AFRL-RI-RS-TR-2018-160

JAVA ANALYSIS AT MASSIVE SCALE (JAMS)

RAYTHEON BBN TECHNOLOGIES

JUNE 2018

INTERIM TECHNICAL REPORT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

STINFO COPY

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report was cleared for public release by the 88th ABW, Wright-Patterson AFB Public Affairs Office and is available to the general public, including foreign nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-RS-TR-2018-160 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE CHIEF ENGINEER:

/ S /
SCOTT F. ADAMS
Work Unit Manager

/ S /
WARREN H. DEBANY, JR
Technical Advisor, Information
Exploitation and Operations Division
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
JUN 2018	INTERIM TECHNICAL REPORT	AUG 2016 – JUL 2017

4. TITLE AND SUBTITLE

JAVA ANALYSIS AT MASSIVE SCALE (JAMS)

5a. CONTRACT NUMBER
FA8750-15-C-0108

5b. GRANT NUMBER
N/A

5c. PROGRAM ELEMENT NUMBER
61101E

6. AUTHOR(S)

Alex Heinricher, Chris Willig

5d. PROJECT NUMBER
STAC

5e. TASK NUMBER
BB

5f. WORK UNIT NUMBER
NT

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Raytheon BBN Technologies Corp.
10 Moulton St.
Cambridge MA 02138

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Research Laboratory/RIGB
525 Brooks Road
Rome NY 13441-4505

10. SPONSOR/MONITOR'S ACRONYM(S)
AFRL/RI

11. SPONSOR/MONITOR'S REPORT NUMBER
AFRL-RI-RS-TR-2018-160

12. DISTRIBUTION AVAILABILITY STATEMENT

Approved for Public Release; Distribution Unlimited. PA# 88ABW-2018-2929
Date Cleared: 11 JUN 2018

13. SUPPLEMENTARY NOTES

14. ABSTRACT

The goal of JAMS was to improve static program analysis through the use of distributed computing techniques. A framework for executing reachability-based analyses on large-scale distributed computing clusters, and a series of client analyses to test the effectiveness of that framework, were developed. This report details the theoretical bases for the approach, the system architecture, and the experimental results.

15. SUBJECT TERMS
Java, program analysis, distributed systems, static analysis, security, malware analysis

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			SCOTT F. ADAMS
U	U	U	UU	26	19b. TELEPHONE NUMBER (*Include area code*)

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

1.1 JAMS Goals

The Java Analysis at Massive Scale (JAMS) seedling sought to improve the speed and maximum complexity of static program analysis, particularly graph reachability analysis (GRA), using off-the-shelf distributed processing techniques. In particular, the goals were:

- Build a framework for running distributed graph reachability analysis (dGRA).
- Measure the benefits/tradeoffs of dGRA versus existing GRA approaches.
- Explore the benefits of using a modular framework for GRA.

This report resulted from research sponsored by the Defense Advanced Research Projects Agency (DARPA) for the Space/Time Analysis for Cybersecurity (STAC) program, under Air Force Research Laboratory (AFRL) agreement number FA8750-15-C-0108. The views, opinions, and/or findings expressed are those of the author(s) and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

1.1.1 Arbitrary Scale through Arbitrary Hardware

Graph reachability analysis is one of the most powerful formal software analysis techniques available. However, it is famed for what is known as "combinatorial explosion" - relatively small increases in analysis precision and target program complexity result in huge increases in the size of the graph being analyzed. State-of-the-art approaches use a combination of highly-optimized implementation and clever limitations in analysis scope and precision to get as much out of one machine as possible. However, there will always be questions that such an approach can't answer, because it involves constructing a model too large for one machine to reason over in the time available, if it can be handled at all.

JAMS sought to remove this barrier using the same force that's driven recent advances in big data and machine learning: the cloud. Instead of avoiding the costs incurred by disk storage and network communication, we explored the possibility of a system that could scale to any formal analysis challenge, using a distributed computing cluster that could be resized to meet the needs of the problem at hand.

1.1.2 Cooperation through Modularity

The key strength of graph reachability is its flexibility. In software analysis, finding a link between two nodes in a graph can tell you how functions call each other, how data moves throughout the program, or if certain events happen in a required order, all depending on the data you use to build your graph.

Ordinarily, due to the heavy optimization required to handle complex software analysis tasks, a given GRA implementation focuses on, at most, a couple of questions. To use the results of one GRA-based analysis to benefit the other, the first analysis must run to completion before the second can begin.

Because JAMS would need a modular platform for testing GRA-based analyses, we explored the idea of running multiple GRA-based analyses cooperatively. Instead of waiting for its dependencies to complete, a given analysis can use intermediate results immediately as they are

generated, allowing more work to be complete per stage of the GRA algorithm, and thus reducing the total time needed to complete analysis.

1.2 Previous Work

JAMS sought to combine well-established fields that had not been readily explored together.

1.2.1 GRA-Based Program Analysis

Reachability-based program analysis is one of the most basic formal analysis techniques. The textbook algorithm (presented as chapter one of [1]), forms the foundation of the JAMS approach. A great deal of inspiration for this approach came from Massachusetts Institute of Technology's (MIT) DroidSafe tool [2], developed for the Defense Advanced Research Projects Agency (DARPA) Automated Program Analysis for Cybersecurity (APAC) program

1.2.2 The MapReduce Algorithm

The MapReduce algorithm was originally developed by Google, Inc. [3] as a simple and versatile framework for distributing and executing computations at arbitrary scales. MapReduce supports tasks that involve computing many partial results on small portions of a large data set (map), and then collating those results into a final answer (reduce). Today, MapReduce is extremely widely used, and is available either as stand-alone applications such as Apache's Hadoop and Spark, or as programming language libraries, such as Python's.

