CRUCIBLE: Towards Unified Secure On- and Off-Line Analytics at Scale

Peter Coetzee, Stephen Jarvis
Performance Computing and Visualisation Group
University of Warwick
Coventry, United Kingdom
p.l.coetzee@warwick.ac.uk,
s.a.jarvis@warwick.ac.uk

ABSTRACT

The burgeoning field of data science benefits from the application of a variety of analytic models and techniques to the oft-cited problems of large volume, high velocity data rates, and significant variety in data structure and semantics. Many approaches make use of common analytic techniques in either a streaming or batch processing paradigm.

This paper presents progress in developing a framework for the analysis of large-scale datasets using both of these pools of techniques in a unified manner. This includes: (1) a Domain Specific Language (DSL) for describing analyses as a set of Communicating Sequential Processes, fully integrated with the Java type system, including an Integrated Development Environment (IDE) and a compiler which builds idiomatic Java; (2) a runtime model for execution of an analytic in both streaming and batch environments; and (3) a novel approach to automated management of cell-level security labels, applied uniformly across all runtimes.

The paper concludes with a demonstration of the successful use of this system with a sample workload developed in (1), and an analysis of the performance characteristics of each of the runtimes described in (2).

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Distributed programming, Parallel programming*;
D.2.11 [**Software Engineering**]: Software Architectures—*Domain-specific architectures*;
D.3.2 [**Language Classifications**]: Concurrent, distributed, and parallel languages; Data-flow languages

General Terms

Algorithms, Languages, Security

DISCS-2013 November 18, 2013, Denver, CO, USA

1. INTRODUCTION

Data Scientists have an increasingly challenging role to play in generating valuable insights across a variety of business and research areas. They must make sense of a broader variety of data than ever before, on an unprecedented scale. This has motivated significant advances in areas pertaining to analytics, from streaming analysis engines such as IBM's InfoSphere Streams, Backtype's Storm or Yahoo!'s S4 to an ecosystem of products built on the MapReduce framework.

When undertaking a piece of analysis, specialists are typically faced with a difficult decision: do they (1) opt to receive continuous insight, but limit their capabilities to a functional or agent-oriented streaming architecture; (2) make use of the bulk data behemoth, MapReduce, but risk their batch analyses taking hours or even days to return responses; or (3) go to the effort of maintaining code targeting both? If they are required to support multiple methodologies, they are faced with an enhanced engineering challenge: ensuring that the analysis they perform is both correct and equivalent on all platforms. These issues are further complicated by deployment scenarios involving multi-tenant cloud systems, or environments with strict access control requirements for data.

Our research seeks to alleviate many of the pain points highlighted above, by demonstrating a Domain Specific Language (DSL) and associated runtime environments: CRUCIBLE. We show how the CRUCIBLE system (named after the containers used in chemistry for high-energy reactions) can be used across multiple data sources to perform highly parallel distributed analyses of data simultaneously in both streaming and batch contexts, efficiently delivering integrated results whilst making best use of existing cloud infrastructure.

Specifically, the contributions of this work are:

- A high-level DSL and associated Integrated Development Environment (IDE) Tooling. This is the first known DSL to be specifically engineered to target both on- and off-line analytics with equal precedence;
- A suite of runtime environments providing consistent execution semantics across on- and off-line data, making use of best-in-class existing backends;
- A framework for managing cell-level security consistently across runtime environments, enabling the security conscious data scientist to trivially manage data visibility using an easily mastered labelling paradigm.

