ANDROMEDA: Accurate and Scalable Security Analysis of Web Applications

Omer Tripp[1], Marco Pistoia[2], Patrick Cousot[3], Radhia Cousot[4], and Salvatore Guarnieri[5]

[1] Tel Aviv University and IBM Software Group, Israel, omert@il.ibm.com
[2] IBM Thomas J. Watson Research Center, USA, pistoia@us.ibm.com
[3] New York University, USA, pcousot@cs.nyu.edu
[4] École Normale Supérieure, France, radhia.cousot@ens.edu
[5] University of Washington and IBM Software Group, USA, sguarni@us.ibm.com

Abstract. Security auditing of industry-scale software systems mandates automation. Static taint analysis enables deep and exhaustive tracking of suspicious data flows for detection of potential leakage and integrity violations, such as cross-site scripting (XSS), SQL injection (SQLi) and log forging. Research in this area has taken two directions: program slicing and type systems. Both of these approaches suffer from a high rate of false findings, which limits the usability of analysis tools based on these techniques. Attempts to reduce the number of false findings have resulted in analyses that are either (i) unsound, suffering from the dual problem of false negatives, or (ii) too expensive due to their high precision, thereby failing to scale to real-world applications.
In this paper, we investigate a novel approach for enabling precise yet scalable static taint analysis. The key observation informing our approach is that taint analysis is a demand-driven problem, which enables lazy computation of vulnerable information flows, instead of eagerly computing a complete data-flow solution, which is the reason for the traditional dichotomy between scalability and precision. We have implemented our approach in ANDROMEDA, an analysis tool that computes data-flow propagations on demand, in an efficient and accurate manner, and additionally features incremental analysis capabilities. ANDROMEDA is currently in use in a commercial product. It supports applications written in Java, .NET and JavaScript. Our extensive evaluation of ANDROMEDA on a suite of 16 production-level benchmarks shows ANDROMEDA to achieve high accuracy and compare favorably to a state-of-the-art tool that trades soundness for precision.

Keywords: Security, Static Analysis, Taint Analysis, Information Flow, Integrity, Abstract Interpretation

1 Introduction

Web-application security is an ever-growing concern. By design, Web applications feed on inputs whose source is untrusted, perform numerous security-sensitive operations (such as database accesses and transfers of Web content to remote machines), and expose data to potentially malicious observers. It is not surprising, then, that six out of the ten most critical Web-application vulnerabilities[6] are *information-flow violations*, which can break *integrity* (whereby untrusted inputs flow into security-sensitive computations) or *confidentiality* (whereby private information is revealed to public observers).

During the last decade, there has been intensive research on methods and algorithms for automatically detecting information-flow violations in Web applications. However, many of the published approaches are not readily applicable to industrial Web applications. Solutions based on type systems tend to be overly complex and conservative [34, 20, 27], and are therefore unlikely to enjoy broad adoption, whereas those based on program slicing are often unsound [33] or limited in scalability [14, 28].

Our Approach. In this paper, we present ANDROMEDA, a sound and highly accurate static security scanner, which also scales to large code bases, being designed for commercial needs as part of a product offering, IBM Security AppScan Source.[7] ANDROMEDA performs a form of *abstract interpretation* [6] known as *taint analysis* [25]: It statically detects data flows wherein information returned by a "source" reaches the parameters of a "sink" without being properly endorsed by a "downgrader". Depending on whether the problem being solved is related to integrity or confidentiality, a *source* is a method that injects untrusted or secret input into a program, a *sink* is a method that performs a security-sensitive computation or exposes information to public observers, and a *downgrader* is a method that sanitizes untrusted data or declassifies confidential data, respectively. ANDROMEDA is equipped with a thorough configuration of triples of sources, sinks and downgraders for all known integrity and confidentiality problems, partitioned into *security rules*, such as XSS and SQLi.

The key idea behind ANDROMEDA is to track vulnerable information flows (emanating from sources) in a demand-driven manner, without eagerly building any complete representation of the subject application. ANDROMEDA builds a call-graph representation of the program based on intraprocedural type inference. Furthermore, when there is a need to compute an aliasing relationship, stemming from flow of vulnerable information into the heap, ANDROMEDA issues a granular aliasing query focused on the flow at hand, thereby obviating the need for whole-program pointer analysis. This enables (i) sound and efficient scanning of large applications, where typically only a small portion of the application requires modeling, and (ii) incremental-analysis capabilities, which allow to preserve valid parts of the old solution when rescanning the application following code changes. Both of these characteristics are enabled by the fact that ANDROMEDA does not need to build any form of whole-program representation.

In another view, ANDROMEDA can be thought of as an extended type system, where a fully automated context-sensitive, interprocedural, incremental inference engine automatically attaches security annotations to program locations and propagates them. ANDROMEDA enforces the following two properties:

1. The inference process is fully automated, and thus no complex, non-standard type system is forced on the developer.
2. The analysis is infinitely context sensitive (up to recursion), and consequently, it does not produce overly conservative results.

[6] `http://owasp.org`.
[7] `http://ibm.com/software/rational/products/appscan/source/`.

These properties lift the two most significant barriers that have so far prevented type systems from enjoying broad industrial adoption.

To our knowledge, ANDROMEDA is the first taint-analysis algorithm that performs demand-driven analysis from the bottom up, including representing the program's type hierarchy, call graph and data-flow propagation graph. This is the key to achieving both accuracy and scalability without sacrificing soundness. We are also not aware of any other security analysis featuring incremental scanning capabilities.

