DEFCon: High-Performance Event Processing with Information Security

Matteo Migliavacca
Department of Computing
Imperial College London

Ioannis Papagiannis
Department of Computing
Imperial College London

David M. Eyers
Computer Laboratory
University of Cambridge

Brian Shand
CBCU, Eastern Cancer Registry
National Health Service UK

Jean Bacon
Computer Laboratory
University of Cambridge

Peter Pietzuch
Department of Computing
Imperial College London

smartflow@doc.ic.ac.uk

Abstract

In finance and healthcare, event processing systems handle sensitive data on behalf of many clients. Guaranteeing information security in such systems is challenging because of their strict performance requirements in terms of high event throughput and low processing latency.

We describe DEFCon, an event processing system that enforces constraints on event flows between event processing units. DEFCon uses a combination of static and runtime techniques for achieving light-weight isolation of event flows, while supporting efficient sharing of events. Our experimental evaluation in a financial data processing scenario shows that DEFCon can provide information security with significantly lower processing latency compared to a traditional approach.

1 Introduction

Applications in finance, healthcare, systems monitoring and pervasive sensing that handle personal or confidential data must provide both strong security guarantees and high performance. Such applications are often implemented as event processing systems, in which flows of event messages are transformed by processing units [37]. Preserving information security in event processing without sacrificing performance is an open problem.

For example, financial data processing systems must support high message throughput and low processing latency. Trading applications handle message volumes peaking in the tens of thousands of events per second during the closing periods on major stock exchanges, and this is expected to grow in the future [1]. Low processing latency is crucial for statistical arbitrage and high frequency trading; latencies above a few milliseconds risk losing the trading initiative to competitors [12].

At the same time, information security is a major concern in financial applications. Internal proprietary traders have to shield their buy/sell message flows and trading strategies from each other, and be shielded themselves from the client buy/sell flows within a bank. Information leakage about other buy/sell activities is extremely valuable to clients, as it may lead to financial gain, motivating them to look for leaks. Leakage of client data to other clients may damage a bank's reputation; leakage of such data to a bank's internal traders is illegal in most jurisdictions, violating rules regarding conflicts of interest [8]. The UK Financial Service Authority (FSA) repeatedly fines major banks for trading on their own behalf based on information obtained from clients [15].

Traditional approaches for isolating information flows have limitations when applied to high-performance event processing. Achieving isolation between client flows by allocating them to separate physical hosts is impractical due to the large number of clients that use a single event processing system. In addition, physical rack space in data centres close to exchanges, a prerequisite for low latency processing, is expensive and limited [23]. Isolation using OS-level processes or virtual machines incurs a performance penalty due to inter-process or inter-machine communication, when processing units must receive multiple client flows. This is a common requirement when matching buy/sell orders, performing legal auditing or carrying out fraud detection. The focus on performance means that current systems do not guarantee end-to-end information security, instead leaving it to applications to provide their own, ad hoc mechanisms.

We enforce information security in event processing using a uniform mechanism. The event processing system prevents incorrect message flows between processing units but permits desirable communication with low latency and high throughput. We describe DEFCon, an event processing system that supports *decentralised event flow control* (DEFC). The DEFC model applies information flow control principles [27] to high-performance event processing: parts of event messages are annotated with appropriate security labels. DEFCon tracks the "taint" caused by messages as they flow through processing units and prevents information leakage when units

lack appropriate privileges by controlling the external visibility of labelled messages. It also avoids the inference of information through implicit information flows—the absence of a unit's messages after that unit becomes tainted would otherwise be observable by other units.

To enforce event flow control, DEFCON uses application-level virtualisation to separate processing units. DEFCON isolates processing units within the same address space using a modified Java language runtime. This lightweight approach allows efficient communication between isolation domains (or *isolates*). To separate isolates, we first statically determine potential storage channels in Java, white-listing safe ones. After that, we add run-time checks by weaving interceptors into potentially dangerous code paths. Our methodology is easily reproducible; it only took us a few days to add isolation to OpenJDK 6.

Our evaluation using a financial trading application demonstrates a secure means of aggregating clients' buy/sell orders on a single machine that enables them to trade at low latency. Our results show that this approach gives low processing latencies of 2 ms, at the cost of a 20% median decrease in message throughput. This is an acceptable trade-off, given that isolation using separate processes results in latencies that are almost four times higher, as shown in §6.