1.2.3 Parallel Program Analysis

We found remarkably little work in attempting to parallelize or distribute graph-reachability-based program analysis. Marcus Edvinsson [4], out of Linaeus University, Sweden, developed an algorithm for parallelizing GRA on a small scale, such as on a single desktop computer. In contrast to JAMS, Edvinsson's core assumption was that optimized traversal of a single directed-acyclic component of a static-single-assignment control flow graph could not be meaningfully improved through parallelism. Based on this assumption, he built a GRA system that identified isolated start nodes of the control flow graph, and explored their descendants in parallel. While extremely efficient, this approach is limited by the number of isolated paths through the program. When paths overlap, Edvinsson's algorithm duplicates work instead of spending time synchronizing state between tasks. Attempting to parallelize beyond a certain point gives no benefit, since there aren't enough independent subgraphs to share amongst the workers.

2. METHODOLOGY, ASSUMPTIONS, AND PROCEDURES

2.1 Assumptions

2.1.1 High Scale, High Risk

JAMS predicated its success on the idea that, given enough workers, one iteration of GRA can be parallelized to the point where the load on a single worker is trivial, or at least significantly lessened. However, the size of the working set for a program analysis task grows very rapidly with the complexity of the analysis being performed. We didn't know if or by how much distributed analysis could alter the asymptotic behavior of the algorithms to bring the answers to previously-intractable questions within reach.

2.1.2 Implementation versus Optimization

The goal of JAMS was to explore whether a distributed approach to formal analysis has any benefit over sequential or small-scale parallel approaches. To support that goal as best as possible, we decided to focus entirely on system construction and testing, and to avoid fine-tuning our use of the supporting frameworks. This means that we were unable to control for the following factors:

- **Framework Scalability:** Successfully scaling distributed systems is often more complicated than just adding more nodes.
- **Framework Overhead:** Using general-purpose tools means that we could not control for the costs of converting our data into common formats, or for safety checks that might not be relevant given our control over the input.
- **Cloud Platform Performance:** Since we used shared hardware, we didn't have guaranteed processor, disk, or network priority.
- **Locality of stored data:** The co-location of workers and their working sets is a difficult problem for JAMS (see 2.3.1), and is entirely dependent on the configuration of the data storage framework. Bad configurations could introduce significant inefficiency as workers need to use the network to look up non-local data.

We expected all of these factors to introduce overhead into our system. We took this expressly into account in our experiment design (see 2.3.3), and we point out several places where we could trace inefficiencies to these factors in our results discussion (see 3.1).

2.1.3 Scope of Program Analysis Tools

The main focus of JAMS was to examine the feasibility of dGRA algorithms, not to build an advanced program analysis platform. JAMS expressly ignored classically-difficult problems in Java Analysis:

- **Library Modeling:** We did not attempt to boost performance using simplified data flow models or stubs for libraries. JAMS performed naïve whole-program analyses, and chose manageable analysis targets rather than spending time forcing the Java Runtime or third-party libraries into manageable forms for analysis.

- **Native Code:** We did not intend to handle native code. While it would be entirely possible to use a dGRA system to analyze native code, indirection and the lack of semantics introduce a whole host of problems that we didn't want to address.
- **Algorithmic Optimization:** We used relatively primitive GRA techniques. While a lot of work has gone into refining and optimizing program analysis using GRA, we preferred to focus on exploring basic dGRA rather than implementing and verifying highly-optimized versions of the algorithm.

2.1.4 Cost of Execution

As imagined, JAMS would run on a lot of hardware. This meant either spending a lot of capital setting up an appropriately-sized computer cluster, or paying for time on someone else's. This gets at a critical ideal behind JAMS; with dGRA, the limitation on the questions you can answer should be the resources you are willing to spend getting that answer, rather than fundamental limits of the analysis system.

We did not expect dGRA systems like JAMS to be a complete replacement for fixed-scale program analysis systems. Classical single-machine solutions are perfectly capable of handling a certain scale of analysis problem. Given the overhead of running a distributed system, we didn't expect JAMS to compete directly with existing tools at this scale. Instead, we hoped to show that dGRA could scale further than traditional approaches, allowing it to tackle more complex problems.

2.2 Methodology

2.2.1 Distributing Graph Reachability

JAMS focused on a single formal analysis method: graph reachability. Graph Reachability Analysis (GRA) answers the question "Which nodes in a directed graph are reachable from a given set of starting nodes?" Depending on what the graph represents, you can answer a number of different questions; "what code blocks are reachable" or "where can data move inside a program" to name a few. The basic GRA algorithm is as follows:

1. Initialize a graph G = (V, E)
2. Let S, subset of V, be the pre-determined set of starting nodes.
3. Let set R, the current reachable nodes, contain all nodes in S.
4. Let set R' be empty.
5. For each directed edge e in E: If e.source is in R and e.sink is not in R, add e.sink to R'.
6. If R' is not empty: Add all nodes in R' to R, return to 5.

Normally, this algorithm is executed sequentially. Each edge e is examined one at a time, sometimes in specific patterns to improve the amount of work achieved per iteration of the algorithm. The problem with using a sequential approach for software analysis is that E can become extremely large. Complex programs will rapidly exceed millions of individual transfers, and high-resolution analyses may increase that by many orders of magnitude, duplicating nodes in order to express the impact of program state on execution. As a result of this precipitous growth, many meaningful questions are beyond the capacity of even the most powerful traditional computers.

The key idea behind JAMS is that each edge examination in step 5 is Hamiltonian: Any examination of an edge only depends on the data contained in R. Since R does not change during step 5, these edge examinations are completely parallelizable. The GRA algorithm now becomes extremely similar to a common parallel/distributed algorithm, MapReduce. Rephrasing the GRA algorithm in terms of a parallel or distributed system, we came up with the following.

1. Initialize a graph G = (V, E), partitioned across shared storage.
2. Let S, subset of V, be the pre-determined set of starting nodes.
3. Map all edges e in E in parallel:
    a. If e.source is in S, mark e.source uniquely.
    b. Apply all marks on e.source to e.sink
    c. If either e.source or e.sink had a new mark added, flag that node as changed.
4. Reduce each node v in V, in parallel:
    a. Merge all sets of marks on all copies of v
    b. If any copy of v was flagged as changed, flag the final copy as changed.
5. Write V, E back to shared storage.
6. If any node in V was flagged as changed, return to 3.