Contributions. This paper makes the following specific contributions:

- **Demand-driven taint analysis.** We present a demand-driven security analysis algorithm that is sound (even in the presence of multi threading), accurate and scalable. We describe the design of the entire analysis stack in support of this feature.
- **Incremental analysis.** ANDROMEDA enables efficient rescanning of the subject application following code changes. This is thanks to its ability to track vulnerable flows in a "local", on-demand fashion, which facilitates invalidation of only parts of the previous data-flow solution. We describe the data structures ANDROMEDA implements for efficient incremental analysis.
- **Framework and library support.** Beyond the core analysis, we describe novel extensions enabling effective modeling of framework and library code. These extensions are important for an analysis targeting real-world Web applications, which are built atop reusable frameworks.
- **Implementation and evaluation.** ANDROMEDA has been fully implemented. It supports Java, .NET and JavaScript programs, and is currently used in a commercial product. We present an extensive evaluation of ANDROMEDA, comparing it to a state-of-the-art security scanner [33] on a suite of 16 real-world Java benchmarks, which shows ANDROMEDA to be superior.

2 Motivation and Overview

To illustrate some of the unique features of ANDROMEDA, we use the `Aliasing5` benchmark from the Stanford SecuriBench Micro suite.[8] Designed for expository purposes, this example shows a Java Web application reading untrusted data from servlet parameters. Specifically, this example highlights the importance of tracking aliasing relationships between program variables and fields for sound security analysis, with `buf` flowing into two formal arguments of method `foo` (line 6).

The flow of the entire program is as follows: The `doGet` handler of the `Aliasing5` servlet first initializes a fresh `StringBuffer` object, `buf`, with the string `"abc"` (line 5). It then invokes method `foo`, such that its first two formal arguments (`buf` and `buf2`) are aliased. Next, `foo` assigns the content of an untrusted parameter, `"name"`, to variable `name`, in the source statement at line 10. This untrusted value subsequently taints the buffer pointed-to by `buf` (line 11). Because of the aliasing relationship between `buf` and `buf2`, the security-sensitive operation at line 13, which renders the content of `buf2` to the response HTML, becomes vulnerable.

[8] `http://suif.stanford.edu/~livshits/work/securibench-micro`.

```
1:  public class Aliasing5 extends HttpServlet {
2:   protected void doGet(HttpServletRequest req,
3:     HttpServletResponse resp)
4:     throws ServletException, IOException {
5:    StringBuffer buf = new StringBuffer("abc");
6:    foo(buf, buf, resp, req); }
7:   void foo(StringBuffer buf,
8:     StringBuffer buf2, ServletResponse resp,
9:     ServletRequest req) throws IOException {
10:   String name = req.getParameter("name");
11:   buf.append(name);
12:   PrintWriter writer = resp.getWriter();
13:   writer.println(buf2.toString()); /* BAD */ } }
```

Fig. 1. The `Aliasing5` Benchmark from the SecuriBench Micro Suite

To detect the vulnerability in this program, the security scanner must account for the aliasing between `buf` and `buf2` in `foo`. Existing approaches have all addressed this requirement by applying a preliminary whole-program pointer analysis, such as Andersen's flow-insensitive analysis [1], to eagerly compute an aliasing solution before starting the security analysis [33]. Perfoming a global aliasing analysis places a significant limitation on the scalability of the client security analysis, which is mitigated (but not lifted) if the aliasing analysis is coarse (*i.e.*, context insensitive, flow insensitive, *etc*). In that case, however, the ensuing security analysis becomes imprecise, often yielding an excess of false reports due to spurious data flows.

ANDROMEDA, instead, performs on-demand alias resolution. It tracks symbolic representations of security facts, known as "access paths", and augments the set of tracked representations to account for aliases of tracked objects. Loosely speaking, an access path is a sequence of field identifiers, rooted at a local variable, such as `x.f.g`. This access path evaluates to the object o reached by dereferencing field `f` of the object pointed-to by `x`, and then dereferencing field `g` of o (or $\perp$ if no such object exists). (We provide a formal definition of an access path later, in Section 3.)

ANDROMEDA starts by modeling the effect of the source statement at line 10 as the seeding data-flow fact `name.*`. The $\star$ notation simply represents the fact that all objects reachable through variable `name` are to be considered untrusted. Then, the flow at line 11 leads the analysis to track both `name.*` and `buf.content.*`. However, because there is a flow into the heap at line 11, the analysis further issues an on-demand interprocedural aliasing query, which establishes that `buf.content` is aliased with `buf2.content`. Therefore, the analysis additionally tracks `buf2.content.*`. This exposes the vulnerability at line 13, where the `toString` call renders `buf2.content` to the response HTML.

3 Core Taint Analysis

The ANDROMEDA algorithm takes as input a Web application, along with its set of supporting libraries, and validates it with respect to a specification in the form of a set of "security rules". A *security rule* is a triple $\langle Src, Dwn, Snk \rangle$, where Src, Dwn and Snk are patterns for matching sources, downgraders and sinks in the subject program, respectively. A pattern match is either a method call or a field dereference. A vulnerability is reported for flows extending between a source and a sink belonging to the same rule, without a downgrader from the rule's Dwn set mediating the flow.

The ANDROMEDA algorithm interleaves call-graph construction with tracking of vulnerable information flows. This is to avoid building eager whole-program representations. Both the call graph and the data-flow solution computed atop the call graph are expanded on demand, ensuring scalability while retaining a high degree of accuracy.

3.1 Type-hierarchy and Call-graph Construction

As mentioned earlier, ANDROMEDA refrains from building global program representations. Instead, it computes its supporting type hierarchy on demand. For this, ANDROMEDA utilizes lazy data structures, which provide sophisticated mechanisms for caching and demand evaluation of type information at the granularity of individual methods and class fields.

Call-graph construction is also performed lazily. The call graph is built based on local reasoning, by resolving virtual calls according to an intra-procedural type-inference algorithm [3]. Call sites are not necessarily expanded eagerly (*i.e.*, before the data-flow analysis stage). Rather, an oracle is used to determine whether any given call site may lead to the discovery of source statements. Our oracle is sound, and is based on control-flow reachability between the calling method and source methods within the type-hierarchy graph [7].