In summary, the main contributions of the paper are:

- a model for decentralised event flow control in event processing systems;
- Java isolation with low overhead for inter-isolate communication using static and runtime techniques;
- a prototype DEFCON implementation and its evaluation in a financial processing scenario.

The next section provides background information on event processing, security requirements and related work on information flow control. In §3, we describe our model for decentralised event flow control. Our approach for achieving lightweight isolation in the Java runtime is presented in §4. In §5, we give details of the DEFCON prototype system, followed by evaluation results in §6. The paper finishes with conclusions (§7).

2 Background

2.1 Event processing

Event processing performs analysis and transformation of flows of event messages, as found in financial, monitoring and pervasive applications [24]. Since events are caused by real-world phenomena, such as buy/sell orders submitted by financial traders, event processing must occur in near real-time to keep up with a continuous flow of events. Popular uses of event processing systems are in fraud detection, Internet betting exchanges [7] and, in the corporate setting, for enterprise application integration and business process management [5]. While we focus on centralised event processing in this paper, event processing also finds applicability in-the-large to integrate "systems of systems" by inter-connecting applications without tightly coupling them [26].

Event processing systems, such as Oracle CEP [38], Esper [14] and Progress Apama [2], use a message-driven programming paradigm. *Event messages* (or *events*) are exchanged between *processing units*. Processing units implement the "business logic" of an event processing application and may be contributed by clients or other third-parties. They are usually reactive in design—events are dispatched to processing units that may emit further events in response. There is no single data format for event messages, but they often have a fixed structure, such as key/value pairs.

Financial event processing. In modern stock trading, low processing latency is key to success. As financial traders use automated algorithmic trading, response time becomes a crucial factor for taking advantage of opportunities before the competition do [20]. To support algorithmic trading, stock exchanges provide appropriate interfaces and event flows. To achieve low latency, they charge for the service of having machines physically co-located in the same data centre as parts of the exchange [16].

It was recently suggested that reducing latency by 6 ms may cost a firm $1.5 million [9]. The advantage that they get from reacting faster to the market than their competition may translate to increased earnings of $0.01 per share, even for trades generated by other traders [12]. However, even with co-location within the same data centre rack, there is a minimum latency penalty due to inter-machine network communication.

Therefore having multiple traders acting for competing institutions share a single, co-located machine has several benefits. First, trading latency is reduced since client processing may be placed on the same physical machine as the order matching itself [34]. Second, the traders can share the financial burden of co-location within the exchange. Third, they can carry out *local brokering* by matching buy/sell orders among themselves—a practice known as a "dark pool"—thus avoiding the commission costs and trading exposure when the stock exchange is involved [44].

Hosting competing traders on the same machine has significant security implications. To avoid disclosing proprietary trading strategies, each trader's stock subscriptions and buy/sell order feeds must be kept isolated. The co-location provider must respect clients' privacy; bugs must never result in information leakage.

2.2 Security in event processing

Today's event processing systems face challenging security requirements as they are complex, process sensitive data and support the integration of third-party code. This increases the likelihood of software defects exposing information. Information leaks have serious consequences because of the sensitive nature of data in domains such as finance or healthcare. As in the stock-trading platform example, the organisation providing the event processing service is frequently not the owner of the processed data. Processing code may also be contributed by multiple parties, for example, when trading strategies are implemented by the clients of a trading platform.

Event processing systems should operate according to data security policies that specify system-wide, end-to-end *confidentiality* and *integrity* guarantees. For example, traders on a trading platform require their trading strategies not to be exposed to other traders (confidentiality). The input data to a trading strategy should only be stock tick events provided by the stock exchange (integrity). This cannot be satisfied by simple access control schemes, such as access control lists or capabilities, because they alone cannot give end-to-end guarantees: any processing unit able to access traders' orders may cause a leak to other traders due to bugs or malicious behaviour. Anecdotal evidence from the (rather secretive) financial industry, and existing open source projects [35], suggest that current proprietary trading systems indeed lack mechanisms to enforce end-to-end information security. Instead, they rely on the correctness (and compliant behaviour) of processing units.

Threat model. We aim to improve information security in event processing by addressing the threat that information in events may be perceived or influenced by unauthorised parties. Our threat model is that processing units may contain unintentional bugs or perform intentional information leakage. We do not target systems that run arbitrary code of unknown provenance: event processing systems are important assets of organisations and are thus carefully guarded. Only accountable parties are granted access to them. As a consequence, we are not concerned about denial-of-service attacks from timing-related attacks or misuse of resources—we leave protection against them for future work. However, we do want protection from parties that may otherwise be tempted not to play by the rules, e.g. by trying to acquire information that they should not access or leak information that they agreed to keep private. We assume that the operating system, the language runtime and our event processing platform can be trusted.