Our hypothesis was that, if given a large enough scale of parallelization and a complex enough problem, the benefit of splitting processing across multiple workers would exceed the increased time costs of communication and system maintenance. Furthermore, by moving away from structural optimizations, we hoped to remove the theoretical limits that plagued Edvinsson's approach. We discussed our hypotheses regarding the performance versus overhead tradeoff in 2.1.1, and our actual implementation of dGRA in 2.3.1.

2.2.2 Building on Existing Technology

Distributed systems engineering for cloud infrastructures is extremely difficult. Fortunately, a number of tools supporting different tasks in this space were reaching maturity. In order to focus on the goal of building and testing a dGRA framework, most of the JAMS system was built using open-source frameworks. We will discuss this more in section 2.3.2.

Additionally, we did not have dozens of cores' worth of distributed computing cluster readily available. In order to perform large-scale distributed processing experiments, we purchased time from commercial cloud services.

The risk in this approach was that we needed to compromise between our requirements, the use-cases supported by the frameworks and services available, and the maturity of those frameworks and services. We discuss our choices of frameworks in section 2.3.2.

2.3 Development Process

2.3.1 System Architecture: dGRA Framework

JAMS itself consists of four major components:

- A dGRA driver
- A data storage interface
- A Preprocessor

- One or more Analysis Modules

Our goal was to make each of these components as decoupled as possible, in order to minimize the risks from bad software choices. Each component communicates using a set of software interfaces defining the components themselves, and the JAMS data representation. If a component didn't meet the needs of the program, we could remove it and re-implement its interfaces with minimal effort.

2.3.1.1 Data Model

To simplify analysis, JAMS operates on an abstract graph representation of Java bytecode. Each basic block of the program is represented as a list of assignment expressions representing the data flow impact of the block, and a list of pointers to other blocks, representing the possible control paths out of the block.

The assignment expressions are simplified abstract syntax trees based on the Jimple intermediate representation. They take the form of a = op(B, C, …), where a is a storage reference – an object describing a direct or indirect memory reference, such as a variable, object field, or formal method parameter. B, C and others are expressions on storage references.

All basic blocks and their related artifacts are assigned a context, a list of string identifiers allowing different analysis modules to track their actions on a given artifact. This is especially useful for creating contextual copies of artifacts, where different clones of the base artifact represent different choices made by the analyses.

In addition to the basic blocks, JAMS also records state documents – data objects that can be attached to an artifact under a specific context. This allows analyses to store conclusions that don't directly impact the program structure.

The basic block is the primary unit of work in JAMS. Each parallel map task handles a fraction of the list of basic blocks; the more mappers, the smaller the apparent size of the graph from the perspective of one mapper and the more tractable the analysis.

The state documents are stored separately from the basic blocks. Because a single artifact – say, a variable – can be referenced from multiple blocks, there needs to be a separate data store to avoid having to synchronize state documents between all of those references. The map tasks lazily load state documents as needed, and add to their local copy based on the data available from the block they are currently processing. This creates multiple conflicting versions of the document; it's the job of the reducer tasks to collate all versions of each artifact modified by the mappers, and produce a single coherent version of the artifact and its state that can be written back to storage.

2.3.1.2 Preprocessor

Because JAMS operates on a high-level representation of a program, the first stage of our system must lift raw Java bytecode into that representation. The preprocessor is built on top of the Soot Java Optimization framework. Soot converts Java bytecode instructions into the Jimple intermediate representation language, and then exposes transformer interfaces that allow developers to write software to operate over that language. JAMS' transformers read each method body in the soot classpath, extract the basic blocks, and build the data transfers. These objects are then serialized, and saved to the permanent data store.

2.3.1.3 Analysis Modules

The actual program analysis in JAMS is performed by a series of pluggable analysis modules. Each module defines a map algorithm that analyzes a single basic block, and one or more reduce algorithms to combine the results of that analysis on different artifacts. The module also has control over how its associated state gets stored. This allows for complicated mappings from state documents to artifacts, or non-trivial serialization schemes, without having to expose the details of the data storage interface to the analysis itself.

When loaded into the system, the modules register with a central manager. The manager's responsibility is to check that all of each module's dependencies are loaded, and to determine the order in which the modules should be executed based on those dependencies.

2.3.1.4 dGRA Driver

The dGRA driver picks up where the preprocessor leaves off, and loads the serialized state and structure data into a form suitable for distributed processing. At the start of an iteration, all basic blocks get loaded from distributed storage and passed to their respective workers.

Each worker is responsible for performing the map operation over each basic block in its partition. For each block in the partition, the worker executes the mapper of each registered analysis module on that block, in order of module dependency.

The results of each map task are a list of key/value pairs, representing basic blocks and all artifacts in them that were changed by the mapper. These are then shuffled to separate reducer tasks based on their keys – all copies of the same artifact will be reduced together, resulting in a single copy at the conclusion of the reduce phase. As with the map tasks, the reduce tasks use the reducers from each analysis module in turn to compute the merge between two copies.

The reduced list is then fed to a second map phase, where any modified state documents attached to an artifact are saved to persistent storage. This map phase also filters out everything except the basic blocks, creating an updated version of the original data set loaded at the start of the iteration.

The final step is to perform a second reduce on the updated list of basic blocks. This checks which analysis modules modified each block, and reports a single bit vector indicating which modules were active this iteration. The updated list of blocks is then saved back to the database, and, if the vector indicates that there are still active modules in the system, the driver starts a new iteration.

2.3.2 Infrastructure Design: Building on Open Source

The dGRA framework represents only a tiny corner of the JAMS system. The tasks of deploying a distributed compute cluster, managing a distributed map-reduce application, and maintaining a distributed data store are all handled by third-party software.

2.3.2.1 Finding Hardware

In order to experiment with distributed software, you need a computer cluster on which to run it. To build such a cluster, we chose to use Amazon Web Services Elastic Compute Cloud (AWS EC2). EC2 allows you to rent virtual machines with varying resource limits and hardware priorities.

