedge (v'_{n-1} \Leftrightarrow f_{n-1}(V))$$

The parallel version distributes evenly all of these functions among threads performing partial conjunctions. These are then combined to form the final function $N(V, V')$. Figure 5.9 shows execution times for the parallel construction of the transition relation of our three test circuits.

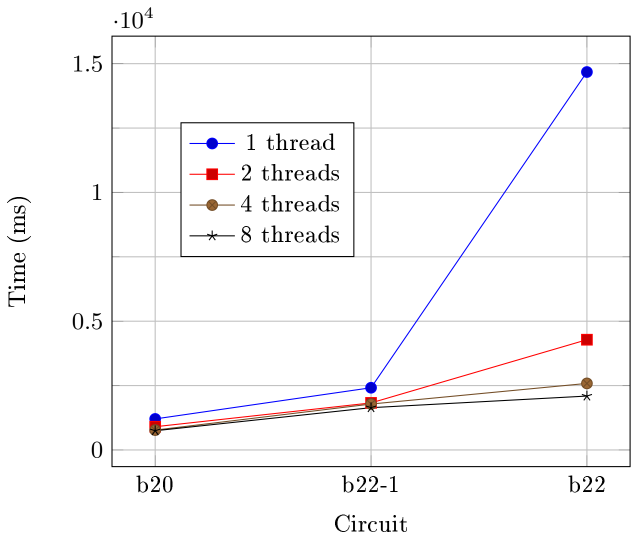


Fig. 5.9: Parallel construction of the transition relation for three circuits from ITC99

6

ER representation

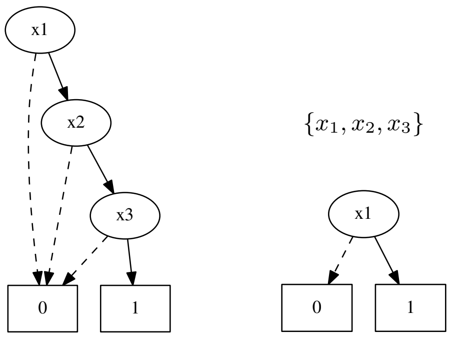
`ERFactory` is a particular factory that represents Boolean functions by keeping information on equivalent variables in a separate data structure. Thus such a representation, that we call *ER*, includes a *BDD* part, and an *equivalence relation* part, a set of disjoint sets of equivalent variables (*equivalence classes*). For each class, we choose a variable l as the *leader* of the class. An equivalence relation can also be seen as a set of pairs of equivalent variables, *i.e.*, $e_i = (x, y)$ is a pair meaning that x and y are equivalent, and we can express this with the formula $f_i = x \leftrightarrow y$. For instance, the equivalence relation $\{\{a, b\}, \{x, y, z\}\}$, can be represented with the pairs $(a, b), (x, y), (y, z), (x, z)$. Moreover, we write $f(b)$ to mean the formula for the function represented by the BDD b. So, given an ER representation formed by the pairs of equivalent variables $e_1, \ldots, e_n$ and the BDD b, the resulting Boolean function is represented by the formula

$$f_1 \wedge \ldots \wedge f_n \wedge f(b) \tag{6.1}$$

Even this representation is kept in canonical form by using a *normalization* operation, that moves variables found equivalent in the BDD part into the equivalence relation part: $norm(\langle E, R\rangle) = \langle E', R'\rangle$. Only the variable leading each class can appear in the BDD. It is implemented by the following algorithm:

```
Function normalize(er)
  newE = er.E
  newR = er.R
  do
      ev = pairs of equivalent variables in er.R
      oldE = newE
      Add pairs in ev to newE
      Rename variables in er.R with their leaders according to er.E
  while $newR \neq oldR \vee newE \neq oldE$;
  return $\langle newE, newR\rangle$
```

Figure 6.1 shows an example of the *ER* representation, besides the original BDD.



(a) BDD (b) ER

Fig. 6.1: Two representations for the function $x_1 \wedge x_2 \wedge x_3$

6.1 Operations

Operations on ERs can be performed on their equivalent BDDs, obtained using the formula (6.1), and transforming the result back into ER form, by normalizing it, but some operations can be carried out directly on ER representations, which is very important for efficiency.

AND Given two ER representations, $\langle E_1, R_1 \rangle$ and $\langle E_2, R_2 \rangle$, we compute their conjunction as $norm(\langle E_1 \cup E_2, R_1 \wedge R_2 \rangle)$

```
**Function** AND(er1,er2)
    $resultE = er1.E \cup er2.E$
    $resultR = er1.R \wedge er2.R$
    $er = ER(resultE, resultR)$
    **return** $normalize(er)$
```

OR Given two ER representations, $\langle E_1, R_1 \rangle$ and $\langle E_2, R_2 \rangle$, their disjunction is $\langle E', R' \rangle$, where

- $E' = E_1 \cap E_2$
- $R' = R'_1 \vee R'_2$

R'_i is obtained by removing equivalent, non-leader variables from R_i, and then re-adding all the equivalences corresponding to pairs of E_i not having non-leaders in common pairs in $E_{(i+1) \mod 2}$.

$\langle E', R' \rangle$ does not need to be normalized.

```
Function OR(er1,er2)
    $resultE = er1.E \cap er2.E$
    $pairs1 = er1.E.pairs$
    $pairs2 = er2.E.pairs$
    foreach pair p in pairs1 do
        if p is in pairs2 then
            Add $p$.second in nonLeaders1
    foreach pair p in pairs2 do
        if p is in pairs1 then
            Add $p$.second in nonLeaders2
    $squeezed1 = squeezeEquiv(er1.R)$
    foreach pair p in pairs1 do
        v1 = $p$.first
        v2 = $p$.second
        if v1 is not in nonLeaders1 $\wedge$ v2 is not in nonLeaders1 then
            squeezed1 = squeezed1 $\wedge (v1 \leftrightarrow v2)$
    $squeezed2 = squeezeEquiv(er2.R)$
    foreach pair p in pairs2 do
        v1 = $p$.first
        v2 = $p$.second
        if v1 is not in nonLeaders2 $\wedge$ v2 is not in nonLeaders2 then
            squeezed2 = squeezed2 $\wedge (v1 \leftrightarrow v2)$
    return squeezed1 $\vee$ squeezed2
```

NOT The negation uses De Morgan's law, and so, given that an *ER* $\langle \{e_1, \ldots, e_n\}, b \rangle$ corresponds to the function $f_1 \wedge \ldots \wedge f_n \wedge f(b)$, its negation is computed as $\neg f_1 \vee \ldots \vee \neg f_n \vee \neg f(b)$. This result is then normalized.

```
Function NOT(er)
    not = not(er.R)
    foreach pair p in er.E.pairs do
        v1 = $p$.first
        v2 = $p$.second
        not = not $\vee \neg (v1 \leftrightarrow v2)$
    return normalize(not)
```

XOR, IMP and BIIMP operations are derived from those above, and so the result does not need to be normalized:

XOR The XOR of two *ER*s er_1, er_2 is computed as $(er_1 \vee er_2) \wedge \neg (er_1 \wedge er_2)$.
IMP The IMP of two *ER*s er_1, er_2 is computed as $\neg er_1 \vee er_2$.
BIIMP The BIIMP of two *ER*s er_1, er_2 is computed as $(\neg er_1 \vee er_2) \wedge (\neg er_2 \vee er_1)$.

EXIST Existential quantification happens over a set of variables $V = \{v_1, \dots, v_k\}$. If a variable in V is in E, it is not quantified in the BDD part, but instead replaced with its leader. If all the variables in a class are in V, there is no leader to be substituted, and the variable is quantified. All variables in V are removed from E.

```
Function EXIST(er,V)
// requires normalized representation
foreach $v$ in $V$ do
    if $v$ is in er.E then
        l = the leader of the equivalence class obtained by removing all
          the variables in $V$ from the equivalence class of $v$
        if $l$ exists then
            Put ($v$, $l$) in renaming
        else
            Add $v$ in $V'$
    else
        Add $v$ in $V'$
Remove all variables in $V$ from er.E
er.R = replace(er.R, renaming)
er.R = exist(er.R, V')
return normalize(er)
```

REPLACE Replace performs variable substitution on the *ER*. Substitution is carried separately on the BDD and on the equivalence relation part. Care is required in case of simultaneous substitution: if some variable is target of some mapping, and is to be substituted itself, then it has to be substituted first. Then, variables in the BDD part are substituted with their leaders according to the equivalence relation.

```
Function REPLACE(er,renaming)
er.R = replace(er.R, renaming)
foreach $v$ in renaming.values do
    if renaming contains $v$ as the key of a value $v'$ then
        Put $(v, v')$ in renameFirst Remove $(v, v')$ from renaming
er.E = replace(er.E, renameFirst)
er.E = replace(er.E, renaming)
Rename variables in er.R with their leaders according to er.E
return normalize(er)
```

The idea of keeping BDDs small by extracting information from them and manipulating it separately was first introduced and formalized in [25]. They managed to keep separate also *ground* variables, for the purpose of groundness analysis of constraint logic programs. This analysis was implemented in the *China* analyzer. It uses the *CORAL* BDD library [29], that employs all of the operations described in this chapter, along with many others.

6.2 Implementation

The equivalence relation part is implemented by the class `EquivalenceRelation`. Equivalence classes, that are sets of variable indexes, are represented by `java.util.BitSets`. When pairs of equivalent variables are needed, they are derived from equivalence classes. `EquivalenceRelation` is an immutable class: adding or removing classes or pairs doesn't affect the current instance, but instead creates another `EquivalenceRelation`.

`BDDER` is the class that implements an ER representation. It is a subclass of `BDD`, and have an additional field for the equivalence relation part.

`BDDER`, like their counterparts `BDD`, are kept in a *factory*, specifically in an `ERFactory`, a subclass of a `Factory`. Basic BDDERs are produced with the methods `makeZero()`, `makeOne()`, `makeVar(int v)` and `makeNotVar(int v)`. Complex BDDERs can be constructed from these basic blocks with methods like `and`, `or`, etc.

6.3 Experiments

For experimenting in ER, we use the *knight's tour problem*, that can create BDDs with a high number of equivalent variables. This algorithm creates BDDs with many equivalent variables, and thus the *ER* representation uses far less BDD nodes than the normal one.

Figure 6.2 compares the number of nodes created by instances of the knight's tour problem by using the two different representations.

It can be seen that for $N = 14$, roughly 7% of the nodes are created by the *ER* factory, with respect to the standard representation.

Figure 6.3 shows execution times.

6.3.1 N-queens

In the n-queens problem, there are little or no equivalences, and so the overhead of the *ER* representation is more pronounced. Figure 6.4 shows the time comparison between the two representations.

6.3.2 Julia

Figure 6.5 shows times for the execution of the Nullness checker of Julia on several programs. Here, too, there is no advantage in using the *ER* representation.

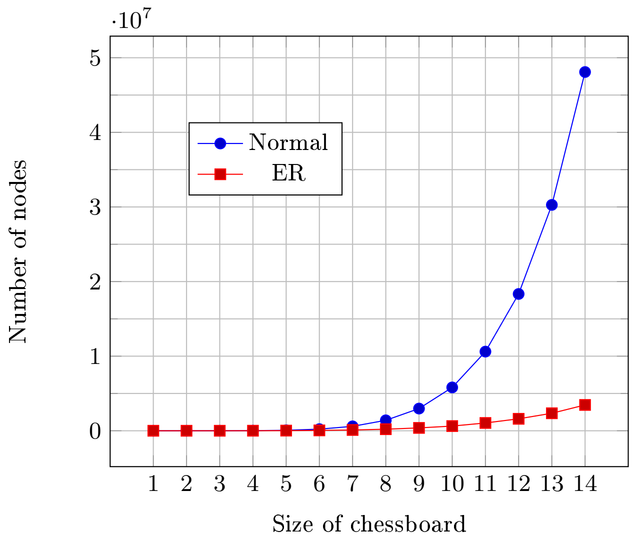


Fig. 6.2: BDD nodes created by instances of the knight's tour problem

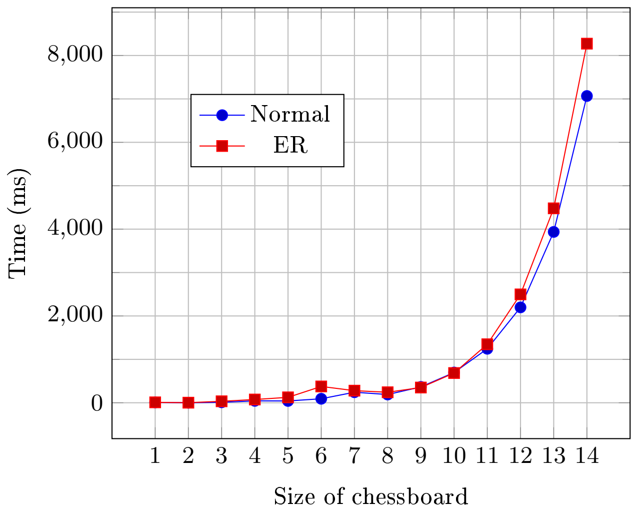


Fig. 6.3: Time to solve instances of the knight's tour problem

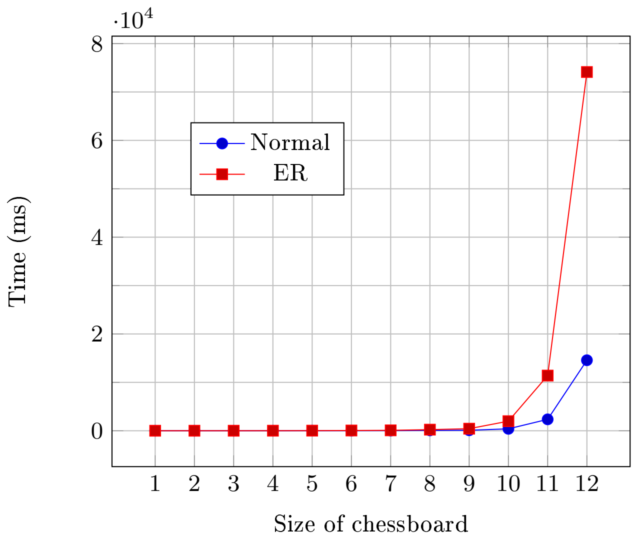


Fig. 6.4: Time to solve instances of the n-queens problem

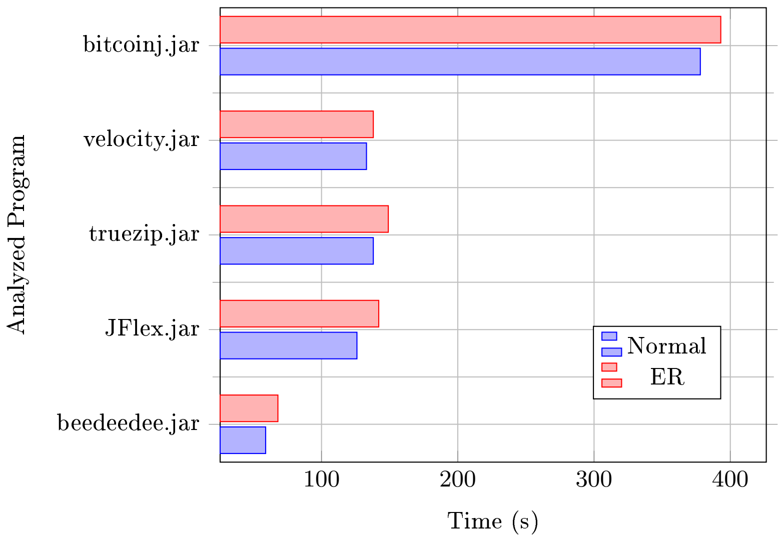


Fig. 6.5: Execution time for the *Nullness* checker of Julia

7 Integrity Check

Due to the use of BDDs in safety critical applications, it is desirable that the probability of error in data stored in memory is reduced to a minimum. Bringing inspiration from [45], we implemented a version of the unique table performing an integrity check on the data it stores at every access, or at the user's request. The table can become corrupt because of defective memory modules, or even because of cosmic rays changing bits in memory chips [15]. If the user requests an *off-line* checksum of the table, it is his responsibility to check it against an old sum saved in the past. If instead he chooses *on-line* checking, at every read access the library takes care to compare the checksum of the current node against that saved at the last write access. Of course, these checks are not free, as they involve reading the table and computing sums, but still some users might prefer to have greater confidence in BDD data stored in memory, at the cost of increased computation time. As far as we know, our library is the only one providing this option.

7.1 Usage

The factory is created similarly to a normal one. There are parameters to indicate initial sizes of the table and the caches, and an additional `boolean` flag indicating the option of on-line checking. This is an example of *off-line* checking:

```
IntegrityCheckFactory factory = new IntegrityCheckFactory(10, 10, false);
long checksum1 = factory.computeChecksum();
... // some time passes...
long checksum2 = factory.computeChecksum();
// checksum1 == checksum2 if the table haven't changed
```

This instead is an example of *on-line* checking:

```
IntegrityCheckFactory factory = new IntegrityCheckFactory(10, 10, true);
```

```
...
factory.makeZero();
...
// cosmic ray changes row for zero!
...
factory.makeZero(); // checksum changed, exception!
```

When an exception is thrown in response to a checksum error, the user needs to restart the computation with a fresh node table, as the old one cannot be relied upon.

7.2 Implementation

To add checks, we need to add code to the factory and the unique table, and so we have two new classes: `IntegrityCheckFactory`, that extends class `Factory`, and `IntegrityCheckUniqueTable`, that extends class `ResizingAndGarbageCollectedUniqueTable`.

`IntegrityCheckFactory` adds a method to the public interface of `Factory`, `long computeChecksum()`, that computes a *CRC32* checksum of the BDD data, namely the `ut` and `H` arrays. Moreover, if *on-line* checking is requested, it also substitutes an instance of `IntegrityCheckUniqueTable` as the unique table to use. Computation of *CRC32* is performed via standard Java methods:

```
public long computeChecksum() {
    long crc32h = computeCRC32(ut.H);
    long crc32ut = computeCRC32(ut.ut);
    return crc32h + crc32ut;
}

private long computeCRC32(int[] a) {
    // convert int[] to byte[]
    ByteBuffer byteBuffer = ByteBuffer.allocate(a.length * 4);
    IntBuffer intBuffer = byteBuffer.asIntBuffer();
    intBuffer.put(a);
    byte[] byteArray = byteBuffer.array();

    Checksum checksum = new CRC32();
    checksum.update(byteArray, 0, byteArray.length);
    return checksum.getValue();
}
```

`IntegrityCheckUniqueTable` extends `ResizingAndGarbageCollectedUniqueTable`. It adds an integer slot to a row in the table, to memorize a checksum for the node, bringing the occupation per node to 28 bytes. At every write access, a checksum of the row being modified is computed and saved in the dedicated slot. Then, before reading data from a row of the table, a checksum is again computed, and compared to the saved one. If the two values differ, an unchecked runtime exception of type `CorruptedNodeException` is thrown. The computation of the checksum for a node is performed in a way similar to the *off-line* checking:

```
private int nodeChecksum(int id) {
    ByteBuffer byteBuffer = ByteBuffer.allocate((getNodeSize() − 1) ∗ 4);
    byteBuffer.putInt(ut[id ∗ getNodeSize() + VAR_OFFSET]);
    byteBuffer.putInt(ut[id ∗ getNodeSize() + LOW_OFFSET]);
    byteBuffer.putInt(ut[id ∗ getNodeSize() + HIGH_OFFSET]);
    byteBuffer.putInt(ut[id ∗ getNodeSize() + NEXT_OFFSET]);
    byteBuffer.putInt(ut[id ∗ getNodeSize() +
                                      HASHCODEAUX_OFFSET]);
    byte[] byteArray = byteBuffer.array();
    Checksum checksum = new CRC32();
    checksum.update(byteArray, 0, byteArray.length);
    return (int) checksum.getValue();
}
```

7.2.1 Cyclic Redundancy Check

Our integrity check uses *CRC32*, an instance of a *Cyclic Redundancy Check* or *CRC*. CRCs were introduced as a means for error detection. It works by first encoding binary strings as polynomials, using the bits of an n-bits string as binary coefficient of a polynomial of degree $n-1$. For example, the string 110101 is encoded by the polynomial $1+x+x^3+x^5$. Then, the *generator polynomial* $P(x)$, of degree k, is chosen to encode a message represented by the polynomial $G(x)$. The encoding works as follows.

$x^kG(x)$ is divided by $P(x)$, in order to find the remainder $R(x)$:

$$x^kG(x) = Q(x)P(x) + R(x)$$

Then the polynomial representing the code is

$$F(x) = Q(x)P(x) = x^kG(x) + R(x)$$

An encoded string may contain errors, and we can represent it with the polynomial

$$H(x) = F(x) + E(x)$$

$F(x)$ represents the correct string, while $E(x)$ represents a string with ones in place of the errors, so that summing them up reproduces the string with errors. To check the string for errors, we can divide it by $P(x)$. If we obtain a remainder, the string contains errors, but if it doesn't we don't know if the string has no errors, or instead the error $E(x)$ is not detectable with the chosen $P(x)$.

7.3 Experiments

Figure 7.1 shows the performance of the n-queens construction for board size up to 12, with the normal factory as well as with that performing on-line integrity check. The resulting overhead is acceptable, given the additional security guarantees on the correctness of results.

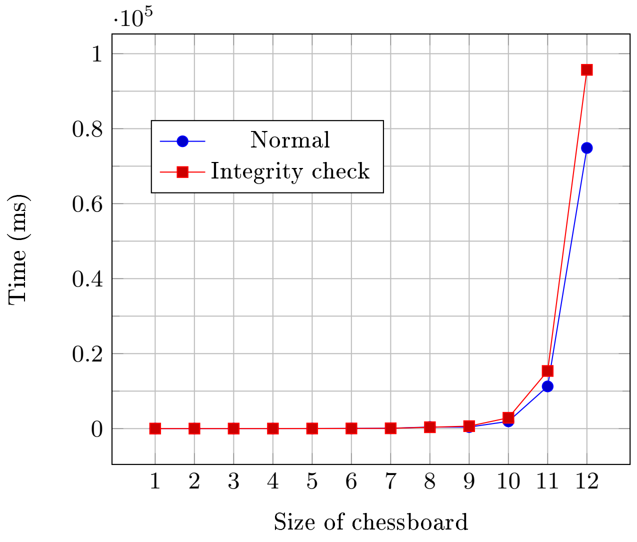


Fig. 7.1: Time to solve instances of the n-queens problem, with normal and integrity check factories

Part II

Program Analysis

8

Identification of Injection Vulnerabilities

BDDs were already used in the past for static analysis. The Nullness analysis implemented in Julia [102] represents propagation of nullness of variables using Boolean functions in the form of BDDs. Groundness analysis of logic programs represents positive Boolean functions with BDDs [25]. In this work, we use BDDs to represent Boolean functions in a novel analysis implemented in Julia, our static program analyzer for Java, that finds possible injections of untrusted, or tainted data, into programs.

The most dangerous security-related software errors, according to CWE 2011, are those leading to injection attacks — user-provided data that result in undesired database access and updates (*SQL-injections*), dynamic generation of web pages (*cross-site scripting-injections*), redirection to user-specified web pages (*redirect-injections*), execution of OS commands (*command-injections*), class loading of user-specified classes (*reflection-injections*), and many others. This chapter describes a flow- and context-sensitive static analysis that automatically identifies if and where injections of tainted data can occur in a program. The analysis models explicit flows of tainted data. Its notion of taintedness applies also to reference (non-primitive) types dynamically allocated in the heap, and is object-sensitive and field-sensitive. The analysis works by translating the program into Boolean formulas that model all possible flows.

This chapter is organized as follows. Sec. 8.1 gives an example of injection and clarifies the importance of a new notion of taintedness for values of reference type. Sec. 8.2 defines a concrete semantics for Java bytecode. Sec. 8.3 defines our new object-sensitive notion of taintedness for values of reference type and its use to induce an object- and field-sensitive abstract interpretation of the concrete semantics. Sec. 8.4 presents experiments with the implementation of the analysis. Extended definitions and proofs are in the appendices.

8.1 Example

Fig. 8.1 is a Java servlet that suffers from SQL-injection and cross-site scripting-injection attacks. (For brevity, the figure omits exception-handling code.)

A *servlet* (lines 1 and 2) is code that listens to HTTP network connection requests, retrieves its parameters, and runs some code in response to each request. The response (line 2) may be presented as a web page, XML, or JSON. This is a standard way of implementing dynamic web pages and web services. The user of a servlet connects to the web site and provides the parameters through the URL, as in `http://my.site.com/myServlet?user=spoto`. Code retrieves these through the `getParameter` method (line 5). Lines 8 and 10 establish a connection to the database of the application, which is assumed to define a table `User` (line 27) of the users of the service. Line 27 builds an SQL query from the user name provided as parameter. This query is reported to the response (line 15) and executed (line 17). The result is a relational table of all users matching the given criterion (the `user` parameter might be a specific name or a wildcard that matches more users). This table is then printed to the response (lines 17–20).

The interesting point here is that the user of this servlet is completely free to specify the value of the `user` parameter. In particular, she can provide a string that actually lets line 17 run *any* possible database command, including malicious commands that erase its content or insert new rows. For instance, if the user supplies the string "`'; DROP TABLE User; --`" as `user`, the resulting concatenation is an SQL command that erases the `User` table from the database. In literature, this is known as an SQL-injection attack and follows from the fact that user (*tainted*) input flows from the `request` *source* into the `executeQuery` *sink* method. There is no SQL-injection at line 22, although it looks very much like line 17, since the query there is not computed from user-provided input.

Another risk exists at lines 15 and 20. There, data is printed to the response object, and is typically interpreted by the client as HTML contents. A malicious user might have provided a `user` parameter that contains arbitrary HTML tags, including tags that will let the client execute scripts (such as Javascript). This might result in evil. For instance, if the user injects a crafted URL such as "`http://my.site.com/myServlet?user=<script>malicious</script>`", the parameter `user` holds "`<script>malicious</script>`". At line 15 this code is sent to the user's browser and interpreted as Javascript, running any `malicious` Javascript. In literature, this is known as cross-site scripting-injection and follows from the fact that user (*tainted*) input from the `request` *source* flows into the *sink* output writer of the response object. The same might happen at line 20, where the flow is more complex: in other parts of the application, the user might save her address to the database and store malicious code instead; line 20 will fetch this malicious code and send it to the browser of the client to run it.

```
protected void doGet(HttpServletRequest request,
                     HttpServletResponse response) {
  String user = request.getParameter("user");
  String url = "jdbc:mysql://192.168.2.128:3306/";
  String dbName = "anvayaV2", driver = "com.mysql.jdbc.Driver";
  String userName = "root", password = "";

  Class.forName(driver).newInstance();
  try (Connection conn = DriverManager.getConnection(url + dbName,
                     userName, password);
       PrintWriter out = response.getWriter()) {

    Statement st = conn.createStatement();
    String query = wrapQuery(user);
    out.println("Query : " + query);

    ResultSet res = st.executeQuery(query);
    out.println("Results:");
    while (res.next())
      out.println("\t\t" + res.getString("address"));

    st.executeQuery(wrapQuery("dummy"));
  }
}

private String wrapQuery(String s) {
  return "SELECT * FROM User WHERE userId=’" + s + "’";
}
}
```

Fig. 8.1: A Java servlet that suffers from SQL and cross-site scripting-injections.

Many kinds of injections exist. They arise from information flows from what the user can specify (the parameter of the request, input from console, data on a database) to specific methods, such as `executeQuery` (SQL-injection), `print` (cross-site scripting-injection), reflection methods (that allow one to load any class or execute any method and lead to a reflection-injection), `execute` (that allows one to run any operating system command and leads to a command-injection), etc. We focus on the identification of flows of tainted information, not on the exact enumeration of sources and sinks. Our approach can be instantiated from well-known lists of sources and sinks in the literature.

8.2 Denotational Semantics of Java Bytecode

This section presents a denotational semantics for Java bytecode, which we will use to define an abstraction for taintedness analysis (Sec. 8.3). The same semantics has been used for nullness analysis [102] and has been proved equivalent [88] to an operational semantics. The only difference is that here primitive values are decorated with their taintedness.

We assume a Java bytecode program P given as a collection of graphs of *basic blocks* of code, one for each method. Bytecodes that might throw exceptions are linked to a handler starting with a `catch`, possibly followed by bytecodes selecting the right kind of exception. For simplicity, we assume that the only primitive type is `int` and the only reference types are *classes*; we only allow *instance* fields and methods; and method parameters cannot be reassigned inside their body. Our implementation handles full Java bytecode.

Definition 8.1 (Classes). *The set of* classes $\mathbb{K}$ *is partially ordered w.r.t. the* subclass relation $\leq$*. A* type *is an element of* $\mathbb{K}\cup\{\texttt{int}\}$*. A class* $\kappa \in \mathbb{K}$ *defines* instance fields $\kappa.f : t$ *(field f of type t defined in κ) and* instance methods $\kappa.m(t_1,\ldots,t_n) : t$ *(method m with arguments of type $t_1,\ldots,t_n$, returning a value of type t, possibly* `void`*). We consider constructors as methods returning* `void`*. If it does not introduce confusion, we write f and m for fields and methods.*

A *state* provides *values* to program variables. *Tainted* values are computed from servlet/user input; others are *untainted.* Taintedness for reference types (such as string `request` in Fig. 8.1) will be defined later as a reachability property from the reference (Def. 8.8); primitive tainted values are explicitly marked in the state.