2.3 Information flow control

We found that information flow control, which provides fine-grained control over the sharing of data in a system, is a natural way to realise the aforementioned kind of security that event processing systems require.

Information flow control is a form of mandatory access control: a principal that is granted access to information stored in an object cannot make this information available to other principals, for example, by storing the information in an unprotected object (no-write-down or *-property) [6]. It was initially proposed in the context of military multi-level security [11]: principals and objects are assigned security *labels* denoting levels, and access decisions are governed by a "can-flow-to" partial order. For example, a principal operating at level "secret" can read a "confidential" object but cannot read a "top-secret" or write to a "confidential" object. Through this model, a system can enforce confinement of "secret" information to principals with "secret" (or higher) clearance.

Equivalently, IFC-protected objects may be thought of as having a *contaminating* or *tainting* effect on the principals that process them—a principal that reads a "secret" document must be contaminated with the "secret" label, and will contaminate all objects it subsequently modifies.

Compartments created by labels are fairly coarse-grained and declassification of information is performed outside of the model by a highly-trusted component. Myers and Liskov [27] introduce *decentralised information flow control* (DIFC) that permits applications to partition their rights by creating fresh labels and controlling declassification privileges for them. Jif [28] applies the DIFC model to variables in Java. Labels are assigned and checked statically by a compiler that infers label information for expressions and rejects invalid programs. In contrast, event-processing applications require fresh labels at runtime, for example, when new clients join the system. Trishul [29] and Laminar [32] use dynamic label checks at the JVM level. However, tracking flows between variables at runtime considerably reduces performance.

Myers and Liskov's model also resulted in a new breed of DIFC-compliant operating systems that use labels at the granularity of OS processes [13, 43, 22]. Asbestos [13] enables processes to protect data and enforces flow constraints at runtime. Processes' labels are dynamic, which requires extra care to avoid implicit information leakage, and Asbestos suffers from covert storage channels. HiStar [43] is a complete OS redesign based on DIFC to avoid covert channels. Flume [22] brings DIFC to Linux by intercepting system calls and augmenting them with labels. All of the above projects isolate processes in separate address spaces and provide IPC abstractions for communication. For event processing, this would require dispatching events to processing units by copying them between isolates, resulting in lower performance (cf. §6).

The approach closest to ours is Resin [41], which discovers security vulnerabilities in applications by modify-

ing the language runtime to attach data flow policies to data. These policies are checked when data flows cross guarded boundaries, such as method invocations. Resin only tracks the policy when data is explicitly copied or altered, making it unsuitable to discover deliberate, implicit leakage of information, as it may be found in financial applications.

3 DEFCON Design

This section describes the design of our event processing system in terms of our approach for controlling the flow of events. We believe that it is natural to apply information flow constraints at the granularity of events because they constitute explicit data flow in the system. This is in contrast to applying constraints with operating system objects or through programming language syntax extensions, as seen in related research [13, 43, 22, 27].

3.1 DEFC model

We first describe our model of *decentralised event flow control* (DEFC). The DEFC model uses information flow control to constrain the flow of events in an event processing system. In this paper, we focus on aspects of the model related to operation within a single machine as opposed to a distributed system.

The DEFC model has a number of novel features, which are specifically aimed at event processing: (1) multiple labels are associated with parts of event messages for fine-grained information security (§3.1.2); (2) privileges are separated from privilege delegation privileges—this lets event flows be constrained to pass through particular processing units (§3.1.3); (3) privileges can be dynamically propagated using privilege-carrying events, thus avoiding implicit, covert channels (§3.1.5); and (4) events can be partially processed by units without tainting all event parts (§3.1.6).

3.1.1 Security labels

Event flow is monitored and enforced through the use of *security labels* (or *labels*), which are similar to labels in Flume [22]. Labels are the smallest structure on which event flow checking operates, and protect confidentiality and integrity of events. For example, labels can act to enforce isolation between traders in a financial application, or to ensure that particularly sensitive aspects of patient healthcare data are not leaked to all users.