For our experiments, we hypothesized that forcing nodes in the distributed cluster to communicate over a network connection versus over a local bus would be inefficient, so we

chose an r3.8xlarge instance, with thirty-two cores, 244 Gigabytes (GB) of Random Access Memory (RAM), and 640 GB of local Solid State Drive (SSD) storage. Ideally, this would allow us to host multiple workers per instance. Our hypothesis was that hosting multiple workers on one machine eliminates network overhead, assuming the machine has enough resources to support those workers.

2.3.2.2 Deployment System

In order to avoid spending cloud dollars for correctness testing, JAMS needed to be deployment agnostic. We wanted to be able to deploy it on a single machine, a local network, or the cloud with minimal changes to the software or its configurations. To do this, we built our entire deployment process around Docker Swarm. Docker is a tool for deploying containerized software – it allows you to create a script that launches a program inside an isolated environment similar to a VM, while avoiding the overhead and heavier setup costs of a full guest operating system.

The Swarm extension aids in creating and connecting a large number of duplicate containers, as well as balancing them across multiple hosts in a network. This ability exactly supports our desired use-case of deploying JAMS in different cluster configurations by specifying the scale of a particular container type, we could effortlessly change the configuration of our system.

JAMS departs from Docker's normal use-case by treating containers as true virtual machines. Normally, a Docker Swarm is made up of containers each running a single service; if you need ten database servers and two web servers, you launch a total of twelve containers. For JAMS, we wanted to keep some services co-located, so we used a tool called supervisord that allowed us to launch multiple services within one container. Our different container types are as follows:

- **Master:** Cluster coordinator. Hosts the master node for the Spark cluster (see 2.3.2.3), and serves as a coordination point when collecting data from across the cluster.
- **Worker:** A single worker in our distributed cluster. Hosts a Spark worker (see 2.3.2.3), a Cassandra database server (2.3.2.4), and a Ganglia daemon (see 2.3.2.6).
- **Logstash/Elastic Search/Kibana (ELK):** Traditional docker containers supporting the ELK logging framework services (see 2.3.2.6).

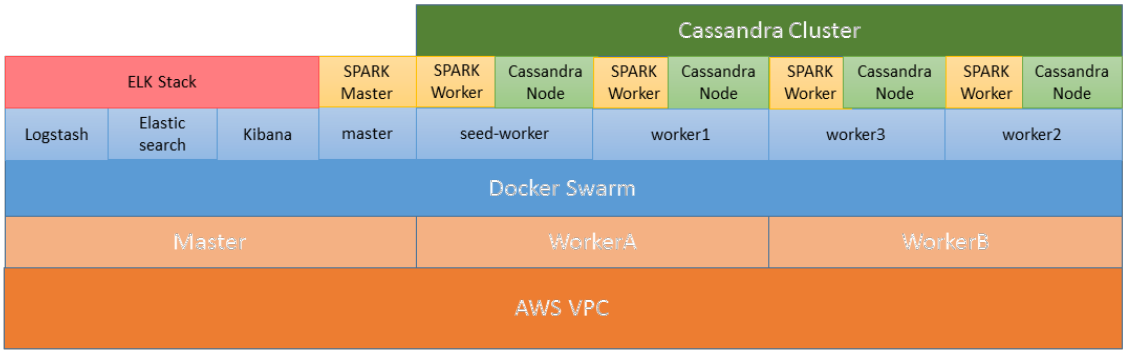

Figure 1: JAMS Cluster Architecture (VPC: Virtual Private Cloud)

A JAMS Swarm includes one master container, one each of the ELK-stack service containers, and at least one worker. One worker – termed the "seed worker" – gets booted ahead of the others to serve as a single point of contact for the other workers' Spark and Cassandra services to join the cluster.

2.3.2.3 Distributed Processing System

Because we based the dGRA algorithm off of MapReduce, there were two main choices of framework to actually host our distributed tasks. The first was Hadoop, the original MapReduce implementation; and Spark, a newer platform built on top of Hadoop.

As a strict improvement over MapReduce, Spark was the obvious choice. On the pure performance side, Spark includes a number of optimizations that are difficult to implement in Hadoop. Key among these is the idea of performing as much work as possible locally and in memory before sharing data between hosts, thus cutting down on overhead from disk input/output (I/O) and network communication compared to Hadoop.

Architecturally, Spark also had a number of benefits over Hadoop. Hadoop is designed for single-pass tasks; plug-in mapper and reducer classes get fed into a fixed Hadoop process that executes them in order, writes out the data, and then terminates. Spark, on the other hand, allows developers to write their own driver classes, exposing distributed data as Resilient Distributed Datasets (RDDs) that can be manipulated like any other Java data structure. Given that GRA is an iterative algorithm, building our own driver made implementing it far easier and more efficient than observing and re-launching a Hadoop process from an external script.

2.3.2.4 Database Cluster

When choosing which distributed data storage tool to use, there were four main criteria it had to meet. It had to support the JAMS data structures, have great performance, scale well, and it had to easily support data co-location. It also had to be Spark compatible. Our three main choices were Apache's Hadoop Distributed File System (HDFS), Cassandra, and MongoDB databases.

HDFS was designed expressly to work as a distributed file system for MapReduce systems, and supports all three of our criteria very well. The problem is that it was designed to distribute pieces of single documents, so it does not have good support for data sets made of large numbers of small arbitrary-sized documents. Its limitations make it unable to support the JAMS use case.

MongoDB is a lightweight non-Structured Query Language (NoSQL) (non-relational) database designed for document storage. It has excellent performance and can be distributed, but the size of a database cluster is difficult to configure, and its distribution algorithms are geared towards creating multiple identical replicas, not separate partitions of the same database.

Apache Cassandra is a distributed relational database. In addition to readily supporting our deployment and data structure needs, there was a third-party library for interfacing Cassandra with Spark's RDD loading process, making it the ideal choice of the three. Cassandra claimed to also support our performance requirements, but offered few details. We will discuss our own experiences in this area in section 3.1.