3.2 Data-flow Analysis

For a formal description of ANDROMEDA's data-flow analysis algorithm, we use a standard description of the program's state, based on the following domains:

$VarId$	Program variables	$Val = Loc \cup \{null\}$	Values
$FldId$	Field identifiers	$Env\colon VarId \to Val$	Environment
Loc	Unbounded set of objects	$Heap\colon Loc \times FldId \to Val$	Heap

A program state, $\sigma = \langle \mathbf{E}, \mathbf{H} \rangle \in States = Env \times Heap$, maintains the pointing from variables to their values, as well as from object fields to their values.

To describe the algorithm we use the following syntactic structures:

Statement	Meaning
$\mathtt{x} = \mathbf{new}\ \mathtt{Object}()$	$[\![\mathtt{x} = \mathbf{new}\ \mathtt{Object}()]\!]\sigma = \sigma[\mathtt{x} \mapsto o \in Loc.\ o \text{ is fresh}]$
$\mathtt{x} = \mathtt{y}$	$[\![\mathtt{x} = \mathtt{y}]\!]\sigma = \sigma[\mathbf{E}(\mathtt{x}) \mapsto \mathbf{E}(y)]$
$\mathtt{x.f} = \mathtt{y}$	$[\![\mathtt{x.f} = \mathtt{y}]\!]\sigma = \sigma[\mathbf{H}(\langle \mathbf{E}(\mathtt{x}), \mathtt{f} \rangle) \mapsto \mathbf{E}(y)]$
$\mathtt{x} = \mathtt{y.f}$	$[\![\mathtt{x} = \mathtt{y.f}]\!]\sigma = \sigma[\mathbf{E}(\mathtt{x}) \mapsto \mathbf{H}(\langle \mathbf{E}(\mathtt{y}), \mathtt{f} \rangle)]$

These are kept to a minimum to simplify the description of the analysis. Extending the core language to contain procedure calls is straightforward [5].

Instrumented Concrete Semantics. To track security facts, we instrument the concrete semantics to further maintain untrusted (or tainted) access paths. Informally, an access path is a symbolic representation of a heap location. For example, access path $\texttt{x.g}$ denotes the heap location pointed-to by field $\texttt{g}$ of the object pointed-to by variable $\texttt{x}$. Security analysis over access paths tracks the set of paths evaluating to untrusted values.

More formally, an *access path* is a (possibly empty) sequence of field identifiers rooted at a local variable; *i.e.*, an element in $VarId \times (FldId)^*$. The meaning of access path $\texttt{x.f}_1 \ldots \texttt{f}_n$ is the unique value $o \in Val$ reached by first dereferencing $\texttt{x}$ using $\mathbf{E}$, and then following the references through $\texttt{f}_1 \ldots \texttt{f}_n$ in $\texttt{H}$, or $\perp$ if there are intermediate null dereferences in the path. This is defined inductively as follows:

$$[\![\texttt{x}.\epsilon]\!]\sigma = \begin{cases} \mathbf{E}(\texttt{x}) & \texttt{x} \in dom(\mathbf{E}) \\ \perp & \text{otherwise} \end{cases}$$

$$[\![\texttt{x.f}_1 \ldots \texttt{f}_n]\!]\sigma = \begin{cases} \mathbf{H}(\langle [\![\texttt{x.f}_1 \ldots \texttt{f}_{n-1}]\!], \texttt{f}_n \rangle) & [\![\texttt{x.f}_1 \ldots \texttt{f}_{n-1}]\!]\sigma \neq \perp, \\ & \langle [\![\texttt{x.f}_1 \ldots \texttt{f}_{n-1}]\!], \texttt{f}_n \rangle \in dom(\mathbf{H}) \\ \perp & \text{otherwise} \end{cases}$$

An instrumented concrete state is a triple, $\sigma = \langle \mathbf{E}, \mathbf{H}, \mathbf{T} \rangle$, where $\mathbf{T}$ is a set of *tainted access paths*. We assume a security specification, $\mathcal{S}$, which seeds the set $\mathbf{T}$ when evaluating certain assignment and field-read statements (according to the Src set of the provided security rules). The semantic rules for updating $\mathbf{T}$ appear in Figure 2.

$$\begin{array}{lll}
\mathbf{T} & \overset{\texttt{x=}\mathbf{new}\ \ldots;}{\rightarrow} & \mathbf{T} \\
\mathbf{T} & \overset{\texttt{x=y;}}{\rightarrow} & \mathbf{T} \cup \{\texttt{x.f}_1 \ldots \texttt{f}_n \colon \texttt{y.f}_1 \ldots \texttt{f}_n \in \mathbf{T}\} \\
\mathbf{T} & \overset{\texttt{x=y.f;}}{\rightarrow} & \mathbf{T} \cup \{\texttt{x.f}_1 \ldots \texttt{f}_n \colon \texttt{y.f f}_1 \ldots \texttt{f}_n \in \mathbf{T}\} \\
\mathbf{T} & \overset{\texttt{x.f=y;}}{\rightarrow} & \mathbf{T} \cup \{\mathbf{A}(\texttt{x}).\texttt{f f}_1 \ldots \texttt{f}_n \colon \texttt{y.f}_1 \ldots \texttt{f}_n \in \mathbf{T}\}
\end{array}$$