The remainder of this paper is structured as follows: Section 2 presents a summary of related work; Section 3 introduces

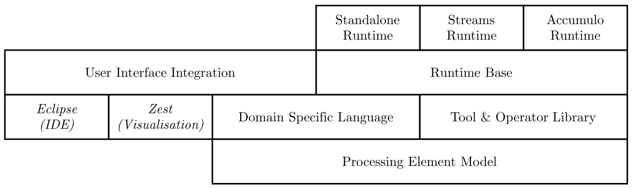

Figure 1: CRUCIBLE Package Layer Cake, with Development Environment on the left and Runtime on the right. Entries in *italics* are external library dependencies.

the CRUCIBLE system and describes its abstract execution model; Section 4 offers a detailed examination of the security labelling approach in CRUCIBLE; Sections 5 and 6 make this abstract model concrete through the introduction of three runtime environments and a standard library; and Section 7 gives a performance analysis of the CRUCIBLE runtimes. Sections 8 and 9 provide some avenues for furthering this research, and conclude the paper.

2. RELATED WORK

Large-scale data warehousing technologies abound in the literature; many of them are inspired by MapReduce [6] and Bigtable [4], a suite of Google technologies which spawned Hadoop [18] and HBase[1]. In parallel, NSA developed Accumulo[2], which added cell-level security, increased fault tolerance through the FATE framework, and a novel server-side processing paradigm [8], called Iterators.

Tools such as Google's Drill [10], and the Apache Software Foundation implementation Dremel [13], promise SQL-like interactive querying over these Bigtable-backed frameworks. Hive [21] and Pig [16] both aim to permit definition of analytics over arbitrarily formatted data in Hadoop [18] by inventing new query languages (with some definitions required in Java). Cascading[3] takes a slightly more engineer-centric approach to definition of analytics over Hadoop.

Perhaps the most common products in the streaming analytics space are IBM's InfoSphere Streams [17], and the open source Storm[4], developed by BackType. Others include Yahoo!'s S4 [14] (now an Apache Incubator project), offering an agent-based programming model; Esper[5] for Java and .NET development; and Microsoft's StreamInsight [2].

Recent research has begun to look at the problem of uplifting offline analytics into the realm of online processing. SAMOA [5] aims to resolve issues around Machine Learning using a streaming processing paradigm. AT&T Research, as part of their Darkstar project [11], have constructed a hybrid stream data warehouse solution, DataDepot [9]. The closest research to CRUCIBLE to date has been in IBM's DEDUCE [12] work, which looked at defining a Mapper and Reducer for MapReduce using SPADE, the programming language used in early versions of InfoSphere Streams.

CRUCIBLE builds on the most desirable attributes of these approaches in order to offer a single engineer-friendly platform for developing secure analytics to be deployed on state of the art multi-tenancy on- and off-line data processing platforms.

[1]`http://hbase.apache.org/`
[2]`http://accumulo.apache.org/`
[3]`http://www.cascading.org/`
[4]`http://storm-project.net/`
[5]`http://esper.codehaus.org/`

```
package eg.gen
import crucible.lib.pe.TimedEmitter
import crucible.lib.pe.FileSink
/**
 * Generate tick tuples
 */
process Generator extends TimedEmitter {
   conf : long Frequency = 10; // ms
   output : Timer
}
/**
 * Write the count of seen tuples
 * to a FileSink, and emit global count
 */
process CountingWriter extends FileSink {
   conf : Filename = '/nfs/tmp/count.txt'
   state : {
      local int seen = 0
      int allSeen = 0
   }
   outputs : [Write, Other];
   input : {
      Generator.Timer -> {
         seen = seen + 1
         allSeen.atomic [
            allSeen = allSeen + 1
         ]
         val id = Thread::currentThread.id
         Write.emit(
            'thread' -> id,
            'global' -> allSeen,
            'local' -> seen
         )
         if (seen % 10 == 0)
            Other.emit('count' -> allSeen)
      }
      CountingWriter.Write -> super
   }
}
```

Listing 1: An Example CRUCIBLE Topology, counting the frequency of words in an input deck.

3. CRUCIBLE DSL

The CRUCIBLE DSL is built on top of the XText [7] language framework. It makes use of XText's XBase language, which is an embeddable version of the XTend JVM (Java Virtual Machine) language. CRUCIBLE's DSL provides a syntactic framework for modelling Processing Elements, using XBase for both PE logic and type declarations.