2.3.2.5 Dependency Management

In order to make JAMS as modular as possible, we decided to use a dependency injection pattern, where analysis modules and other system components are connected at runtime. Java comes packaged with the Contexts and Dependency Injection (CDI) framework, which provides

seamless dependency injection in its enterprise software framework. To use CDI without trying to run Spark in a web server, we used Apache's DeltaSpike tool, which allows the use of CDI without a full Java Enterprise Edition (EE) environment.

The benefit of this approach is that CDI, unlike other dependency injection libraries, requires no configuration whatsoever; it works purely off of Java introspection. The downside is that DeltaSpike is an extremely heavy-weight library, and required a steep learning curve to integrate unobtrusively (in its default configuration, DeltaSpike can generate gigabytes of log data).

2.3.2.6 Instrumentation Design: Gathering the data we need

Our key challenge in trying to measure system performance was that we wanted to focus on the timing of certain events that occur solely within the confines of our system without drastically impacting system performance. To that end, we did not want to use a traditional profiler; although accurate and purpose-built for our goals, we were not aware of any that would work on a distributed platform, and the ones we were aware of had such a heavy impact on system performance that they would not be useful for us.

We wound up writing what amounts to our own profiler using a technique we termed Semantic Logging. Using the log4j logging framework backed by the ELK stack, we built a library that allowed us to send custom messages marking the start and end of arbitrary tasks. We included these.

The benefit of this approach is that we could instrument our own system events, such as single analysis module executions or entire analysis ticks, which don't have a clear analog in any of the frameworks we used. Additionally, log4j is optimized to have almost no impact on its host program, allowing for zero-profile profiling.

The downside is that we could not measure the costs of events outside our control. The greatest of these "blind spots" is when Spark shuffles data between map and reduce tasks. However, we can estimate these costs based on the missing time between the phases we did control.

In addition, we experimented with using Ganglia, a distributed system monitoring tool, to measure load on different system resources, such as Central Processing Unit (CPU) and RAM.

2.3.3 Experiment Design

2.3.3.1 Experimental Data Set

In a twist of good luck, the previous work performed by Edvinsson used a large list of open-source Java programs and libraries as its test set. In order to save ourselves work in vetting test programs, and to allow nearly-direct comparisons with existing work, we decided to use as many of these programs as we could find.

We ran into two complications with this approach. The first was that, in the years since Edvinsson was published, the test projects had gone through several versions. We decided that, since our approach was already too dissimilar to make a scientifically-precise comparison to Edvinsson's results, we would use the more-readily-available current releases of the applications.

The second was that a number of the programs proved difficult to feed into our preprocessor. In order for Soot to meaningfully process a Java program, it needs access to all the program's dependencies. Rather than spend a lot of time understanding complex and undocumented build configurations, we chose a subset of the test programs that we knew our platform could handle.

We also added two test applications of our own. The first, jams-example, is a very small (11 basic block) program we constructed specifically to test the correctness of our analyses. The program is a series of primitive data and object manipulations designed to cover code structures significant to our analyses. To keep things as simple as possible, the program does not reference any external libraries.

The second program we chose to add was the Soot Java Analysis Framework. Aside from Jython, most of the applications used by Edvinsson were relatively small. Since larger programs are generally easier to examine through timing-based analyses, and because Jython proved difficult to process in early tests, we added Soot to give us another large test app.

The list is as follows:

- Jams-example
- Antlr-4.6
- Emma-2.0.5312
- Jython-2.7.0
- Sablecc-3.7
- Soot-3.0.0

2.3.3.2 dGRA Scaling Tests

Our first goal is to measure exactly how far we can scale a dGRA-based system. To measure this, we will build a primitive intraprocedural data taint analysis tool.

- **Wall-Clock Time:** Time to run the analysis from end-to-end. Gathered from Spark, logging frameworks.
- **Performance Multiplier:** Ratio of Wall-Clock Time for a reference configuration to Wall-Clock Time for the experimental configuration.
- **Overhead Ratio:** Ratio of number of workers in the cluster (an ideal maximum performance multiplier for a given cluster) to the Performance Multiplier of that configuration.

As stated in 2.1.1, we expect our prototype tool to suffer from a great deal of computational overhead, so we do not expect to compare Wall-Clock Time directly to existing work. Instead, we are focusing on Performance Multiplier. As described in 1.2.1, Edvinsson's approach suffered from a theoretical maximum performance multiplier due to limitations in partitioning their working set for parallel operations. The JAMS approach has no such limitation, so our hope for this project is to create a tool that can achieve a better maximum multiplier than past approaches.

We hoped to present data on CPU and RAM usage in order to analyze what fraction of the cluster resources each experiment used. However, we discovered late in the program that Ganglia cannot correctly monitor resource usage for multiple Docker containers on the same host. Since Ganglia references the host's hardware directly, running one Ganglia daemon per container lead to over- and – in some cases – under-counting that we could not trace accurately enough to correct.

2.3.3.3 Cooperative Analysis Tests

Compared to measuring the performance of a distributed system, testing the cooperative analysis hypothesis is extremely simple. The goal of running cooperative analyses is to reduce the number of GRA iterations the analyses need to reach a fixed-point, so this experiment focuses solely on measuring that reduction.

To create a chain of cooperative analyses, we wrote a pair of interdependent analysis modules implementing the following analyses:

- **Points-To Analysis:** Starting at object allocation sites – wherever the "new" operator gets used in the code – trace the handoff of objects from one storage location to another. In a way, this is a simpler version of taint analysis; each allocation site defines a unique source of taint, and object references are passed between storage locations by simple assignments.
  When run on its own, the Points-To analysis module is only capable of analyzing inside individual methods. To reason over the whole program, this analysis must take advantage of the inter-procedural links established by the call graph analysis.
- **Call Graph Construction:** Starting at the entrypoint method, link callsites and their arguments to specific methods and their parameters. As each link is established, the module builds an approximation of the current call-stack, or an execution context. Every basic block in the called method – and the storage references it contains – is cloned for each context under which it's executed, representing the possibility of different behavior when a method gets called from different places.
  Because of polymorphism, all non-static method calls in Java are indirect; you can't tell exactly which method is getting called until you know the exact class of the receiver object. As such, Call Graph Construction depends heavily on Points-To Analysis to provide possible receiver object types.