Fig. 2. Forward Data-flow Equations

$$\begin{array}{lll}
\mathbf{A} & \overset{\texttt{x=}\mathbf{new}\ \ldots;}{\rightarrow} & \mathbf{A} \\
\mathbf{A} & \overset{\texttt{x=y;}}{\rightarrow} & \mathbf{A} \cup \{\texttt{y.f}_1 \ldots \texttt{f}_n \colon \texttt{x.f}_1 \ldots \texttt{f}_n \in \mathbf{A}\} \\
\mathbf{A} & \overset{\texttt{x=y.f;}}{\rightarrow} & \mathbf{A} \cup \{\texttt{y.f f}_1 \ldots \texttt{f}_n \colon \texttt{x.f}_1 \ldots \texttt{f}_n \in \mathbf{A}\} \cup \{\texttt{x.f}_1 \ldots \texttt{f}_n \colon \texttt{y.f f}_1 \ldots \texttt{f}_n \in \mathbf{A}\} \\
\mathbf{A} & \overset{\texttt{x.f=y;}}{\rightarrow} & \mathbf{A} \cup \{\texttt{y.f}_1 \ldots \texttt{f}_n \colon \texttt{x.f f}_1 \ldots \texttt{f}_n \in \mathbf{A}\} \cup \{\texttt{x.f f}_1 \ldots \texttt{f}_n \colon \texttt{y.f}_1 \ldots \texttt{f}_n \in \mathbf{A}\}
\end{array}$$

Fig. 3. Backward Data-flow Equations

Access-path Widening. The key difficulty in using the symbolic access-path representation for static security analysis is that this representation of the heap, which is known as *storeless* [10], is unbounded. This problem manifests when dealing with recursive data structures, such as linked lists. To deal with this problem, we apply widening by introducing a special symbol, $*$. An access path now has either the concrete form $\mathtt{x.f_1 \ldots f_n}$, or the widened form $\mathtt{x.f_1 \ldots f_n} *$, where

$$[\![\mathtt{x.f_1 \ldots f_n} *]\!]\sigma = \{o \colon \exists \mathtt{f_{n+1} \ldots f_k} \in (\mathit{FldId})^*.\ o = [\![\mathtt{x.f_1 \ldots f_n\, f_{n+1} \ldots f_k}]\!]\sigma\}$$

That is, a widened access path potentially points to more than one object.

In this way, the analysis can track a bounded number of access paths in a sound manner by restricting the length of an access path to some constant c, and allowing for insertion of $*$ at the end of a path of length c instead of extending it when accounting for the effect of a field-assignment statement.

On-demand Aliasing. As mentioned earlier, ANDROMEDA features the ability to soundly track symbolic security facts. The key idea is to perform alias analysis on demand, when an untrusted value flows into an object field (*i.e.*, untrusted data flows into the heap). We first illustrate this situation through a simple example, where we assume that initially there is a single taint fact, $\mathbf{T} = \{\mathtt{z.g}\}$, and the last statement—assigning a value to `o.sinkfld`—is a sink, and as such must not be assigned an untrusted value:

$$\begin{tikzcd}
\mathtt{x = y.f} \arrow[d, bend right, "{[2]\ \mathtt{y.f.h}}"'] \arrow[dd, bend right=60, "{[2]\ \mathtt{y.f.h}}"'] & \mathbf{T} = \{\mathtt{z.g}\} \\
\mathtt{x.h = z.g} \arrow[u, bend right, "{[1]\ \mathtt{x.h}}"'] & \mathbf{T} = \{\mathtt{z.g}, \mathtt{x.h}, \mathtt{y.f\ h}\} \\
\mathtt{w = y.f.h} \arrow[d, bend right, "{[3]\ \mathtt{w}}"'] & \mathbf{T} = \{\mathtt{z.g}, \mathtt{x.h}, \mathtt{y.f\ h}, \mathtt{w}\} \\
\mathtt{o.sinkfld = w} & \mathbf{T} = \{\mathtt{z.g}, \mathtt{x.h}, \mathtt{y.f\ h}, \mathtt{w}\}
\end{tikzcd}$$

We highlight in red the access paths that would be missed by a forward dataflow analysis without on-demand alias-analysis capabilities, such as the IFDS framework [24]. Such an analysis would ignore the assignment $\mathtt{x = y.f}$ because it is not affected by $\mathbf{T}$, thereby missing the aliasing relation between `x.h` and `y.f.h` at the point when it becomes relevant, which is the following two statements: The first, $\mathtt{x.h = z.g}$, contaminates `x.h`, and thus also `y.f.h`, and the second dereferences `y.f.h` into `w`.

In constrast, ANDROMEDA is fully sound, as we prove in Theorem 1. ANDROMEDA handles cases such as the above by performing on-demand alias analysis. Upon encountering the field-assignment statement $\mathtt{x.h = z.g}$, ANDROMEDA traverses the control-flow graph backwards seeking aliases of `x.h`. It then finds that `y.f.h` is an alias of `x.h`, and propagates this additional security fact forward, which ensures that the security vulnerability is discovered. The ANDROMEDA propagation steps are visualized above using labeled edges, the label consisting of the step index (in square brackets) followed by the learned taint fact.

Formally, ANDROMEDA computes a fixpoint solution for the equations in Figure 3 while traversing the control flow backwards from the statement performing the heap update. The seeding value for $\mathbf{A}$ in our example is the singleton set $\{\mathtt{x}\}$.

Theorem 1. *The* ANDROMEDA *data-flow analysis algorithm is sound. That is, in the fixpoint solution $\mathcal{F}$ computed by* ANDROMEDA *for program P with respect to specification S, for every control location c in the program and set A of access paths arising in $\mathcal{F}$ c, $\gamma\ A \supseteq A'$, where A' is the set of all concrete access paths that may arise in c in an execution of P.*

*Proof (Sketch). First, we make the observation that our transformers (in Figure 2) are distributive (*i.e.*, $\tau\ X \cup Y \equiv \tau\ X \cup \tau\ Y$). This simplifies the proof by letting us consider singleton sets of access paths [24], making it clear that all the transformers not modifying the heap (*i.e.*, all transformers except* `x.f=y`*) are trivially sound. Finally, for field assignment, the backward equations (in Figure 3) guarantee that all aliases of* `x` *due to preceding statements (according to the control-flow order) are accounted for. The equations handle all possible cases, including forward and backward propagation due to field accesses, and thus result in a complete aliasing solution.*

3.3 Extensions: Library and Framework Modeling

Modern Web applications are often built atop one or more frameworks, such as Struts, Spring and JSF [29, 35]. Frameworks typically invoke application code using reflective constructs, based on information provided in external configuration files, which complicates static analysis of Web applications.