CRUCIBLE compiles a topology (a collection of interconnected PEs) into idiomatic Java, based on the CRUCIBLE PE Model (the bottom layer in Figure 1). This is in contrast to many other JVM languages, such as [15], which directly generate far less readable bytecode. Compiler support is used to provide syntactic sugar for access to global shared state and the security labelling mechanism, both of which are discussed in more detail later in this section.

Listing 1 contains a sample topology, demonstrating CRUCIBLE's syntax and JVM integration. At a high level, it is structured similarly to a Java code file; a package declaration, a set of imports, followed by one or more classes. In the CRUCIBLE DSL, each `process` models a class, with a name and an optional superclass. These classes are referred to as *Processing Elements*, or PEs for short. The body of a process is divided into a set of unordered blocks:

- `conf` - Compile-time configuration constants. The calculation of these may involve an arbitrary expression.
- `state` - Runtime mutable state; shared globally between instances of this PE. These variables may be declared `local`, in which case no global state is utilised for their storage (see Section 3.2).

- `output` / `outputs` - Declaration of the named output ports from the `process`.
- `input` - A block of key/value pairs mapping the qualified name (in the form `ProcessName.OutputName`) of an output to a block of code to execute upon arrival of a tuple from that port.

3.1 Message Passing

CRUCIBLE PEs communicate using message passing; a call to `*OutputName*.emit(...)` causes all subscribers to that output to receive the same message. No guarantees are given about the ordering of messages interleaved from different sources. Messages are emitted as a set of key-value pairs (as in lines 32-35; this makes use of the XTend Pair binary operator, `x -> y`, which is syntactic sugar for `new Pair<>(x,y)`). At compile time CRUCIBLE performs type inference on all of the `emit` calls in the topology to generate a correctly typed `receive` method interface on each subscriber.

3.2 Global Synchronisation & State

CRUCIBLE's global synchronisation and shared state components make use of the `GlobalStateProvider` and `LockingProvider` implementations which are injected at runtime. As discussed previously, `state` variables exist in global scope if they are not marked `local`. Thus, if multiple instances of a PE are run simultaneously, they will share any updates to their state; these changes are made automatically. This mechanism is applied without any promises about transactional integrity, which in limited circumstances is acceptable (*e.g.*, when sampling for 'a recent value').

In those circumstances which require distributed locking, an `atomic` extension method is provided to lock a given state element, and apply the given closure to the locked state (as in lines 28-30). The behaviour of this is similar to Java's `synchronized` keyword, with two key distinctions. The first of these is that the locking is guaranteed across multiple instances of a PE within a job, even across multiple hosts. The second key feature is that the `atomic` method may be applied to multiple objects by locking a literal list of variables (*e.g.*, `#[x, y, z].atomic[...]`), in which case all locks are acquired before executing the closure. A fixed ordering of locking and unlocking is applied in order to prevent deadlock, as well as a protocol lock to ensure that interleaving of lock requests on different critical regions do not deadlock.

4. SECURITY LABELLING

CRUCIBLE's Security Labelling protocol is built on the idea of cell-level visibility expressions, similar to those described by Bell and La Padula [3]. An expression is given as a conjunction of disjunctions across named labels. For example, the expression "*foo* & (*bar*||*baz*)" requires that a user is authorised to read the `foo` label, as well as either `bar` or `baz`. If they lack sufficient authorisation, then they are not permitted knowledge of the existence of that cell. This mechanism is particularly valuable in a multi-tenancy security scenario, where data with different security caveats or classifications are processed by a single system.

In CRUCIBLE, this concept is implemented by declaring an empty security label for every variable in the system. This label is accessible to the user by calling the `label` extension method on a variable. A user may add to a label using the `+=` operator. For example, the label of the `x` variable is expanded by either calling `x.label += "A | B"` (literal expansion), or `x.label += y.label` (label reference).