Definition 8.2 (State). *A* value *is an element of* $\mathbb{Z}\cup\boxed{\mathbb{Z}}\cup\mathbb{L}\cup\{\texttt{null}\}$*, where* $\mathbb{Z}$ *are untainted integers,* $\boxed{\mathbb{Z}}$ *are tainted integers, and* $\mathbb{L}$ *is a set of* locations*. A* state *is a triple* $\langle l \,\|\, s \,\|\, \mu\rangle$ *where l are the values of the* local variables*, s the values of the* operand stack*, which grows leftwards, and μ a* memory *that binds locations to* objects*. The empty stack is written ε. Stack concatenation is $::$ with $s::\varepsilon$ written as just s. An object o belongs to class $o.\kappa \in \mathbb{K}$ (is an* instance *of $o.\kappa$) and maps identifiers (the fields f of $o.\kappa$ and of its superclasses) into values $o.f$. The set of states is Ξ. We write $\Xi_{i,j}$ when we want to fix the number i of local variables and j of stack elements. A value v* has type *t in a state $\langle l \,\|\, s \,\|\, \mu\rangle$ if $v \in \mathbb{Z}\cup\boxed{\mathbb{Z}}$ and $t = \texttt{int}$, or $v = \texttt{null}$ and $t \in \mathbb{K}$, or $v \in \mathbb{L}$, $t \in \mathbb{K}$ and $\mu(v).\kappa \leq t$.*

Example 8.3. Let state $\sigma = \langle[3,\texttt{null},\boxed{4},\ell] \,\|\, \boxed{3}::\ell''::\ell'' \,\|\, \mu\rangle \in \Xi_{4,3}$, with $\mu = [\ell \mapsto o, \ell' \mapsto o', \ell'' \mapsto o'']$, $o.f = \ell'$, $o.g = 13$, $o'.g = \boxed{17}$ and $o''.g = 10$. Local 0 holds the integer 3 and local 2 holds the integer 4, marked as computed from servlet/user input. The top of the stack holds 3, marked as computed from servlet/user input. The next two stack elements are aliased to ℓ''. Location ℓ is bound to object o, whose field f holds ℓ' and whose field g holds the untainted integer 13. Location ℓ' is bound to o' whose field g holds a tainted integer $\boxed{17}$. Location ℓ'' is bound to o'' whose field g holds the untainted value 10.

The Java Virtual Machine (JVM) allows exceptions. Hence we distinguish *normal* states $\sigma \in \Xi$, arising during the normal execution of a piece of code, from *exceptional* states $\underline{\sigma} \in \underline{\Xi}$, arising *just after* a bytecode that throws an exception. The latter have only one stack element, *i.e.*, the location of the thrown exception object, also in the presence of nested exception handlers [72]. The semantics of a bytecode is then a *denotation* from an *initial* to a *final* state.

Definition 8.4 (JVM State and Denotation). *The set of* JVM states *(from now just* states*) with i local variables and j stack elements is $\Sigma_{i,j} = \Xi_{i,j} \cup \underline{\Xi}_{i,1}$. A* denotation *is a partial map from an* input *or* initial *state to an* output *or* final *state; the set of denotations is Δ or $\Delta_{i_1,j_1 \to i_2,j_2} = \Sigma_{i_1,j_1} \to \Sigma_{i_2,j_2}$ to fix the number of local variables and stack elements. The* sequential composition *of $\delta_1, \delta_2 \in \Delta$ is $\delta_1;\delta_2 = \lambda\sigma.\delta_2(\delta_1(\sigma))$, which is undefined when $\delta_1(\sigma)$ or $\delta_2(\delta_1(\sigma))$ is undefined.*

In $\delta_1;\delta_2$, the idea is that δ_1 describes the behavior of an instruction ins_1, δ_2 that of an instruction ins_2 and $\delta_1;\delta_2$ that of the execution of ins_1 and then ins_2.

At each program point, the number i of local variables and j of stack elements and their types are statically known [72], hence we can assume the semantics of the bytecodes undefined for input states of wrong sizes or types. Readers can find the denotations of bytecode instructions in appendices, together with the construction of the concrete fixpoint collecting semantics of Java bytecode, explicitly targeted at abstract interpretation, since it only requires to abstract three concrete operators ;, $\cup$, and *extend* on $\wp(\Delta)$, *i.e.*, on the subsets of Δ and the denotation of each single bytecode distinct from `call`. The operator *extend* plugs a method's denotation at its calling point and implements `call`. The concrete fixpoint computation is in general infinite, but its abstractions converge in a finite number of steps if, as in Sec. 8.3, the abstract domain has no infinite ascending chain.

Basic instructions. Bytecode `const` v pushes $v \in \mathbb{Z} \cup \{\texttt{null}\}$ on the stack: $const\ v = \lambda\langle l \,\|\, s \,\|\, \mu\rangle.\langle l \,\|\, v :: s \,\|\, \mu\rangle$ (s might be ε). The λ-notation defines a partial map, undefined on exceptional states since $\langle l \,\|\, s \,\|\, \mu\rangle$ is not underlined. That is, *const* v is executed when the JVM is in a normal state. This holds for *all* bytecodes but `catch`, that starts the exceptional handlers from an exceptional state. Bytecode `dup` t duplicates the top of the stack, of type t: $dup\ t = \lambda\langle l \,\|\, top :: s \,\|\, \mu\rangle.\langle l \,\|\, top :: top :: s \,\|\, \mu\rangle$. Bytecode `load` k t pushes on the stack the value of local variable number k, that must exist and have type t: $load\ k\ t = \lambda\langle l \,\|\, s \,\|\, \mu\rangle.\langle l \,\|\, l[k] :: s \,\|\, \mu\rangle$. Conversely, bytecode `store` k t pops the top of the stack of type t and writes it in local variable k: $\mathit{store}\ k\ t = \lambda\langle l \,\|\, \mathit{top} :: s \,\|\, \mu\rangle.\langle l[k := \mathit{top}] \,\|\, s \,\|\, \mu\rangle$. If l has less than $k+1$ variables, the resulting set of local variables gets expanded. Binary arithmetic bytecodes such as `add` consume the operands, *i.e.*, the topmost two elements of the stack, and produce the arithmetic result: $\mathit{add} = \lambda\langle l \,\|\, v_1 :: v_2 :: s \,\|\, \mu\rangle.\langle l \,\|\, (v_2 + v_1) :: s \,\|\, \mu\rangle$. Here, $+$ is the extended addition operator that yields a tainted sum if and only if at least one of its operands is tainted. The semantics of a conditional bytecode is undefined when its condition is false. For instance, `ifne` t checks if the top of the stack, of type t, is not 0 nor $\boxed{0}$ when $t = \texttt{int}$, and is not `null` otherwise. Its semantics $\mathit{ifne}\ t$ is

$$\lambda\langle l \,\|\, \mathit{top} :: s \,\|\, \mu\rangle. \begin{cases} \langle l \,\|\, s \,\|\, \mu\rangle & \text{if } \mathit{top} \notin \{0, \boxed{0}, \texttt{null}\}, \\ \mathit{undefined} & \text{otherwise.} \end{cases}$$

Memory-manipulating instructions. Some bytecodes deal with objects in memory: `new` κ pushes on the stack a reference to a new object n of class κ, with reference fields set to `null`. Its semantics $\mathit{new}\ \kappa$ is

$$\lambda\langle l \,\|\, s \,\|\, \mu\rangle. \begin{cases} \langle l \,\|\, \ell :: s \,\|\, \mu[\ell := n]\rangle & \text{if there is memory} \\ \underline{\langle l \,\|\, \ell \,\|\, \mu[\ell := \mathit{oome}]\rangle} & \text{otherwise} \end{cases}$$

with $\ell \in \mathbb{L}$ fresh and oome new instance of `java.lang.OutOfMemoryError`. This is the first bytecode that throws an exception. Bytecode `getfield` $\kappa.f{:}t$ reads the field $\kappa.f{:}t$ of the object pointed by the top rec (the *receiver*) of the stack, of type κ. Its semantics $\mathit{getfield}\ \kappa.f{:}t$ is

$$\lambda\langle l \,\|\, \mathit{rec} :: s \,\|\, \mu\rangle. \begin{cases} \langle l \,\|\, \mu(\mathit{rec}).f :: s \,\|\, \mu\rangle & \text{if } \mathit{rec} \neq \texttt{null}, \\ \underline{\langle l \,\|\, \ell \,\|\, \mu[\ell \mapsto \mathit{npe}]\rangle} & \text{otherwise} \end{cases}$$

with $\ell \in \mathbb{L}$ fresh and npe new instance of `java.lang.NullPointerException`. Bytecode `putfield` $\kappa.f{:}t$ moves the top of the stack, of type t, in the field $\kappa.f{:}t$ of the object pointed by a value rec of type κ below top. Its semantics $\mathit{putfield}\ \kappa.f{:}t$ is (ℓ and npe are as before)

$$\lambda\langle l \| \mathit{top} :: \mathit{rec} :: s \| \mu\rangle. \begin{cases} \langle l \| s \| \mu[\mu(\mathit{rec}).f := \mathit{top}]\rangle & \text{if } \mathit{rec} \neq \texttt{null}, \\ \underline{\langle l \,\|\, \ell \,\|\, \mu[\ell := \mathit{npe}]\rangle} & \text{otherwise.} \end{cases}$$

Exception handling instructions. Bytecode `throw` κ throws the object of type $\kappa \leq$ `java.lang.Throwable` pointed by the top of the stack. Its semantics $\mathit{throw}\ \kappa$ is (ℓ and npe are as before)

$$\lambda\langle l \,\|\, \mathit{top} :: s \,\|\, \mu\rangle. \begin{cases} \underline{\langle l \,\|\, \mathit{top} \,\|\, \mu\rangle} & \text{if } \mathit{top} \neq \texttt{null}, \\ \underline{\langle l \,\|\, \ell \,\|\, \mu[\ell \mapsto \mathit{npe}]\rangle} & \text{if } \mathit{top} = \texttt{null}. \end{cases}$$

Bytecode `catch` starts an exception handler from an exceptional state: it transforms it into a normal one, used by the implementation of the handler: $\mathit{catch} = \lambda\underline{\langle l \,\|\, \mathit{top} \,\|\, \mu\rangle}.\langle l \,\|\, \mathit{top} \,\|\, \mu\rangle$ where $\mathit{top} \in \mathbb{L}$ is an instance of `java.lang.Throwable`. The correct handler for a specific class of exception is then selected on the basis of the runtime class of the exception object by a bytecode `exception_is` K that filters the states whose stack top points to an instance of a class in $K \subseteq \mathbb{K}$. Its semantics *exception_is* K is

$$\lambda\langle l \parallel top \parallel \mu\rangle . \begin{cases} \langle l \parallel top \parallel \mu\rangle & \text{if } top \in \mathbb{L},\ \mu(top).\kappa \in K, \\ \mathit{undefined} & \text{otherwise.} \end{cases}$$

Method call and return instructions. A method $M = \kappa.m(t_1, \dots, t_n) : t$ starts with a bytecode `receiver_is` K asserting that the runtime class of the receiver (local variable 0) is in a set K statically computed from the look-up rules of the language. The semantics of *receiver_is* K is

$$\lambda\langle l \parallel \varepsilon \parallel \mu\rangle . \begin{cases} \langle l \parallel \varepsilon \parallel \mu\rangle & \text{if } l[0] \in \mathbb{L},\ \mu(l[0]).\kappa \in K, \\ \mathit{undefined} & \text{otherwise,} \end{cases}$$

with ℓ and *npe* as before. At the beginning of M, the stack is ε and local variables hold exactly its $n+1$ actual arguments (including `this`). At its end, a `return` t bytecode leaves on the stack the return value of type t only, or a `return` bytecode just returns, if $t =$ `void`. Without loss of generality, we assume that `return` is only executed when there is no other value on the stack. Hence $\mathit{return}\ t = \lambda\langle l \parallel top \parallel \mu\rangle.\langle l \parallel top \parallel \mu\rangle$ and $\mathit{return} = \lambda\langle l \parallel \varepsilon \parallel \mu\rangle.\langle l \parallel \varepsilon \parallel \mu\rangle$. Overall, the semantics of the code of M is hence a denotation δ from a state $\langle [v_0, \dots, v_n] \parallel \varepsilon \parallel \mu\rangle$ to a state $\sigma = \langle l' \parallel top \parallel \mu'\rangle$, with $top = \varepsilon$ when $t =$ `void`, if M returns normally, or to a state $\sigma = \underline{\langle l' \parallel top \parallel \mu'\rangle}$, with *top* pointing to an exception e if M throws e. From the point of view of the caller of M, its i local variables l are not affected by the call and the actual arguments $v_0, \dots, v_n$ are popped from its stack, of height $j = b+n+1$, and replaced with *top* (if any). We model this through $\mathit{extend}^{i,j}_M \in \Delta_{n+1,0 \to i',r} \to \Delta_{i,j \to i,b+r}$, with $r = 0$ if $t =$ `void` and $r = 1$ otherwise, defined as (ℓ and *npe* are as before)

$$\lambda\langle l \| v_n :: \cdots :: v_0 :: s \| \mu\rangle . \begin{cases} \underline{\langle l \,\| \ell \| \, \mu[\ell := npe]\rangle} & \text{if } v_0 = \texttt{null} \\ \langle l \,\| top :: s \| \, \mu'\rangle & \text{if } v_0 \in Loc,\ \sigma \in \Xi, \\ \underline{\langle l \| top \| \mu'\rangle} & \text{if } v_0 \in Loc,\ \sigma \in \underline{\Xi}. \end{cases}$$

The Denotational Semantics. A semantics ι of P is an *interpretation* that specifies the behavior of each *block* b in P by providing a set $\iota(b)$ of denotations. They represent possible executions starting at b and continuing with b's successor blocks until a block with no successor is reached. *Sets* are typical of a *collecting* semantics [39], able to model *properties* of denotations. The operators *extend* and ; over denotations are consequently extended to sets of denotations.

Definition 8.5 (Interpretation). *An* interpretation *is a map from P's blocks into $\wp(\Delta)$. The set of interpretations $\mathbb{I}$ is ordered by pointwise set-inclusion.* $\square$

Given $\iota \in \mathbb{I}$, we define the set $[\![b]\!]^\iota \subseteq \Delta$ of all the executions, induced by ι, that start at b and continue with b's successors until a block with no successors is reached: we compose sequentially the denotations of the instructions inside b and those of the successor blocks $b_1, \ldots, b_n$, as given by ι. For `call`s, we *extend* the denotations of the first block of the called method(s), as given by ι.

Definition 8.6 (Denotations of Instructions and Blocks). *Let $\iota \in \mathbb{I}$. The* denotations in ι of an instruction *are $[\![\texttt{ins}]\!]^\iota = \{ins\}$ if* `ins` *is not a* `call`*, and $[\![\texttt{call}\ M_1, \ldots, M_q]\!]^\iota$ $= \cup_{1 \le s \le q} extend^{i,j}_{M_s}(\iota(b_{M_s}))$ otherwise, where $\{M_1, \ldots, M_q\}$ is a superset of the methods that might be called (computed by some class analysis), b_{M_s} the block where M_s starts, i the number of local variables and j the height of the stack where the* `call` *occurs. Function $[\![\_]\!]^\iota$ is extended to blocks:*

$$\left[\!\!\left[\ \boxed{\begin{array}{c}\texttt{ins}_1\\ \cdots\\ \texttt{ins}_\texttt{n}\end{array}}\begin{array}{l}\to b_1\\ \cdots\\ \to b_m\end{array}\right]\!\!\right]^\iota = [\![\texttt{ins}_1]\!]^\iota ; \cdots ; [\![\texttt{ins}_\texttt{n}]\!]^\iota \underbrace{; (\iota(b_1) \cup \cdots \cup \iota(b_m))}_{Cont}$$

where Cont is missing when $m = 0$. □

Note that Def. 8.6 uses an operator $\cup$ over $\wp(\Delta)$.

Loops and recursion make the blocks of P interdependent and hence a denotational semantics needs a fixpoint computation: it starts from the *empty* interpretation ι_0, such that $\iota_0(b) = \emptyset$ for all blocks b of P, computes $\iota_1 = T_P(\iota_0)$ and iterates the application of T_P until a fixpoint (for efficiency, one can perform local, smaller fixpoints on the strongly-connected components of blocks).

Definition 8.7 (Denotational Semantics). *We define $T_P : \mathbb{I} \to \mathbb{I}$ as $T_P(\iota)(b) = [\![b]\!]^\iota$ for every $\iota \in \mathbb{I}$ and block b of P. Its least fixpoint exists and can be computed with a (possibly infinite) iterative application of T_P from ι_0 [88]. It is the* denotational semantics *of P.* □

Abstractions of T_P over a domain with no infinite ascending chains (such as that in Sec. 8.3) reach the abstract fixpoint in a finite number of iterations.

8.3 Taintedness Analysis

This section defines an abstract interpretation [39] of the concrete semantics of Sec. 8.2, whose abstract domain is made of Boolean formulas whose models are consistent with all possible ways of propagating taintedness in the concrete semantics. The concrete semantics works over $\wp(\Delta)$ and is built from singletons (sets made of a single $\delta \in \Delta$), one for each bytecode, with three operators ;, $\cup$, and *extend*. Hence we define here correct abstractions of those sets and operators.

Our analysis assumes that three other analyses have been performed in advance. (1) $\mathit{reach}(v, v')$ is true if (the location held in) v' is reachable from (the location held in) v. (2) $\mathit{share}(v, v')$ is true if from v and v' one can reach a common location. (3) $\mathit{updated}_M(l_k)$ is true if some call in the program to method M might ever modify an object reachable from local variable l_k. All three analyses are conservative overapproximations of the actual (undecidable) relations. Our implementation computes these predicates as in [85], [96], and [57], respectively.

Primitive values are explicitly marked as tainted (Def. 8.2), while taintedness for references is indirectly defined in terms of reachability of tainted values. Hence, this notion allows $a.f$ and $b.f$ to have distinct taintedness, depending of the taintedness of variables a and b (object-sensitivity).

Definition 8.8 (Taintedness). *Let $v \in \mathbb{Z} \cup \boxed{\mathbb{Z}} \cup \mathbb{L} \cup \{\texttt{null}\}$ be a value and μ a memory. The property of being* tainted *for v in μ is defined recursively as: $v \in \boxed{\mathbb{Z}}$ or ($v \in \mathbb{L}$ and $o = \mu(v)$ and there is a field f such that $o(f)$ is tainted in μ).*

A first abstraction step selects the variables that, in a state, hold tainted data. It yields a logical model where a variable is true if it holds tainted data.

Definition 8.9 (Tainted Variables). *Let $\sigma \in \Sigma_{i,j}$. Its* tainted variables *are*

$$\mathit{tainted}(\sigma) = \begin{cases} \{l_k \mid l[k] \textit{ is tainted in } \mu,\ 0 \le k < i\} \cup \{s_k \mid v_k \textit{ is tainted in } \mu,\ 0 \le k < j\} \\ \quad \textit{if } \sigma = \langle l \,\|\, v_{j-1} :: \cdots :: v_0 \,\|\, \mu \rangle \\ \{l_k \mid l[k] \textit{ is tainted in } \mu,\ 0 \le k < i\} \cup \{e, s_0\} \\ \quad \textit{if } \sigma = \underline{\langle l \,\|\, v_0 \,\|\, \mu \rangle} \textit{ and } v_0 \textit{ is tainted in } \mu \\ \{l_k \mid l[k] \textit{ is tainted in } \mu,\ 0 \le k < i\} \cup \{e\} \\ \quad \textit{if } \sigma = \underline{\langle l \,\|\, v_0 \,\|\, \mu \rangle} \textit{ and } v_0 \textit{ is not tainted in } \mu. \end{cases}$$

Example 8.10. Consider σ from Ex. 8.3. We have $\mathit{tainted}(\sigma) = \{l_2, l_3, s_2\}$, since tainted data is reachable from both locations ℓ and ℓ', but not from ℓ''.

To make the analysis flow-sensitive, distinct variables abstract the input (marked with $\check{\ }$) and output (marked with $\hat{\ }$) of a denotation. If S is a set of identifiers, then $\check{S} = \{\check{v} \mid v \in S\}$ and $\hat{S} = \{\hat{v} \mid v \in S\}$. The abstract domain contains Boolean formulas that constraint the relative taintedness of local variables and stack elements. For instance, $\check{l}_1 \to \hat{s}_2$ states that if local variable l_1 is tainted in the input of a denotation, then the stack element s_2 is tainted in its output.

Definition 8.11 (Taintedness Abstract Domain $\mathbb{T}$). *Let $i_1, j_1, i_2, j_2 \in \mathbb{N}$. The taintedness abstract domain $\mathbb{T}_{i_1,j_1 \to i_2,j_2}$ is the set of Boolean formulas over $\{\check{e}, \hat{e}\} \cup \{\check{l}_k \mid 0 \le k < i_1\} \cup \{\check{s}_k \mid 0 \le k < j_1\} \cup \{\hat{l}_k \mid 0 \le k < i_2\} \cup \{\hat{s}_k \mid 0 \le k < j_2\}$ (modulo logical equivalence).*

Example 8.12. $\phi = (\check{l}_1 \leftrightarrow \hat{l}_1) \wedge (\check{l}_2 \leftrightarrow \hat{l}_2) \wedge (\check{l}_3 \leftrightarrow \hat{l}_3) \wedge \neg\check{e} \wedge \neg\hat{e} \wedge (\check{s}_0 \leftrightarrow \hat{l}_0) \in \mathbb{T}_{4,1 \to 4,0}$.

The concretization map γ states that a $\phi \in \mathbb{T}$ abstracts those denotations whose behavior, w.r.t. the propagation of taintedness, is a model of ϕ.

Proposition 8.13 (Abstract Interpretation). *$\mathbb{T}_{i_1,j_1\to i_2,j_2}$ is an abstract interpretation of $\wp(\Delta_{i_1,j_1\to i_2,j_2})$ with $\gamma : \mathbb{T}_{i_1,j_1\to i_2,j_2} \to \wp(\Delta_{i_1,j_1\to i_2,j_2})$ given by*

$$\gamma(\phi) = \left\{ \delta \in \Delta_{i_1,j_1\to i_2,j_2} \;\middle|\; \begin{array}{l} \textit{for all } \sigma \in \Sigma_{i_1,j_1} \textit{ s.t. } \delta(\sigma) \textit{ is defined} \\ \check{tainted}(\sigma) \cup \hat{tainted}(\delta(\sigma)) \models \phi \end{array} \right\}.$$

Lemma 8.14. *The map γ of Prop. 8.13 is co-additive.*

Proof. Let $i_1, i_1, j_2, j_2 \in \mathbb{N}$, $I \subseteq \mathbb{N}$ and $\{\phi_i\}_{i\in I} \subseteq \mathbb{T}_{i_1,j_1\to i_2,j_2}$. We prove that $\gamma(\wedge_{i\in I}\phi_i) = \cap_{i\in I}\gamma(\phi_i)$:

$$\begin{aligned}
\gamma(\wedge_{i\in I}\phi_i) &= \left\{ \delta \in \Delta_{i_1,j_1\to i_2,j_2} \;\middle|\; \begin{array}{l} \textit{for all } \sigma \in \Sigma_{i_1,j_1} \textit{ s.t. } \delta(\sigma) \textit{ is defined} \\ \check{tainted}(\sigma) \cup \hat{tainted}(\delta(\sigma)) \models \wedge_{i\in I}\phi_i \end{array} \right\} \\
&= \left\{ \delta \in \Delta_{i_1,j_1\to i_2,j_2} \;\middle|\; \begin{array}{l} \textit{for all } \sigma \in \Sigma_{i_1,j_1} \textit{ s.t. } \delta(\sigma) \textit{ is defined} \\ \check{tainted}(\sigma) \cup \hat{tainted}(\delta(\sigma)) \models \phi_i \textit{ for all } i \in I \end{array} \right\} \\
&= \bigcap_{i\in I} \left\{ \delta \in \Delta_{i_1,j_1\to i_2,j_2} \;\middle|\; \begin{array}{l} \textit{for all } \sigma \in \Sigma_{i_1,j_1} \textit{ s.t. } \delta(\sigma) \textit{ is def.} \\ \check{tainted}(\sigma) \cup \hat{tainted}(\delta(\sigma)) \models \phi_i \end{array} \right\} \\
&= \cap_{i\in I}\gamma(\phi_i).
\end{aligned}$$

Given the previous lemma, we can prove Prop. 8.13:

Proof. The domain $\mathbb{T}_{i_1,j_1\to i_2,j_2}$ is a complete lattice w.r.t. logical entailment with $\wedge$ as greatest lower bound operator. The domain $\wp(\Delta_{i_1,j_1\to i_2,j_2})$ is a complete lattice w.r.t. set inclusion with $\cap$ as greatest lower bound operator. The map γ is co-additive (Lemma 8.14), hence the thesis follows by a general result of abstract interpretation [39].

Example 8.15. Consider ϕ from Ex. 8.12 and bytecode `store 0` at a program point with $i = 4$ locals and $j = 1$ stack elements. Its denotation *store 0* $\in \gamma(\phi)$ since that bytecode does not modify locals 1, 2 and 3, hence their taintedness is unchanged $((\check{l}_1 \leftrightarrow \hat{l}_1) \wedge (\check{l}_2 \leftrightarrow \hat{l}_2) \wedge (\check{l}_3 \leftrightarrow \hat{l}_3))$; it only runs if no exception is thrown just before it $(\neg\check{e})$; it does not throw any exception $(\neg\hat{e})$; and the output local 0 is an alias of the topmost and only element of the input stack $(\check{s}_0 \leftrightarrow \hat{l}_0)$.

Fig. 8.2 defines correct abstractions for the bytecodes from Sec. 8.2, but `call`. A formula U (for *unchanged*) is a frame condition for input local variables and stack elements, that are also in the output and with unchanged value: their taintedness is unchanged. For the stack, this is only required when no exception is thrown, since otherwise the only output stack element is the exception.

$$
\begin{aligned}
(\mathit{const}\ v)^{\mathbb{T}} &= U \wedge \neg\check{e} \wedge \neg\hat{e} \wedge \neg\hat{s}_j \\
(\mathit{load}\ k\ t)^{\mathbb{T}} &= U \wedge \neg\check{e} \wedge \neg\hat{e} \wedge (\check{l}_k \leftrightarrow \hat{s}_j) \\
(\mathit{store}\ k\ t)^{\mathbb{T}} &= U \wedge \neg\check{e} \wedge \neg\hat{e} \wedge (\check{s}_{j-1} \leftrightarrow \hat{l}_k) \\
(\mathit{add})^{\mathbb{T}} &= U \wedge \neg\check{e} \wedge \neg\hat{e} \wedge (\hat{s}_{j-2} \leftrightarrow (\check{s}_{j-2} \vee \check{s}_{j-1})) \\
(\mathit{throw}\ \kappa)^{\mathbb{T}} &= U \wedge \neg\check{e} \wedge \hat{e} \wedge (\hat{s}_0 \rightarrow \check{s}_{j-1}) \\
(\mathit{new}\ \kappa)^{\mathbb{T}} &= U \wedge \neg\check{e} \wedge (\neg\hat{e} \rightarrow \neg\hat{s}_j) \wedge (\hat{e} \rightarrow \neg\hat{s}_0) \\
(\mathit{catch})^{\mathbb{T}} &= U \wedge \check{e} \wedge \neg\hat{e} \\
(\mathit{getfield}\ \kappa.f\!:\!t)^{\mathbb{T}} &= U \wedge \neg\check{e} \wedge (\neg\hat{e} \rightarrow (\hat{s}_{j-1} \rightarrow \check{s}_{j-1})) \wedge (\hat{e} \rightarrow \neg\hat{s}_0) \\
(\mathit{putfield}\ \kappa.f\!:\!t)^{\mathbb{T}} &= \wedge_{v\in L} R_j(v) \wedge (\neg\hat{e} \rightarrow \wedge_{v\in S} R_j(v)) \wedge (\hat{e} \rightarrow \neg\hat{s}_0) \wedge \neg\check{e}.
\end{aligned}
$$

Fig. 8.2: Bytecode abstraction for taintedness, in a program point with j stack elements. Bytecodes not reported in this figure are abstracted into the default $U \wedge \neg\check{e} \wedge \neg\hat{e}$.

Definition 8.16. *Let sets S (of stack elements) and L (of local variables) be the input variables that after all executions of a given bytecode in a given program point (only after the normal executions for S) survive with unchanged value. Then $U = \wedge_{v\in L}(\check{v} \leftrightarrow \hat{v}) \wedge (\neg\hat{e} \rightarrow \wedge_{v\in S}(\check{v} \leftrightarrow \hat{v}))$.*

Consider Fig. 8.2. Bytecodes run only if the preceding one does not throw any exception ($\neg\check{e}$) but `catch` requires an exception to be thrown ($\check{e}$). Bytecode `const` v pushes an untainted value on the stack: its abstraction says that no variable changes its taintedness (U), the new stack top is untainted ($\neg\hat{s}_j$) and `const` v never throws an exception ($\neg\hat{e}$). Most abstractions in Fig. 8.2 can be explained similarly. The result of `add` is tainted if and only if at least one operand is tainted ($\hat{s}_{j-2} \leftrightarrow (\check{s}_{j-2} \vee \check{s}_{j-1})$). For `new` κ, no variable changes its taintedness (U), if its execution does not throw any exception then the new top of the stack is an untainted new object ($\neg\hat{e} \rightarrow \neg\hat{s}_j$); otherwise the only stack element is an untainted exception ($\hat{e} \rightarrow \neg\hat{s}_0$). Bytecode `throw` κ always throws an exception ($\hat{e}$); if this is tainted, then the top of the initial stack was tainted as well ($\hat{s}_0 \rightarrow \check{s}_{j-1}$). The abstraction of `getfield` says that if it throws no exception and the value of the field is tainted, then the container of the field was tainted as well ($\neg\hat{e} \rightarrow (\hat{s}_{j-1} \rightarrow \check{s}_{j-1})$). This follows from the object-sensitivity of our notion of taintedness (Def. 8.8). Otherwise, the exception is untainted ($\hat{e} \rightarrow \neg\hat{s}_0$). For `putfield`, we cannot use U and must consider each variable v to see if it might reach the object whose field is modified ($\check{s}_{j-2}$). If that is not the case, v's taintedness is not affected ($\check{v} \leftrightarrow \hat{v}$); otherwise, if its value is tainted then either it was already tainted before the bytecode or the value written in the field was tainted ($(\check{v} \vee \check{s}_{j-1}) \leftarrow \hat{v}$). In this last case, we must use $\leftarrow$ instead of $\leftrightarrow$ since our reachability analysis is a *possible* approximation of actual (undecidable) reachability. This is expressed by formula $R_j(v)$, used in Fig. 8.2, where $R_j(v) = \check{v} \leftrightarrow \hat{v}$ if $\neg reach(v, s_{j-2})$, and $R_j(v) = (\check{v} \vee \check{s}_{j-1}) \leftarrow \hat{v}$, if $reach(v, s_{j-2})$.