To address this concern, ANDROMEDA is fully integrated with Framework For Frameworks (F4F), a recent solution augmenting taint-analysis engines with precise framework support [29]. F4F automatically generates Web Application Framework Language (WAFL) static-analysis artifacts, which can be integrated into a taint-analysis engine to ensure that the interaction of a Web application with the frameworks it uses is modeled soundly and accurately.

ANDROMEDA's integration with F4F exploits the fact that static analysis can operate on non-executable yet legal Java code. We transform the F4F output into synthetic code that soundly models data flows involving framework code. This choice has several advantages compared to direct modeling of frameworks within the ANDROMEDA engine, being (i) more lightweight (no need to directly generate Intermediate-Representation (IR) code), (ii) more portable and reusable (the synthetic Java code generated from the WAFL specification can be plugged into *any* existing analysis), as well as (iii) more intelligible to the developer (being presented with simple Java code instead of IR code).

Before statically analyzing an application, ANDROMEDA takes the WAFL output of F4F and transforms it as follows. Each call replacement has a synthetic method associated with it. This is the method that ANDROMEDA should consider in place of the one specified in the application source code. For every synthetic method, ANDROMEDA creates Java code corresponding to the instructions for that synthetic method that are specified in the output of F4F. In most cases, this can be done straightforwardly. However, there were several interesting problems that we had to address.

One case is simulating method invocations from synthetic methods. Such invocations are on uninitialized variables, which causes ANDROMEDA's intra-procedural type inference to ignore them. Solving this by initializing the variables is problematic: Some declared types are abstract, and some do not have a default constructor. Instead, ANDROMEDA solves this problem by adding a level of indirection via a method call that returns `null`. Since the assignment to `null` is performed in a different procedure, ANDROMEDA's type inference accepts the call as valid, with a result sufficient to model taint propagation faithfully.

Another problem arises when synthetic methods invoke default-scope or protected methods in a class of another package. Since these methods can only be invoked from classes in the same package, ANDROMEDA extends that package with an additional public synthetic class containing a public synthetic method that calls the default-scope or protected method, and returns its return value. Being public and in the same package as the restricted method, this synthetic method can be invoked without restrictions.

4 Incremental Security Analysis

A key feature of ANDROMEDA is its ability to update the scan report incrementally following code changes. For industry-scale Web applications, this feature is of crucial importance. Without it, long waiting times need be spent on reanalysis of the entire application following any code change, which complicates the integration of security scanning into the development lifecycle. Moreover, incremental scanning allows verification of fixes on the spot, which makes for a fluent and rapid remediation process.

The design of ANDROMEDA, emphasizing local and demand-driven representation of the subject program, is geared toward incremental scanning. We have implemented this feature such that neither the soundness nor the accuracy of the analysis are lost in rescanning, which leaves responsiveness as the main challenge. We address this concern by combining several optimizations and algorithms, which are described in the remainder of this section.

4.1 Change Impact Analysis

ANDROMEDA's ability to respond to code changes efficiently is founded on a change-impact analysis (CIA) algorithm spanning all the layers of data structures comprising ANDROMEDA, from the type hierarchy, through the call graph, and up to the propagation graph. Upon receiving a notification that a given compilation unit (CU) has changed, the CIA algorithm compares its new version to the previous one, which it caches exactly for this purpose. By the end of the comparison, CUs where differences were found are marked as either modified or deleted or added.

ANDROMEDA then localizes the changes to determine what the bottommost layer they affect is. For example, if a class is marked as modified due to a change made in one of its methods, then ANDROMEDA will reason only about that method in the ensuing stages of the update process. Moreover, if the method has changed in a way that affects neither the call sites it declares nor the (intraprocedural) type-inference solution computed for it, then there are no implications with respect to the call graph, and the notification can immediately flow to the highest layer of the hierarchy, which is the propagation graph. This focusing strategy translates into a major optimization, whereby lower layers of the hierarchy can often be skipped.

4.2 Efficient Data Structures

Since the type hierarchy and call graph of ANDROMEDA are built in a local fashion, using intra-procedural type inference (*cf.* Section 3.1), change notifications arriving at these data structures can be handled efficiently. For example, if the call graph is notified that method m has changed, then only the subgraph rooted at m needs to be modified. Furthermore, call sites in m where the type-inference solution for the receiver remains unchanged can safely be preserved throughout the update process. The challenge is with the propagation graph, which records transitive information flows.

Following a code change, certain parts of the propagation graph are invalidated, but detecting the obsolete data-flow edges is difficult without additional bookkeeping, because they are due to transitive flow of information. To this end, similarly to [26], we use a *support graph*, which is an auxiliary graph structure documenting how edges in the propagation graph, henceforth referred to as taint facts, were formed. Similarly to [24], we distinguish between two types of edges in the propagation graph: *path edges* and *summary edges*. Path edges correspond to normal intraprocedural flow, whereas summary edges bridge across call sites. This implies two types of edges in the support graph:

- Normal edges are of the form $tf_1 \rightarrow tf_2$, where tf_i denotes a taint fact.
- Summary edges are of the form $\langle tf_{1,1}, tf_{1,2} \rangle \rightarrow \langle tf_{2,1}, tf_{2,2} \rangle$, with the interpretation that summary edge $tf_{2,1} \rightarrow tf_{2,2}$ in the caller was learned based on edge $tf_{1,1} \rightarrow tf_{1,2}$ in the callee.