Additionally, the basic taint analysis was upgraded to use the context data provided by the call graph analysis, improving the sensitivity of the results and allowing taint to pass between methods.

To create a reference data set, we performed two analysis runs on each test program using the same database. The first pass exercised the Call Graph Construction and Points-To Analysis modules and saved their results to the database. The second pass exercised the improved Taint Analysis module, using results from the previous GRA pass to complete its work. For each pass, we recorded the number of GRA iterations needed to reach a fixed point.

The experimental run involved running all three analyses as part of the same GRA pass. We noted the number of iterations needed to reach a fixed point, and compared it to the sum of the iterations from the reference analysis. Ideally, the number of iterations for the experimental run should be less than the total for the reference analysis.

3. RESULTS AND DISCUSSION

3.1 dGRA Scaling Results

Our scaling experiments clearly demonstrated that distributing GRA can provide a benefit. As shown in Table 1, all test applications performed better with increased parallelism:

Table 1: Scaling Results

Application	Base Time (min.)	Best Time (min.)	Best Config. (Hosts x Workers)	Multiplier
Soot-3.0.0	25	9.7	8x9, 16x17 (tie)	2.58
Antlr-4.6	8.2	3.3	8x9	2.48
Sablecc-3.7	1.6	1	8x9	1.6
Emma-2.0.5312	4.1	1.9	16x17	2.15
Jython-2.7.0	29	13	8x9, 16x17 (tie)	2.23

All applications showed two definite trends. One was that performance increased with the number of instances hosting a given number of workers. This suggested that local overhead or resource starvation outweighed the overhead from network communications.

The second trend was that increasing the number of workers would improve performance to a point, after which. This suggests a communications bottleneck somewhere in our system that does not come into play until a certain level of parallelism.

In examining the fine-grained timing data, it turned out that the overwhelming majority of our execution time was spent writing to Cassandra. As shown in Figure 2, the vast majority of time spent in one iteration of the GRA algorithm is spent on writing the basic blocks out to the Cassandra data store. The only other noticeable time sinks were the map and consolidate phases, both of which also involve reads and writes to Cassandra, to load and save state documents. Their 9% represents about one minute of work out of twelve minutes for the entire iteration.

It is interesting to note that there was no unaccounted time. If the Spark framework incurred overhead when shuffling data between workers, it would have increased the time for the iteration without increasing the cost of any monitored phase. This means that the impact of distribution itself is negligible.

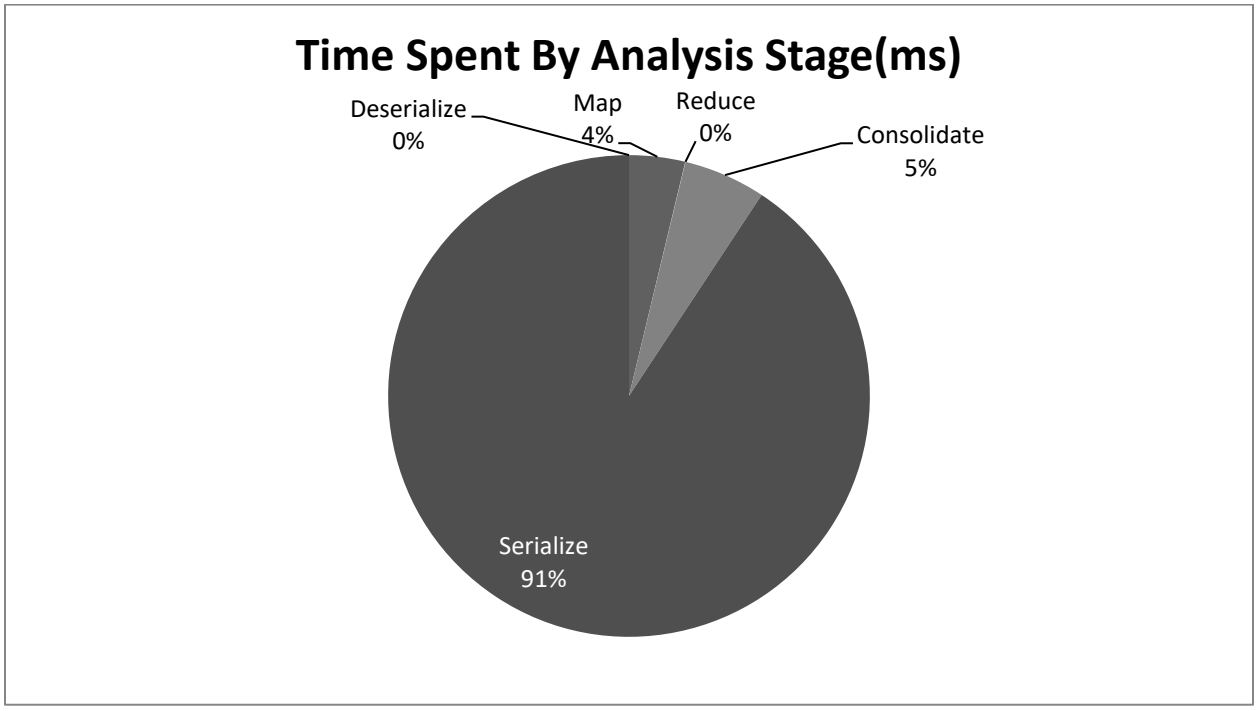

Figure 2: Time Spent By Analysis Stage (ms: milliseconds)

Additionally, we suspect that inefficiencies in our Cassandra configuration are the reason for both major trends in our scaling data. We could not run experiments with more than thirty-three workers because the Cassandra cluster would fail while writing the program structure. This initially confused us, because Cassandra's claim to fame is scaling to tens of thousands of nodes. However, reading more about these examples showed that they involved extremely careful communication configurations between the nodes in the Cassandra cluster. Since we used an extremely naïve method for establishing connectivity between nodes, we suspect that our configuration could not keep up with the demands placed upon it.