Example 8.17. According to Fig. 8.2, the abstraction of `store 0` at a program point with $i = 4$ local variables and $j = 1$ stack elements is the formula ϕ of Ex. 8.12.

Example 8.18. Consider a `putfield f` at a program point p where there are $i = 4$ local variables, $j = 3$ stack elements and the only variable that reaches the receiver s_1 is the underlying stack element s_0. A possible state at p in Ex. 8.3. According to Fig. 8.2, the abstraction of that bytecode at p is $\phi' = (\check{l}_0 \leftrightarrow \hat{l}_0) \wedge (\check{l}_1 \leftrightarrow \hat{l}_1) \wedge (\check{l}_2 \leftrightarrow \hat{l}_2) \wedge (\check{l}_3 \leftrightarrow \hat{l}_3) \wedge (\neg\hat{e} \rightarrow ((\check{s}_0 \vee \check{s}_2) \leftarrow \hat{s}_0)) \wedge (\hat{e} \rightarrow \neg\hat{s}_0) \wedge \neg\check{e} \in \mathbb{T}_{4,3\rightarrow 4,1}$.

Proposition 8.19. *The approximations in Fig. 8.2 are correct w.r.t. the denotations of Sec. 8.2,* i.e., *for all bytecode* `ins` *distinct from* `call` *we have* $ins \in \gamma(ins^{\mathbb{T}})$.

Lemma 8.20. *Let* `ins` *be a bytecode distinct from* `putfield` *and* `call` *and let U be the formula constructed for* `ins` *according to Def. 8.16. Then U is correct w.r.t. ins,* i.e., $ins \in \gamma(U)$.

Proof. Let S and L be as in Def. 8.16 and $\sigma \in \Sigma$ be such that $\sigma' = ins(\sigma)$ is defined.

Let $v \in L$. Since v survives to all executions of `ins`, *it is a local variable of both σ and σ' where it has the same value. For the hypothesis on* `ins`, *no object reachable from that value is modified by ins. Hence either $\{\check{v}, \hat{v}\} \subseteq \widecheck{tainted}(\sigma) \cup \widehat{tainted}(\sigma')$ or $\{\check{v}, \hat{v}\} \cap (\widecheck{tainted}(\sigma) \cup \widehat{tainted}(\sigma')) = \emptyset$. In both cases we conclude that $\widecheck{tainted}(\sigma) \cup \widehat{tainted}(\sigma') \models \check{v} \leftrightarrow \hat{v}$,* i.e., $ins \in \gamma(\check{v} \leftrightarrow \hat{v})$.

Let now $v \in S$. If $\sigma' \in \Xi$, since v survives to all normal executions of `ins`, *it is a stack element of both σ and σ' where it has the same value. For the hypothesis on* `ins`, *no object reachable from that value is modified by ins. Hence $\hat{e} \notin \widehat{tainted}(\sigma')$ and either $\{\check{v}, \hat{v}\} \subseteq \widecheck{tainted}(\sigma) \cup \widehat{tainted}(\sigma')$ or $\{\check{v}, \hat{v}\} \cap (\widecheck{tainted}(\sigma) \cup \widehat{tainted}(\sigma')) = \emptyset$, so that $\widecheck{tainted}(\sigma) \cup \widehat{tainted}(\sigma') \models \neg\hat{e} \rightarrow (\check{v} \leftrightarrow \hat{v})$. If $\sigma' \in \underline{\Xi}$ we have $\hat{e} \in \widehat{tainted}(\sigma')$ and also in this case $\widecheck{tainted}(\sigma) \cup \widehat{tainted}(\sigma') \models \neg\hat{e} \rightarrow (\check{v} \leftrightarrow \hat{v})$. We conclude that $ins \in \gamma(\neg\hat{e} \rightarrow (\check{v} \leftrightarrow \hat{v}))$.*

The result follows by Lemma 8.14.

We can now prove Prop. 8.19.

Proof. We consider each bytecode instruction. We start from `putfield`, which modifies the memory and requires a specific proof.

`putfield` $\kappa.f\!:\!t$

Let σ be such that $\sigma' = (\mathit{putfield}\ \kappa.f\!:\!t)(\sigma)$ is defined. Let ϕ be the formula for this instruction in Fig. 8.2. The execution of this instruction can only modify field $\kappa.f\!:\!t$ of the object bound to s_{j-2} (the receiver) in σ. For a given variable v in σ, that still exists in σ', if $\neg \mathit{reach}(v, s_{j-2})$ before the instruction then the memory reachable from v in σ is not affected by the execution of this bytecode and the taintedness of v is not affected either (Def. 8.8). It follows that, in this case, $\check{\mathit{tainted}}(\sigma) \cup \widehat{\mathit{tainted}}(\sigma') \models \check{v} \leftrightarrow \hat{v}$. If, instead, $\mathit{reach}(v, s_{j-2})$, then from v one reaches tainted data in σ' if that was already possible in σ or if that data was made reachable by this instruction through the updated field, that has been updated to s_{j-1}, from which it must have been possible to reach tainted data then. Hence, in this case we have $\check{\mathit{tainted}}(\sigma) \cup \widehat{\mathit{tainted}}(\sigma') \models (\check{v} \vee \check{s}_{j-1}) \leftarrow \hat{v}$. That is, in both cases, we have $\check{\mathit{tainted}}(\sigma) \cup \widehat{\mathit{tainted}}(\sigma') \models R_j(v)$. The variables in σ that still exist in σ' are those in L and, if $\sigma' \in \Xi$, also those in S. It follows that $\check{\mathit{tainted}}(\sigma) \cup \widehat{\mathit{tainted}}(\sigma') \models \wedge_{v \in L} R_j(v) \wedge (\neg \hat{e} \to \wedge_{v \in S} R_j(v))$. If $\sigma' \in \underline{\Xi}$ then the top of the stack of σ' is a reference to an untainted exception object and we have $\hat{s}_0 \notin \widehat{\mathit{tainted}}(\sigma')$. Moreover, this instruction is only executed from a normal state, hence $\sigma \in \Xi$. We conclude that $\check{\mathit{tainted}}(\sigma) \cup \widehat{\mathit{tainted}}(\sigma') \models (\hat{e} \to \neg \hat{s}_0) \wedge \neg \check{e}$. We conclude that $\check{\mathit{tainted}}(\sigma) \cup \widehat{\mathit{tainted}}(\sigma') \models \wedge_{v \in L} R_j(v) \wedge (\neg \hat{e} \to \wedge_{v \in S} R_j(v)) \wedge (\hat{e} \to \neg \hat{s}_0) \wedge \neg \check{e}$, and hence $\mathit{putfield}\ \kappa.f\!:\!t \in \gamma(\phi)$.

For the other bytecodes, by Lemma 8.20 we know that $\mathit{ins} \in \gamma(U)$. Let σ be such that $\sigma' = \mathit{ins}(\sigma)$ is defined. If `ins` is not `catch` then $\sigma \in \Xi$ (Sec. 8.2). Hence $\check{e} \notin \check{\mathit{tainted}}(\sigma)$ and $\mathit{ins} \in \gamma(\neg \check{e})$. If instead `ins` is `catch`, we must have $\sigma \in \underline{\Xi}$ and hence $\check{e} \in \check{\mathit{tainted}}(\sigma)$. Then $\mathit{ins} \in \gamma(\check{e})$. By Lemma 8.14, it remains to prove that $\mathit{ins} \in \gamma(\phi)$, where ϕ is the portion of the formulas in Figure 8.2 that follows the prefix $U \wedge \neg \check{e}$ ($U \wedge \check{e}$ for `catch`). We prove it for each kind of bytecode instruction.

`const` v

We have $\phi = \neg \hat{e} \wedge \neg \hat{s}_j$. We have $\sigma' \in \Xi$ so $\hat{e} \notin \widehat{\mathit{tainted}}(\sigma')$. Moreover, the top s_j of the stack of σ' holds v. If $v \in \mathbb{Z}$ or $v = \texttt{null}$ then $\hat{s}_j \notin \widehat{\mathit{tainted}}(\sigma')$. v is a constant value, so it cannot be tainted. We conclude that $\check{\mathit{tainted}}(\sigma) \cup \widehat{\mathit{tainted}}(\sigma') \models \phi$ and hence $\mathit{const}\ v \in \gamma(\phi)$.

`load` $k\ t$

We have $\phi = \neg \hat{e} \wedge (\check{l}_k \leftrightarrow \hat{s}_j)$. Since $\sigma' \in \Xi$ we have $\hat{e} \notin \widehat{\mathit{tainted}}(\sigma')$. Moreover, the ith local variable of σ is a copy of the top of the stack of σ'. Hence they are both tainted, in which case $\{\check{l}_k, \hat{s}_j\} \subseteq \check{\mathit{tainted}}(\sigma) \cup \widehat{\mathit{tainted}}(\sigma')$, or they are both untainted, in which case $\{\check{l}_k, \hat{s}_j\} \cap (\check{\mathit{tainted}}(\sigma) \cup \widehat{\mathit{tainted}}(\sigma')) = \emptyset$. In both cases we have $\check{\mathit{tainted}}(\sigma) \cup \widehat{\mathit{tainted}}(\sigma') \models \phi$ and hence $\mathit{load}\ k\ t \in \gamma(\phi)$.

`store` $k\ t$

We have $\phi = \neg\hat{e} \wedge (\check{s}_{j-1} \leftrightarrow \hat{l}_k)$. Since $\sigma' \in \Xi$ we have $\hat{e} \notin \widehat{\mathit{tainted}}(\sigma')$. Moreover, the top of the stack of σ is a copy of the kth local variable of σ'. Hence they are both tainted, in which case $\{\check{s}_{j-1}, \hat{l}_k\} \subseteq \check{\mathit{tainted}}(\sigma) \cup \widehat{\mathit{tainted}}(\sigma')$, or they are both untainted, in which case $\{\check{s}_{j-1}, \hat{l}_k\} \cap (\check{\mathit{tainted}}(\sigma) \cup \widehat{\mathit{tainted}}(\sigma')) = \emptyset$. In both cases we have $\check{\mathit{tainted}}(\sigma) \cup \widehat{\mathit{tainted}}(\sigma') \models \phi$ and hence $\mathit{store}\ k\ t \in \gamma(\phi)$.

`add`

We have $\phi = \neg\hat{e} \wedge (\hat{s}_{j-2} \leftrightarrow (\check{s}_{j-2} \vee \check{s}_{j-1}))$. Since $\sigma' \in \Xi$ we have $\hat{e} \notin \widehat{\mathit{tainted}}(\sigma')$. Moreover, the top of the stack in σ' (the result) is tainted only if at least one of the operands is tainted in σ. So $\{\hat{s}_{j-2}, \check{s}_{j-2}\} \subseteq \check{\mathit{tainted}}(\sigma) \cup \widehat{\mathit{tainted}}(\sigma')$ or $\{\hat{s}_{j-2}, \check{s}_{j-1}\} \subseteq \check{\mathit{tainted}}(\sigma) \cup \widehat{\mathit{tainted}}(\sigma')$ or $\{\hat{s}_{j-2}, \check{s}_{j-2}, \check{s}_{j-1}\} \cap (\check{\mathit{tainted}}(\sigma) \cup \widehat{\mathit{tainted}}(\sigma')) = \emptyset$. In all cases we have $\check{\mathit{tainted}}(\sigma) \cup \widehat{\mathit{tainted}}(\sigma') \models \phi$ and hence $\mathit{add} \in \gamma(\phi)$.

`ifne` t

We have $\phi = \neg\hat{e}$. Since $\sigma' \in \Xi$ we have $\hat{e} \notin \widehat{\mathit{tainted}}(\sigma')$. We conclude that $\check{\mathit{tainted}}(\sigma) \cup \widehat{\mathit{tainted}}(\sigma') \models \phi$ and hence $\mathit{ifne}\ t \in \gamma(\phi)$.

`new` κ

We have $\phi = (\neg\hat{e} \to \neg\hat{s}_j) \wedge (\hat{e} \to \neg\hat{s}_0)$. If $\sigma' \in \underline{\Xi}$ then the top of the stack is a reference to an untainted exception object, and we have $\hat{s_0} \notin \widehat{\mathit{tainted}}(\sigma')$. If $\sigma' \in \Xi$ then the top of the stack of σ' is a reference to a new object of class t, hence untainted. We conclude that $\check{\mathit{tainted}}(\sigma) \cup \widehat{\mathit{tainted}}(\sigma') \models \phi$ and hence $\mathit{new}\ \kappa \in \gamma(\phi)$.

`throw` κ

We have $\phi = \hat{e} \wedge (\hat{s}_0 \to \check{s}_{j-1})$. We have $\sigma' \in \underline{\Xi}$ and hence $\hat{e} \in \widehat{\mathit{tainted}}(\sigma')$. Moreover if $\hat{s}_0 \in \widehat{\mathit{tainted}}(\sigma')$ then $\check{s}_{j-1} \in \check{\mathit{tainted}}(\sigma)$. Hence we have $\check{\mathit{tainted}}(\sigma) \cup \widehat{\mathit{tainted}}(\sigma') \models \phi$. In conclusion, $\mathit{throw}\ \kappa \in \gamma(\phi)$.

`catch`

We have $\phi = \neg\hat{e}$. We have $\sigma' \in \Xi$ and hence $\hat{e} \notin \widehat{\mathit{tainted}}(\sigma')$. Then we have $\check{\mathit{tainted}}(\sigma) \cup \widehat{\mathit{tainted}}(\sigma') \models \phi$. In conclusion, $\mathit{catch} \in \gamma(\phi)$.

`exception_is` K

We have $\phi = \neg\hat{e}$. We have $\sigma' \in \Xi$ and hence $\hat{e} \notin \widehat{\mathit{tainted}}(\sigma')$. Then we have $\check{\mathit{tainted}}(\sigma) \cup \widehat{\mathit{tainted}}(\sigma') \models \phi$. In conclusion, $\mathit{exception\_is}\ K \in \gamma(\phi)$.

`receiver_is` K

We have $\phi = \neg\hat{e}$. We have $\sigma' \in \Xi$ and hence $\hat{e} \notin \widehat{\mathit{tainted}}(\sigma')$. Then we have $\check{\mathit{tainted}}(\sigma) \cup \widehat{\mathit{tainted}}(\sigma') \models \phi$. In conclusion, $\mathit{receiver\_is}\ K \in \gamma(\phi)$.

`return` t

We have $\phi = \neg\hat{e}$. Since $\sigma' \in \Xi$, we have $\hat{e} \notin \widehat{\mathit{tainted}}(\sigma')$. Hence we have $\widecheck{\mathit{tainted}}(\sigma) \cup \widehat{\mathit{tainted}}(\sigma') \models \phi$. In conclusion, *return* $t \in \gamma(\phi)$.

`return`

We have $\phi = \neg\hat{e}$. Since $\sigma' \in \Xi$, we have $\hat{e} \notin \widehat{\mathit{tainted}}(\sigma')$. Hence we have $\widecheck{\mathit{tainted}}(\sigma) \cup \widehat{\mathit{tainted}}(\sigma') \models \phi$. In conclusion, *return* $\in \gamma(\phi)$.

`getfield` $\kappa.f : t$

We have $\phi = (\neg\hat{e} \to (\hat{s}_{j-1} \to \check{s}_{j-1})) \wedge (\hat{e} \to \neg\hat{s}_0)$. If $\sigma' \in \underline{\Xi}$ then the top of the stack is a reference to an untainted exception object, and we have $\hat{s}_0 \notin \widehat{\mathit{tainted}}(\sigma')$. If $\sigma' \in \Xi$ then, by the definition of taintedness (Def. 8.8), if $\hat{s}_{j-1} \in \widehat{\mathit{tainted}}(\sigma')$ then $\check{s}_{j-1} \in \widecheck{\mathit{tainted}}(\sigma)$. We conclude that $\widecheck{\mathit{tainted}}(\sigma) \cup \widehat{\mathit{tainted}}(\sigma') \models \phi$ and hence *getfield* $\kappa.f : t \in \gamma(\phi)$.

Denotations are composed by ; and their abstractions by $;^{\mathbb{T}}$. The definition of $\phi_1;^{\mathbb{T}}\phi_2$ matches the output variables of ϕ_1 with the corresponding input variables of ϕ_2. To avoid name clashes, they are renamed apart and then projected away.

Definition 8.21. *Let* $\phi_1, \phi_2 \in \mathbb{T}$. *Their* abstract sequential composition $\phi_1;^{\mathbb{T}}\phi_2$ *is* $\exists_{\overline{V}}(\phi_1[\overline{V}/\hat{V}] \wedge \phi_2[\overline{V}/\check{V}])$, *where* $\overline{V}$ *are fresh overlined variables.*

Example 8.22. Consider the execution of `putfield f` at program point p and then `store 0`, as in Ex. 8.18. The former is abstracted by ϕ' from Ex. 8.18; the latter by ϕ from Ex. 8.17. Their sequential composition is $\phi';^{\mathbb{T}}\phi = \exists_{\overline{V}}(\phi'[\overline{V}/\hat{V}] \wedge \phi[\overline{V}/\check{V}]) = \exists_{\overline{V}}([(\check{l}_0 \leftrightarrow \bar{l}_0) \wedge (\check{l}_1 \leftrightarrow \bar{l}_1) \wedge (\check{l}_2 \leftrightarrow \bar{l}_2) \wedge (\check{l}_3 \leftrightarrow \bar{l}_3) \wedge (\neg\bar{e} \to ((\check{s}_0 \vee \check{s}_2) \leftarrow \bar{s}_0)) \wedge (\bar{e} \to \neg\bar{s}_0) \wedge \neg\check{e}] \wedge [(\bar{l}_1 \leftrightarrow \hat{l}_1) \wedge (\bar{l}_2 \leftrightarrow \hat{l}_2) \wedge (\bar{l}_3 \leftrightarrow \hat{l}_3) \wedge \neg\bar{e} \wedge \neg\hat{e} \wedge (\bar{s}_0 \leftrightarrow \hat{l}_0)])$ which simplifies into $(\check{l}_1 \leftrightarrow \hat{l}_1) \wedge (\check{l}_2 \leftrightarrow \hat{l}_2) \wedge (\check{l}_3 \leftrightarrow \hat{l}_3) \wedge ((\check{s}_0 \vee \check{s}_2) \leftarrow \hat{l}_0) \wedge \neg\check{e} \wedge \neg\hat{e}$.

The second semantical operator is $\cup$ of two sets, approximated as $\cup^{\mathbb{T}} = \vee$. The third is *extend*, that makes the analysis context-sensitive by plugging the behavior of a method at each distinct calling context. Let ϕ approximate the taintedness behavior of method $M = \kappa.m(t_1, \ldots, t_n) : t$; ϕ's variables are among $\check{l}_0, \ldots, \check{l}_n$ (the actual arguments including `this`), $\hat{s}_0$ (if M does not return `void`), $\hat{l}_0, \hat{l}_1 \ldots$ (the final values of M's local variables), $\check{e}$ and $\hat{e}$. Consider a call M at a program point where the $n+1$ actual arguments are stacked over other b stack elements. The operator plugs ϕ at the calling context: the return value $\hat{s}_0$ (if any) is renamed into $\hat{s}_b$; each formal argument $\check{l}_k$ of the callee is renamed into the actual argument $\check{s}_{k+b}$ of the caller; local variable $\hat{l}_k$ at the end of the callee is temporarily renamed into $\bar{l}_k$. Then a frame condition is built: the set $SA_{b,M,v}$ contains the formal arguments of the caller that might share with variable v of the callee at call-time and might be updated during the call. If this set is empty, then nothing reachable from v is modified during the call and v keeps its taintedness unchanged. This is expressed by the first case of formula $A_{b,M}(v)$. Otherwise, if v is tainted at the end of the call then either it was already tainted at the beginning or at least one of the variables in $SA_{b,M,v}$ has become tainted during the call. The second case of formula $A_{b,M}(v)$ uses the temporary variables to express that condition, to avoid name clashes with the output local variables of the caller. The frame condition for the b lowest stack elements of the caller is valid only if no exception is thrown, since otherwise the stack contains the exception object only. At the end, all temporary variables $\{\bar{l}_0, \ldots, \bar{l}_{i'}\}$ are projected away.

Definition 8.23. *Let $i, j \in \mathbb{N}$ and $M = \kappa.m(t_1, \ldots, t_n) : t$ with $j = b + n + 1$ and $b \geq 0$. We define $(\mathit{extend}_M^{i,j})^{\mathbb{T}} : \mathbb{T}_{n+1,0\to i',r} \to \mathbb{T}_{i,j\to i,b+r}$ with $r = 0$ if $t =$ `void` and $r = 1$ otherwise, as*
$(\mathit{extend}_M^{i,j})^{\mathbb{T}}(\phi) = \neg\check{e} \wedge \exists_{\{\bar{l}_0,\ldots,\bar{l}_{i'}\}}\big(\phi[\hat{s}_b/\hat{s}_0][\bar{l}_k/\hat{l}_k \mid 0 \leq k < i'][\check{s}_{k+b}/\check{l}_k \mid 0 \leq k \leq n] \wedge \bigwedge_{0\leq k<i} A_{b,M}(l_k) \wedge \big(\neg\hat{e} \to \bigwedge_{0\leq k<b} A_{b,M}(s_k)\big)\big)$, with

$$SA_{b,M,v} = \{l_k \mid 0 \leq k \leq n,\ \mathit{share}(v, s_{b+k}) \text{ and } \mathit{updated}_M(l_k)\},$$

$$A_{b,M}(v) = \begin{cases} \check{v} \leftrightarrow \hat{v} & \text{if } SA_{b,M,v} = \emptyset \\ ((\check{v} \vee (\bigvee_{w\in SA_{b,M,v}} \overline{w})) \leftarrow \hat{v}) & \text{otherwise} \end{cases}$$

Proposition 8.24. *The operators $;^{\mathbb{T}}$, $\mathit{extend}^{\mathbb{T}}$ and $\cup^{\mathbb{T}}$ are correct.*

Proof. Let $\phi_1, \phi_2 \in \mathbb{T}$, $d_1 \subseteq \gamma(\phi_1)$ and $d_2 \subseteq \gamma(\phi_2)$. We must prove that $d_1; d_2 \in \gamma(\phi_1;^{\mathbb{T}} \phi_2)$. Let $\delta_1 \in d_1$ and $\delta_2 \in d_2$. It is enough to prove that $\delta_1; \delta_2 \in \gamma(\phi_1;^{\mathbb{T}} \phi_2)$. Let hence σ be such that $(\delta_1; \delta_2)(\sigma)$ is defined, *i.e.*, both $\sigma' = \delta_1(\sigma)$ and $\sigma'' = \delta_2(\sigma')$ are defined (Def. 8.4). From $\delta_1 \in \gamma(\phi_1)$ we conclude that $\check{\mathit{tainted}}(\sigma) \cup \hat{\mathit{tainted}}(\sigma') \models \phi_1$. From $\delta_2 \in \gamma(\phi_2)$ we conclude that $\check{\mathit{tainted}}(\sigma') \cup \hat{\mathit{tainted}}(\sigma'') \models \phi_2$. Hence

$$\check{\mathit{tainted}}(\sigma) \cup \{\overline{v} \mid \hat{v} \in \hat{\mathit{tainted}}(\sigma')\} \models \phi_1[\overline{V}/\hat{V}]$$
$$\{\overline{v} \mid \check{v} \in \check{\mathit{tainted}}(\sigma')\} \cup \hat{\mathit{tainted}}(\sigma'') \models \phi_2[\overline{V}/\check{V}]$$

so that $\check{\mathit{tainted}}(\sigma) \cup \{\overline{v} \mid v \in \mathit{tainted}(\sigma')\} \cup \hat{\mathit{tainted}}(\sigma'') \models \phi_1[\overline{V}/\hat{V}] \wedge \phi_2[\overline{V}/\check{V}]$. We conclude that $\check{\mathit{tainted}}(\sigma) \cup \hat{\mathit{tainted}}(\sigma'') \models \exists_{\overline{V}}(\phi_1[\overline{V}/\hat{V}] \wedge \phi_2[\overline{V}/\check{V}]) = \phi_1;^{\mathbb{T}} \phi_2$. Hence $\delta_1; \delta_2 \in \gamma(\phi_1;^{\mathbb{T}} \phi_2)$.

Let $\phi \in \mathbb{T}_{n+1,0\to i',r}$ as in Def. 8.23. Let $d \subseteq \gamma(\phi)$. We must prove that for all $i, j \in \mathbb{N}$ with $j = b+n+1$ and $b \geq 0$ we have $\mathit{extend}_M^{i,j}(d) \subseteq \gamma((\mathit{extend}_M^{i,j})^{\mathbb{T}}(\phi))$, where $\mathit{extend}_M^{i,j}$ has been defined in Sec. 8.2. Let hence $\delta \in d$. It is enough to prove that $\mathit{extend}_M^{i,j}(\delta) \in \gamma((\mathit{extend}_M^{i,j})^{\mathbb{T}}(\phi))$. Let $\sigma = \langle l \parallel v_n :: \cdots :: v_0 :: s \parallel \mu\rangle$ be such that $\sigma' = \mathit{extend}_M^{i,j}(\delta)(\sigma)$ is defined. This corresponds to an execution of M from $\sigma'' = \langle [v_0, \ldots, v_n] \parallel \epsilon \parallel \mu\rangle$ to some $\sigma''' = \langle l' \parallel \mathit{top} \parallel \mu'\rangle$. By the definition of $\mathit{extend}_M^{i,j}$ we know that σ and σ' have the same set of local variables with unchanged values; and that when $\sigma' \in \Xi$, the b lowest stack elements are in both σ and σ' and with unchanged value. Let hence v be one of such unchanged variables. The taintedness of v might well change during the execution of M, but only if M can access a location reachable from v and updates one of the fields of the object at that location, *i.e.*, only if v shares with a parameter of M that gets updated. Hence, if $SA_{b,M,v} = \emptyset$ then the taintedness of v cannot be changed by the call to M and $\check{\mathit{tainted}}(\sigma) \cup \hat{\mathit{tainted}}(\sigma') \models \check{v} \leftrightarrow \hat{v}$. If, instead, $SA_{b,M,v} \neq \emptyset$ then v might be tainted at the end of the call if it was already tainted before or if a formal parameter of M in $SA_{b,M,v}$, sharing with v at call-time and updated during the call, is tainted at the end of the call, that is, in σ''' (we have assumed that method parameters cannot be reassigned inside its body, see Sec. 8.2). Hence $\check{\mathit{tainted}}(\sigma) \cup \hat{\mathit{tainted}}(\sigma') \cup \{\overline{x} \mid x \in \mathit{tainted}(\sigma''')\} \models (\check{v} \vee (\bigvee_{w \in SA_{b,M,v}} \overline{w})) \leftarrow \hat{v}$. Since the stack elements remain unchanged only if the call does not throw any exception, by Lemma 8.14 we have $\check{\mathit{tainted}}(\sigma) \cup \hat{\mathit{tainted}}(\sigma') \cup \{\overline{x} \mid x \in \mathit{tainted}(\sigma''')\} \models \bigwedge_{0 \le k < i} A_{b,M}(l_k) \wedge (\neg\hat{e} \to \bigwedge_{0 \le k < b} A_{b,M}(s_k))$. By the definition of σ, σ', σ'' and σ''' and from $\delta \in \gamma(\phi)$ we have $\check{\mathit{tainted}}(\sigma) \cup \hat{\mathit{tainted}}(\sigma') \cup \{\overline{x} \mid x \in \mathit{tainted}(\sigma''')\} \models \phi[\hat{s}_b/\hat{s}_0][\bar{l}_k/\hat{l}_k \mid 0 \le k < i'][\check{s}_{k+b}/\check{l}_k \mid 0 \le k \le n]$. By Lemma 8.14, we conclude that $\check{\mathit{tainted}}(\sigma) \cup \hat{\mathit{tainted}}(\sigma') \models \exists_{\{\bar{l}_0,\ldots,\bar{l}_{i'}\}}\big(\phi[\hat{s}_b/\hat{s}_0][\bar{l}_k/\hat{l}_k \mid 0 \le k < i'][\check{s}_{k+b}/\check{l}_k \mid 0 \le k \le n] \wedge \bigwedge_{0 \le k < i} A_{b,M}(l_k) \wedge (\neg\hat{e} \to \bigwedge_{0 \le k < b} A_{b,M}(s_k))\big)$. By the definition of $\mathit{extend}^{i,j}_M$, we know that $\sigma \in \Xi$, so that $\check{e} \notin \check{\mathit{tainted}}(\sigma)$ and $\check{\mathit{tainted}}(\sigma) \cup \hat{\mathit{tainted}}(\sigma') \models \neg\check{e}$. By Lemma 8.14 and Def. 8.23 we conclude that $\check{\mathit{tainted}}(\sigma) \cup \hat{\mathit{tainted}}(\sigma') \models (\mathit{extend}^{i,j}_M)^{\mathbb{T}}(\phi)$. Since σ and σ' are arbitrary, we conclude that $\mathit{extend}^{i,j}_M(\delta) \in \gamma((\mathit{extend}^{i,j}_M)^{\mathbb{T}}(\phi))$.

Let $\phi_1, \phi_2 \in \mathbb{T}$, $d_1 \subseteq \gamma(\phi_1)$ and $d_2 \subseteq \gamma(\phi_2)$. We must prove that $d_1 \cup d_2 \subseteq \gamma(\phi_1 \cup^{\mathbb{T}} \phi_2) = \phi_1 \vee \phi_2$. Let hence $\delta \in d_1 \cup d_2$. It is enough to prove that $\delta \in \gamma(\phi_1 \vee \phi_2)$. If $\delta \in d_1$ then $\delta \in \gamma(\phi_1) \subseteq \gamma(\phi_1 \vee \phi_2)$. If $\delta \in d_2$ then $\delta \in \gamma(\phi_2) \subseteq \gamma(\phi_1 \vee \phi_2)$.