More formally, consider a label function λ, and a function ϵ, for label expansion:

$$\begin{array}{c} \lambda(a) : \text{Label} \ \ \text{for} \ \ \text{identifier } a \\ \epsilon(a,b) : \lambda_1(a) \ = \ \{\lambda_0(a), \lambda(b)\} \end{array} \tag{1}$$

Labelling of object-oriented method invocation makes the assumption that the invocation receiver's state may be mutated by the supplied arguments. Therefore:

$$\begin{array}{c} c.foo(d,e,f) \ \Rightarrow \ \begin{cases} \epsilon(c,d) \\ \epsilon(c,e) \\ \epsilon(c,f) \end{cases} \\ \lambda_1(c) \ = \ \{\lambda_0(c), \lambda(d), \lambda(e), \lambda(f)\} \end{array} \tag{2}$$

Assignment of a value to a non-final Java variable (*e.g.*, in `g = h`, where `h` is any expression; not to be confused with `g.label = h.label`) is equivalent to clearing the contents of its label, as all state for that reference is lost:

$$g = h \ \Rightarrow \ \lambda_1(g) = \emptyset \tag{3}$$

If the right-hand-side of the assignment (`h`) contains any identifiers, expansion must occur;

$$\begin{array}{c} \forall(i) \in h, \ identifier(i) \Rightarrow \epsilon(g,i) \\ \lambda_2(g) \ = \ \{\lambda(i_0)..\lambda(i_n)\} \end{array} \tag{4}$$

As many Java objects contain references to mutable state, when a label for `x` expands to include the label for `y`, and `y` is later expanded, `x`'s label must include the additions to `y`:

`# Ass: y.label == ''`	$\lambda_0(y) \ = \ \emptyset$
`x.label += 'foo'`	$\lambda_0(x) \ = \ \{\text{"}foo\text{"}\}$
`x.doSomething(y)`	$\epsilon(x,y)$
`y.label += 'bar'`	$\lambda_1(y) \ = \ \{\text{"}bar\text{"}\}$
`x.label == 'bar&foo'`	$\lambda_2(x) \ = \ \{\text{"}bar\text{"}, \text{"}foo\text{"}\}$

This labelling requires support from the CRUCIBLE compiler to transform invocations of `emit(Pair<String,?> ... tuple)` into invocations of `emit(Pair<SecurityLabel,Pair <String,?>> ... tuple)`. Note that in the Java type system this has the same type erasure as the original method. Concordantly, when generating the method signature for a `receive` method (the naming convention for these generated methods takes the form `receive$*PEName*$*OutputName*`), the compiler interleaves parameters with their respective labels. Thus, a signature of:

```
    receive$Proc$Out(String, Integer)
```

becomes:

```
    receive$Proc$Out(SecurityLabel, String,
            SecurityLabel, Integer)
```

It is important to note that due to CRUCIBLE's close integration with the JVM, this mechanism can not be considered secure for arbitrary untrusted code; it aims to assist the security-conscious engineer by making it easier to comply with security protocols than to ignore them.

5. CRUCIBLE RUNTIMES

Figure 2 shows how classes in the model interact. Instances of many of these classes (shaded in Figure 2) are injected at runtime (using [22]), permitting the behaviour of the topology to be integrated with the relevant runtime engine without changes or specialisation in the user code.

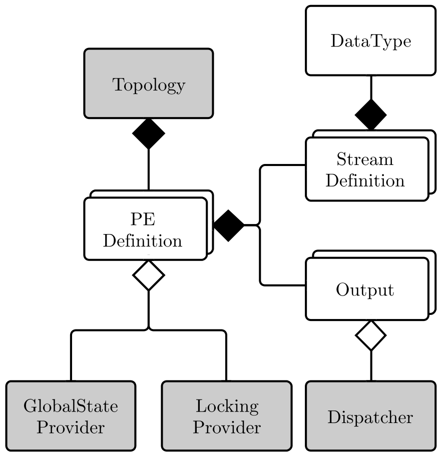

Figure 2: CRUCIBLE Model Composition diagram, showing the composition of the core model (white) and the runtime injectable components (grey).