When the propagation graph is notified of a change in a particular method, it establishes the set I of taint facts that can immediately be discarded based on the change. It then consults the support graph, which computes the transitive closure of the facts in I. Corresponding edges are then removed from the propagation graph, and the fixpoint iteration process is renewed by updating the IR of every changed method, and then searching for new seeds and extending existing path edges.

5 Empirical Evaluation

In this section, we describe the experiments we conducted to measure ANDROMEDA's accuracy, performance and incremental capabilities.

5.1 Experimental Setup

ANDROMEDA is a client of the WALA framework.[9] It is written in Java and implemented as an Eclipse plugin. We have conducted two sets of experiments to evaluate ANDROMEDA:

1. **Standard Analysis.** We measure ANDROMEDA's performance and accuracy by applying it to a suite of 16 benchmarks, including applications appearing in [33] and [17], as well as several contemporary commercial applications. Benchmark characteristics are provided at the leftmost columns of Figure 4. We compare ANDROMEDA with Taint Analysis for Java (TAJ) [33] on 8 common benchmarks.

[9] `http://wala.sf.net`.

TAJ, which is the most recent and advanced work on industrial taint analysis, is also a WALA client. The main difference is that TAJ utilizes whole-program pointer analysis, ensuring accuracy and scalability by enforcing *unsound* bounds. For scalability, TAJ uses a preset budget for call-graph and pointer-analysis construction. Similar bounds are used for accuracy (*e.g.*, filtering out witness flows beyond a given length).

2. **Incremental Scanning.** We measure average response time for reanalysis of two applications following several common code changes, such as deleting or adding a statement or a method.

We performed the experiments on a MacBook Pro laptop computer with a 2.66-GHz processor and 8 GB of RAM, running OS X V10.8 and Java Standard Edition Runtime Environment (JRE) V1.6.0_35 with 2.6GB of heap space.[10]

5.2 Standard Analysis

The results of the first experiment appear in Figure 4. To assess the accuracy of ANDROMEDA, a security expert sampled at random 10 findings per benchmark, and classified them as either true positive (TP), false positive (FP) or unknown. A finding was classified as *unknown* if there was missing source code (*e.g.*, if the flow goes through library code), or the flow was valid but of low exploitability. The TAJ data comes from the original TAJ paper [33].

The experimental data gives a clear indication of ANDROMEDA's high accuracy. Compared to TAJ (on 8 of the benchmarks), ANDROMEDA finds substantially more issues, reporting 578 findings compared to a total of 280 findings reported by TAJ. Moreover, ANDROMEDA's findings are more accurate on 4 of the 5 benchmarks where accuracy data is available for TAJ, the only exception being Webgoat. The accuracy statistics are summarized in Figure 6. For performance, ANDROMEDA's average running time (on the common benchmarks) is 114 seconds, whereas the average scanning time of TAJ is 112 seconds, which is almost identical.

Our analysis of the findings suggests that the combination of soundness and framework modeling allows ANDROMEDA to find more application entrypoints, as well as follow data flows through more parts of the application, compared to TAJ. These account for ANDROMEDA's ability to find more quality findings than TAJ while retaining a better signal-to-noise ratio.

For the entire suite, ANDROMEDA's accuracy statistics show an average of 53% TPs, 11% FPs and 36% unknowns. ANDROMEDA's average running time is 298 seconds (AppA being an outlier). These numbers point to ANDROMEDA's high precision, which comes at the reasonable cost of 5 minutes on average per scan.

5.3 Incremental Scanning

To measure ANDROMEDA's incremental features, we considered a set of common editing operations, including addition and deletion of statements and methods, which we

[10] The running times reported for TAJ are drawn from the original paper [33], where another execution environment, involving a Windows desktop machine, was used. Running-time comparisons should thus be considered with a grain of salt.

Benchmark	**Characteristics**			TAJ				ANDROMEDA			
	Version	**Class Count**	**Line Count**	**Findings**	**Time (s)**	**TP**	**FP**	**Findings**	**Time (s)**	**TP**	**FP**
AjaxChat	0.8.3	29	4147	-	-	-	-	14	30	70%	30%
AltoroJ	1.0	43	746	37	23	80%	20%	35	4	90%	10%
AppA	N/A	250	N/A	-	-	-	-	301	2555	20%	0%
Blojsom	3.1	254	19984	123	207	-	-	139	494	60%	30%
BlueBlog	1.0	38	650	12	6	50%	50%	13	16	100%	0%
Contineo	2.2.3	79	65744	-	-	-	-	228	573	90%	0%
Dlog	3.0-BETA-2	268	17229	6	221	-	-	30	51	60%	20%
Friki	2.1.1-58	35	2339	7	9	70%	20%	81	3	100%	0%
GestCV	1.0	124	107494	7	209	50%	50%	89	10	60%	10%
Ginp	1.0	73	387	49	28	-	-	122	62	40%	0%
JBoard	0.3	185	17500	-	-	-	-	74	330	10%	0%
JPetstore	2.5.6	116	25820	-	-	-	-	179	73	10%	0%
JugJobs	1.0	30	4815	-	-	-	-	39	32	60%	40%
Photov	2.1	239	210304	-	-	-	-	178	229	10%	0%
StrutsArticle	1.1	45	7897	-	-	-	-	25	35	10%	0%
Webgoat	5.1-20080213	192	17656	39	193	90%	10%	69	275	60%	40%

Fig. 4. Performance and Accuracy Results for TAJ and ANDROMEDA in Standard Scanning

Change Type	**Response Time (s)**			
	AltoroJ		Webgoat	
	Deletion	**Addition**	**Deletion**	**Addition**
Taint-propagator statement	2	2.2	1.9	2.2
Security sink	0.5	2	1.9	2.5
Security source	2.1	2.1	1.8	3.2
Irrelevant statement	1.9	2	2.5	2.8
Relevant method	2.2	1.9	1.8	2.7
Irrelevant method	2.2	1.7	1.7	1.7

Fig. 5. Response Times for Various Incremental Changes

	ANDROMEDA	TAJ
Average TPs	82%	68%
Average FPs	12%	30%
Average Unknowns	6%	2%

Fig. 6. Accuracy Statistics

classified according to the relevance of the statement or method to the solution computed by ANDROMEDA. A statement may either be a source, a sink, a taint propagator (participating in a vulnerable flow), or an operation lying oustide the ANDROMEDA data-flow solution. Similarly, a method may or may not participate in the solution.