Since the number of Boolean formulas over a given finite set of variables is finite (modulo equivalence), the abstract fixpoint is reached in a finite number of iterations. Hence this abstract semantics is a static analysis tool if one specifies the sources of tainted information and the sinks where it should not flow.

Sources. Some formal parameters or return values must be considered as sources of tainted data, that can be freely provided by the external world. Our implementation uses a database of library methods for that, such as the `request` argument of `doGet` and `doPost` methods of servlets and the return value of console and database methods. Moreover, it lets users specify their own sources through annotations. The abstract denotation in Fig. 8.2 is modified at *receiver_is* (a special bytecode at the beginning of each method) and *return* to force to true those formal arguments and return values that are injected tainted data, respectively.

Sinks. Our implementation has a database of library methods that need untainted parameters (users can add their own through annotations). Hence it knows which `call`s in P need an untainted parameter v (such as `executeQuery` in Fig. 8.1). But a denotational semantics is an input/output description of the behavior of P's methods and does not say what is passed *at* a `call`. For that, a *magic-sets transformation* [88] of P adds new blocks of code whose denotation gives information at internal program points, as traditional in denotational static analysis. It computes a formula ψ that holds at the `call`. If ψ entails $\neg\hat{v}$ then the `call` receives untainted data for v. Otherwise, the analysis issues a warning.

8.3.1 Making the Analysis Field-Sensitive

The approximation of getfield f in Fig. 8.2 specifies that if the value of field f (pushed on the stack) is tainted then the container of f must be tainted as well ($\hat{s}_{j-1} \rightarrow \check{s}_{j-1}$). Read the other way round, if the container is untainted then f's value is untainted, otherwise it is conservatively assumed as tainted. This choice is sound and object-sensitive, but field-insensitive: when $\check{s}_{j-1}$ is tainted, both its fields f and g are conservatively assumed as tainted. But if the program never assigns tainted data to f, then f's value can only be untainted, regardless of the taintedness of $\check{s}_{j-1}$. If the analyzer could spot such situations, the resulting analysis would be field-sensitive and hence more precise (fewer false positives).

We apply here a technique pioneered in [102]: it uses a set of fields O (the *oracle*) that might contain tainted data. For getfield f, it uses a better approximation than in Fig. 8.2: it assumes that f's value is tainted if its container is tainted *and* $f \in O$. The problem is now the computation of O. As in [102], this is done iteratively. The analyzer starts with $O = \emptyset$ and runs the analysis in Sec. 8.3, but with the new abstraction for getfield f seen in this paragraph. Then it adds to O those fields g such that there is at least one putfield g that stores tainted data. The analysis is repeated with this larger O. At its end, O is further enlarged with other fields g such that there is at least one putfield g that stores tainted data. The process is iterated until no more fields are added to O. As proved in [102], this process converges to a sound overapproximation of O and the last analysis of the iteration is sound. In practice, repeated analyses with larger and larger O are made efficient by caching abstract computations. On average, this process converges in around 5 iterations, also for large programs. By using caching, this only doubles the time of the analysis. Since preliminary analyses are more expensive than information flow analysis, this technique increases the total time by around 25% on average. (Sec. 8.4 shows effects on cost and precision.) This technique is not identical to statically, manually classifying fields as tainted and untainted, as [26, 56] do. The classification of the fields is here dynamic, depending on the program under analysis, and completely automatic. Moreover, a field might be in O (and hence be potentially tainted) but the analyzer might still consider its value untainted, because its container is untainted.

8.4 Experiments

We have implemented our analysis inside Julia (`http://www.juliasoft.com/julia`). Julia represents Boolean formulas via BDDs (binary decision dia-

Test	Tool	TP	FP	FN	Analysis Time
CWE89	CodePro Analytix	1332	0	888	20 minutes
	FindBugs	1776	2400	444	2 minutes
	Fortify SCA	700	0	1520	2.5 days
	Julia fs/fi	2220/2220	0/0	0/0	79/65 minutes
WebGoat	CodePro Analytix	26	7	1	1 minute
	FindBugs	22	12	5	20 seconds
	Fortify SCA	23	0	4	164 minutes
	Julia fs/fi	27/27	14/15	0/0	3/2 minutes

Fig. 8.3: Experiments with the identification of SQL injections.

grams). We have compared Julia with other tools that identify injections (Sec. 3.2). For Julia we have compared a field-sensitive analysis with an oracle (Sec. 8.3.1, *Julia fs*) with a field-insensitive analysis without oracle (*Julia fi*).

Our experiments analyze third-party tests developed to assess the power of a static analyzer to identify injection attacks: WebGoat 6.0.1 (`https://www.owasp.org/index.php/Category:OWASP_WebGoat_Project`) and 4 tests from the Samate suite (`http://samate.nist.gov/SARD/testsuite.php`). The following table reports their number of non-blank, non-comment lines of application source code (LoC), without supporting libraries.

Test	LoC
WebGoat	25070
CWE80	68967
CWE81	34317
CWE83	34317
CWE89	748962

Section 8.5 describes a more general and standardized benchmark.

Fig. 8.3 reports the evaluation for SQL injections using CWE89 and WebGoat. It shows that only Julia is sound (no false negatives: if there is an injection, Julia finds it). Julia issued no false positives to CWE89: possibly these tests just propagate information, without side-effects that degrade the precision of Julia (Def. 8.23; we do not know if and how other tools deal with side-effects). Julia issued 14 false alarms for WebGoat, often where actual information flows from source to sink exist, but constrained in such a way to be unusable to build an SQL-injection attack. Only here the field-insensitive version of Julia is slightly less precise (one false positive more). In general, its cost is around 25% higher than the field-sensitive version. The conclusion is that field sensitivity is not relevant when object sensitivity is used to distinguish different objects. Analysis time indicates the efficiency, roughly: CodePro Analytix and FindBugs work on the client machine in Eclipse, Fortify SCA on its cloud like Julia, that is controlled from an Eclipse client. Times include all supporting analyses.

We evaluated the same tools for the identification of cross-site scripting injections in CWE80/81/83, and WebGoat. As shown in Fig. 8.4, Julia is perfectly precise. It missed 11 cross-site scripting attacks in JSP (not in the main Java code of the application), found only by Fortify SCA. If we translate JSP's into Java through Jasper (as a servlet container would do, automatically) and include its bytecode in the analysis, Julia finds the missing 11 attacks. Nevertheless, this process is currently manual and we think fairer to count 11 false negatives.

Test	Tool	TP	FP	FN	Analysis Time
CWE80	CodePro Analytix	180	0	486	9 minutes
	FindBugs	19	0	647	18 seconds
	Fortify SCA	282	0	384	590 minutes
	Julia fs/fi	666/666	0/0	0/0	5/4 minutes
CWE81	CodePro Analytix	0	0	333	10 seconds
	FindBugs	19	0	314	4 seconds
	Fortify SCA	141	0	192	303 minutes
	Julia fs/fi	333/333	0/0	0/0	3/2 minutes
CWE83	CodePro Analytix	90	0	243	5 minutes
	FindBugs	19	0	314	4 seconds
	Fortify SCA	141	0	192	296 minutes
	Julia fs/fi	333/333	0/0	0/0	3/2 minutes
WebGoat	CodePro Analytix	5	0	11	1 minute
	FindBugs	0	0	16	20 seconds
	Fortify SCA	15	21	1	164 minutes
	Julia fs/fi	5/5	0/0	11/11	3/2 minutes

Fig. 8.4: Experiments with the identification of XSS injections.

We have run Julia on real code from our customers. Julia found 6 real SQL-injections in the Internet banking services (575995 LoC) of a large Italian bank, and found 5 more in its customer relation management system (346170 LoC). The analysis never took more than one hour. This shows that Julia is already able to scale to real software and automatically find evidence of security attacks.

8.5 The OWASP Cybersecurity Benchmark

As far as we know, the OWASP Benchmark Project represents the most relevant attempt to establish a universal security benchmark, *i.e.*, a suite of thousands of small Java programs containing security threats. According to their web page [86]:

The OWASP Benchmark for Security Automation is a free and open test suite designed to evaluate the speed, coverage, and accuracy of automated software vulnerability detection tools and services. Without the ability to measure these tools, it is difficult to understand their strengths and weaknesses, and compare them to each other.

The benchmark has benefited from the critical contribution of many organizations, so that it has also served as a way of clarifying the actual nature of the threats. During the years, it has emerged as *the* reference for the comparison of security analysis tools, and it is nowadays a must-do for any tool that asserts to find software security vulnerabilities in Java code. Most tests of the OWASP benchmark are servlets that might allow unconstrained information flow from their inputs to dangerous routines. A few tests are not related to injections, but rather to unsafe cookie exchange or to the use of inadequate cryptographic algorithms, hash functions or random number generators. Injection attacks are however the most complex to spot and have larger scientific interest. The benchmark sets traps for tools, *i.e.*, contains also harmless servlets that *seem* to feature security threats, at least at a superficial analysis. In this way, the benchmark measures the number of true positives (that is, real vulnerabilities reported by the tool), and false positives (that is, vulnerabilities reported by the tool that are not real issues). They represent a deep and wide stress test for the tools: a perfect analyzer should not be caught in a trap while still reporting all the real vulnerabilities. In an ideal world, a tool would get 100% true positive and 0% false positives. However, to achieve such a result one should be able to explore all possible executions of a program; therefore, existing tools make a compromise between soundness, precision, and efficiency of the analysis.

An important feature of this benchmark is the automatic generation of reports to compare different tools: there are several scripts to plot coverage and accuracy of the tools, inside comparative *scorecards*. This gives an immediate graphical picture of the relative positioning of the tools. Free tools are plotted in the scorecards. Commercial tools are anonymized into their overall average [86].

8.5.1 Analysis of the OWASP Benchmark with Julia

A (simplified) example of OWASP benchmark test is shown in Fig. 8.5. Julia warns about a possible XSS attack at the last line, since the `bar` parameter to `format()` (at line 14) is tainted. In fact, this parameter is built from the content of local variable `param` (line 9), that received (line 5) an input that the user can control (the header of the connection). Note that Julia correctly spots the information flow through the constructor of `StringBuilder` and the call to `replace()`. Consider the test in Fig. 8.6 now. Julia does not issue any warning here. Actually, no possible XSS attack is possible this time, since variable

```
 1 public void doPost(HttpServletRequest request,
 2                    HttpServletResponse response) throws ... {
 3   response.setContentType("text/html;charset=UTF-8");
 4   String param = "";
 5   if (request.getHeader("Referer") != null)
 6     param = request.getHeader("Referer");
 7   param = java.net.URLDecoder.decode(param, "UTF-8");
 8   String bar = param;
 9   if (param != null && param.length() > 1) {
10     StringBuilder sbxyz67327 = new StringBuilder(param);
11     bar = sbxyz67327.replace(param.length()-"Z".length(), param.length(),
12                "Z").toString();
13   }
14   response.setHeader("X-XSS-Protection", "0");
15   Object[] obj = { "a", "b" };
16   response.getWriter().format(java.util.Locale.US,bar,obj);
17 }
```

Fig. 8.5: Benchmark 146: This test suffers from a real XSS attack and Julia spots it.

`param` (tainted at line 6) is sanitized into `bar` by Spring method `htmlEscape()` (line 8). Julia uses a dictionary of sanitizing methods and others can be specified by the user. Consider the test in Fig. 8.7 now. This time Julia falls in the

```
 1 public void doPost(HttpServletRequest request,
 2                    HttpServletResponse response) throws ... {
 3   response.setContentType("text/html;charset=UTF-8");
 4   String param = "";
 5   java.util.Enumeration<String> headers = request.getHeaders("Referer");
 6   if (headers != null && headers.hasMoreElements())
 7     param = headers.nextElement();
 8   param = java.net.URLDecoder.decode(param, "UTF-8");
 9   String bar = org.springframework.web.util.HtmlUtils.htmlEscape(param);
10   response.setHeader("X-XSS-Protection", "0");
11   response.getWriter().print(bar);
12 }
```

Fig. 8.6: Benchmark 278: This is a trap, since no XSS attack is possible here, and Julia does not fall in it.

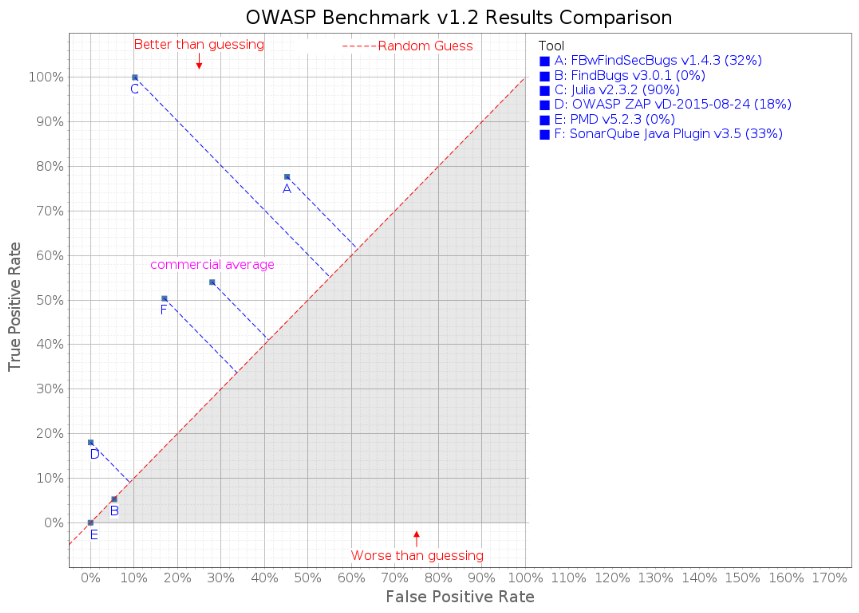
trap and issues a spurious warning about a potential XSS attack at the call to `format()`, since it thinks that `bar` (and hence `obj`) is tainted. But this is not actually the case, since this test manipulates a `valueList` in such a way that the value finally stored into `bar` is untainted. This list manipulation is too complex for the taint analysis of Julia, that cannot distinguish each single element of the list and conservatively assumes all elements of the list to be tainted.

```
public void doPost(HttpServletRequest request,
                   HttpServletResponse response) throws ... {
  response.setContentType("text/html;charset=UTF-8");
  String param = "";
  if (request.getHeader("Referer") != null)
    param = request.getHeader("Referer");
  param = java.net.URLDecoder.decode(param, "UTF-8");
  String bar = "alsosafe";
  if (param != null) {
    java.util.List<String> valuesList =
                   new java.util.ArrayList<String>( );
    valuesList.add("safe");
    valuesList.add( param );
    valuesList.add( "moresafe" );
    valuesList.remove(0); // remove the 1st safe value
    bar = valuesList.get(1); // get the last 'safe' value
  }
  response.setHeader("X-XSS-Protection", "0");
  Object[] obj = { "a", bar };
  response.getWriter().format("Formatted like: %1$s and %2$s.",obj);
}
```

Fig. 8.7: Benchmark 147: This is a trap, since no XSS attack is possible here, and Julia is caught in it.

8.5.2 Results

Fig. 8.8 shows, on the left, a summarizing scorecard generated by the OWASP benchmark, Julia to other free (explicitly) and commercial (anonymously) static analyzers. Scorecards report soundness on the left and precision horizontally. Hence, a perfect (*i.e.*, sound and precise) tool should stay on the top left corner of the scorecard. Fig. 8.8 shows that Julia is very close to that corner, much more than all free analyzers and of the anonymous average of the commercial analyzers. Fig. 8.8 also reports the results of Julia for the eleven categories of threats considered by the OWASP benchmark. Julia is always close to the top left corner of the scorecard and always finds all threats, since it is the only sound analyzer in this comparison. Hence its results lie on the 100% line for soundness (true positive rate).



Category	**FN**	**TP**	**FP**
Command Injection	0	126	20
Cross-Site Scripting	0	246	19
Insecure Cookie	0	36	0
LDAP Injection	0	27	4
Path Traversal	0	133	22
SQL Injection	0	272	36
Trust Boundary Violation	0	83	12
Weak Encryption Algorithm	0	130	0
Weak Hash Algorithm	0	129	0
Weak Random Number	0	218	0
XPath Injection	0	15	3

Fig. 8.8: The scorecard comparing Julia with free and commercial tools and Julia's detailed results.

Fig. 8.8 reports also the number of false negatives (**FN**), true positives (**TP**), and false positives (**FP**) obtained by Julia on the OWASP benchmark. For all categories Julia obtained zero false negatives: this proves the (practical) soundness of the analysis, meaning that Julia is always able to spot security vulnerabilities if the programs contain them. In addition, the number of false positives is always a small percentage (below 20%) of the number of warnings produced: this proves the (practical) precision of the analysis, meaning that a developer using Julia to identify vulnerabilities will need to discard (at least on the OWASP benchmark) only a warning out of six.

Below, all categories are presented in detail.

Command Injection

This vulnerability makes possible to execute arbitrary system commands by injecting a command line string in an application through unvalidated user input. The command is then executed in the host system with the privileges of the vulnerable application. The exploit works by appending the desired command after a command chaining character sequence, such as `‘;’`, `‘|’`, `‘&&’`, etc. For example, the following C program is meant to list the file named by the first argument passed to it:

```c
int main(char* argc, char** argv) {
    char cmd[CMD_MAX] = "/usr/bin/cat ";
    strcat(cmd, argv[1]);
    system(cmd);
}
```

By using a filename like “`file.txt ; rm -rf /`”, the command “`rm -rf /`” is executed with the privileges of the executing program.

The Java function `Runtime.exec` behaves differently from the C function `system`. It considers the first word of input as the name of the program, and the remaining words as arguments to this program, whereas the `system` function passes the input to the shell to be interpreted, thus allowing all the concatenation sequences listed before to inject a command in the input and execute it.

Figure 8.9 shows the comparison with other tools for command injection.

Cross-Site Scripting

Using this vulnerability, an attacker can inject a script in a webpage presented to the user. This script is executed by the victim’s browser, as it comes from a seemingly trusted source. In a *stored* Cross Site Scripting (XSS), the malicious script is saved in the server, and presented to the user at a later time. The script can be injected in a forum entry, a comment, or similar structure. In a *reflected* XSS, the script is delivered by the server directly to the browser which sent it, for example in an error message or search result page.

Figure 8.10 shows the comparison with other tools for XSS.

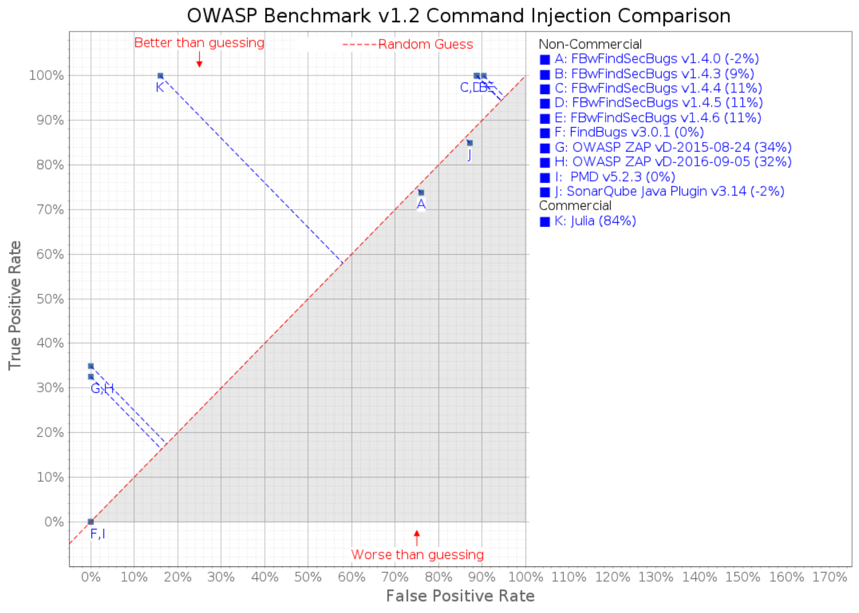


Fig. 8.9: The scorecard comparing Julia with free and commercial tools in detecting Command Injection

Insecure Cookie

This refers to the fact that the application uses insecure cookies, , *i.e.*, sent in plaintext. This vulnerability can be fixed simply by setting the *secure* attribute. Figure 8.11 shows the comparison with other tools for insecure cookie.

LDAP Injection

LDAP is a standard protocol for accessing directory services over a network. Similarly to other injections, strings containing LDAP statements can be manipulated so to modify their intended meaning. In the following code

```
String query = "(cn=" + userName + ")";
```

If `userName` is provided by the user and not sanitized, it can contain LDAP code that whenever injected in the executed statement may compromise the security of the LDAP database. For example, if `userName` is `*`, all usernames in the database will be included in the query.

Figure 8.12 shows the comparison with other tools for LDAP injection.

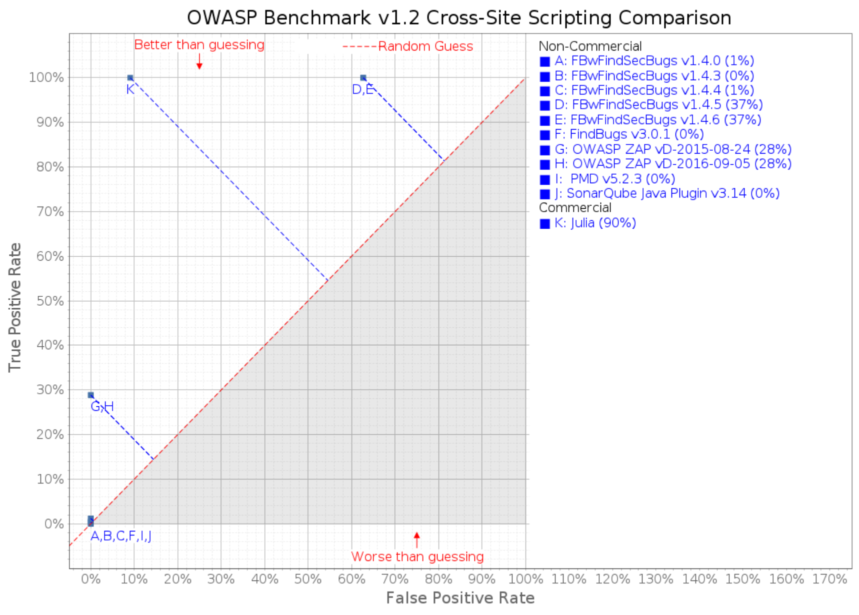


Fig. 8.10: The scorecard comparing Julia with free and commercial tools in detecting Cross-Site Scripting

Path Traversal

A path traversal attack is the injection of special sequences of characters in an *URL*, such that the attacker can access private files in the server. A typical sequence is `../`, that refers to the enclosing directory and, under certain conditions, allows the attacker to build an URL that traverses the directory of the whole application, or of the whole system.

Figure 8.13 shows the comparison with other tools for path traversal.

SQL Injection

SQL injections are the most widespread vulnerabilities for web applications. They consist in inserting SQL code in an already existing SQL statement, such that unintended operations may be performed on the underlying database.

Figure 8.14 shows the comparison with other tools for SQL injection.

Trust Boundary Violation

This vulnerability refers to mixing trusted and untrusted data in the same data structure. This is mainly an architectural issue, as it becomes more difficult

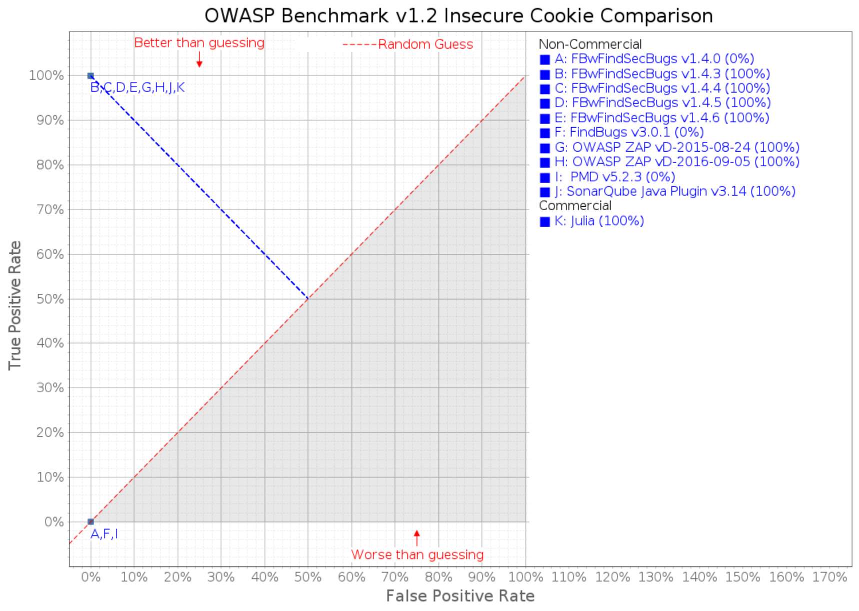


Fig. 8.11: The scorecard comparing Julia with free and commercial tools in detecting Insecure Cookies attacks

for programmers to track what has been validated as trusted and what has not, and mistakenly trust unvalidated data.

For example, in the following code an unvalidated username is inserted into the `session` object, that should instead remain trusted:

```
user = request.getParameter("user");
if (session.getAttribute(ATTR_USR) == null) {
   session.setAttribute(ATTR_USR, user);
}
```

Figure 8.15 shows the comparison with other tools for trust boundary violation.

Weak Encryption Algorithm

This refers to the use of obsolete or non-standard algorithms for cryptography.

Figure 8.16 shows the comparison with other tools for weak encryption algorithm.

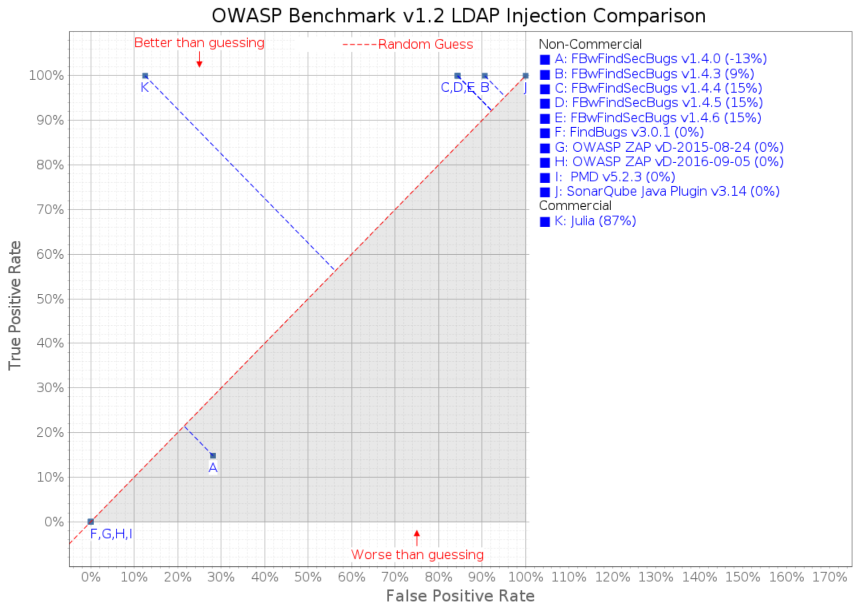


Fig. 8.12: The scorecard comparing Julia with free and commercial tools in detecting LDAP Injection

Weak Hash Algorithm

This problem happens when the application uses a hashing function that is easy to invert, or for which it is easy to find a collision. This consideration is important when security is involved, such as with password hashes.

Figure 8.17 shows the comparison with other tools for weak hash algorithm.

Weak Random Number

This refers to the use of a non cryptographically secure random number generator (RNG). In a security context a predictable generator may allow an attacker to guess values that should be kept secret, and impersonate another user or access sensitive information. Java provides a secure RNG in the class `java.security.SecureRandom`, as opposed to `java.util.Random`.

Figure 8.18 shows the comparison with other tools for weak random number.

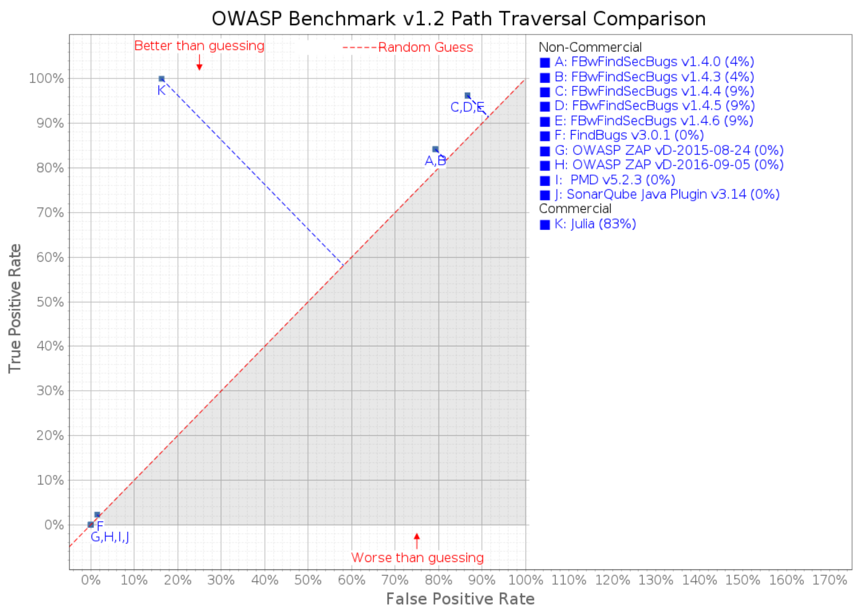


Fig. 8.13: The scorecard comparing Julia with free and commercial tools in detecting Path Traversal attacks

XPath Injection

XPath is a standard query language for XML documents. As it happens with SQL injections, fragments of XPath code can be injected in an XPath query, to perform tasks not intended in the original formulation.

For example, the following XPath query returns the employee having the supplied username and password:

```
String query = "//Employee[UserName/text()='" + username
    + "' And Password/text()='" + password + "']"
```

By injecting a string like "`john' or 1=1 or 'a'='a`" as the username, an attacker could access to all of the records in the XML file.