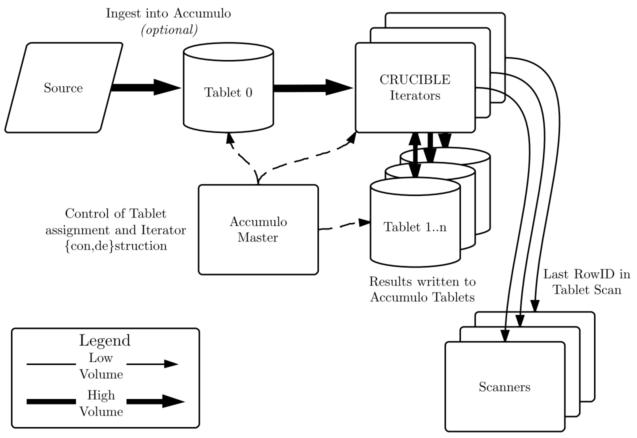

Figure 3: CRUCIBLE Accumulo Runtime Message Dispatch, demonstrating how Scanners are used to pull data through a collection of custom Iterators to analyse data sharded across Accumulo Tablets.

5.1 Standalone Processing

The first, and simplest, runtime environment is designed for easy local testing of a CRUCIBLE topology, without any need for a distributed infrastructure. This Standalone environment simply executes a given topology locally, in a single JVM, relying heavily on Java's multithreading capabilities. Locking and global state are provided based on this in-JVM assumption.

Message passing is performed entirely in-memory, using a singleton `Dispatcher` implementation with a blocking concurrent queue providing backpressure [20] in the event that some PEs are slower than others. This prevents the topology from using an excessive amount of memory.

5.2 On-Line Processing

IBM's InfoSphere Streams product forms the basis of CRUCIBLE's streaming (on-line) runtime engine. An extension to the CRUCIBLE DSL compiler generates a complete SPL (IBM's Streams Processing Language) project from the given topology. This project can be imported into InfoSphere Streams Studio; it consists of the required project infrastructure (including classpath dependencies), and a single SPL Main Composite describing the topology. Each Streams SPL PE is an instance of the `CruciblePE` class. This class handles invocation of the `receive$...` tuple methods, dispatch between Streams and the `CruciblePE`s, and tuple serialisation.

There is a one-to-one mapping between tuples emitted in CRUCIBLE and tuples emitted in Streams. Each key in a CRUCIBLE tuple has a defined field in a Streams tuple, and all keys are transmitted with each emission. Keys are interleaved with their security labels, such that the label for a key always precedes it. Tuple values are converted between Streams and CRUCIBLE using an injected serialisation provider; the default implementation of this leverages Kryo[6] for time and space efficiency reasons [1, 19], but it would be trivial to add, for example, a Protocol Buffers[7]-based implementation if interoperability with external systems were required. Security Labels are *not* serialised through Kryo, in order to facilitate their inspection by debug tooling on the Streams instance. Security Labels are written as `rstring` values, while all others are serialised as a `list<int8>` (representing an immutable array of bytes).

Each of these `CruciblePE` instances could potentially be scheduled into separate JVMs running on different hosts, according to the behaviour of the Streams deployment manager. Manual editing of the SPL *e.g.*, to use SPLMM (SPL Mixed Mode, using Perl as a preprocessor), can be used to split a single PE across multiple hosts, as well. The global synchronisation primitives discussed in Section 3.2 must be enabled on the runtime to facilitate this parallelism.

[6]`https://code.google.com/p/kryo/`
[7]`http://code.google.com/p/protobuf/`

5.3 Off-Line Processing

The mapping from CRUCIBLE's execution model to Accumulo for off-line processing is more involved. In order to exploit the data locality and inherent parallelism available in HDFS, while maintaining the level of continuous insight offered by Streams, the Accumulo runtime makes use of Accumulo Iterators [8]. An `Iterator` may scan multiple tablets in parallel, and will stream ordered results to the `Scanner` which invoked the iterator. CRUCIBLE makes use of this paradigm by spawning a `CrucibleIterator` for each PE in the topology, along with a multithreaded `Scanner` to consume results. Each `CrucibleIterator` may be instantiated, destroyed, and re-created repeatedly as the scan progresses through the data store.