As illustrated in Figure 1, security labels are pairs, (S, I), consisting of a *confidentiality component* S and an *integrity component* I. S and I are each sets of *tags*. Each tag is used to represent an individual, indivisible concern either about the privacy, placed in S, or the integrity, placed in I, of data. Tags are opaque values,

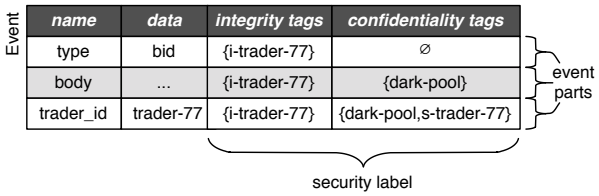

Figure 1: An event message with multiple named parts, each containing data protected by integrity and confidentiality tags.

implemented as unique, random bit-strings. We refer to them using a symbolic name, such as i-trader-77 (an integrity tag in this case).

Tags in confidentiality components are "sticky": once a tag has been inserted into a label component, data protected by that label cannot flow to processing units without that tag, unless privilege over the tag is exercised. In contrast, tags in integrity components are "fragile": they are destroyed when information with such tags is mixed with information not containing the tag, again unless a privilege is exercised.

For example, if a processing unit in a trading application receives data from two other units with confidentiality components {s-trading, s-client-2402} and {s-trading, s-trader-77} respectively, then any resulting data will include all of the tags {s-trading, s-client-2402, s-trader-77}. This reflects the sensitivity with respect to both sources of the data. Similarly, if data from a stock ticker with an integrity component {i-stockticker} is combined with client data with integrity {i-trader-77}, the produced data will have integrity {}. This shows that the data cannot be identified as originating directly from the stock ticker any more.

Labels form a lattice: for the confidentiality component (S), information labelled S_a can flow to places holding component S_b if and only if $S_a \subseteq S_b$; here $\subseteq$ is the "can flow to" ordering relation [42]. For integrity labels (I), "can flow to" order is the superset relation $\supseteq$. Thus we define the "can flow to" relationship $L_a \prec L_b$ for labels as:

$$L_a \prec L_b \quad \text{iff} \quad S_a \subseteq S_b \text{ and } I_a \supseteq I_b$$
$$\text{where } L_a = (S_a, I_a) \text{ and } L_b = (S_b, I_b)$$

3.1.2 Anatomy of events

A key aspect of our model is the use of information flow control at the granularity of events. An event consists of a number of event *parts*. Each part has a *name*, associated *data* and a *security label*. Using parts within an event allows it to be processed by the system as a single, connected entity, but yet to carry data items within its parts that have different security labels. Dispatching a single event with secured parts supports the principle of least privilege—processing units only obtain access to parts of the event that they require.

Figure 1 shows a bid event in a financial trading application with three parts. The event is tagged with the

trader's integrity tag. The information contained in the bid has different sensitivity levels: the type part of the event is public, while the body part is confined to match within the dark pool by the dark-pool tag. The identity part of the trader is further protected by a trader-private confidentiality tag.

Access to event parts is controlled by the system that implements DEFC. When units want to retrieve or modify event parts, or to create new events, they must use an API such as the one described in §5.

3.1.3 Constraining tags and labels

Each processing unit can store state—its data can persist between event deliveries. Rather than associate labels with each piece of state in that unit, a single label (S_u, I_u) is maintained with the overall confidentiality and integrity of the unit's state. (We also refer to this as the unit's *contamination level*.) This avoids the need for specific programming language support for information flow control, as most enforcement can be done at the API level.

The ability of a unit to add or remove a tag to/from its label is a *privilege*. A unit u's run-time privileges are represented using two sets: O_u^+ and O_u^-. If a tag appears in O_u^+, then u can add it to S_u or I_u. Likewise, u can remove any tag in O_u^- from any of its components.

If unit u adds tag $t \in O_u^+$ to S_u, then t is used as a confidentiality tag, moving u to a higher level of secrecy. This lets u "read down" no less (and probably more) data than before. If t is used as an integrity tag, then adding it to I_u would be exercising an *endorsement privilege*. Conversely, removing a confidentiality tag $t \in O_u^-$ from S_u involves unit u exercising a *declassification privilege*, while removing an integrity tag t from I_u is a transition to operation at lower integrity.

For dynamic privilege management, privileges over tag privileges themselves are represented in two further sets per unit: $O_u^{-\text{auth}}$ and $O_u^{+\text{auth}}$. We define their semantics with a short-hand notation: t_u^+ means that $t \in O_u^+$; t_u^- means $t \in O_u^-$; $t_u^{+\text{auth}}$ means $t \in O_u^{+\text{auth}}$; $t_u^{-\text{auth}}$ means $t \in O_u^{-\text{auth}}$ for tag t and unit u. We will omit the u subscript when the context is clear.