8.6 Multithreaded Experiments

The implementation in Julia is able to perform multiple analyses at the same time, for different kinds of injection.

Figure 8.20 shows the time needed to perform from 1 to 4 injection analyses with the Julia Static Analyzer, in parallel on a quad-core processor, with the three libraries BeeDeeDee, JavaBDD and JDD. When we use BeeDeeDee, the

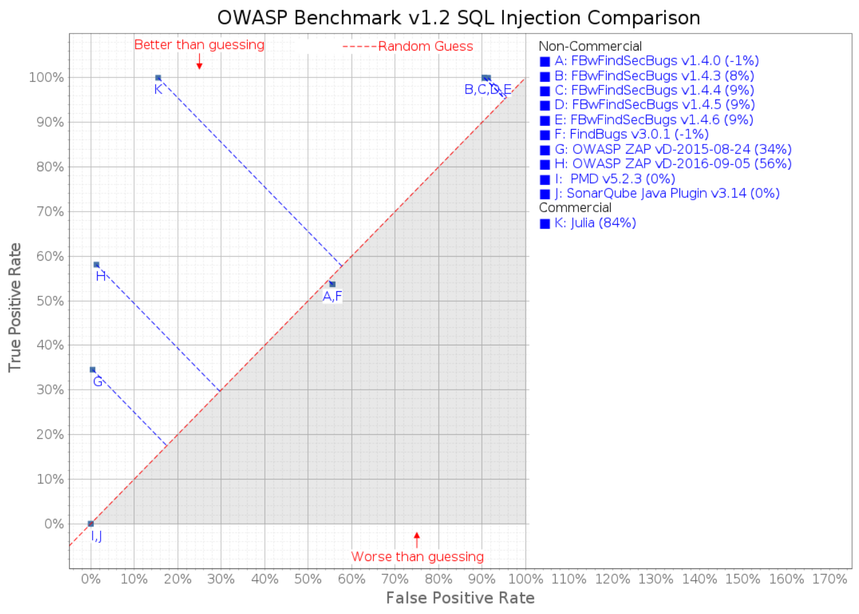


Fig. 8.14: The scorecard comparing Julia with free and commercial tools in detecting SQL Injection

BDD unique table and caches are shared, while this is not possible for the other, non-thread-safe libraries. JDD's poor performance is due to the fact that the injection analysis heavily uses quantification and replacement operations, for which JDD is not optimized. JavaBDD is slightly faster in this case (it pays no synchronization overhead), but it consumes more memory, as we show next with Figure 8.21. There, a parallel information flow analysis with 4 kinds of injection is performed by using the two libraries; with BeeDeeDee it never requires more than 9 gigabytes of RAM, whereas with JavaBDD around 11 gigabytes are required, 2 more than with BeeDeeDee. The information flow analysis is only the tip of the iceberg, resting on previous processing and analyses, that amount to many gigabytes of RAM, independently from the BDD library. For a fairer comparison, we hence have to consider only the memory occupied by the BDD library, that is, the BDD table size. For that, BeeDeeDee uses single shared unique table and caches, that in this example reach a size of 2,200,000 nodes; whereas JavaBDD needs four different unique table and caches, for a cumulative size of 5,500,000 nodes. This shows that BeeDeeDee is an effective choice when it is sensible to share unique node table and caches among different threads to reduce the memory footprint of the overall computation. We stress the fact that the precision of the information flow analyses is always the same, independently from the BDD library that we

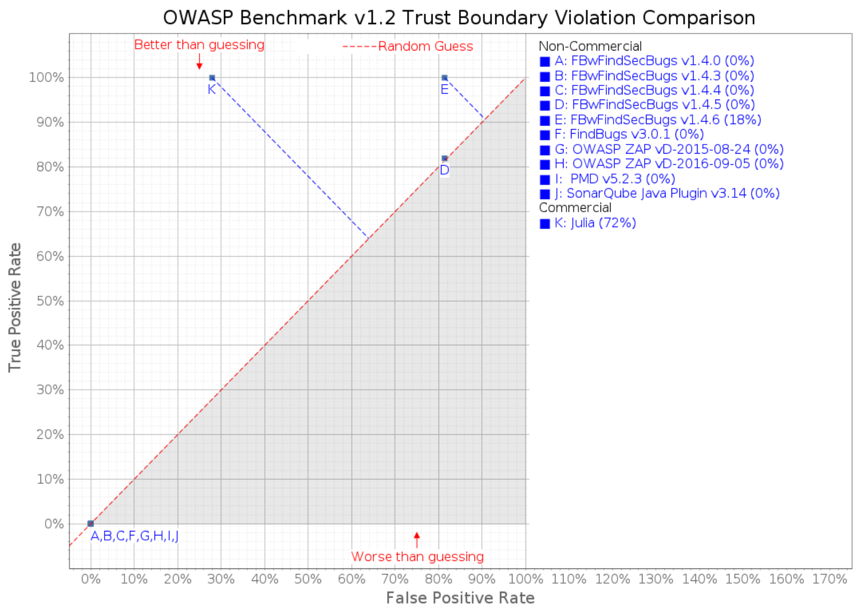


Fig. 8.15: The scorecard comparing Julia with free and commercial tools in detecting Trust Boundary Violation

use, since it depends on the definition of the abstraction and of the abstract operations, not on the BDD library used for their implementation.

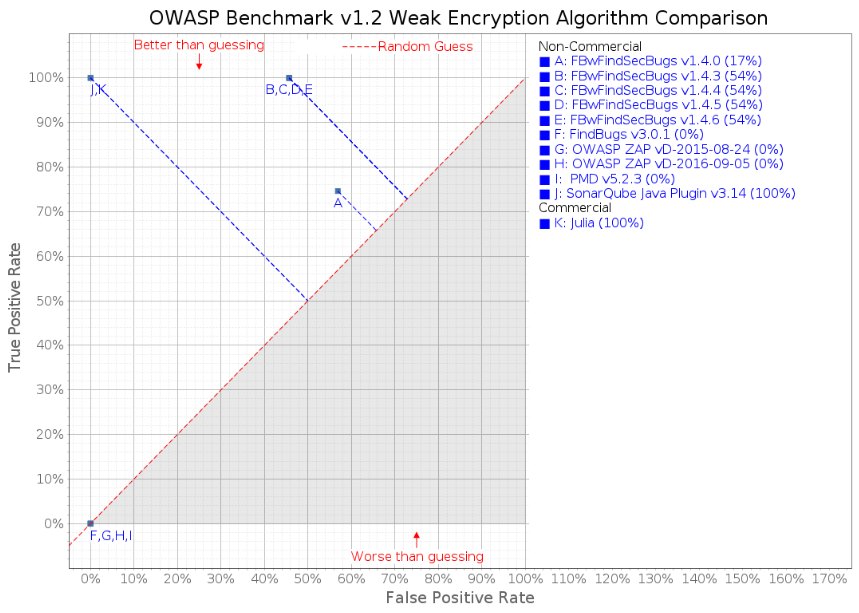


Fig. 8.16: The scorecard comparing Julia with free and commercial tools in detecting Weak Encryption Algorithms

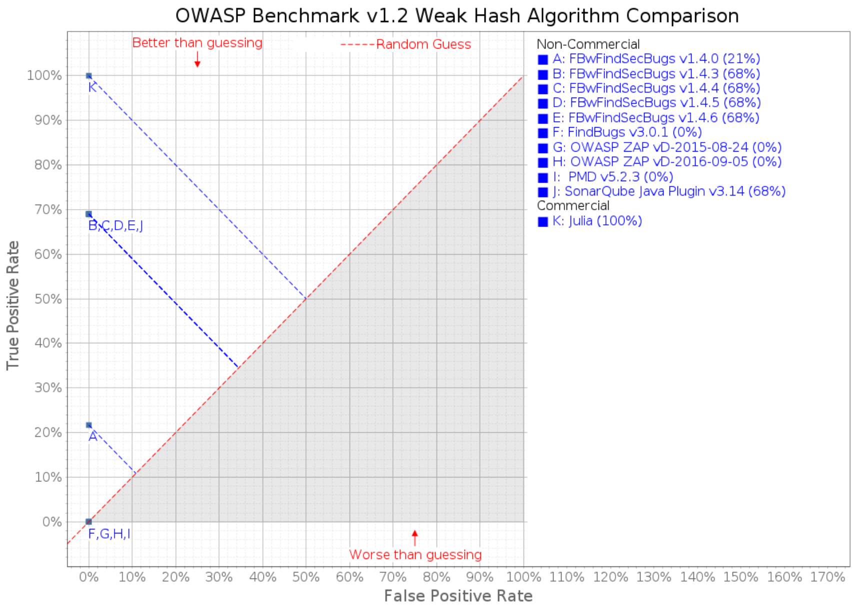


Fig. 8.17: The scorecard comparing Julia with free and commercial tools in detecting Weak Hash Algorithms

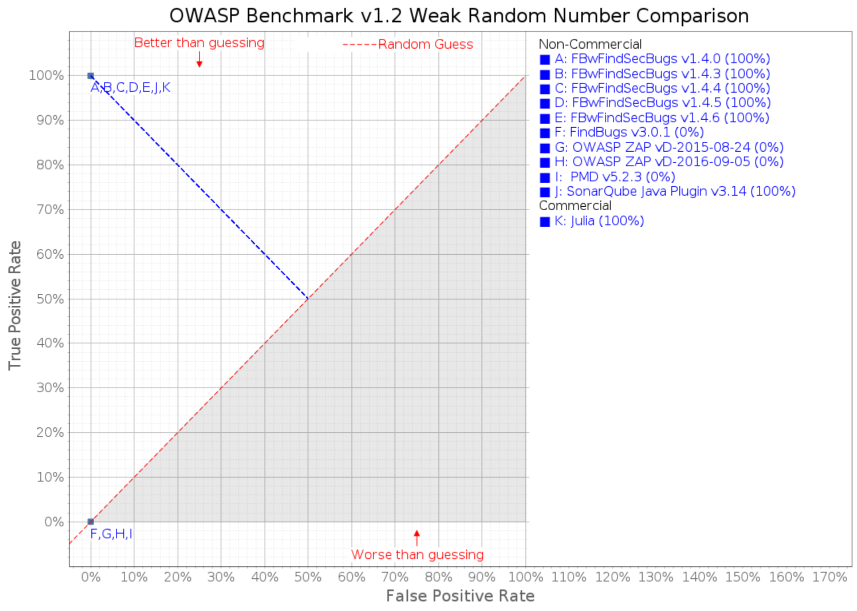


Fig. 8.18: The scorecard comparing Julia with free and commercial tools in detecting Weak Random Number Generation

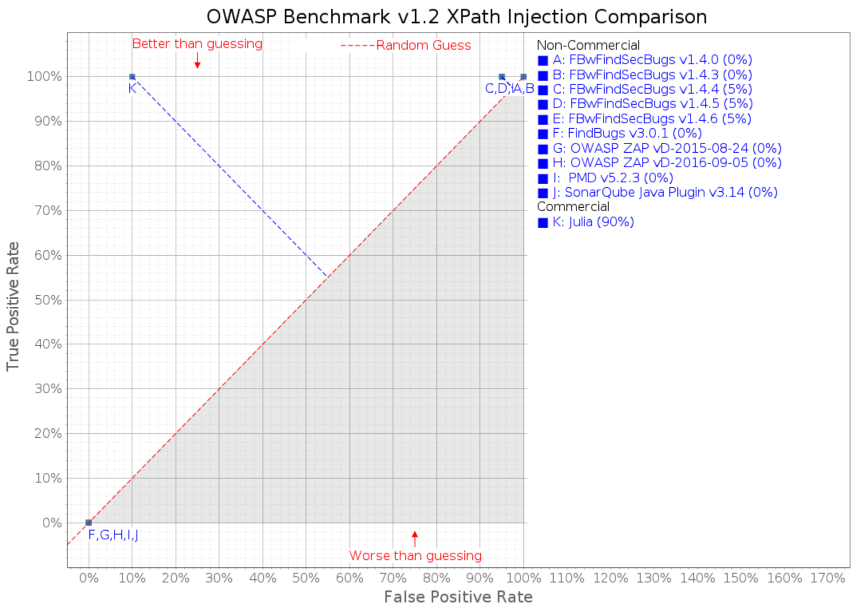


Fig. 8.19: The scorecard comparing Julia with free and commercial tools in detecting XPath Injection

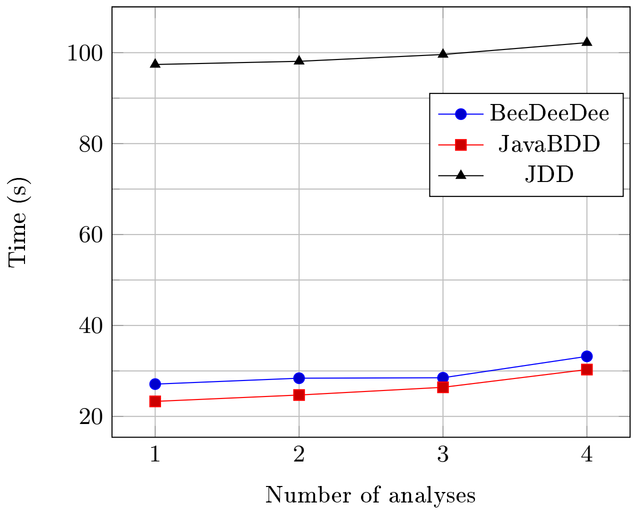
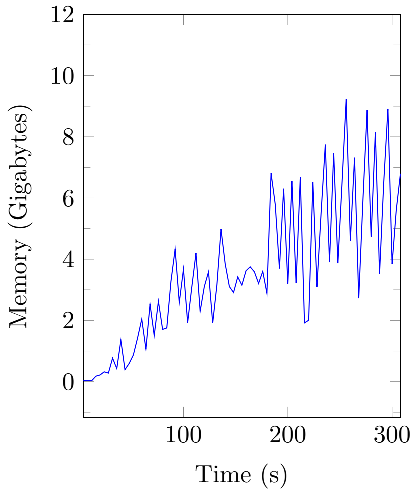
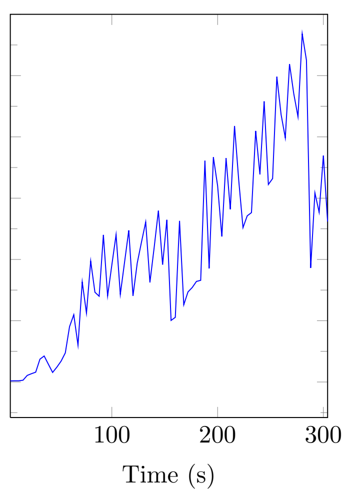


Fig. 8.20: Parallel information flow analysis for different kinds of injection



(a) BeeDeeDee



(b) JavaBDD

Fig. 8.21: Memory consumption for a flow analysis with 4 kinds of injection

9

Locking Discipline Inference

Concurrency is a requirement for much modern software, but the implementation of multi-threaded algorithms comes at the risk of errors such as data races. Programmers can prevent data races by documenting and obeying a *locking discipline*, which indicates which locks must be held in order to access which data.

This chapter introduces a formal semantics for locking specifications that gives a guarantee of data race freedom. A notable difference from most other semantics is that it is in terms of values (which is what the runtime system locks) rather than variables.

Experiments compare the annotations inferred by our analysis with those written by programmers, showing that the ambiguities and unsoundness of previous formulations are a problem in practice.

9.1 Locking Discipline Semantics

This section shows how a locking discipline can enforce mutual exclusion and the absence of data races; lays out the design space for a locking discipline semantics; and discusses why such a semantics should provide value protection rather than name protection.

9.1.1 Dining Philosophers Example

To illustrate how to specify a locking discipline, consider the traditional dining-philosophers example. More examples are given later.

A group of philosophers sit around a table; there is a fork between each pair of philosophers; and each philosopher needs its left and right forks to eat. The locking discipline provides each fork with a lock, and a philosopher must hold the lock in order to use the fork; this guarantees mutual exclusion and the absence of race conditions.

```
public class Fork implements Comparable<Fork> {
  private static int nextId = 0;
  private final int id = nextId++;
  // who is holding the fork, or null if on the table
  private Philosopher usedBy = null;

  void pickUp(Philosopher philosopher) {
    this.usedBy = philosopher;
  }

  void drop() {
    this.usedBy = null;
  }

  public int compareTo(@GuardedBy("itself") Fork other) {
    return id − other.id;
  }

  public synchronized String toString() {
    if (usedBy != null)
      return "fork " + id + " used by " + usedBy.getName();
    else
      return "fork " + id + " on the table";
  }
}
```

Fig. 9.1: A fork, possibly held by a philosopher.

Figure 9.1 shows Java code for the fork. The fork contains mutable information (which philosopher holds it) in order to demonstrate how a locking discipline can protect access to a mutable field. A philosopher (Figure 9.2) is modeled as a thread whose `run` method repeatedly thinks, locks its two forks, eats, and unlocks the forks. The code illustrates a situation in which classes cooperate to implement a synchronization policy, rather than the less challenging case of all code being in the same class.

In Java, each object is associated with a monitor [62, S 17.1] or *intrinsic* lock. A `synchronized` statement or method locks the monitor, and exiting the statement or method unlocks the monitor. Java also provides *explicit* locks, which our theory and implementation handles.

The `@GuardedBy` type qualifiers express the locking discipline. In the semantics that we will introduce, the type qualifier `@GuardedBy("itself")` on a variable's type states that the variable holds a value v whose non-`final` fields are only accessed at moments when v's monitor is locked by the current thread.

```
public class Philosopher extends Thread {
  private final @GuardedBy("itself") Fork left;
  private final @GuardedBy("itself") Fork right;

  Philosopher(String name, @GuardedBy("itself") Fork left,
                           @GuardedBy("itself") Fork right) {
    super(name);
    // a fixed ordering avoids deadlock
    if ( left .compareTo(right) < 0) {
      this. left  = left ; this.right  = right;
    } else {
      this. left  = right; this.right  = left ;
    }
  }

  public void run() {
    while (true) {
      think();
      synchronized (left) {
        left .pickUp(this);
        synchronized (right) {
          right .pickUp(this);
          eat ();
          right .drop();
        }
        left .drop();
      }
    }
  }

  private void think() { ... }

  @Holding({ "left", "right" })
  private void eat() { ... }
}
```

Fig. 9.2: A philosopher.

Our tool infers and verifies the `@GuardedBy` annotations in these figures. The `@GuardedBy("itself")` type qualifiers on fields `left` and `right` guarantee that philosophers use their forks only after properly locking them. The unlocked access to the `final` field `id` on line 16 of fig. 9.1 does not violate the `@GuardedBy("itself")` specification.

9.1.2 Design Space for Locking Discipline Semantics

Recall the informal definition of `@GuardedBy`: when a programmer writes `@GuardedBy(`E`)` on a program element, then a thread may use the program element only while holding the lock E. This definition suffers the following ambiguities related to the guard expression E.

1. May a definite alias of E be locked? Given a declaration `@GuardedBy ("lock") Object shared;`, is the following permitted?
   ```
   Object lockAlias = lock;
   synchronized (lockAlias) {
     ... use shared ...
   }
   ```
2. Is E allowed to be reassigned while locked? Given a declaration `@GuardedBy("lock") Object shared;`, is the following permitted?
   ```
   synchronized (lock) {
     lock = new Object();
     ... use shared ...
   }
   ```
   What about other side effects to E? Given a declaration `@GuardedBy ("anObject.field") Object shared;`, are the following permitted?
   ```
   synchronized (anObject.field) {
     foo();   // might side-effect anObject and reassign field
     ... use shared ...
   }
   synchronized (anObject.field) {
     foo();   // might side-effect but not reassign field
     ... use shared ...
   }
   ```
3. Should E be interpreted at the location where it is defined or at the location where it is used? Given a declaration
   ```
   class C {
     @GuardedBy("this") Object field;
     ...
   }
   ```
   are the following permitted?
   ```
   C c;
   synchronized (this) {
     ... use c.field ...
   }
   synchronized (c) {
     ... use c.field ...
   }
   ```

The latter use assumes some kind of contextualization, such as viewpoint adaptation [43].

The informal definition suffers further ambiguities in the interpretation of the program element being guarded. These can be summarized by asking, what is a "use" of the shared program element? Is it any occurrence of the variable name or only certain operations; do uses of aliases count, and are re-assignment and side effects permitted? More relevantly, does the `@GuardedBy` annotation specify restrictions on uses of a variable name ("name protection"), or restrictions on uses of values ("value protection")?

Current definitions of `@GuardedBy` do not provide guidance about any of the ambiguities regarding the lock expression. Thus, there is a danger that different tools interpret them differently, including unsound interpretations that do not prevent data races. There is also a danger that programmers will assume a different definition than a tool provides, and thus do not obtain the guarantee they expect.

Current definitions of `@GuardedBy` are clearer about what constitutes a use of the program element — any access to (that is, lexical occurrence of) the name. This definition provides name protection, but unfortunately it does not prevent data races. A program that obeys this locking discipline might not be thread-safe and may still suffer data races, as illustrated below. Therefore, any definition that provides name protection is in general incorrect, because it does not satisfy the stated goals of the `@GuardedBy` annotation.

9.1.3 Name Protection and Value Protection

Name protection and value protection are distinct and incomparable. Neither one implies the other. To illustrate the differences, consider an implementation of the observer design pattern [54], which is a key part of model-view-controller and other software architectures. Figures 9.3 and 9.4 are patterned after the implementation found in the Java JDK. An `Observable` object allows clients to concurrently register listeners. When an event of interest occurs, a callback method is invoked on each listener.

Synchronization is required to avoid data races. Synchronization in the `register` method and copy constructor prevents simultaneous modifications of the `listeners` list, which might result in a corrupted list or lost registrations. Synchronization is needed in the `getListeners()` method as well, or otherwise the Java memory model would not guarantee the inter-thread visibility of the registrations. In fig. 9.3, synchronization is performed on the container object, and in fig. 9.4, synchronization is performed on a field.

Figure 9.3 satisfies all interpretations of the name protection semantics: every use of `listeners` occurs at a program point where the current thread locks its container.[1] Nevertheless, a data race is possible, since two threads

[1] It also satisfies an interpretation of `@GuardedBy` that does not do contextualization or viewpoint adaptation, since the constructor is implicitly synchronized on `this`.

```
public class Observable {
  private @GuardedBy("this") List<Listener> listeners
    = new ArrayList<>();
  public Observable() {}
  public Observable(Observable original) {  // copy constructor
    synchronized (original) {
      listeners.addAll(original.listeners);
    }
  }
  public void register(Listener listener) {
    synchronized (this) {
      listeners.add(listener);
    }
  }
  public List<Listener> getListeners() {
    synchronized (this) {
      return listeners;
    }
  }
}
```

Fig. 9.3: An implementation of the observer design pattern in which locking is performed on the container `Observable` object. This implementation suffers data races. The implementation satisfies the name-protection semantics for `@GuardedBy`, but not the value-protection semantics.

could call `getListeners()` and later access the returned value concurrently. This demonstrates that the name protection semantics does not prevent data races. Figure 9.3 does not satisfy the value-protection semantics (which prevent data races), because the return type of `getListeners()` is not compatible with the `return` statement. Figure 9.3 could be made to satisfy the value-protection semantics by annotating the return type of `getListeners()` as `@GuardedBy("this")`, which would force the client program to do its own locking and would prevent data race.

Figure 9.4 specifies a different locking discipline. First consider the value-protection semantics. `@GuardedBy("itself")` means that all dereferences (field accesses) of the value of `listeners` occur while the current thread locks that value. The annotation on the return type of `getListeners()` imposes the same requirement on clients of `Observable`. The field `listeners` could have been annotated `@GuardedBy("listeners")`, but the syntax for the return type of `getListeners()` would have been more complex, thus the `@GuardedBy("itself")` syntax. Figure 9.4 also satisfies the name-protection semantics. Depending on how the semantics handles aliasing and side effects, the semantics may prevent clients of this program from suffering data races.

```
public class Observable {
  private @GuardedBy("itself") List<Listener> listeners
    = new ArrayList<>();
  public Observable() {}
  public Observable(Observable original) {      // copy constructor
    synchronized (original.listeners) {
      listeners.addAll(original.listeners);
    }
  }
  public void register(Listener listener) {
    synchronized (listeners) {
      listeners.add(listener);
    }
  }
  public @GuardedBy("itself") List<Listener> getListeners() {
    synchronized (listeners) {
      return listeners;
    }
  }
}
```

Fig. 9.4: An implementation of the observer design pattern in which locking is performed on the `listeners` field.

Figure 9.4's choice of locking the field rather than the container permits additional flexibility. Consider the following client code:

```
List<Listener> l = new Observable(original).getListeners();
... use l ...
```

At the use of `l`, there is no syntactic handle for the container, and it might even have been garbage-collected. Instead, the annotation `@GuardedBy("itself")` is perfectly meaningful for `l`.

Regardless of other choices for the semantics of `@GuardedBy`, the name-protection and value-protection variants are not comparable: neither entails the other. In fig. 9.5, field `x` is declared as `@GuardedBy("itself")`. This annotation holds in the value-protection semantics, since its value is only accessed at line 11 inside a synchronization on itself, but not in name-protection semantics: `x` is used at line 8. Field `y` is `@GuardedBy("this.x")` for name protection but not for value protection: its value is accessed at line 14 via `w`. In some cases the semantics do coincide. Field `z` is `@GuardedBy("itself")` according to both semantics: its name and value are only accessed at line 11, where they are locked. Field `w` is not `@GuardedBy` according to any semantics: its name and value are accessed at line 14.

```
public class K {
  private K1 x = new K1();
  private K2 y = new K2();
  private K1 z;
  private K2 w;

  public void m() {
    z = x;
    w = new K2();
    synchronized (z) {
      y = z.f;
      w = y;
    }
    w.g = new Object();
  }
}

class K1 {
  K2 f = new K2();
}

class K2 {
  Object g = new Object();
}
```

var	name protection	value protection
x	–	`@GB("itself")`
y	`@GB("this.x")`	–
z	`@GB("itself")`	`@GB("itself")`
w	–	–

Fig. 9.5: Comparison of name-protection and value-protection semantics for `@GuardedBy` (abbreviated as `@GB`).

9.1.4 Definition of `@GuardedBy`

We can now state our semantics for the `@GuardedBy` annotation. By *dereference* of a value v we mean the access of a non-`final` field of v. The key idea is that values are protected rather than names, and that dereferences of v are considered uses of v.

Suppose that the type of expression x contains the qualifier `@GuardedBy(`E`)`. A program satisfies the locking discipline if, at program point p where the program dereferences a value that has ever been bound to x, the current thread holds the lock on the value of expression E. Furthermore, the value of E must not change (in any thread) during the time that the thread holds the lock. The protection is *shallow*, since it applies to the value that x evaluates to, not to all values reachable from it. There is no restriction on copying values, including passing values as arguments (including as the receiver) or returning values.

This definition resolves the ambiguities noted in section 9.1.2. A definite alias of the guard expression E is permitted to be locked. The guard expression is not allowed to be reassigned to a different value while locked. Side effects to the guard value are permitted, since they do not affect the monitor. The lock expression undergoes viewpoint adaptation so that it makes sense in the context of use. A use of the program element is a dereference of any value it may hold, regardless of aliasing, reassignment, and side effects.

A set of `@GuardedBy` annotations expresses a locking discipline. Julia infers a maximal locking discipline that the program satisfies. Every program trivially satisfies the empty locking discipline.

9.1.5 Definition of `@Holding`

The `@GuardedBy` annotation is sufficient for expressing a locking discipline. Inferring or checking a locking discipline requires reasoning about which locks are held at any given point in the program. Our implementation provides a `@Holding(`E`)` annotation to express these facts explicitly to aid in program comprehension or modular checking.[2] It annotates a method declaration to indicate that when the method is called, the current value of E (possibly viewpoint-adapted) is locked. An example appears on line 58 of fig. 9.2.

9.2 Locking Discipline Inference

Our abstract-interpretation-based, whole-program inference has been implemented inside the Julia static analyzer [13]. It uses four static analyses to infer `@GuardedBy` annotations (fig. 9.6), as described in this section. Inference of `@Holding` is based on similar techniques but is simpler. Creation points and definite aliasing analysis have been previously published [104, 83], including technical details of their abstract domains, and hence sections 9.2.1 and 9.2.2 only describe their use for the inference of a locking discipline. Definite locked expression analysis and locking discipline inference are described in sections 9.2.3 and 9.2.5. These four static analyses are sound, up to the use of reflection and native methods, where the analysis conservatively assumes the method may return any value of any known type, but optimistically assumes that the call has no side effects. Soundness and the use of a definite aliasing analysis entail that our analysis never mistakenly infers a field/variable as `@GuardedBy(E)`. However, it might fail to infer some `@GuardedBy(E)` annotations that actually hold in the program, since the aliasing analysis might be approximated or since `E` might be too complex or the creation points analysis might be too coarse. Also note that an inference tool infers not what the programmer intended, but what the programmer implemented.

Julia only infers E made up of final fields and the special variable `itself`, which refers to the same value being protected. This is a common, safe programming practice and caused no problems in our case studies.

9.2.1 Creation Points Analysis

Creation points analysis is an instance of *class analysis* [106] and its first use in Julia is to build the call-graph of the program under analysis. Julia implements a concretization of Parlsberg and Schwarzbach's class analysis [87, 104]. For each variable and field of reference type, creation points analysis infers an

[2] JCIP overloads the name `@GuardedBy` for two distinct purposes as a field annotation and a method precondition. For clarity, we always refers to the latter as `@Holding`.