Each `CrucibleIterator` is assigned to its own table, named after the `UUID` of the Job and the PE to which it refers. Values map onto an Accumulo Key by using a timestamp for the Row ID, the Source PE of a tuple as Column Qualifier, and the emitted item's key as Column Key. Column Visibility and Security Label are mapped directly onto their Accumulo equivalents, making efficient use of native constructs.

In this way, the `CrucibleIterator` can invoke the correct `receive` method on a PE, by collecting all $(key, value, label)$ triples of a given `RowID` together. By mapping CRUCIBLE Security Labels onto Accumulo Visibilities, all message passing data (and final results) are persisted to HDFS with their correct labels: external Accumulo clients may read that state, provided they have suitable `Authorization`(s).

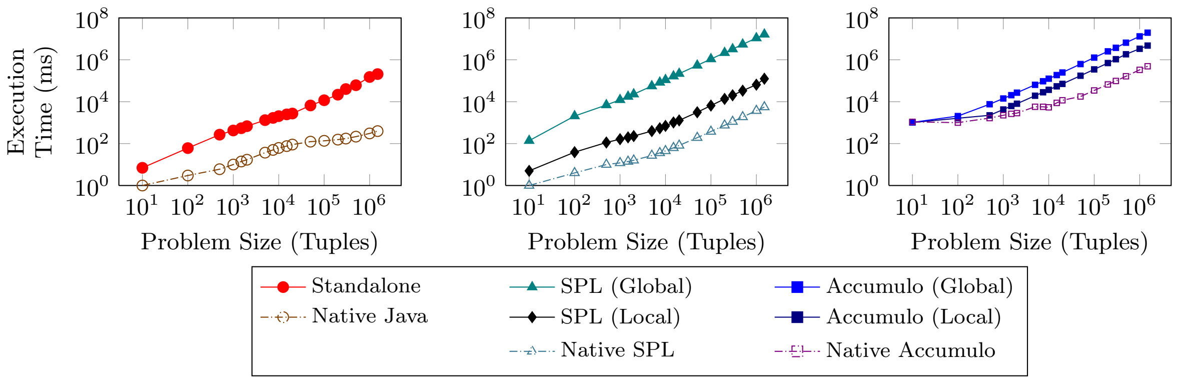

Figure 4: Scalability Comparison of CRUCIBLE Runtimes and Native Implementations.

CRUCIBLE's `AccumuloDispatcher` takes tuples emitted by a PE, and writes them to the tables of each subscriber to that stream, for the relevant `CrucibleIterators` to process in parallel. The final component is the multithreaded `Scanner`, which will restart from the last key scanned, thus ensuring that the Accumulo-backed job fully processes all tuples in all tables. This flow is laid out in Figure 3, for clarity.

6. STANDARD LIBRARY

The final CRUCIBLE component is the standard library. This includes the injected components necessary for operation of the runtimes described in Section 5, along with a set of base PE implementations to simplify creation of CRUCIBLE topologies. These provide examples of data ingest from a variety of sources, such as the various APIs of Flickr and Twitter, along with primitives to read and write File data. An XPath PE is valuable for extracting data from XML. Work is already underway to expand this library to include operators for parallel JOINs, Bloom-Filters, and serialisation to/from common data formats such as JSON.

This library is implemented in standard Java, and no special infrastructure is required to extend it. It is intended that any users of CRUCIBLE may extend this library with custom PEs, or publish their own, simply by writing the relevant Java, conforming to an interface, and making it available on the deployed classpath. For single use Java operators this may even be done within the CRUCIBLE topology's Eclipse IDE project - the Java compiler will pick this up and integrate it with CRUCIBLE automatically.