We would have liked to have presented CPU and RAM metrics to show how much of the hardware a given configuration used, but we discovered that we had an error in our instrumentation after our experiments were complete. Ganglia's CPU and RAM reporting are not reliable when run inside of docker containers; they measure the usage of the entire system, leading to over-counting if there are multiple docker containers on one machine. We could not find a way to undo Ganglia's errors with any confidence, and did not want to redo all experiments.

3.2 Cooperative Analysis Results

The experimental run took 24 iterations to complete all three analyses on jams-example. Our baseline implementation took 19 iterations to finish Call Graph and Points-to analysis, and a further 11 iterations to complete taint analysis, for a total of 30 iterations, so by sharing iterative cycles between analyses, we saved six iterations, proving our hypothesis.

This test involved an 11-block program, less than 100 lines of code. Since the majority of JAMS' costs come from moving data between workers, using tandem analysis to reduce analysis cycles would probably give a bigger benefit for more complicated programs.

4. CONCLUSIONS

4.1 Success of System

The JAMS system proved that distributing GRA-based program analyses can provide a benefit. Across all test apps, distributed computation showed a distinct improvement over a base configuration. However, our results show mixed improvements when compared to existing approaches.

JAMS may not be constrained by structural limitations, but we do suffer from the predictable massive overhead of distributed computation. Because we focused on implementation instead of optimization, we can't tell from these experiments if a distributed analysis system could ever become competitive with single-machine platforms, or whether dGRA could significantly change the asymptotic behavior of an analysis. Our current results showed that performance improved logarithmically with the number of workers in the cluster. However, we cannot tell if that trend is innate to distributed algorithms, or due to our choice of supporting software. In particular, we can't tell if our returns diminished because of distributed communications in general, or because of poor performance from our naïve Spark and Cassandra configurations.

Of separate interest is the benefit discovered from having multiple algorithms work together in one reachability computation. This concept isn't tied to distributed program analysis; it could also be used to improve other reachability-based analysis platforms.

4.2 Examination of Assumptions

As stated above, our focus on implementing our framework and leveraging open source software instead of precisely optimizing our approach proved to be a tradeoff. It limited the value of our ultimate results, since we incurred a number of large sources of overhead that added uncontrolled variables to our data. However, it made our problem tractable. Without Spark, Cassandra, and our other open-source components, it is extremely unlikely we could have completed JAMS on time, and we would have left ourselves open to a host of extra risks from trying to correctly implement and optimize complex software from scratch. Now that JAMS has proven the initial feasibility of dGRA, future efforts can go back and control for the costs that we could not address on this program.

Another assumption we definitely proved incorrect was the benefit of hosting multiple workers on large hosts. We could not identify the contested resources responsible for starving the workers, so we can't say if our workers required large amounts of standard resources such as CPU or RAM, if we were overloading more basic parts of the system such as communications busses, or if running multiple Cassandra servers per host was simply too expensive regardless of the workload on the host.

Because we had relatively little experimental control, it's difficult to say whether using basic blocks as the unit of work for map tasks was a good idea. In Java programs, basic blocks tend to be extremely small due to the reliance on indirect and exceptional control flow. How this impacted JAMS' performance, and how this performance would change for different languages, is still unknown.

4.3 Avenues for Further Work

4.3.1 Refine Data Storage Architecture

As stated in 3.1, a huge amount of overhead came from using Cassandra as a data storage engine. In hindsight, Cassandra was not the best choice of data storage solution.

JAMS' transactional model is extremely simple; each document will be written at most once per iteration after all reads are complete, so there is no risk of conflicting reads and writes. Cassandra focuses on providing transactional guarantees – protecting against conflicting reads and writes, especially in the case of redundant replicas of the database.

What we really needed is something closer to a distributed file system or a very simple shared no-sql database. We had originally ignored MongoDB because of its aggressive load balancer. However, we now know this can be disabled, preventing costly data moves and allowing us to experiment with our own data locality algorithms (see 4.3.2).

4.3.2 Explore Locality of Reference

One source of overhead in JAMS is the time needed to load data from one host in the distributed cluster onto another. This happens if a map task needs to reference a part of the state or structure that was partitioned onto another server. While Spark's partition-based mapping helps keep basic blocks local, JAMS currently does nothing to ensure that state documents are stored locally to the blocks that need them, or that blocks which reference similar state are stored together. With an improved data storage architecture, it would be interesting to see how much benefit a good locality algorithm could bring to dGRA.

4.3.3 Explore New Cluster Architectures

We tested JAMS under one cluster architecture: a few high-capacity machines hosting multiple workers each. Our experiments proved that this isn't the most efficient configuration for our current implementation. Since resource starvation contributed more to overhead than network communication, it makes sense to explore a cluster configuration where each host is scoped appropriately for a single worker. Determining this "appropriate scope" would also tell us exactly where our system ran into resource starvation issues, which we were unable to determine in this round of experiments.

4.3.4 Beyond Reachability

While graph reachability is a powerful technique for program analysis, it isn't the only technique being researched. Symbolic execution engines use a combination of graph-based value analysis and expression satisfiability to track the conditions under which execution can reach a particular state. Satisfiability solvers are extremely computationally-intensive, and are often implemented to take advantage of multi-core processors or GPUs. With performance improvements to the JAMS system, it could serve as a platform to expand satisfiability-based analysis even further, with each host in the JAMS cluster running its own Satisfiability Modulo Theories (SMT) solver.

4.3.5 Native Code

We chose to focus on Java on the JAMS program because it presents a tantalizingly easy target for program analysis. An obvious next step would be to alter JAMS to work on machine code. The basic idea of performing reachability analysis on a semantic program representation isn't

novel to JAMS; we borrowed the idea from the Binary Analysis Platform (BAP) framework [5], which is expressly designed to work on native code. There are a few challenges that would need to be overcome before JAMS could operate on assembly:

- New Structural Representation: Several components of JAMS' program representation are specific to Java. Most significantly, JAMS' representation of indirect control and data transfers are specifically tailored to use Java's object references. These constructs would need to be redesigned to fit a language with pointer-based indirection.
- New Preprocessor: The Soot framework only handles Java bytecode; we would need to identify a new disassembly/code analysis framework to help convert native code into the JAMS representation language.
- Improved Analysis: Tracing data through a native application is far more difficult than it is in Java. Resolving pointer arithmetic requires value analysis to track the actual arithmetic manipulation of referenced addresses, as well as an efficient means of representing the program's memory space. While the basic dGRA framework would still be valid, new analysis modules would be needed to perform the more complex indirection analysis. JAMS may have an advantage in this area; its use of disk in place of main memory may allow for more verbose and precise memory models than are possible in a memory-constrained system.