We examined the effect of either adding or deleting a syntactic construct chosen from each of these 6 categories, which yielded 12 kinds of possible changes. For each change type, and each of the two benchmarks we used for this experiment, we applied the change 10 times. For each round, we chose a target at random from a pool of suitable candidates that we prepared in advance. The reported numbers are the average (in wall-clock seconds) across these 10 rounds.

The results of this evaluation are listed in Figure 5. Response times are largely within the range of 2-3 seconds per change, whereas the overall scanning time of Webgoat is 275 seconds. For AltoroJ, incremental scanning is less motivated, because analysis from scratch takes 4 seconds to complete. Still, the average response time for an incremental change in AltoroJ is 1.9 seconds, which is less than half of the time required for complete reanalysis. For Webgoat, a response is obtained after 2.2 seconds on average, which is less than 1% of the time needed for a fresh scan of this benchmark.

6 Related Work

There is a rich body of work on taint analysis. We here concentrate on static taint analysis, and refer the reader to [4, 22] for a survey of dynamic taint-analysis techniques. An detailed overview of works on program slicing is given in [30] and references therein.

The notion of *tainted variables* became known with the Perl language. Typically, the data manipulated by a program can be tagged with security levels [9], which assume a poset structure. Under certain conditions, this poset is a lattice [8]. Given a program, the principle of *noninterference* dictates that low-security behavior of the program be not affected by any high-security data, unless that high-security data has been previously downgraded [12]. Taint analysis is an information-flow problem in which high data is the untrusted output of a source, low-security operations are those performed by sinks, and untrusted data is downgraded by sanitizers.

Volpano *et al.* [34] show a type-based algorithm that certifies implicit and explicit flows and also guarantees noninterference. Shankar *et al.* present a taint analysis for C using a constraint-based type-inference engine based on `cqual` [27]. Similarly to the propagation graph built by ANDROMEDA, a constraint graph is constructed for a `cqual` program, and paths from tainted nodes to untainted nodes are flagged.

Myers' Java Information Flow (Jif) [20] uses type-based static analysis to track information flow. Based on the Decentralized Label Model [21], Jif considers all memory as a channel of information, which requires that every variable, field, and parameter used in the program be statically labeled. Labels can either be declared or inferred. Ashcraft and Engler [2] also use taint analysis to detect software attacks due to tainted variables. Their approach provides user-defined sanity checks to untaint potentially tainted variables. Pistoia *et al.* [23] present a static analysis to detect tainted variables in privilege-asserting code in access-control systems based on stack inspection.

Snelting *et al.* [28] make the observation that Program Dependence Graphs (PDGs) and noninterference are related in that $\mathit{dom}(s_1) \not\rightsquigarrow \mathit{dom}(s_2)$ implies $s_1 \notin \mathit{backslice}(s_2)$, where *backslice* is maps each statement s to its static backwards slice. Based on this observation, Hammer *et al.* [14] present an algorithm for verifying noninterference: For output statement s, $\mathit{backslice}(s)$ must contain only statements whose security label is lower than s. Though promising, this approach has not been shown to scale.

Livshits and Lam [17] analyze Java EE applications by tracking taint through heap-allocated objects. Their solution requires prior computation of Whaley and Lam's flow-insensitive, context-sensitive may-points-to analysis, based on Binary Decision Diagrams (BDDs) [38], which limits the scalability of the analysis [16]. The points-to relation is the same for the entire program ignoring control flow. By contrast, the PDG-based algorithm in [14] handles heap updates in a flow-sensitive manner, albeit at a much higher cost. Livshits and Lam's analysis requires programmer-supplied descriptors for sources, sinks and library methods dealing with taint carriers. Guarnieri *et al.* [13] present a taint analysis for JavaScript. Their work relies on Andersen's whole-program analysis [1]. While being sound, the analysis is not incremental, and has not been shown to scale to large programs.

Wassermann and Su extend Minamide's string-analysis algorithm [19] to syntactically isolate tainted substrings from untainted substrings in PHP applications. They label nonterminals in a context-free grammar with annotations reflecting taintedness and untaintedness. Their expensive yet elegant mechanism is applied to detect both SQLi [36] and XSS [37] vulnerabilities. Subsequent work by Tateishi *et al.* [32] enhances taint-analysis precision through a string analysis that automatically detects and classifies downgraders in the application scope.

McCamant and Ernst [18] take a quantitative approach to information flow: Instead of using taint analysis, they cast information-flow security to a network-flow-capacity problem, and describe a dynamic technique for measuring the amount of secret data that leaks to public observers.

ANDROMEDA's scalability stems from its demand-driven analysis strategy. Demand-driven pointer analysis was originally introduced by Heintze and Tardieu [15]. Since there have been several works on demand-driven points-to analysis via context-free-language reachability [31, 40, 39]. For taint analysis, our empirical data suggests that only a small fraction of a large program is expected to be influenced by source statements. Fuhrer *at al.* [11] take a demand-driven approach in replacing raw references to generic library classes with parameterized references. At a high level, this analysis resembles the alias analysis performed by ANDROMEDA, as constraints on type parameters are first propagated backwards to allocation sites and declarations, and from there they are propagated forward.