7. EXPERIMENTAL RESULTS

A key part of validating this approach, beyond its functional correctness, is demonstrating the performance of the various CRUCIBLE runtimes. As the CRUCIBLE system is in its early phases of development, we present results from the pre-optimisation codebase, with a focus on comparing the scaling behaviour of each CRUCIBLE runtime against a functionally equivalent native implementation.

7.1 Experimental Setup

A simple CRUCIBLE benchmark topology was written to count the frequency of letter occurrence in a dictionary (akin to Listing 1), limited to the top N results, where N is the problem size. Due to the lack of available support in native code, CRUCIBLE's security labelling capabilities have been omitted from this benchmark.

These results were collected on a small development cluster, consisting of three Tablet Servers, one Master, and three Streams nodes. Each node hosts two dual-core 3.0GHz Intel Xeon 5160 CPUs, 8GB RAM, and 2x1GbE interfaces.

7.2 Analysis

The three graphs in Figure 4 shows the results of this testing across the Standalone (Section 5.1), Streams SPL (Section 5.2), and Accumulo (Section 5.3) runtimes respectively. It is clear from these results that the CRUCIBLE runtimes, in the main, scale proportionally to their native equivalents. There is a noticeable performance gap for most of the runtimes at present; future work is scheduled to enhance the per-tuple processing delay in all CRUCIBLE runtimes. The "Global" and "Local" data series are worth noting, as they highlight the performance difference between Global (ZooKeeper based) and Local (in memory) shared state providers (see Section 3.2, and the discussion at the end of Section 5.2). Some runtimes do not require all of the features of CRUCIBLE at all times, and thus it is valuable to be able to disable performance-hampering features such as these.

While comparing the absolute performance of CRUCIBLE and the native implementations, it is important to consider the engineering implications of our approach in CRUCIBLE. Removing much of the "scaffolding" of other solutions has enhanced the expressivity of the CRUCIBLE DSL to the point where the above benchmark was implemented in ∼40 lines of code, as opposed to ∼260 for the three native implementations. Furthermore, the CRUCIBLE implementation can be executed across multiple runtimes, whereas the native implementations are specific to either on- or off-line environments. In our experience, the 2–3 days taken to write and debug the suite of native analytics was reduced to under a day with CRUCIBLE.

8. FURTHER WORK

There are a number of valuable avenues of further work to explore. Ultimately, we intend to improve CRUCIBLE's usability through the use of improved interfaces for planning and development of analytics. Alongside that, we intend to consider two key areas of improvement; the CRUCIBLE DSL, and the runtime environments.

CRUCIBLE DSL improvements include: (1) New language features - topology composition, runtime PE reuse, cross-job Subscription, etc; and (2) Enhanced in-IDE debug tooling, through the use of mock data sources, probes, and visualisations of data flow and security labels.

Further work on the runtimes includes: (1) Performance enhancements for CRUCIBLE runtimes, including alternative compilation strategies; (2) Workload-based optimisation of a topology for alternative architectures; and (3) PE Fusion and Fission Techniques to enhance data parallelism.

9. CONCLUSIONS

We have demonstrated CRUCIBLE; a scalable system for implementation of security-conscious analytics over high volume, velocity, and variety data by allowing users to make the most of the on- and off-line systems at their disposal. Three key aspects of CRUCIBLE have been detailed; (1) a high-level DSL and associated IDE Tooling; (2) a suite of runtime environments providing consistent execution semantics across on- and off-line data; and (3) a consistent cross-runtime cell-level security labelling framework.

Our results show that the *absolute* performance of the CRUCIBLE runtimes must be improved, but their *scalability* is in line with native implementations. We have demonstrated the power of basing CRUCIBLE on a DSL; the tight integration of key language features (in particular security labelling and atomic operations) enables the implementation of sample analytics in one sixth the amount of code as an equivalent suite of native implementations – a substantial improvement in engineering time, cost, and risk.

The web page `http://go.warwick.ac.uk/crucible` contains further information on CRUCIBLE.