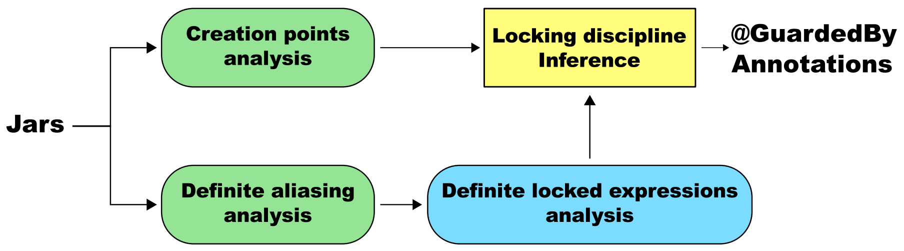

Jars
Creation points analysis
Definite aliasing analysis
Locking discipline Inference
Definite locked expressions analysis
@GuardedBy Annotations

Fig. 9.6: The structure of the abstract interpretation inference.

overapproximation of the set of program points where the value bound to that variable or field might have been created. This is a *concretization* since it does not track types of values, but rather their creation point, from which the type can be derived. The approximation of local variables in this analysis is flow-sensitive, while the approximation of object fields is flattened, context-insensitive [104]. Hence this analysis is sound for concurrent programs. For efficiency, allocation sites and function call sites are not context-sensitive (it is a 0-CFA analysis [104]).

The use of creation point analysis in the inference of `@GuardedBy` is for computing an overapproximation of run-time values, since two variables that hold the same object (value) must have the same creation point, while the converse does not hold in general. Figure 9.7 shows the result of Julia's creation points analysis at some selected points of the program of figs. 9.1 and 9.2 and a client program that creates forks and philosophers and starts the philosopher processes. It reports where the values of the variables at those program points and of the fields of the objects have been created by a `new` statement. For instance, the figure shows that variable `other` at line 16 contains a value of type `Fork` that can only be created in the driver program. The same holds for the values held in fields `left` and `right` of all `Philosopher` objects in memory. Figure 9.7 also reports the creation points of the objects passed to the Java library, including the implicit argument (receiver) of `getName`, which will be needed later. Note that, in Java bytecode, those arguments are held in stack variables, hence the creation points analysis computes that information. In this simple example, the approximation is always a singleton, but in general it could be a set of creation points. If the line numbers are dropped from column *creation points*, one gets a class analysis. That extra information makes it into a creation points analysis.

9.2.2 Definite Aliasing Analysis

This analysis infers, at each program point and for each local variable, expressions that are definitely aliased with that variable [83]. *Definite* means that aliasing must hold at the program point, however it is reached. This analysis is limited to alias expressions built from variables and final fields (or fields

lines	variable/field	creation points
8,12,16	`this`	{`Fork@80`}
20,21,23	`this`	{`Fork@80`}
16	`other`	{`Fork@80`}
35,37	`this`	{`Philosopher@84`}
-	`Philosopher.left`	{`Fork@80`}
-	`Philosopher.right`	{`Fork@80`}
21,21,23	arg. to `String.concat`	$\{\kappa, \pi\}$
21	arg. to `Thread.getName`	{`Philosopher@84`}

Fig. 9.7: Creation points analysis of our example. Creation point π stands for a generic creation point inside the Java library code; κ stands for an object held in the constant pool.

that are never modified after being initialized). Hence this analysis is sound for concurrent programs.

In particular, we are interested in definite aliases of values used in the `synchronized` statements in our example. Those values are held in a stack variable in bytecode, whose definite aliases are shown in fig. 9.8, as computed by the analysis. Note that the approximation is semantic. For instance, the analysis would not change if one modified the code at line 44 into `Fork f = left; synchronized (f) ...` Later, it will be useful to know the definite aliases of the container E in each field access expression $E.f$ where f is a non-`final` field. Figure 9.8 provides that information for our example as well.

lines	definite aliases of locked value
19	{`this`}
44	{`this.left`}
46	{`this.right`}

lines	definite aliases of the container of the field
8,12,20	{`this`}

Fig. 9.8: Expression aliasing analysis of our example.

9.2.3 Definite Locked Expressions Analysis

At each program point p, the definite locked expressions analysis builds an under-approximation A_p of the set of definitely locked expressions at p. This is a solution of a constraint built from the statements of the program. Normally, these constraints just propagate the approximation from the statement at p to the next at $p+1$:

$$A_p \supseteq A_{p+1}$$

For instance, a constraint for a method call does not modify the set since, according to the semantics of Java, locks cannot be released by callees, and locks acquired in a method must be released before returning to the caller.[3] We assume that this property is also valid in bytecode, and indeed JVM implementations can enforce it, by throwing an exception on its violation.

Some statements instead affect the set of definitely locked expressions. At synchronization points, the definite aliases of the locked value are inserted, such as at line 44:

$$A_{44} \cup \{\texttt{this.left}\} \supseteq A_{45}$$

Conversely, at the end of a synchronization block, the constraint conservatively kills all definitely locked expressions whose type is compatible with that of the unlocked expression, such as at line 50:

$$A_{50} \setminus \{E \in A_{50} \mid E \text{ has type compatible with } \texttt{Fork}\} \supseteq A_{51}$$

The analysis is interprocedural. Namely, definitely locked expressions are renamed at method call, such as at line 45, to implement parameter passing:

$$\left\{ E \begin{bmatrix} a_1 \mapsto \texttt{this} \\ a_2 \mapsto \texttt{philosopher} \end{bmatrix} \;\middle|\; \begin{array}{l} E \in A_{45},\ \text{the receiver of } \texttt{pickUp} \\ \text{is definitely aliased to } a_1, \\ \text{the parameter of } \texttt{pickUp} \\ \text{is definitely aliased to } a_2 \end{array} \right\} \supseteq A_7$$

Expressions being propagated should only contain variables forming the actual parameters of the method, since they are available in the callee, after the renaming.

In order to define a sound analysis for concurrent programs, aliases must be unmodifiable expressions, as in our example, where `left` and `right` are `final` fields.

These inclusion constraints are built for each pair of consecutive statements and from callers to callees, eventually composing a set constraint. The result of the analysis is computed as a maximal fixpoint of the set constraint, since the analysis is definite. Fig. 9.9 shows the resulting sets for our example program.

lines	definitely locked expressions
8,12,20,21,23	{`this`}
16,35,37	{}

Fig. 9.9: Definite locked expressions analysis of our example.

[3] For simplicity, we do not consider explicit locks, that can be acquired and released freely, across procedure calls. Our implementation deals with explicit locks as well.

Operational Semantics

Our analysis operates on a language similar to Java bytecode. It is used also in [103] and inspired by the standard informal semantics [72]. Bytecode is simpler than Java source code, since it has a narrower choice of instructions, and exposes fewer constructs (for example, inner classes are only expressible in Java code).

This section introduces a formal operational semantics of this target language, containing the instructions const v, dup, load, store, inc, ifeq, ifne, new, getfield, putfield, throw, call, monitorenter and monitorexit, each abstracting a set of Java bytecode instructions such as iconst_v, ldc, bipush, dup, iload, aload, istore, astore, iinc, ifeq, ifne, if_null, if_nonnull, new, getfield, putfield, athrow, invokevirtual, invokespecial, monitorenter and monitorexit. An instruction op abstracts arithmetic bytecode instructions such as iadd, isub, imul, idiv, irem, and an instruction catch starts the exception handlers. These instructions operate on *variables*, which encompass both stack elements and local variables. A standard algorithm [72] infers their static types.

The Java Virtual Machine supports exception handling. We therefore distinguish between *normal* and *exceptional* states. Exceptional states arise *immediately after* a bytecode throwing an exception and in that case there is only one element on the stack: a location bound to the thrown exception. catch starts the exception handlers from an exceptional state and is, therefore, undefined on a normal state. Any other instruction is defined only when the JVM is in a normal state.

Variables can be of primitive or reference types. We assume that the only primitive type is *int* and for reference types (*classes*) we consider only *instance fields* and *instance methods*. This is a simplification, in our implementation we handle all Java types and bytecodes, and static fields and methods.

The analysis takes as input a control flow graph (CFG), *i.e.*, a directed graph of *basic blocks*. Basic blocks contain a list of bytecode instructions, in which only the last can be a branch. The block containing the list $\{\mathsf{ins}, rest\}$, connected to m subsequent blocks $b_1, \ldots, b_m$, is denoted with the diagram $\boxed{\begin{array}{c}\mathsf{ins}\\ rest\end{array}}\begin{array}{l}\dashrightarrow b_1\\ \quad\cdots\\ \dashrightarrow b_m\end{array}$.

Example 9.1. Fig. 9.10 shows a Java method `increment` next to the corresponding CFG. There are branches at instructions monitorenter, getfield, putfield and monitorexit, since they might throw a `NullPointerException` which would be temporarily caught and then re-thrown to the caller of the method. Otherwise, the execution continues with the normal flow. Every bytecode instruction except return and throw always has one or more immediate successors. The latter are placed at the end of a method or constructor and typically have no successors.

An object-oriented program consists of a hierarchy of classes, each defining fields and methods.

```
public class Counter {
   private int c;

   public void increment() {
      synchronized (this) {
         c++;
      }
   }
    ...
}
```

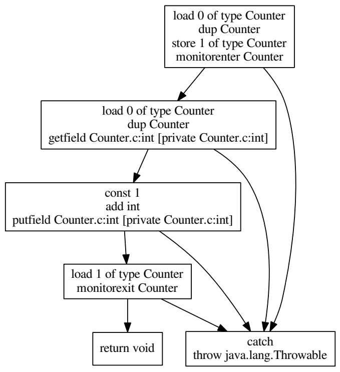


Fig. 9.10: Our running example, in Java and bytecode forms

Definition 9.2 (Classes). *We let* $\mathbb{K}$ *denote the set of classes and we define* $\mathbb{T} = \{int\} \cup \mathbb{K}$ *the set of all possible types. Every class* $\kappa \in \mathbb{K}$ *might have* instance fields $\kappa.f : t$ *(field* f *of type* $t \in \mathbb{T}$ *defined in class* κ*) and* instance methods $\kappa.m(\vec{t}) : t$ *(method* m *defined in class* κ*, with arguments of type* $\vec{t}$ *taken from* $\mathbb{T}$*, returning a value of type* $t \in \mathbb{T} \cup \{void\}$*), where* κ*,* $\vec{t}$ *and* t *are often omitted. We let* $\mathbb{F}(\kappa)$ *denote the set of all fields contained in* κ*. We define a partial order* $\preceq$*, that represents the subclass relation between classes. Two types are compatible if variables of a type can be assigned values of the other. Two primitive types* t_1, t_2 *are always compatible, since they are both int. Two classes* κ_1 *and* κ_2 *are compatible if either* $\kappa_1 \preceq \kappa_2$ *or* $\kappa_2 \preceq \kappa_1$*.*

Java bytecode is statically typed, hence a *type environment* provides static types for local and stack variables at a given program point.

Definition 9.3 (Type Environment). *Let* $\mathcal{V}$ *be the set of* variables *from* $\mathcal{L} = \{l_0, \ldots, l_{i-1}\}$ *(i* local *variables) and* $\mathcal{S} = \{s_0, \ldots, s_{j-1}\}$ *(j* stack *elements). A* type environment *is a function* $\tau : \mathcal{V} \to \mathbb{T}$*, and its domain is written as* $\text{dom}(\tau)$*. The set of all type environments is* $\mathcal{T}$*. For simplicity, we write* $\text{dom}(\tau) = \mathcal{L} \cup \mathcal{S}$*. Moreover, we let* $|\tau|$ *denote* $|\text{dom}(\tau)| = i + j$*.*

The state of the computation is represented as an environment mapping variables to locations and a memory binding those locations to objects.

Definition 9.4 (State). *A* value *is an element of* $\mathbb{V} = \mathbb{Z} \cup \mathbb{L} \cup \{null\}$*, where* $\mathbb{L}$ *is an infinite set of memory locations. A* state *over a type environment* τ *is* $\langle \rho, \mu \rangle$*, where* $\rho \in \text{dom}(\tau) \to \mathbb{V}$ *is called* environment *and assigns a value to each variable from* $\text{dom}(\tau)$*, while* $\mu \in \mathbb{M}$ *is called* memory *and binds locations to objects. Every object* o *has class* $o.\kappa$ *and an internal state* $o.\phi$ *mapping each field* f *of* o *into its value* $(o.\phi)(f)$*. It also has a lock counter* $o.locks$*, and an* *owner thread o.owner. The set of all states over τ is Σ_τ. We assume that states are well-typed,* i.e., *variables hold values consistent with their static types as expressed by τ and objects have exactly the fields required by their class and are well-typed themselves.*

Example 9.5. Let $\tau = [l_0 \mapsto \mathit{Counter}, l_1 \mapsto \mathit{Counter}, s_0 \mapsto \mathit{Counter}, s_1 \mapsto \mathit{int}] \in \mathcal{T}$. Fig. 9.11 shows the (normal) state $\sigma = \langle \rho, \mu \rangle \in \Sigma_\tau$ that might be current just after the execution of the getfield instruction in our example code in Fig. 9.10, when the c field of the object o_0 has value 3. There are two local variables l_0, l_1, and two stack variables s_0, s_1. The environment ρ maps these variables to their values: l_0, l_1, s_0 map to the same location ℓ_0, whereas s_1 maps to the integer value 3. The memory μ maps the location l_0 to the object o_0. The lock counter of the object o_0 has value 1, since it was incremented by the previous monitorenter instruction. The lock owner is the thread executing the method.

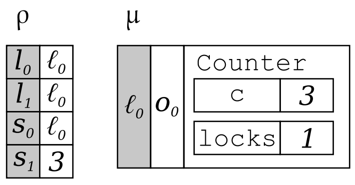


Fig. 9.11: A JVM state $\sigma = \langle \rho, \mu \rangle$

Bytecode instructions can be categorized as follows:

Basic Instructions. const v pushes an integer v on the stack. dup t duplicates the top of the stack, of type t. load k t pushes on the stack the value of local variable number k, l_k, which must exist and have type t. Conversely, store k t pops the top of the stack of type t and writes it in local variable l_k; it might potentially enlarge the set of local variables. In our formalization, conditional bytecodes are used in complementary pairs (such as ifne t and ifeq t), at the two branches of a conditional. For instance, ifeq t checks whether the top of the stack, of type t, is 0 when t $= \mathit{int}$ or `null` when t $\in \mathbb{K}$. Otherwise, its semantics is undefined. Bytecode inc kx increments the integer held in local variable l_k by a constant x. Bytecode op pops two integers from the operand stack, performs a suitable binary algebraic operation on them, and pushes the integer result back onto the stack. op may be add, sub, mul, div and rem, and the corresponding algebraic operations are $+, , \times, \div$ and %.

Object-Manipulating Instructions. These create or access objects in memory. new κ pushes on the stack a reference to a new object o of class κ, whose fields are initialized to a default value: `null` for reference fields, and 0 for integer fields [72]. getfield f reads field f of a receiver object r popped from the stack. putfield f writes the top of the stack inside field f of the object pointed to by the underlying value r.

Exception-Handling Instructions. throw κ throws the top of the stack, whose type κ is a subclass of `Throwable`. catch starts an exception handler: it takes an exceptional state and transforms it into a normal state at the beginning of the handler. After catch, an appropriate handler dependent on the run-time class of the exception is selected.

Method Call and Return. We use an activation stack of states. Methods can be redefined in object-oriented code, so a call instruction has the form call $m_1, \ldots, m_k$, enumerating an over-approximation of the set of its possible run-time targets. See [103] for details.

Synchronization Instructions. These control the execution of threads. monitorenter tries to acquire the lock on the object o referenced by the top stack variable. If $o.locks = 0$, the object has no owner. The thread executing monitorenter becomes then the owner of the object, and sets the lock counter to 1. If $o.locks > 0$ and the executing thread is the owner of o, the thread executing monitorenter increments $o.locks$. If $o.locks > 0$ and the executing thread is *not* the owner of o, the thread executing monitorenter blocks waiting for $o.locks$ to become 0, and then tries again to acquire the lock. monitorexit decrements the lock counter of the object o referenced by the top stack variable.

The semantics of an instruction ins of our target language is a partial map $ins : \Sigma_\tau \rightarrow \Sigma_{\tau'}$ from *initial* to *final* states. Number of local variables and stack elements at its start, as well as their static types, are specified by $\tau \in \mathcal{T}$. In the following we assume that $\mathrm{dom}(\tau)$ contains i local variables and j stack elements. Moreover, we suppose that the semantics is undefined for input states of wrong sizes or types, as is required in [72]. The formal semantics is given in [103].

Locked Expressions

In this section, we define the expressions that our analysis is able to identify as definitely locked (Definition 9.6), their *evaluation* (Definition 9.7), that might modify the content of some memory locations, and the notion of *locked expression* (Definition 9.9).

Definition 9.6 (Expressions). *Let $\mathcal{F}$ and $\mathcal{M}$ be the sets of the names of all possible fields and methods, respectively. Let τ be a type environment. The set of expressions over τ is $\mathbb{E}_\tau \ni E ::= n \mid v \mid E \oplus E \mid E.f \mid E.m(E, \ldots)$, where $n \in \mathbb{Z}$, $v \in \mathrm{dom}(\tau)$,$\oplus \in \{+, -, \times, \div, \%\}$, $f \in \mathcal{F}$ and $m \in \mathcal{M}$. We assume that expressions are well-typed, that is, variables and fields are used in accordance to their declared type. The function* vars $: \mathbb{E}_\tau \rightarrow \mathcal{P}(\mathrm{dom}(\tau))$ *returns the set of variables contained in an expression.*

Some of the expressions defined above represent the result of a method invocation. Their evaluation, in general, might modify the memory, so we must be aware of the side-effects of the methods appearing in these expressions. We define the evaluation of an expression E in a state $\langle\rho,\mu\rangle$ as a pair $\langle w,\mu'\rangle$, where w is the computed value of E, while μ' is the updated memory obtained from μ after the evaluation of E.

Definition 9.7 (Evaluation of expressions). *The evaluation of an expression $E \in \mathbb{E}_\tau$ in a state $\sigma = \langle\rho,\mu\rangle \in \Sigma_\tau$ is a partial map $[\![E]\!]^* : \Sigma_\tau \to \mathbb{V} \times \mathbb{M}$ defined as:*

E	$[\![E]\!]^*\sigma$	*defined only if*
$n \in \mathbb{Z}$	$\langle n,\mu\rangle$	
$v \in \mathrm{dom}(\tau)$	$\langle\rho(v),\mu\rangle$	
$[\![E_1 \oplus E_2]\!]^*\sigma$	$\langle w_1 \oplus w_2, \mu_2\rangle$	$[\![E_1]\!]^*\sigma = \langle w_1,\mu_1\rangle$, $[\![E_2]\!]^*\langle\rho,\mu_1\rangle = \langle w_2,\mu_2\rangle$, $w_1, w_2 \in \mathbb{Z}$
$[\![E.f]\!]^*\sigma$	$\langle(\mu_1(\ell).\phi)(f),\mu_1\rangle$	$[\![E]\!]^*\sigma = \langle\ell,\mu_1\rangle$, $\ell \in \mathbb{L}$, $f \in \mathbb{F}(\mu_1(\ell).\kappa)$
$[\![E_0.m(E_1,\ldots,E_\pi)]\!]^*\sigma$	$\langle w,\mu'\rangle$	$[\![E_0]\!]^*\sigma = \langle w_0,\mu_0\rangle$, $[\![E_{i+1}]\!]\langle\rho,\mu_i\rangle = \langle w_{i+1},\mu_{i+1}\rangle$, $w_0 \in \mathbb{L}$, $(\mu_\pi(w_0)).m(w_1,\ldots,w_\pi)$ *terminates with no exception, result* w *and final memory* μ'

We write $[\![E]\!]\sigma$ for the value of E, without the updated memory.

Definition 9.8 (Locked Object). *An object o is* locked *if $o.locks > 0$.*

Definition 9.9 (Locked Expression). *We say that $E \in \mathbb{E}_\tau$ is* locked *in $\sigma = \langle\rho,\mu\rangle \in \Sigma_\tau$ if and only if $\mu([\![E]\!]\sigma)$ is locked.*

Definite Locked Expressions Analysis

The concrete semantics works over concrete states, and our abstract interpretation abstracts them into sets of expressions.

Definition 9.10 (Concrete and Abstract Domain). *The concrete domain over $\tau \in \mathcal{T}$ is $\mathsf{C}_\tau = \langle\mathcal{P}(\Sigma_\tau),\subseteq\rangle$ and the abstract domain over τ is $\mathsf{A}_\tau = \langle\mathcal{P}(\mathbb{E}_\tau),\supseteq\rangle$.*

An abstract element $A \in \mathsf{A}_\tau$ represents those concrete states $\sigma = \langle\rho,\mu\rangle$ where the expressions in A evaluate to locked objects.

Definition 9.11 (Concretization Map). *Let $\tau \in \mathcal{T}$ and $A \in \mathsf{A}_\tau$. We define the concretization map $\gamma_\tau : \mathsf{A}_\tau \to \mathsf{C}_\tau$ as $\gamma_\tau(A) = \{\sigma = \langle\rho,\mu\rangle \in \Sigma_\tau \mid \forall E \in A.(\mu([\![E]\!]\sigma)$ is locked$)\}$.*

C_τ and A_τ are both complete lattices. For every subset $c \subseteq \mathcal{P}(\Sigma_\tau)$ the *infimum* is given by the intersection and the *supremum* by the union of the elements in c. For every $a \subseteq \mathcal{P}(\mathbb{E}_\tau)$ the infimum is the union and the supremum is the intersection of the elements in a.

Furthermore, γ_τ is coadditive:

$$\begin{aligned}
\gamma_\tau(\bigcap_{i\geq 0} A_i) &\overset{def}{=} \{\sigma = \langle\rho,\mu\rangle \in \Sigma_\tau \mid \forall E \in \bigcap_{i\geq 0} A_i.(\mu(\llbracket E\rrbracket\sigma) \text{ is locked})\}\\
&= \{\sigma = \langle\rho,\mu\rangle \in \Sigma_\tau \mid \forall E.(E \in \bigcap_{i\geq 0} A_i) \Rightarrow (\mu(\llbracket E\rrbracket\sigma) \text{ is locked})\}\\
&= \{\sigma = \langle\rho,\mu\rangle \in \Sigma_\tau \mid \forall E.(\bigwedge_{i\geq 0} E \in A_i) \Rightarrow (\mu(\llbracket E\rrbracket\sigma) \text{ is locked})\}\\
&= \{\sigma = \langle\rho,\mu\rangle \in \Sigma_\tau \mid \forall E. \bigwedge_{i\geq 0}(E \in A_i \Rightarrow (\mu(\llbracket E\rrbracket\sigma) \text{ is locked}))\}\\
&= \{\sigma = \langle\rho,\mu\rangle \in \Sigma_\tau \mid \bigwedge_{i\geq 0} \forall E.(E \in A_i \Rightarrow (\mu(\llbracket E\rrbracket\sigma) \text{ is locked}))\}\\
&= \{\sigma = \langle\rho,\mu\rangle \in \Sigma_\tau \mid \bigwedge_{i\geq 0}(\forall E \in A_i.(\mu(\llbracket E\rrbracket\sigma) \text{ is locked}))\}\\
&= \bigcap_{i\geq 0}\{\sigma = \langle\rho,\mu\rangle \in \Sigma_\tau \mid \forall E \in A_i.(\mu(\llbracket E\rrbracket\sigma) \text{ is locked})\}\\
&= \bigcap_{i\geq 0} \gamma_\tau(A_i)
\end{aligned}$$

Therefore, γ_τ is the concretization map of a Galois connection [39], and A_τ is actually an abstract domain, in the sense of abstract interpretation.

The Abstract Constraint Graph

Our analysis is constraint-based: we construct an *abstract constraint graph* from the program under analysis and then solve it. For each bytecode of the program there is a node containing the locked expressions at that point. Arcs of the graph propagate these approximations, reflecting, in abstract terms, the effects of the concrete semantics on the locking information. That is, the arc between the node for bytecode ins and that for a subsequent bytecode ins′ propagates the locking information at ins into that at ins′. The exact meaning of *propagates* depends here on ins, since each bytecode has a different abstract effect.

Definition 9.12 (ACG). *Let P be the program under analysis, already in the form of a CFG of basic blocks for each method or constructor (Section 9.2.3). The abstract constraint graph (ACG) for P is a directed graph where:*

- *there is a node* $\boxed{\mathsf{ins}}$ *for each bytecode* ins *in P, containing an approximation $A \in \mathsf{A}_\tau$, where τ is the type environment at the beginning of* ins*. This*

ins	A'
#1 const v, dup t, load k t, ifeq t, ifne t, op, new κ, getfield f, call $m_1 \dots m_k$, store k t, inc k t, new κ, throw κ, catch	$\{E \in A \mid \mathsf{vars}(E)$ do not contain modified locals or stack elements$\}$
#2 putfield f	$\{E \in A \mid s_{top}, s_{top-1} \notin \mathsf{vars}(E)$ and f does not occur in $E\}$
#3 monitorenter	$\{E \in A \mid s_{top} \notin \mathsf{vars}(E)\} \cup \{E \mid E$ is a definite alias of $s_{top}\}$
#4 monitorexit	$\{E \in A \mid$ the static type of E is not compatible with $\tau(s_{top})\}$
#5 call $m_1 \dots m_k$	$\{E[s_{j-\pi} \mapsto l_0, \dots, s_{j-1} \mapsto l_{\pi-1}] \mid E \in A$ and $\mathsf{vars}(E) \subseteq \{s_{j-\pi}, \dots, s_{j-1}\}\}$

Fig. 9.12: Propagation rules

abstract element represents a definite approximation of the actual locking information at ins*;*

- *each arc has a propagation rule,* i.e., *a function over A, from the locking information at its source to the locking information at its sink.*

Arcs are built as follows. We assume that τ is the static type information at the execution of a bytecode ins. *Moreover, we assume that τ contains j stack elements and consequently $top = j - 1$ is the height of the topmost stack element there.*

Sequential Arcs. If ins *is immediately followed by* ins′*, there is an arc from* $\boxed{\mathsf{ins}}$ *to* $\boxed{\mathsf{ins}'}$*, with propagation rule $\lambda A.A'$, where A' is defined by rules #1 - #4 in Fig. 9.12.*

Parameter Passing Arcs. For each call $m_1 \dots m_k$ *to a method with π parameters (including* this*) and each $1 \leq w \leq k$, there is an arc from* $\boxed{\mathsf{call}\ m_1 \dots m_k}$ *to the node for the first bytecode of m_w, with propagation rule $\lambda A.A'$, where A' is defined by rule #5 in Fig. 9.12.*

Example 9.13. In Fig. 9.13 it is shown the ACG of the method `increment` from Fig. 9.10. Node **a** belongs to the caller of this method and shows the arc related to the call. Arcs are decorated with the number of their associated propagation rules, with the exception of rule #1, which is the frequent default.

The **sequential arcs** link an instruction ins to its immediate successor ins′ propagating the set of locks held by the current thread. Only expressions unaffected by the instruction are propagated, *i.e.*, those not containing the modified variables, since we are not sure if they are definitely locked anymore. For putfield, only expressions not containing the target field or the two

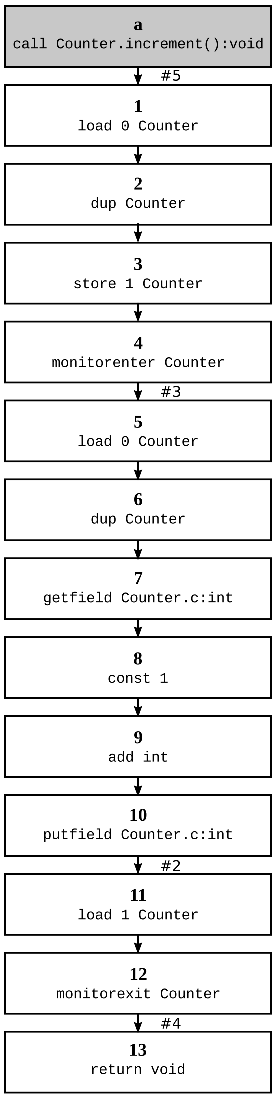


Fig. 9.13: The ACG of the method `increment` in Fig. 9.10

variables at the top of the stack are propagated. The rule for monitorenter adds all expressions aliased to the locked object, and propagates expressions not containing s_{top}. Conversely, that for monitorexit removes all expressions with a type compatible with that of the locked value. This is a conservative assumption, necessary for a sound definite approximation.

The **parameter passing arcs** link call $m_1 \ldots m_k$ nodes to the node representing the first bytecode of each callee method implementation m_i. They propagate the locked expressions that only refer to stack elements used as actual parameters, thus implementing an inter-procedural static analysis. These stack elements become formal arguments $l_0, \dots, l_{\pi-1}$ in the callee.

Definition 9.14 (Locked Expressions Analysis). *A solution of an ACG is an assignment of an abstract element A_n to each node n of the ACG such that $A_{first(\mathtt{main})} = \emptyset$ and the propagation rules of the arcs are satisfied,* i.e., *for every arc from node n to n' with propagation rule $\lambda A.\Pi(A)$, the condition $A_{n'} \supseteq \Pi(A_n)$ holds. The locked expression analysis of the program is the maximal solution of its ACG w.r.t. $\supseteq$.*

According to this definition, the abstract information assigned to the first statement of the program, $first(\mathtt{main})$, is the empty set, as at the beginning no expression is locked. Moreover, the set of definitely locked expressions at a generic node n is the intersection of all the sets resulting from the propagation rules of the incoming arcs in n. The maximal solution of the ACG is thus the greatest set satisfying the constraints, and its existence is guaranteed by the fact that the abstract domain A_τ is finite, since we fix upper bounds on the height of the locked expressions (e.g., a maximal number of field accesses and method invocations). The solution of the constraint can hence be computed by starting with the bottom approximation for every node: the set of all possible locked expressions; and then applying the propagation of the arcs and computing the intersection at each node entry, until stabilization.