7 Conclusion

We have presented ANDROMEDA, a security-analysis algorithm featuring local, demand-driven tracking of vulnerable information flows. Thanks to this design choice, ANDROMEDA scales to large codes while being highly accurate, and additionally features incremental scanning capabilities. ANDROMEDA is part of a commercial product. Our experimental evaluation of ANDROMEDA, comparing it to a state-of-the-art scanner that sacrifices soundness for accuracy, shows ANDROMEDA to be favorable.

References

1. Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, University of Copenhagen, Copenhagen, Denmark, May 1994.
2. K. Ashcraft and D. Engler. Using Programmer-Written Compiler Extensions to Catch Security Holes. In *S&P*, 2002.
3. D. F. Bacon and P. F. Sweeney. Fast static analysis of c++ virtual function calls. In *OOPSLA*, pages 324–341, 1996.
4. W. Chang, B. Streiff, and C. Lin. Efficient and Extensible Security Enforcement Using Dynamic Data Flow Analysis. In *CCS*, 2008.
5. B. Cheng and W. W. Hwu. Modular interprocedural pointer analysis using access paths: design, implementation, and evaluation. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 57–69, 2000.
6. P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL*, pages 238–252, 1977.
7. J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of the 9th European Conference on Object-Oriented Programming*, ECOOP '95, pages 77–101, 1995.

8. D. E. Denning. A Lattice Model of Secure Information Flow. *CACM*, 19(5), 1976.
9. D. E. Denning and P. J. Denning. Certification of Programs for Secure Information Flow. *CACM*, 20(7), 1977.
10. A. Deutsch. A Storeless Model of Aliasing and Its Abstractions Using Finite Representations of Right-regular Equivalence Relations. In *ICCL*, 1992.
11. R. Fuhrer, F. Tip, A. Kieżun, J. Dolby, and M. Keller. Efficiently Refactoring Java Applications to Use Generic Libraries. In *ECOOP*, 2005.
12. J. A. Goguen and J. Meseguer. Security Policies and Security Models. In *S&P*, 1982.
13. S. Guarnieri, M. Pistoia, O. Tripp, J. Dolby, and S. Teilhet. Saving the World Wide Web from Vulnerable JavaScript. In *ISSTA*, 2011.
14. C. Hammer, J. Krinke, and G. Snelting. Information Flow Control for Java Based on Path Conditions in Dependence Graphs. In *S&P*, 2006.
15. N. Heintze and O. Tardieu. Demand-Driven Pointer Analysis. In *PLDI*, 2001.
16. O. Lhoták and L. J. Hendren. Context-Sensitive Points-to Analysis: Is It Worth It? In *CC*, 2006.
17. V. B. Livshits and M. S. Lam. Finding Security Vulnerabilities in Java Applications with Static Analysis. In *USENIX Security*, 2005.
18. S. McCamant and M. D. Ernst. Quantitative Information Flow as Network Flow Capacity. In *PLDI*, 2008.
19. Y. Minamide. Static Approximation of Dynamically Generated Web Pages. In *WWW*, 2005.
20. A. C. Myers. JFlow: Practical Mostly-static Information Flow Control. In *POPL*, 1999.
21. A. C. Myers and B. Liskov. A Decentralized Model for Information Flow Control. In *SOSP*, 1997.
22. J. Newsome and D. Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *NDSS*, 2005.
23. M. Pistoia, R. J. Flynn, L. Koved, and V. C. Sreedhar. Interprocedural Analysis for Privileged Code Placement and Tainted Variable Detection. In *ECOOP*, 2005.
24. T. Reps, S. Horwitz, and M. Sagiv. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *POPL*, 1995.
25. A. Sabelfeld and A. C. Myers. Language-based Information-flow Security. *IEEE Journal on Selected Areas in Communications*, 21:5–19, 2003.
26. D. Saha. *Incremental Evaluation of Tabled Logic Programs*. PhD thesis, State University of New York at Stony Brook, Stony Brook, NY, USA, 2006.
27. U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting Format String Vulnerabilities with Type Qualifiers. In *USENIX Security*, 2001.
28. G. Snelting, T. Robschink, and J. Krinke. Efficent Path Conditions in Dependence Graphs for Software Safety Analysis. *TOSEM*, 15(4), 2006.
29. M. Sridharan, S. Artzi, M. Pistoia, S. Guarnieri, O. Tripp, and R. Berg. F4F: Taint Analysis of Framework-based Web Applications. In *OOPSLA*, 2011.
30. M. Sridharan, S. J. Fink, and R. Bodík. Thin Slicing. In *PLDI*, 2007.
31. Manu Sridharan and Rastislav Bodík. Refinement-based Context-sensitive Points-to Analysis for Java. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2006)*, pages 387–400, Ottawa, ON, Canada, June 2006.
32. T. Tateishi, M. Pistoia, and O. Tripp. Path- and Index-sensitive String Analysis Based on Monadic Second-order Logic. In *ISSTA*, 2011.
33. O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman. TAJ: Effective Taint Analysis of Web Applications. In *PLDI*, 2009.
34. D. Volpano, C. Irvine, and G. Smith. A Sound Type System for Secure Flow Analysis. *JCS*, 4(2-3), 1996.
35. I. Vosloo and D. G. Kourie. Server-centric web frameworks: An overview. *ACM Comput. Surv.*, 40(2):4:1–4:33, 2008.

36. G. Wassermann and Z. Su. Sound and Precise Analysis of Web Applications for Injection Vulnerabilities. In *PLDI*, 2007.
37. G. Wassermann and Z. Su. Static Detection of Cross-site Scripting Vulnerabilities. In *ICSE 2008*, 2008.
38. J. Whaley and M. S. Lam. Cloning Based Context-Sensitive Pointer Alias Analysis Using Binary Decision Diagrams. In *PLDI*, 2004.
39. D. Yan, G. Xu, and A. Rountev. Demand-driven context-sensitive alias analysis for java. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pages 155–165, 2011.
40. X. Zheng and R. Rugina. Demand-driven alias analysis for c. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 197–208, 2008.