5. REFERENCES

[1] F. Nielson, Principals of Program Analysis, Berlin: Springer, 2005.

[2] M. Gordon, "Information-Flow Analysis of Android Applications in DroidSafe," in *NDSS*, 2015.

[3] J. Dean, "MapReduce: Simplified Data Processing on Large Clusters," Google, Inc., 2004.

[4] M. Edvinsson, "Parallel Reachability and Escape Analyses," in *NDSS*, 2010.

[5] D. Brumley, "BAP: A binary analysis platform," in *International Conference on Computer Aided Verification*, Berlin, 2011.

APPENDIX: FULL SCALABILITY RESULTS

The following tables show the complete experimental timing results for the dGRA scaling tests. Each row indicates a different number of hosts, while each column is a different number of worker containers. Results are presented in minutes. Blank cells represent experiments that failed because the preprocessor could not successfully connect to the Cassandra cluster.

Table A 1: Soot-3.0.0

	2	3	5	9	17	33	65
1	25	17	14	18	21	N/A	N/A
2	N/A	16	13	14	19	46	N/A
4	N/A	N/A	11	11	15	29	
8	N/A	N/A	N/A	9.7	12	23	44
16	N/A	N/A	N/A	N/A	9.7	16	34
32	N/A	N/A	N/A	N/A	N/A	13	29
64	N/A	N/A	N/A	N/A	N/A	N/A	

Table A 2: Antlr-4.6

	2	3	5	9	17	33	65
1	8.2	6.2	4.8	5.2	6.8	N/A	N/A
2	N/A	5.7	5	5.1	7.3		N/A
4	N/A	N/A	4.6	3.6	4.4	8.5	
8	N/A	N/A	N/A	3.7	4.8	7.6	15
16	N/A	N/A	N/A	N/A	3.3	6.8	11
32	N/A	N/A	N/A	N/A	N/A	5.7	8.7
64	N/A	N/A	N/A	N/A	N/A	N/A	

Table A 3: Sabblecc-3.7

	2	3	5	9	17	33	65
1	1.6	1.5	1.5	1.3	1.8	N/A	N/A
2	N/A	1.5	1.3	1.1	1.3	2.4	N/A
4	N/A	N/A	1.3	1	1.1	1.8	
8	N/A	N/A	N/A	1	1.3	1.9	1.1
16	N/A	N/A	N/A	N/A	1.1	1.4	2.5
32	N/A	N/A	N/A	N/A	N/A	1.2	2.5
64	N/A	N/A	N/A	N/A	N/A	N/A	

Table A 4: emma-2.0.5312

	1	2	4	8	16	32	64
1	4.1	3.7	3.4	2	2.8	N/A	N/A
2	N/A	3.3	2.9	1.9	2	4.2	N/A
4	N/A	N/A	2.7	1.8	1.9	3.1	
8	N/A	N/A	N/A	2	2.2	2.9	
16	N/A	N/A	N/A	N/A	1.9	2.4	4.3
32	N/A	N/A	N/A	N/A	N/A	2	3.8
64	N/A	N/A	N/A	N/A	N/A	N/A	

Table A 5: Jython-2.7

	2	3	5	9	17	33	65
1	29	20	18	20	28	N/A	N/A
2	N/A	19	16	16	23		N/A
4	N/A	N/A	16	16	19	36	
8	N/A	N/A	N/A	13	16	26	60
16	N/A	N/A	N/A	N/A	13	19	41
32	N/A	N/A	N/A	N/A	N/A	17	34
64	N/A	N/A	N/A	N/A	N/A	N/A	

GLOSSARY/ACRONYMS

- **AFRL:** Air Force Research Laboratory
- **APAC:** Automated Program Analysis for Cybersecurity, DARPA program
- **AWS:** Amazon Web Services
- **BAP:** Binary Analysis Platform
- **Basic Block:** A section of computer code where control flow neither branches nor merges.
- **Call Graph:** A graph that shows how the execution of a computer program can progress from one procedure to another, separated by specific "call" instructions.
- **CDI:** Contexts and Dependency Injection
- **Control Flow Graph:** A graph that shows how the execution of a computer program can progress from one basic block to another.
- **CPU:** Central Processing Unit
- **DARPA:** Defense Advanced Research Projects Agency
- **DB:** database
- **dGRA:** Distributed Graph Reachability Analysis. A modified version of Graph Reachability Analysis developed on the JAMS program that can be performed as a distributed computation.
- **EC2:** Elastic Compute Cloud
- **ELK:** Logstash/Elastic Search/Kibana logging framework services
- **GB:** Gigabytes
- **GRA:** Graph Reachability Analysis. Also known as Reaching Definition Analysis or Reachability Analysis, GRA seeks to determine which nodes in a graph can be reached from a set of defined starting nodes.
- **HDFS:** Hadoop Distributed File System
- **I/O:** input/output
- **Iteration**: One step in the GRA algorithm. GRA and dGRA work by repeatedly analyzing a graph until no additional information can be gathered. An iteration is one repetition of that analysis.
- **Java EE**: Java Platform, Enterprise Edition
- **MIT:** Massachusetts Institute of Technology
- **N/A:** Not Applicable
- **NoSQL:** non-Structured Query Language
- **RAM:** Random Access Memory
- **RDD:** Resilient Distributed Dataset
- **SMT:** Satisfiability Modulo Theories
- **SQL:** Structured Query Language
- **SSD:** Solid State Drive
- **STAC:** Space/Time Analysis for Cybersecurity, DARPA program
- **VPC:** Virtual Private Cloud