Example 9.15. Fig. 9.14 shows the solution of the ACG from Fig. 9.13. The rule for the monitorenter node (4) adds aliases of `this` to the set of definitely locked expressions, while the rule for the monitorexit node (12) removes them.

n	A_n
a, 1, 2, 3	$\emptyset$
4, ..., 11	$\{l_0, l_1\}$
12, 13, 14	$\emptyset$
15, ..., 23, b	$\emptyset$

Fig. 9.14: The solution of the ACG from Fig. 9.13

Theorem 9.16 (Soundness). *Let an execution of a program lead to a state $\sigma \in \Sigma_\tau$ and $A^{\boldsymbol{ins}} \in \mathsf{A}_\tau$ be the approximation at the node* ***ins*** *corresponding to* ins*, computed by our static analysis. Then, $\sigma \in \gamma_\tau(A^{\boldsymbol{ins}})$.*

9.2.4 Implementation

The definite locked expressions analysis uses the result of other analyses to find locked expressions (Fig. 9.15):

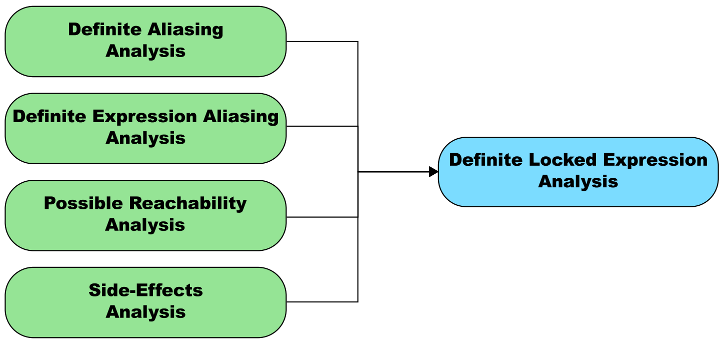

Definite Aliasing Analysis
Definite Expression Aliasing Analysis
Possible Reachability Analysis
Side-Effects Analysis
Definite Locked Expression Analysis

Fig. 9.15: Analyses supporting the definite locked expressions analysis

- a *definite aliasing analysis*, which finds variables that are aliases at a program point;
- a *definite expression aliasing analysis*, which finds expressions aliased to variables;
- a *possible reachability analysis*, which finds for each variable those variables that can be reached from it;
- a *side-effects analysis*, which finds for every alias if it can be affected by a given bytecode.

The alias analyses are used when building the propagation for monitorenter, in determining all expressions that are aliases of the object being locked, and parameter passing, in determining aliases of call parameters. They are described in [84]. Side-effects and reachability analyses are used when building propagation arcs for bytecodes writing to a field, or calling a method, to determine if members of the incoming set of expressions can be affected by the bytecode. Reachability is also used to refine the analysis when building arcs for monitorexit, to keep in the set of the definitely locked expressions only those aliases that are not reachable from, or do not reach, the object being unlocked. The possible reachability analysis is described in [85].

The definite locked expressions analysis is used by the Julia static analyzer in its GuardedBy checker, which checks and infers `@GuardedBy` and `@Holding` annotations. `@GuardedBy` is applied to variables, and documents that the annotated object is accessed only when the given expression is locked. `@Holding` has the same meaning, but is applied to methods. For example, in the code in Fig. 9.10, the field `c` is `@GuardedBy(this)`, since it is accessed only in a `synchronized(this)` block.

Experiments

We ran the definite locked expressions analysis on real-world programs. Fig. 9.16 lists our benchmark programs, along with some statistics. BitcoinJ [4] is a java library for working with bitcoins, used in many Bitcoin projects. Eclipse ECJ is the compiler for Java used in the Eclipse IDE. Guava [60] is a library providing various utilities to Java developers. Jetty Server is a web server, part of the Jetty project [46]. Velocity [20] is a template engine library. Apache Zookeeper [21] is a server providing services to distributed applications. The other programs are part of the Apache Tomcat project [19]. All experiments were executed on a Linux machine with an Intel Core i7 4770 CPU and 8 gigabytes of RAM.

Project	Precision	LoC	Time	JAR file name
BitcoinJ	19% (19550/102775)	103598	2633	`bitcoinj-core-0.12.2.jar`
Eclipse ECJ	0.5% (1514/316609)	162423	10376	`ecj-4.4.jar`
Guava 18.0	7% (8362/116501)	119299	3187	`guava-18.0.jar`
Jetty Server	19% (7173/37533)	60160	1369	`jetty-server-9.2.6.v20141205.jar`
Velocity	5% (2526/43377)	55355	1380	`velocity-1.7.jar`
Zookeeper	15% (10607/69165)	73982	1810	`zookeeper-3.4.6.jar`
Catalina	18% (26875/144641)	123450	5924	`tomcat-catalina-8.0.15.jar`
Coyote	11% (7264/63134)	72194	2184	`tomcat-coyote-8.0.15.jar`
Dbcp	18% (4816/26386)	53811	1354	`tomcat-dbcp-8.0.15.jar`
Jasper	3% (2080/58772)	68166	2343	`tomcat-jasper-8.0.15.jar`
Jni	35% (1777/4996)	33804	640	`tomcat-jni-8.0.15.jar`
Util	8% (962/10814)	43086	1329	`tomcat-util-8.0.15.jar`
Websocket	7% (1041/13776)	39244	715	`tomcat-websocket-8.0.15.jar`

Fig. 9.16: Benchmark programs for our analysis. *Precision* is the ratio of the total number of definitely locked expressions over the total number of *watchpoints*, *i.e.*, interesting bytecodes. *LoC* is the approximate number of lines of code reached by during the analysis. It is the count of the entries in the line number table of each class included in the analysis, plus 3 for each method or constructor. *Time* is the execution time of the analysis, in milliseconds.

9.2.5 Inference of the Locking Discipline

Once the three previous supporting analyses have been performed, Julia infers `@GuardedBy(`E`)` annotations for fields and method parameters (fig. 9.6). This amounts to finding expressions E such that the non-`final` fields of all possible values ever held in those fields or parameters are only accessed at a program point where E is locked by the current thread. Julia uses creation points as a conservative approximation of the identity of run-time values. Objects created at distinct creation points must be distinct, while the converse might not hold. Namely, it uses the following algorithm to infer the `@GuardedBy` annotations for a field or parameter x:

1. it uses the creation points analysis to determine an overapproximation C of the creation points of the values ever held in x;
2. it computes the set of program points where a field of an object created at C might be accessed, that is, $A = \{p \mid$ a non-`final` field f is accessed at p as $E_p.f$ and the set $C^p_{E_p}$ of all possible creation points of E_p at p is such that $C^p_{E_p} \cap C \neq \emptyset\}$;
3. for each $p \in A$, it computes a set of expressions that are definitely locked there, using `itself` as a shorthand for the expression itself:
   $L_p = \{E[E_p \mapsto \texttt{itself}] \mid E$ is a definite alias of E_p at p
   and E is definitely locked at $p\}$;
4. it computes $L = \bigcap_{p \in A} L_p$;
5. it infers the annotations `@GuardedBy("E")` for each $E \in L$ where no variable occurs, but for `itself`.

Consider for instance field `left` in fig. 9.2. According to the creation point analysis (fig. 9.7), we have $C = \{\texttt{Fork@80}\}$. Access to non-`final` fields occur as `this.usedBy` at lines 8, 12, 20 and we have $C^8_{\texttt{this}} = C^{12}_{\texttt{this}} = C^{20}_{\texttt{this}} = \{\texttt{Fork@80}\}$ (fig. 9.7). Hence $A = \{8, 12, 20\}$. At those program points, `this` is obviously a definite alias of itself (fig. 9.8). According to fig. 9.9, the expression `this` is always locked at 8, 12 and 20. Then $L_p = \{\texttt{this}[\texttt{this} \mapsto \texttt{itself}]\} = \{\texttt{itself}\}$ for each $p \in A$, and hence $L = \{\texttt{itself}\}$. Therefore, Julia infers the annotation `@GuardedBy("itself")` for field `left`.

9.2.6 Calls to Library Methods

The algorithm sketched in section 9.2.5, at its step 2, requires to check all program points A where a non-`final` field in accessed. This includes the program points inside the libraries as well. Hence the inference of `@GuardedBy("itself")` for field `left` above should be corrected by considering in A also the program points outside the application shown in figs. 9.1 and 9.2 and the driver program. However, a simplifying and computationally effective alternative solution is to consider only program points A inside the application under analysis, as long as we also include in A the program points where a value is passed to the libraries. That is, point 2 of the algorithm from section 9.2.5 can be modified to

2. it computes the set of program points $A = \{p$ in the application $\mid$ a non-`final` field f is accessed at p as $E_p.f$ *or an expression E_p is passed as an argument to libraries* and the set $C^p_{E_p}$ of all possible creation points of E_p at p is such that $C^p_{E_p} \cap C \neq \emptyset\}$;

By applying this inference algorithm, to figs. 9.1 and 9.2 and the driver program, Julia infers the `@GuardedBy` annotations in figs. 9.1 and 9.2.

9.3 Experiments

We performed experiments to understand how programmers currently use `@GuardedBy` and to evaluate the utility of our semantics.

9.3.1 Subject Programs and Methodology

We chose 15 open-source subject programs that use locking (fig. 9.17). The programmers had partially documented the locking discipline in 5 of them. We counted not only `@GuardedBy` and `@Holding` annotations but also commented annotations and English comments containing the string “guard”. The programmers sometimes used comments to document a locking discipline without adding a compile-time and run-time dependency on the `@GuardedBy` annotation. However, the documented locking discipline may be incorrect because it was not checked by any tool.

We determined a goal set of correct annotations, *i.e.*, those whose locking discipline the program obeys. To determine this set, we manually analyzed every annotation written by the programmer or inferred by Julia.[4] We retained every annotation from either set such that the program is guaranteed not to suffer a data race on the annotated program element. (We did not observe any data races that appeared to be intentional.) Then, we compared the goal annotations to both the programmer-written and the inferred ones. This comparison was not syntactical: annotations that are conceptually the same or are expressing the same thing are considered equal.

As is standard for an information retrieval problem [95], we report results in terms of precision (number of correct reported annotations divided by total number of reported annotations) and recall (number of correct reported annotations divided by total number of goal annotations). Precision and recall are measurements between 0% and 100% inclusive, and larger numbers are better.

We used Julia to infer the locking discipline in terms of `@GuardedBy` and `@Holding` with value-protection semantics.[5] Experimental results for `@GuardedBy` annotations appear in fig. 9.18, and results for `@Holding` appear in fig. 9.19. Programmers made significant numbers of mistakes (as shown by low precision) and omitted significant numbers of annotations (as shown by low recall).

Programmer mistakes. In every program where programmers documented a locking discipline, they wrote incorrect annotations that express a locking discipline that the code does not satisfy. For example, Guava’s `LocalCache` and `MapMakerInternalMap` classes incorrectly use `Segment.this` as a guard

[4] There might exist other correct annotations that neither Julia, the original programmer, nor we are aware of.

[5] Julia has two modes and can also infer annotations for name protection, but here we focus on value protection.

			Programmer-written		Inference
Project	Version	LoC	`@GuardedBy`	`@Holding`	time
BitcoinJ	0.12.2	102458	46	14	238
Daikon	5.2.24	169710	0	0	1596
Derby Engine	10.11.1.1	119594	12	9	4077
Eclipse ECJ	4.4	161701	0	0	924
Guava	18.0	118190	64	72	621
Jetty Server	9.2.6.v20141205	59611	0	0	109
Velocity	1.7	54549	0	0	94
Zookeeper	3.4.6	75475	0	0	118
Catalina	8.0.15	121959	0	0	472
Coyote	8.0.15	71527	1	0	110
Dbcp	8.0.15	53181	16	0	84
Jasper	8.0.15	67380	0	0	105
Jni	8.0.15	32682	0	0	49
Util	8.0.15	42115	0	0	58
Websocket	8.0.15	39928	0	0	75

Fig. 9.17: Subject programs. The last 7 are part of Tomcat. *LoC* is the approximate number of lines of code reached by Julia during the analysis. It is the count of the entries in the line number table of each class analyzed, plus 3 for each method or constructor. *Inference time* is measured in seconds.

	Goal	Programmer-written					Inference		
			name		value			value	
Project	#	#	P%	R%	P%	R%	#	P%	R%
BitcoinJ	47	46	87	85	30	30	7	100	15
Daikon	5	0	-	0	-	0	1	100	20
Derby Engine	16	12	83	63	58	44	6	100	38
Eclipse ECJ	6	0	-	0	-	0	6	100	100
Guava	22	64	19	55	14	41	5	100	23
Jetty Server	1	0	-	0	-	0	1	100	100
Velocity	4	0	-	0	-	0	4	100	100
Zookeeper	5	0	-	0	-	0	5	100	100
Catalina	2	0	-	0	-	0	2	100	100
Coyote	24	1	100	4	0	0	23	100	100
Dbcp	20	16	88	70	56	45	6	100	30
Jasper	7	0	-	0	-	0	7	100	100
Jni	1	0	-	0	-	0	1	100	100
Util	4	0	-	0	-	0	4	100	100
Websocket	9	0	-	0	-	0	9	100	100

Fig. 9.18: Experimental results for `@GuardedBy` annotations. The table lists the number of annotations written by the programmer and inferred by Julia. *Goal* is the number of goal annotations. The precision (R%) and recall (R%) are given separately when annotations are interpreted according to the name-protection or value-protection semantics. Computations whose denominator is zero are reported as "-".

			Programmer-written					Abstract interp.			
Project	Goal	OGoal	#	Corr	P%	R%	OR%	#	Corr	P%	R%
BitcoinJ	113	45	14	14	100	12	31	113	113	100	100
Daikon	3	0	0	0	-	0	-	3	3	100	100
Derby Engine	121	13	9	7	78	6	54	120	120	100	99
Eclipse ECJ	1	0	0	0	-	0	-	1	1	100	100
Guava	126	45	72	38	53	30	84	110	110	100	87
Jetty Server	4	0	0	0	-	0	-	4	4	100	100
Velocity	20	0	0	0	-	0	-	20	20	100	100
Zookeeper	16	0	0	0	-	0	-	16	16	100	100
Catalina	98	0	0	0	-	0	-	98	98	100	100
Coyote	13	0	0	0	-	0	-	13	13	100	100
Dbcp	18	0	0	0	-	0	-	18	18	100	100
Jasper	2	0	0	0	-	0	-	2	2	100	100
Jni	1	0	0	0	-	0	-	1	1	100	100
Util	4	0	0	0	-	0	-	4	4	100	100
Websocket	4	0	0	0	-	0	-	4	4	100	100

Fig. 9.19: Experimental results for `@Holding` annotations. Numbers are as in fig. 9.18, but `@Holding` means the same thing in both the name- and value-protection semantics. The number of correct annotations (Corr) is given together with the precision and recall. *OGoal* (for "omission-tolerant goal") is the number of goal annotations whose guard expression the programmer used elsewhere, and *OR%* is the programmer recall based on the omission-tolerant goal set.

expression. Julia infers the correct guard `this`. In other cases, a lock is acquired only at write accesses but not at read accesses to a variable. This can lead to corrupted data reads for data larger than 32 bits (*i.e.*, `long` and `double` values, that on some machines are accessed in two steps). For 32-bit data, it can lead to inconsistent multiple reads of a variable because the Java memory model permits delayed publication. An example is in the Guava class `SerializingExecutor`: the field `private boolean isThreadScheduled` is annotated as `@GuardedBy("internalLock")`, but it is read without protection at line 135, despite being always written after acquiring the lock.

The most common programmer mistake, however, was creating external aliases to a value. If a reference to a variable's value leaks, then a data race can occur even if a lock is held whenever the variable is read or written. In other words, in the presence of aliasing the value-protection semantics provides no guarantee. This is a natural problem, given the lack of automated checking and even the lack of a mention of the danger of aliasing in references such as JCIP [59], where only instance confinement is mentioned. An example is BitcoinJ field `PaymentChannelClient.conn`. It is always accessed holding a lock inside the class, but the field is initialized with a parameter of a `public` constructor. So there exists an external alias to the object that can potentially be used to access the object without protection.

Programmer omissions. The private BitcoinJ method PaymentChannel-Server.truncateTimeWindow(long) is inferred to be @Holding("lock"), and is indeed called always with lock held. Nevertheless, the programmer didn't write the annotation.

In Apache Velocity, a template engine, Julia finds four objects that are @GuardedBy("itself"): the field XPATH_CACHE, in XPathCache, is accessed in a synchronized(XPATH_CACHE) block; the field SimplePool pool, in ParserPoolImpl, uses methods put and get of SimplePool, that modify the object's state inside a synchronized(this) block; the receivers of the same two methods are thus guarded as well.

Julia mistakes. Julia's output was correct: its precision is 100%, just as for any sound tool that infers definite information.

Julia omissions. There are two reasons that Julia fails to infer a correct programmer-written locking discipline: either (1) the program's correctness is too subtle for Julia to reason about, or (2) the locking discipline is inexpressible in the value-protection semantics.

(1) Julia incompleteness: Julia missed 1 @Holding in Derby Engine and 16 in Guava because methods in the Monitor, AbstractService, and ServiceManager classes use complex reasoning, ensuring for instance that a call to a method happens only in flows of execution where the lock is held by the executing thread. At the moment Julia does not understand these tricks.

Julia only allows itself and final fields in a guard expression. This is sufficient but not necessary to ensure that the guard expression evaluates to the same value throughout the scope of the guard (section 9.1.4). Programmers usually use variables in guard expressions (sometimes correctly, sometimes incorrectly). As future work, we plan to support the container this in guard expressions, which still protects against data races if it is never aliased.

(2) Value-protection semantics inflexibility: Only one example seems a genuine value-protection programmer-written annotation that is not inferred by Julia. The static field in Dbcp

```
private static Timer _timer; //@GuardedBy("EvictionTimer.class")
```

is always accessed in synchronized static methods, it never escapes, and is assigned with _timer = AccessController.doPrivileged(new Privileged-NewEvictionTimer()). The doPrivileged method is native, and executes the run method of the PrivilegedNewEvictionTimer class, that simply returns a new Timer object. The guard refers to the class object and is permitted under the value-protection semantics.

Omission-tolerant Results

We computed two sets of recall numbers for programmer-written @Holding annotations (fig. 9.19). First, we determined the overall recall, based on the full set of goal annotations. Second, we determined the recall based on a reduced set of goal annotations. The reduced, or omission-tolerant, set contains only `@Holding` annotations whose guard expressions appear in `@GuardedBy` annotations that the programmer wrote. This latter metric considers only locks that the programmer deemed significant enough to document.

The rationale for reporting two different measurements is that there are two different reasons that a `@Holding` annotation might be missing from the programmer-written set:

- The programmer wrote `@GuardedBy` on some variable v but omitted `@Holding(`v`)`. This incomplete specification of the locking discipline for v is a programmer error. For example, the programmer correctly annotated the unary method `Wallet.maybeUpgradeToHD` as `@Holding("keychainLock")` in BitcoinJ, but didn't annotate the no-argument overloaded version.
- The programmer omitted `@GuardedBy` on some variable v and also omitted `@Holding(`v`)`. It is conceivable that the programmer only intended to write specifications for some guarded variables and intentionally omitted the `@GuardedBy` annotation on other variables. The OR% measurement assumes every such omission was intentional, even though the practice is undesirable because someone calling or modifying the code could misuse it. For example, Julia infers `@Holding("enumConstantCache")` for Guava's private method `Enums.populateCache`, which needs it for a call to `put`. Indeed, the only invocation of `populateCache` is in a `synchronized (enumConstantCache)` block. Nevertheless, the programmer did not annotate it as `@GuardedBy("enumConstantCache")`.

10 Implementation

In this chapter we briefly present the implementation of the taintedness analysis and the locking discipline inference in the Julia program analyzer. The taintedness analysis was implemented in the Injection checker, and the locking discipline inference in the GuardedBy checker.

10.1 The Julia Analyzer

Julia is a static analyzer of Java bytecode programs. It uses abstract interpretation to derive analyses on properties defined from a denotational semantics of Java bytecode, as described in Section 8.2. It also performs constraint-based analyses. These analyses are implemented in *pluggable checkers*, so that the analyzer can be easily extended.

10.2 Injection

The taintedness analysis is implemented by the `InjectionChecker` class. In its entry method it configures and starts a `FlowsChecker`, a more general checker that tracks flows of information from arbitrary sources to arbitrary sinks. The flow checker is configured for sources of information of type `UNTRUSTED`. This in turn execute the `FlowAnalyser`, that performs the analysis for each bytecode, before combining the results. Depending on the type of bytecode, the analysis proceeds as follows:

CALL For `CALL`, the environment of the called method must be plugged in that of the caller, as described by the *extend* operator (Definition 8.23). At line 11, a method (listed later) is called that renames variables and removes temporary ones. The for loop at line 23 implements $\bigwedge_{0\leq k<b} A_{b,M}(s_k)$; the *then* branch of the `if` statement, taken when $SA_{b,M,v}$ is empty, creates $\check{v} \leftrightarrow \hat{v}$, while the `else` creates $((\check{v} \vee (\bigvee_{w\in SA_{b,M,v}} \overline{w})) \leftarrow \hat{v})$. The same does the for loop at line 58, for local variables.

```
protected final FlowDomainElement analyse(CALL call,
      FlowDomainElement denotation) {
   return (call.getNumberOfDynamicTargets() == 0) ?
      worstCase(call) :
      domain.mk(analyseCALLAux(call, denotation));
}

protected BDD analyseCALLAux(CALL call,
      FlowDomainElement denotation) {

   //...

   bdd = domain.plugIntoTheCallingContext(call,
         denotation.getBDD());

   CodeImplementation target =
         call.getDynamicTargets().iterator().next();
   BDD stack = domain.inputIsNotExceptional();
   boolean modifiesOnlyFieldsOfParameters =
         call.modifiesOnlyFieldsOfParameters();
   boolean sideEffectsFree = call. isSideEffectsFree ();
   boolean isConstructorCall = call.getStaticTarget()
         instanceof ConstructorReference;
   boolean doesNotIncreaseSecrecy =
         doesNotIncreaseSecrecy(call);

   DefiniteAliasingInformation alias = aliasing.get().at( call );
   int rec = call.getReceiverStackPosition();

   // we consider all variables that survive the call
   for (TypeCursor cursor: call.getStack().upToSlot(rec))
      if ((sideEffectsFree && (!isConstructorCall ||
            ! alias .stackStack(cursor. getSlotPosition (), rec)))
             || reachability .get() .stackIsImmutableWrt(
                   cursor, call ) || doesNotIncreaseSecrecy ||
                      (! isConstructorCall &&
                      secrecyDoesIncreaseBySideEffect(
                            cursor.getType())))
         stack.andWith(domain.iff(
               domain.codeStackAtTheEndOf(cursor, call),
                  domain.codeStackAtTheBeginningOf(
                        cursor, call)));
      else {
         TypeListIterable pars =
               modifiesOnlyFieldsOfParameters ?
```

```
                reachability .get()
                        .sideEffectedParametersReachedFromStack(
                                cursor,  call) :
                sharing.get()
                        .sideEffectedParametersSharingWithStack(
                                cursor,  call);

            List<Integer> or = new ArrayList<Integer>();
            or.add(domain.codeStackAtTheBeginningOf(cursor,
                  call));

            for (TypeCursor parsCursor: pars)
                if ((isConstructorCall &&
                      parsCursor.getSlotPosition() == 0) ||
                      reachability.get().sideEffected(call)
                              [parsCursor.getSlotPosition()])
                    or.add(domain
                        .codeTempLocalAtTheEndOfMethodOrConstructor(
                                parsCursor, target));

            stack.andWith(domain.onlyIf(
            domain.codeStackAtTheEndOf(cursor, call),
                  intoArray(or)));
        }

    bdd.andWith(stack);

    BDD locals = domain.getFactory().one();

    for (TypeCursor cursor: call.getLocals())
        if ((sideEffectsFree && (!isConstructorCall ||
              ! alias.localStack(cursor.getSlotPosition(), rec))) ||
              reachability .get().localIsImmutableWrt(cursor, call)
              || doesNotIncreaseSecrecy || (!isConstructorCall &&
             secrecyDoesIncreaseBySideEffect(cursor.getType())))
            locals .andWith(domain.iff(
                domain.codeLocalAtTheEndOf(cursor, call),
                      domain.codeLocalAtTheBeginningOf(cursor,
                      call)));
        else {
            TypeListIterable pars =
                  modifiesOnlyFieldsOfParameters ?
                reachability .get()
                    .sideEffectedParametersReachedFromLocal(
                                cursor,  call) :
```

```
                    sharing.get()
                        .sideEffectedParametersSharingWithLocal(
                                cursor, call);

                List<Integer> or = new ArrayList<Integer>();
                or.add(domain.codeLocalAtTheBeginningOf(cursor,
                     call));

                for (TypeCursor parsCursor: pars)
                    if ((isConstructorCall &&
                         parsCursor.getSlotPosition() == 0) ||
                         reachability.get()
                    .sideEffected(call)[parsCursor.getSlotPosition()])
                        or.add(domain.codeTempLocalAtTheEndOfMethodOrConstructor(
                            parsCursor, target));

                locals.andWith(domain.onlyIf(domain.codeLocalAtTheEndOf(
                    cursor, call), intoArray(or)));
            }

        // if it becomes too complex, we give up
        if (bdd.nodeCount() * locals.nodeCount() > 100000)
            return worstCase(call).getBDD().id();

        bdd.andWith(locals);

        // we remove the temporary variables
        BDD temp = bdd;
        bdd = temp.exist(domain.getTemps());
        temp.free();


        // we assume that library methods never throw tainted
             exceptions
        if (call.getStaticTarget().getReferencedClass()
                .isSystemAPI()) {
            BDD temp = bdd;
            bdd = temp.exist(domain.outputException());
            bdd.andWith(domain.outputIsExceptional().impWith(
                domain.nOutputException()));
            temp.free();
        }

        return bdd;
    }
```

The method `analyseCALLAux` calls `plugIntoTheCallingContext` of the class `VarBDDAbstractDomain` to rename variables $\hat{s}_0$ into $\hat{s}_b$, $\check{l}_k$ into $\check{s}_{k+b}$, $\hat{l}_k$ into $\bar{l}_k$, and quantify over temporary variables $\bar{l}_k$:

```
public final BDD plugIntoTheCallingContext(CALL call, BDD
      bdd) {
    BDDPairing renaming = getFactory().makePair();
    int rec = call.getReceiverStackPosition();

    BDD remove = null;

    if (rec > 0 &&
          !call.getStaticTarget().getReturnType().isVOID())
        if (isTrackedAndRepresentable(codeStackAtTheEndOf(rec,
              call))) {
            int into = codeStackAtTheEndOf(rec, call);
            if (into != codeReturnValue())
                renaming.set(codeReturnValue(), into);
        }
        else
            remove = returnValue();

    if (call.getNumberOfDynamicTargets() > 0) {
        CodeImplementation target =
              call.getDynamicTargets().iterator().next();
        TypeListIterable formals = call.requiredStackTypes();
        Bytecode start = call.getDynamicTargets().iterator().next()
              .getInitialBytecode();
        BDDFactory factory = getFactory();

        for (TypeCursor cursor: formals) {
            int inputLocal = codeLocalAtTheBeginningOf(cursor,
                  start);

            if (isTrackedAndRepresentable(inputLocal)) {
                int inputStack = codeStackAtTheBeginningOf(rec +
                      cursor.getSlotPosition(), call);

                if (isTrackedAndRepresentable(inputStack))
                    renaming.set(inputLocal, inputStack);
                else
                    if (remove == null)
                        remove = factory.ithVar(inputLocal);
                    else
                        remove.andWith(factory.ithVar(inputLocal));
            }

            int fromCode =
                codeLocalAtTheEndOfMethodOrConstructor(cursor,
                target);
            if (isTrackedAndRepresentable(fromCode))
              renaming.set(fromCode, outputToTemp(fromCode));
          }
        }

        if (remove != null) {
          // we remove the renamed variables that we could not
              represent
          bdd = bdd.exist(remove).replaceWith(renaming);
          remove.free();

          return bdd;
        }
        else
          return bdd.replace(renaming);
      }
```

Other bytecodes For other bytecodes the abstraction is a boolean function as described in Figure 8.2. For example the function for the `LOAD` bytecode is $(load\ k\ t)^{\mathbb{T}} = U \wedge \neg\check{e} \wedge \neg\hat{e} \wedge (\check{l}_k \leftrightarrow \hat{s}_j)$ and the corresponding implementation is:

```
        protected FlowDomainElement analyseLOAD(LOAD load) {
          // no local variable and no stack element is modified − U
              function
          BDD bdd = localsAndStackDoNotChange(load, 0);

          // the new top of the stack behaves exactly as local
              variable varNum
          return domain.mk(bdd.andWith(domain.iff
            (domain.codeStackAtTheEndOf(load.stackHeight(), load),
            domain.codeLocalAtTheBeginningOf(load.getVarNum(),
                load))));
        }
```

In `PUTFIELD` we must consider reachability:
$(putfield\ \kappa.f\!:\!t)^{\mathbb{T}} = \wedge_{v \in L} R_j(v) \wedge (\neg\hat{e} \rightarrow \wedge_{v \in S} R_j(v)) \wedge (\hat{e} \rightarrow \neg\hat{s}_0) \wedge \neg\check{e}$
and indeed the implementation uses previously computed reachability information:

```
        protected FlowDomainElement analysePUTFIELD(PUTFIELD
            putfield) {
```

```
        int rec = putfield.getDereferenced();
        BDD bdd = domain.inputIsNotExceptional();
        if (!source.mightHoldForInternalExceptions())
            bdd.andWith(domain.nOutputException());

        int codeValue = domain.codeStackAtTheBeginningOf(rec +
              1, putfield);
        PossibleReachabilityInformation reach =
              reachability.get().at(putfield);

        // if we write a possibly secret value, everything that
              might reach the receiver
        // becomes possibly secret
        for (TypeCursor cursor: putfield.getStack().upToSlot(rec))
            if (reach.stackStack(cursor.getSlotPosition(), rec) &&
            !reachability.get().stackIsImmutableWrt(cursor, putfield))
                bdd.andWith(domain.onlyIf(
                        domain.codeStackAtTheEndOf(cursor, putfield),
                    domain.codeStackAtTheBeginningOf(cursor,
                          putfield), codeValue));
            else
                bdd.andWith(domain.iff(
                    domain.codeStackAtTheEndOf(cursor, putfield),
                    domain.codeStackAtTheBeginningOf(
                            cursor, putfield)));

        for (TypeCursor cursor: putfield.getLocals())
            if (reach.localStack(cursor.getSlotPosition(), rec) &&
            !reachability.get().localIsImmutableWrt(cursor, putfield))
                bdd.andWith(domain.onlyIf(
                    domain.codeLocalAtTheEndOf(cursor, putfield),
                    domain.codeLocalAtTheBeginningOf(cursor,
                          putfield), codeValue));
            else
                bdd.andWith(domain.iff(
                    domain.codeLocalAtTheEndOf(cursor, putfield),
                    domain.codeLocalAtTheBeginningOf(cursor,
                          putfield)));

        return domain.mk(bdd);
    }
```

10.3 GuardedBy

The `GuardedByChecker` uses the result of the `DefiniteLockedExpAnalyser`, that is a `ConstraintAnalyser`. As explained in Section 9.2.3, it builds a constraint graph and then solves it. As examples, we consider below the arc construction for the `MONITORENTER` and `MONITOREXIT` bytecodes, the interesting ones.