Acknowledgments

This work was funded under an Industrial EPSRC CASE Studentship, entitled "Platforms for Deploying Scalable Parallel Analytic Jobs over High Frequency Data Streams". The lead author thanks their employer for funding the course of study that led to this paper.

References

[1] T. Aihkisalo and T. Paaso. A Performance Comparison of Web Service Object Marshalling and Unmarshalling Solutions. In *Proceedings of the 2011 IEEE World Congress on Services*, pages 122–129. IEEE, 2011.

[2] M. Ali. An Introduction to Microsoft SQL Server StreamInsight. In *Proceedings of the 1st International Conference and Exhibition on Computing for Geospatial Research & Applications*, page 66, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0031-5.

[3] D. E. Bell and L. J. La Padula. Secure Computer System: Unified Exposition and Multics Interpretation. Technical report, DTIC Document, 1976.

[4] F. Chang et al. Bigtable: A Distributed Storage System for Structured Data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[5] G. De Francisci Morales. SAMOA: A Platform for Mining Big Data Streams. In *Proceedings of the 22nd International Conference on the World Wide Web Companion*, pages 777–778. International World Wide Web Conferences Steering Committee, 2013.

[6] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[7] S. Efftinge and M. Völter. oAW xText: A Framework for Textual dsls. In *Proceedings of the Workshop on Modeling Symposium at Eclipse Summit*, 2006.

[8] A. Fuchs. Accumulo - Extensions to Google's Bigtable Design. Technical report, National Security Agency, March 2012.

[9] L. Golab et al. Stream Warehousing With DataDepot. In *Proceedings of the 35th SIGMOD international conference on Management of data*, pages 847–854. ACM, 2009.

[10] M. Hausenblas and J. Nadeau. Apache Drill: Interactive Ad-Hoc Analysis at Scale. *Big Data*, June 2013.

[11] C. R. Kalmanek et al. Darkstar: Using Exploratory Data Mining to Raise the Bar on Network Reliability and Performance. In *Proceedings of the 7th International Workshop on Design of Reliable Communication Networks, 2009*, pages 1–10. IEEE, 2009.

[12] V. Kumar et al. DEDUCE: At the Intersection of MapReduce and Stream Processing. In *Proceedings of the 13th International Conference on Extending Database Technology*, pages 657–662. ACM, 2010.

[13] S. Melnik et al. Dremel: Interactive Analysis of Web-Scale Datasets. *Proceedings of the VLDB Endowment*, 3(1-2):330–339, 2010.

[14] L. Neumeyer et al. S4: Distributed Stream Computing Platform. In *Proceedings of the 2010 IEEE International Conference on Data Mining Workshops (ICDMW)*, pages 170 –177, Dec. 2010.

[15] M. Odersky et al. An Overview of the Scala Programming Language. Technical report, Citeseer, 2004.

[16] C. Olston et al. Pig Latin: A Not-So-Foreign Language for Data Processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1099–1110. ACM, 2008.

[17] R. Rea and K. Mamidipaka. IBM InfoSphere Streams: Enabling Complex Analytics with Ultra-Low Latencies on Data in Motion. *IBM White Paper*, 2009.

[18] K. Shvachko et al. The Hadoop Distributed File System. In *Proceedings of the 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10. IEEE, 2010.

[19] E. Smith. JVM Serializers Project. URL `https://github.com/eishay/jvm-serializers/wiki`.

[20] L. Tassiulas and A. Ephremides. Stability Properties of Constrained Queueing Systems and Scheduling Policies for Maximum Throughput in Multihop Radio Networks. *IEEE Transactions on Automatic Control*, 37 (12):1936–1948, 1992. ISSN 0018-9286.

[21] A. Thusoo et al. Hive: A Warehousing Solution Over a Map-Reduce Framework. *Proceedings of the VLDB Endowment*, 2(2):1626–1629, 2009.

[22] R. Vanbrabant. *Google Guice: Agile Lightweight Dependency Injection Framework*. Apress, 2008.