When entering a synchronized block, all definite aliases of the object being locked are inserted in the set of definitely locked expressions:

```
private boolean buildArcsForMONITORENTER(Bytecode first,
      Bytecode second) {
   if ( first instanceof MONITORENTER) {
      // all aliases of the receiver becomes locked
      MONITORENTER monitorenter = (MONITORENTER) first;
      int rec = monitorenter.getDereferenced();
      BitSet<Alias> added = getConstraint().createEmptySet();
      for (Alias alias : expAliasing.get().at(monitorenter).stack(rec))
         added.add(alias);

      DefiniteAliasingInformation alias =
            aliasing.get().at(monitorenter);
      for (TypeCursor local: monitorenter.getLocals())
         if ( alias.localStack( local.getSlotPosition(), rec))
            added.add(Local.mk(local.getSlotPosition(),
                  local.getType(), getProgram()));

      getConstraint().arc(new UnionArc<Alias>(position(first),
            getConstraint().constant(added), position(second)));

      return true;
   }
   else
      return false;
}
```

When exiting a synchronized block, all expressions having type compatible with the locked object are removed from the set:

```
private boolean buildArcsForMONITOREXIT(Bytecode first,
      Bytecode second) {
   if ( first instanceof MONITOREXIT) {
      final MONITOREXIT monitorexit = (MONITOREXIT) first;

      getConstraint().arc(new Arc<Alias>(position(first),
            position(second)) {

```

```
    @Override
    protected BitSet<Alias> pass(BitSet<Alias> set) {
        BitSet<Alias> result = getConstraint().createEmptySet();
        ReferenceType unlockedType = monitorexit.getType();
        int rec = monitorexit.getDereferenced();
        PossibleReachabilityInformation reach =
              reachability.get().at(monitorexit);

        // we determine which aliases keep being definitely locked
              after this instruction
        for (Alias alias: set) {
            Type aliasType = alias.getType();
            if (!aliasType.possiblySubtypeOf(unlockedType)
                    && !unlockedType.possiblySubtypeOf(aliasType))
                // the static type information is enough to conclude
                      that the alias is not unlocked
                result.add(alias);
            else if (alias instanceof Local) {
                int l = ((Local) alias).getLocal();
                if (!reach.stackLocal(rec, l) ||
                      !reach.localStack(l, rec))
                    result.add(alias);
            }
            else if (alias instanceof StaticFieldAlias) {
                if (!reach.stackStatic(rec) ||
                      !reach.staticStack(rec))
                    result.add(alias);
            }
            else if (alias instanceof FieldOfLocal) {
                int l = ((FieldOfLocal) alias).getLocal();
                if (!reach.localStack(l, rec))
                    result.add(alias);
            }
            else if (alias instanceof FieldOf) {
                int l = startingLocal((FieldOf) alias);
                if (l >= 0 && !reach.localStack(l, rec))
                    result.add(alias);
            }
            else if (alias instanceof ArrayLocalAlias) {
                int l = ((ArrayLocalAlias) alias).getArray();
                if (!reach.localStack(l, rec))
                    result.add(alias);
            }
            else if (alias instanceof ArrayLocalLocal) {
                int l = ((ArrayLocalLocal) alias).getArray();
```

```
                    if (!reach.localStack(l, rec))
                        result.add(alias);
                }
            }

            return result;
        }

        /**
         * Goes back to the starting local of a chain of field
              dereferences, if any.
         *
         * @param alias the chain of field dereferences
         * @return the local, or -1 if no such local exists
         */

        private int startingLocal(FieldOf alias) {
            Alias base = alias.getBase();
            if (base instanceof FieldOfLocal)
                return ((FieldOfLocal) base).getLocal();
            else if (base instanceof FieldOf)
                return startingLocal((FieldOf) base);
            else
                return -1;
        }
    });

    return true;
  }
  else
    return false;
}
```

11 Conclusion

We have developed a new multi-threaded BDD library in Java. Several threads can share the same unique table of nodes and caches. This reduces the memory footprint and avoids repeating computations in different threads. Moreover, we implemented distinctive features like a representation for Boolean functions that keeps equivalent variables separate from the BDD, and a factory that checks the integrity of the node table, either on user's demand or on every access. Its use in our program analyzer Julia led to the reduction of the memory occupied by BDD nodes when performing analyses for several injection types in parallel.

We have formalized an object-sensitive notion of taintedness that can be applied to reference types. We have built a new, flow-, context- and field-sensitive static taintedness analysis based on this notion, proved it sound, implemented it in the Julia analyzer, and evaluated it. It scales to real code and gives useful results. As far as we know, this is the first object-sensitive taintedness analysis. As usual in static analysis, soundness is jeopardized by the use of reflection or non-standard class loaders. However, soundness is still relevant since it increases the confidence on the results, up to those features. Julia deals with the full bytecode generated by Java 8, including the new `invokedynamic`.

The novelty of the approach stems from Def. 8.8 of a property of reference types as a reachability property, whose relevance goes beyond the case of taintedness analysis. Here, we mean reachability of data from a memory reference, which is not reachability of abstract states through execution paths as in [90]. Def. 8.8 results in an object-sensitive analysis: the taintedness of an object determines that of its fields; a drawback is that a sound analysis must consider side-effects at `putfield` and `call`. The analysis becomes then field sensitive through an oracle-based approach (Sec. 8.3.1), already used for nullness analysis [102]. Hence the oracle is a general technique for building sound field-sensitive static analyses.

The extension of this work to implicit and hidden flows would provide a stronger guarantee against injections of tainted information into a set of sinks. The problem is complex: implicit flows in Java are not just due to conditionals but also to exception branches and dynamic resolution of method calls. The risk is that a sound analysis w.r.t. implicit flows would end up being very conservative and imprecise. Declassification might be helpful here, but its meaning for reference types (not just primitive values) must be studied. The extension of this work to the analysis of JSP, that are non-Java code mixed and interacting with Java code, currently not analyzed by Julia (only partially by concurrent tools), would avoid missed alarms, as Sec. 8.4 shows. It is also important to explain the warnings to the users, with an execution trace where data flows from sources into sinks. Fortify SCA already provides some support in that direction.

A locking discipline makes concurrent programming manageable. Used properly, it guarantees the lack of data races. Used improperly (with vague definitions or no mechanical checking), it is error-prone at best and misleading at worst.

Current definitions of locking disciplines and their implementations suffer from many ambiguities; furthermore, they often specify name protection rather than value protection, even though name protection does not in general provide a guarantee of freedom from data races. These ambiguities and unsoundness are a real issue in practice. We have formalized and proved a value-protection semantics, eliminating both the ambiguities and the unsoundness. Our locking discipline formalism is a common language for discussing data races and can also be used to express value protection. The leap from name to value semantics may be desirable in other domains as well.

Our case studies of real-world code show that programmers often make mistakes (precision 19–100%, recall 6–84%): they write locking-discipline specifications that their programs do not follow, and they fail to write ones that their programs do follow. Programmers seem to often assume an unsound name-protection semantics for the locking-discipline specifications. We have shown that the value-protection semantics is more restrictive and possibly harder to use; but the more accurate documentation and the reduction in bugs should be worth it.

References

1. The Akka Framework. `http://akka.io`.
2. Android. `https://developer.android.com`.
3. The Apache Hadoop Project. `http://hadoop.apache.org/`.
4. The bitcoinj library. `http://bitcoinj.github.io`.
5. BuDDy. `http://buddy.sourceforge.net`.
6. CAL. `http://embedded.eecs.berkeley.edu/Research/cal_bdd`.
7. The Checker Framework. `http://types.cs.washington.edu/checker-framework/`.
8. CUDD. `http://vlsi.colorado.edu/~fabio/CUDD`.
9. ITC99 Benchmark Circuits. `http://www.cerc.utexas.edu/itc99-benchmarks/bench.html`.
10. Java Modeling Language. `http://www.eecs.ucf.edu/~leavens/JML/jmlrefman/`.
11. The JavaBDD Library. `http://javabdd.sourceforge.net`.
12. JDD. `http://javaddlib.sourceforge.net/jdd`.
13. The Julia Static Analyzer. `http://www.juliasoft.com`.
14. SableJBDD. `http://www.sable.mcgill.ca/~fqian/SableJBDD`.
15. Soft Error. `https://en.wikipedia.org/wiki/Soft_error`.
16. Spring. `http://spring.io`.
17. M. Abadi, C. Flanagan, and S. N. Freund. Types for safe locking: Static race detection for java. *ACM Trans. Program. Lang. Syst.*, 28(2):207–255, 2006.
18. A. Aiken, J. S. Foster, J. Kodumal, and T. Terauchi. Checking and inferring local non-aliasing. In Cytron and Gupta [42], pages 129–140.
19. Apache. Tomcat. `http://tomcat.apache.org`.
20. Apache. Velocity. `http://velocity.apache.org`.
21. Apache. Zookeeper. `https://zookeeper.apache.org`.
22. D. Appelt, C. D. Nguyen, L. C. Briand, and N. Alshahwan. Automated Testing for SQL Injection Vulnerabilities: An Input Mutation Approach. In *ISSTA*, pages 259–269, San Jose, CA, USA, 2014.
23. S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. L. Traon, D. Octeau, and P. McDaniel. Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In M. F. P. O'Boyle and K. Pingali, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, page 29. ACM, 2014.

24. A. Baddeley. Working memory: looking back and looking forward. *Nat Rev Neurosci*, 4(10):829–839, Oct 2003.
25. R. Bagnara and P. Schachte. Factorizing equivalent variable pairs in robdd-based implementations of pos. In A. M. Haeberer, editor, *Algebraic Methodology and Software Technology, 7th International Conference, AMAST '98, Amazonia, Brasil, January 4-8, 1999, Proceedings*, volume 1548 of *Lecture Notes in Computer Science*, pages 471–485. Springer, 1998.
26. G. Barthe, D. Pichardie, and T. Rezk. A certified lightweight non-interference java bytecode verifier. *Mathematical Structures in Computer Science*, 23(5):1032–1081, 2013.
27. G. Barthe, T. Rezk, and A. Basu. Security types preserving compilation. *Computer Languages, Systems & Structures*, 33(2):35–59, 2007.
28. D. Bogdanas and G. Rosu. K-Java: A complete semantics of Java. In *ACM SIGPLAN-SIGACT POPL*, pages 445–456, Mumbai, India, 2015.
29. F. Bossi. Coral: a modern c++ library for the manipulation of boolean functions. `http://www.cs.unipr.it/Informatica/Tesi/Fabio_Bossi_20090225.pdf`.
30. C. Boyapati, R. Lee, and M. C. Rinard. Ownership types for safe programming: preventing data races and deadlocks. In M. Ibrahim and S. Matsuoka, editors, *Proceedings of the 2002 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2002, Seattle, Washington, USA, November 4-8, 2002.*, pages 211–230. ACM, 2002.
31. C. Boyapati and M. C. Rinard. A parameterized type system for race-free java programs. In L. M. Northrop and J. M. Vlissides, editors, *Proceedings of the 2001 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2001, Tampa, Florida, USA, October 14-18, 2001.*, pages 56–69. ACM, 2001.
32. K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient implementation of a BDD package. In *DAC*, pages 40–45, 1990.
33. R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986.
34. J. R. Burch, E. M. Clarke, and D. E. Long. Representing circuits more efficiently in symbolic model checking. In *DAC*, pages 403–407, 1991.
35. J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^20 states and beyond. In *Proceedings of the Fifth Annual Symposium on Logic in Computer Science (LICS '90), Philadelphia, Pennsylvania, USA, June 4-7, 1990*, pages 428–439. IEEE Computer Society, 1990.
36. D. Clark, C. Hankin, and S. Hunt. Information flow for algol-like languages. *Computer Languages*, 28(1):3–28, 2002.
37. D. G. Clarke, J. Potter, and J. Noble. Ownership types for flexible alias protection. In B. N. Freeman-Benson and C. Chambers, editors, *Proceedings of the 1998 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '98), Vancouver, British Columbia, Canada, October 18-22, 1998.*, pages 48–64. ACM, 1998.
38. J. C. Corbett, M. B. Dwyer, J. Hatcliff, and Robby. Expressing checkable properties of dynamic systems: the bandera specification language. *STTT*, 4(1):34–56, 2002.
39. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints.

In R. M. Graham, M. A. Harrison, and R. Sethi, editors, *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, pages 238–252. ACM, 1977.

40. N. Cowan. Chapter 20 what are the differences between long-term, short-term, and working memory? In V. F. C. Wayne S. Sossin, Jean-Claude Lacaille and S. Belleville, editors, *Essence of Memory*, volume 169 of *Progress in Brain Research*, pages 323 – 338. Elsevier, 2008.
41. D. Cunningham, S. Drossopoulou, and S. Eisenbach. Universes for race safety. In *VAMP 2007: Verification and Analysis of Multi-threaded Java-like Programs (VAMP)*, pages 20–51, Lisbon, Portugal, Sept. 2007.
42. R. Cytron and R. Gupta, editors. *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation 2003, San Diego, California, USA, June 9-11, 2003.* ACM, 2003.
43. W. Dietl, S. Drossopoulou, and P. Müller. Generic universe types. In E. Ernst, editor, *ECOOP 2007 - Object-Oriented Programming, 21st European Conference, Berlin, Germany, July 30 - August 3, 2007, Proceedings*, volume 4609 of *Lecture Notes in Computer Science*, pages 28–53. Springer, 2007.
44. J. C. Doshi, M. Christian, and B. H. Trivedi. SQL FILTER - SQL injection prevention and logging using dynamic network filter. In J. L. Mauri, S. M. Thampi, D. B. Rawat, and D. Jin, editors, *Security in Computing and Communications - Second International Symposium, SSCC 2014, Delhi, India, September 24-27, 2014. Proceedings*, volume 467 of *Communications in Computer and Information Science*, pages 400–406. Springer, 2014.
45. R. Drechsler. Verifying integrity of decision diagrams. In W. D. Ehrenberger, editor, *Computer Safety, Reliability and Security, 17th International Conference, SAFECOMP'98, Heidelberg, Germany, October 5-7, 1998, Proceedings*, volume 1516 of *Lecture Notes in Computer Science*, pages 380–389. Springer, 1998.
46. Eclipse. The Jetty project. `http://eclipse.org/jetty`.
47. M. D. Ernst, A. Lovato, D. Macedonio, C. Spiridon, and F. Spoto. Boolean formulas for the static identification of injection attacks in java. In M. Davis, A. Fehnker, A. McIver, and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning - 20th International Conference, LPAR-20 2015, Suva, Fiji, November 24-28, 2015, Proceedings*, volume 9450 of *Lecture Notes in Computer Science*, pages 130–145. Springer, 2015.
48. M. D. Ernst, A. Lovato, D. Macedonio, F. Spoto, and J. Thaine. Locking discipline inference and checking. In L. K. Dillon, W. Visser, and L. Williams, editors, *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, pages 1133–1144. ACM, 2016.
49. C. Flanagan and S. N. Freund. Type-based race detection for java. In M. S. Lam, editor, *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Vancouver, Britith Columbia, Canada, June 18-21, 2000*, pages 219–232. ACM, 2000.
50. C. Flanagan and S. N. Freund. Type inference against races. In R. Giacobazzi, editor, *Static Analysis, 11th International Symposium, SAS 2004, Verona, Italy, August 26-28, 2004, Proceedings*, volume 3148 of *Lecture Notes in Computer Science*, pages 116–132. Springer, 2004.
51. C. Flanagan, S. N. Freund, and S. Qadeer. Thread-modular verification for shared-memory programs. In D. L. Mêtayer, editor, *Programming Languages*

and Systems, 11th European Symposium on Programming, ESOP 2002, held as Part of the Joint European Conference on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings, volume 2305 of *Lecture Notes in Computer Science*, pages 262–277. Springer, 2002.

52. C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for java. In J. Knoop and L. J. Hendren, editors, *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Berlin, Germany, June 17-19, 2002*, pages 234–245. ACM, 2002.
53. C. Flanagan and S. Qadeer. A type and effect system for atomicity. In Cytron and Gupta [42], pages 338–349.
54. E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides. *Design Patterns.* Addison-Wesley, Reading, MA, 1995.
55. S. Genaim, R. Giacobazzi, and I. Mastroeni. Modeling secure information flow with boolean functions. In P. Ryan, editor, *WITS'04*, April 2004.
56. S. Genaim and F. Spoto. Information flow analysis for java bytecode. In R. Cousot, editor, *Verification, Model Checking, and Abstract Interpretation, 6th International Conference, VMCAI 2005, Paris, France, January 17-19, 2005, Proceedings*, volume 3385 of *Lecture Notes in Computer Science*, pages 346–362. Springer, 2005.
57. S. Genaim and F. Spoto. Constancy analysis. In M. Huisman, editor, *FTfJP*, Paphos, Cyprus, July 2008. Radboud University.
58. P. Gerakios, N. Papaspyrou, and K. F. Sagonas. A type and effect system for deadlock avoidance in low-level languages. In S. Weirich and D. Dreyer, editors, *Proceedings of TLDI 2011: 2011 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, Austin, TX, USA, January 25, 2011*, pages 15–28. ACM, 2011.
59. B. Goetz, T. Peierls, J. Bloch, J. Bowbeer, D. Holmes, and D. Lea. *Java Concurrency in Practice.* Addison-Wesley, 2006.
60. Google. Guava: Google Core Libraries for Java 1.6+. `https://code.google.com/p/guava-libraries`.
61. C. S. Gordon, M. D. Ernst, and D. Grossman. Static lock capabilities for deadlock freedom. In B. C. Pierce, editor, *Proceedings of TLDI 2012: The Seventh ACM SIGPLAN Workshop on Types in Languages Design and Implementation, Philadelphia, PA, USA, Saturday, January 28, 2012*, pages 67–78. ACM, 2012.
62. J. Gosling, B. Joy, G. Steele, G. Bracha, and A. Buckley. *The Java Language Specification.* Addison Wesley, Boston, MA, Java SE 8 edition, 2014.
63. J. Gray. Why do computers stop and what can be done about it? In *Symposium on Reliability in Distributed Software and Database Systems*, pages 3–12, 1986.
64. D. Grossman. Type-safe multithreading in cyclone. In Z. Shao and P. Lee, editors, *Proceedings of TLDI'03: 2003 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, New Orleans, Louisiana, USA, January 18, 2003*, pages 13–25. ACM, 2003.
65. J. Huang, Q. Luo, and G. Rosu. Gpredict: Generic predictive concurrency analysis. In A. Bertolino, G. Canfora, and S. G. Elbaum, editors, *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, pages 847–857. IEEE Computer Society, 2015.
66. S. T. Iqbal and E. Horvitz. Disruption and recovery of computing tasks: field study, analysis, and directions. In M. B. Rosson and D. J. Gilmore, editors,

Proceedings of the 2007 Conference on Human Factors in Computing Systems, CHI 2007, San Jose, California, USA, April 28 - May 3, 2007, pages 677–686. ACM, 2007.
67. Y. Jang and J. Choi. Detecting SQL injection attacks using query result size. *Computers and Security*, 44:104–118, 2014.
68. N. Kobayashi and K. Shirane. Type-based information analysis for low-level languages. In *The Third Asian Workshop on Programming Languages and Systems, APLAS'02, Shanghai Jiao Tong University, Shanghai, China, November 29 - December 1, 2002, Proceedings*, pages 302–316, 2002.
69. D. G. Kumar and M. Chatterjee. MAC based solution for SQL injection. *J. Computer Virology and Hacking Techniques*, 11(1):1–7, 2015.
70. L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Computers*, 28(9):690–691, 1979.
71. P. Laud. Semantics and program analysis of computationally secure information flow. In D. Sands, editor, *Programming Languages and Systems, 10th European Symposium on Programming, ESOP 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001, Proceedings*, volume 2028 of *Lecture Notes in Computer Science*, pages 77–91. Springer, 2001.
72. T. Lindholm, F. Yellin, G. Bracha, and A. Buckley. *The Java Virtual Machine Specification, Java SE 7 Edition.* Addison-Wesley Professional, 1st edition, 2013.
73. L. Liu, J. Xu, M. Li, and J. Yang. A dynamic SQL injection vulnerability test case generation model based on the multiple phases detection approach. In *37th Annual IEEE Computer Software and Applications Conference, COMPSAC 2013, Kyoto, Japan, July 22-26, 2013*, pages 256–261. IEEE Computer Society, 2013.
74. B. Long and B. W. Long. Formal specification of Java concurrency to assist software verification. In *IPDPS*, page 136, Nice, France, April 2003.
75. A. Lovato, D. Macedonio, and F. Spoto. A thread-safe library for binary decision diagrams. In D. Giannakopoulou and G. Salaün, editors, *Software Engineering and Formal Methods - 12th International Conference, SEFM 2014, Grenoble, France, September 1-5, 2014. Proceedings*, volume 8702 of *Lecture Notes in Computer Science*, pages 35–49. Springer, 2014.
76. Y. Lu, J. Potter, and J. Xue. Structural lock correlation with ownership types. In M. Felleisen and P. Gardner, editors, *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, volume 7792 of *Lecture Notes in Computer Science*, pages 391–410. Springer, 2013.
77. A. Makiou, Y. Begriche, and A. Serhrouchni. Improving web application firewalls to detect advanced SQL injection attacks. In *10th International Conference on Information Assurance and Security, IAS 2014, Okinawa, Japan, November 28-30, 2014*, pages 35–40. IEEE, 2014.
78. C. Meinel and T. Theobald. *Algorithms and Data Structures in VLSI Design: OBDD - Foundations and Applications.* Springer, 1998.
79. MITRE/SANS. Top 25 Most Dangerous Software Errors. `http://cwe.mitre.org/top25`, September 2011.
80. M. Mizuno. A least fixed point approach to inter-procedural information flow control. In *NCSC*, pages 558–570, 1989.

81. M. Naik, A. Aiken, and J. Whaley. Effective static race detection for java. In M. I. Schwartzbach and T. Ball, editors, *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation, Ottawa, Ontario, Canada, June 11-14, 2006*, pages 308–319. ACM, 2006.
82. NASA. Java PathFinder. `http://babelfish.arc.nasa.gov/trac/jpf`.
83. D. Nikolic and F. Spoto. Definite expression aliasing analysis for Java bytecode. In *9th International Colloquium on Theoretical Aspects of Computing (ICTAC 2012)*, pages 74–89, Bangalore, India, September 2012.
84. D. Nikolic and F. Spoto. Definite expression aliasing analysis for java bytecode. In A. Roychoudhury and M. D'Souza, editors, *Theoretical Aspects of Computing - ICTAC 2012 - 9th International Colloquium, Bangalore, India, September 24-27, 2012. Proceedings*, volume 7521 of *Lecture Notes in Computer Science*, pages 74–89. Springer, 2012.
85. D. Nikolic and F. Spoto. Reachability analysis of program variables. In B. Gramlich, D. Miller, and U. Sattler, editors, *Automated Reasoning - 6th International Joint Conference, IJCAR 2012, Manchester, UK, June 26-29, 2012. Proceedings*, volume 7364 of *Lecture Notes in Computer Science*, pages 423–438. Springer, 2012.
86. OWASP. Benchmark. `https://www.owasp.org/index.php/Benchmark`. Checked on Oct 13, 2016.
87. J. Palsberg and M. I. Schwartzbach. Object-oriented type inference. In A. Paepcke, editor, *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'91), Sixth Annual Conference, Phoenix, Arizona, USA, October 6-11, 1991, Proceedings.*, pages 146–161. ACM, 1991.
88. É. Payet and F. Spoto. Magic-sets transformation for the analysis of java bytecode. In H. R. Nielson and G. Filé, editors, *Static Analysis, 14th International Symposium, SAS 2007, Kongens Lyngby, Denmark, August 22-24, 2007, Proceedings*, volume 4634 of *Lecture Notes in Computer Science*, pages 452–467. Springer, 2007.
89. V. Pech. Concurrency is hot, try the JCIP annotations. `http://jetbrains.dzone.com/tips/concurrency-hot-try-jcip`, February 2010.
90. T. W. Reps, S. Horwitz, and S. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In R. K. Cytron and P. Lee, editors, *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995*, pages 49–61. ACM Press, 1995.
91. E. Rodríguez, M. B. Dwyer, C. Flanagan, J. Hatcliff, G. T. Leavens, and Robby. Extending JML for modular specification and verification of multi-threaded programs. In A. P. Black, editor, *ECOOP 2005 - Object-Oriented Programming, 19th European Conference, Glasgow, UK, July 25-29, 2005, Proceedings*, volume 3586 of *Lecture Notes in Computer Science*, pages 551–576. Springer, 2005.
92. J. Rose, N. Swamy, and M. Hicks. Dynamic inference of polymorphic lock types. *Sci. Comput. Program.*, 58(3):366–383, 2005.
93. A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.
94. A. Sabelfeld and D. Sands. A per model of secure information flow in sequential programs. *Higher-Order and Symbolic Computation*, 14(1):59–91, 2001.
95. G. Salton. *Automatic Information Organization and Retrieval.* McGraw-Hill, 1968.

96. S. Secci and F. Spoto. Pair-sharing analysis of object-oriented programs. In C. Hankin and I. Siveroni, editors, *Static Analysis, 12th International Symposium, SAS 2005, London, UK, September 7-9, 2005, Proceedings*, volume 3672 of *Lecture Notes in Computer Science*, pages 320–335. Springer, 2005.
97. H. Shahriar and M. Zulkernine. Information-theoretic detection of SQL injection attacks. In *14th International IEEE Symposium on High-Assurance Systems Engineering, HASE 2012, Omaha, NE, USA, October 25-27, 2012*, pages 40–47. IEEE Computer Society, 2012.
98. L. K. Shar and H. B. K. Tan. Defeating SQL injection. *IEEE Computer*, 46(3):69–77, 2013.
99. N. M. Sheykhkanloo. Employing neural networks for the detection of SQL injection attack. In R. Poet and M. Rajarajan, editors, *Proceedings of the 7th International Conference on Security of Information and Networks, Glasgow, Scotland, UK, September 9-11, 2014*, page 318. ACM, 2014.
100. B. Simic and J. Walden. Eliminating SQL injection and cross site scripting using aspect oriented programming. In J. Jürjens, B. Livshits, and R. Scandariato, editors, *Engineering Secure Software and Systems - 5th International Symposium, ESSoS 2013, Paris, France, February 27 - March 1, 2013. Proceedings*, volume 7781 of *Lecture Notes in Computer Science*, pages 213–228. Springer, 2013.
101. C. Skalka and S. F. Smith. Static enforcement of security with types. In M. Odersky and P. Wadler, editors, *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000.*, pages 34–45. ACM, 2000.
102. F. Spoto. Nullness analysis in boolean form. In A. Cerone and S. Gruner, editors, *Sixth IEEE International Conference on Software Engineering and Formal Methods, SEFM 2008, Cape Town, South Africa, 10-14 November 2008*, pages 21–30. IEEE Computer Society, 2008.
103. F. Spoto and M. D. Ernst. Inference of field initialization. In R. N. Taylor, H. C. Gall, and N. Medvidovic, editors, *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, pages 231–240. ACM, 2011.
104. F. Spoto and T. P. Jensen. Class analyses as abstract interpretations of trace semantics. *ACM Trans. Prog. Lang. Syst.*, 25(5):578–630, 2003.
105. N. Sterling. WARLOCK - A static data race analysis tool. In *Proceedings of the Usenix Winter 1993 Technical Conference, San Diego, California, USA, January 1993*, pages 97–106. USENIX Association, 1993.
106. F. Tip and J. Palsberg. Scalable propagation-based call graph construction algorithms. In M. B. Rosson and D. Lea, editors, *Proceedings of the 2000 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA 2000), Minneapolis, Minnesota, USA, October 15-19, 2000.*, pages 281–293. ACM, 2000.
107. O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman. TAJ: effective taint analysis of web applications. In M. Hind and A. Diwan, editors, *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, pages 87–97. ACM, 2009.
108. T. van Dijk and J. van de Pol. Sylvan: Multi-core decision diagrams. In C. Baier and C. Tinelli, editors, *Tools and Algorithms for the Construction and Analysis*

of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings, volume 9035 of *Lecture Notes in Computer Science*, pages 677–691. Springer, 2015.

109. D. M. Volpano, C. E. Irvine, and G. Smith. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2/3):167–188, 1996.
110. T. Wu, J. Pan, C. Chen, and C. Lin. Towards SQL injection attacks detection mechanism using parse tree. In H. Sun, C. Yang, C. Lin, J. Pan, V. Snásel, and A. Abraham, editors, *Genetic and Evolutionary Computing - Proceeding of the Eighth International Conference on Genetic and Evolutionary Computing, ICGEC 2014, October 18-20, 2014, Nanchang, China*, volume 329 of *Advances in Intelligent Systems and Computing*, pages 371–380. Springer, 2014.
111. Y. Zhao and J. Boyland. Assuring lock usage in multithreaded programs with fractional permissions. In *20th Australian Software Engineering Conference (ASWEC 2009), 14-17 April 2009, Gold Cost, Australia*, pages 277–286. IEEE Computer Society, 2009.