$t_u^{-\text{auth}}$ lets u *delegate* the corresponding privilege over tag t to a target unit v. After delegation, t_v^- holds. Likewise for $t_u^{+\text{auth}}$. If $t_u^{-\text{auth}}$, u can also delegate to v the ability to delegate privilege, yielding $t_v^{-\text{auth}}$ (likewise for $t_u^{+\text{auth}}$). Delegation is done by passing privilege-carrying events between units (cf. §3.1.5), ensuring that the DEFC model is enforced without creating a covert channel.

The separation of O_u^+ and $O_u^{+\text{auth}}$, in contrast to Asbestos/HiStar or Flume, allows our model to enforce specific processing topologies. For example, a Broker unit can send data to the Stock Exchange unit only through a Regulator unit, by preventing the Regulator from delegating to the Broker the right to communicate with the Stock Exchange directly.

Units can request that tags be created for them at runtime by the system. Although opaque to the units, tags and tag privilege delegations are transmittable objects. When a tag t is successfully created for a unit u, then $t_u^{-\text{auth}}$ and $t_u^{+\text{auth}}$. In many cases, u will apply these privileges to itself to obtain t_u^- or t_u^+.

A unit can have both t_u^- and t_u^+; then u has complete privilege over t. Note that the privilege alone does not let u transfer its privileges to other units.

3.1.4 Input/Output labels

Processing units need a convenient way to express their intention to use privileges when receiving or sending events. A unit u applies privileges by controlling an *input* label $(S_u^{\text{in}}, I_u^{\text{in}})$, which is equivalent to its contamination level (S_u, I_u), and an *output label* $(S_u^{\text{out}}, I_u^{\text{out}})$. Changes to these labels cause the system automatically to exercise privileges on behalf of the unit when it receives or sends events, in order to reach a desired level. Input/output labels increase convenience for unit programmers: they avoid repeated API calls to add and remove tags from labels when outputting events, or to change a unit's contamination label temporarily in order to be able to receive a given event.

For example, a Broker unit can add an integrity tag i to I_u^{out} but not to I_u^{in}. This enables it to vouch for the integrity of the stock trades that it publishes without having to add tag i explicitly each time. Similarly, adding tag t temporarily to S_u^{in} but not to S_u^{out} allows a Broker to receive and declassify t-protected orders without changing the code that handles individual events. In both cases, the use of privileges is only required when changing the input and output labels and not every time when handling an event.

Note that systems that allow for implicit contamination risk leaking information. For example, one could posit a model in which a unit's input and output labels rose automatically if that unit read an event part that included tags that were not within the unit's labels. The problem with this is that if unit u observes that it can no longer communicate with unit v that has been implicitly contaminated, then information has leaked to u. Therefore we require explicit requests for all changes to the input/output labels.

3.1.5 Dynamic privilege propagation

We use *privilege-carrying events* as an in-band mechanism to delegate privileges between processing units. A request to read a privilege-carrying part will bestow privileges on the requesting unit—but only if the unit already has a sufficient input label to read the data in that part.

An example of this is a Regulator unit trying to learn the identity of a trader mentioned in a trade event. The trader's identity is protected against disclosure by a unique tag t, but t^+ and t^- are included in another part visible to the Regulator unit only. This means that the Regulator can read this part, thus gaining t^+ and t^-, and then use these privileges to learn the trader's identity.

Although the bestowing of privileges is implicit, the privileges relate to a particular tag t, and the receiving unit cannot invoke the privileges without a reference to tag t itself. This reference is carried in the data part of an event: units, by design, will know in advance when to expect tags to be transferred to them, and when accessing a part will result in a privilege delegation. In the previous example, the tag t itself has to be in the data part that the Regulator accesses.

3.1.6 Partial event processing

Event processing frequently involves units transforming events along a *main dataflow path*, augmenting events as they flow through the system. To allow units to update only some parts of an event, we distinguish event processing on the main path from events generated by units themselves. In the former case, a unit that adds a part does not cause the labels of all parts of that event to change to the unit's output label. In the latter case, all parts' labels match the unit's output label.

For example, partial event processing enables a Broker unit to operate on orders without knowing the identity of the originating trader. The Broker can have access to some parts, such as the bid/ask price, and subsequently add new parts, such as a reason why an order was rejected, without being aware of or affecting a protected part with the trader identity.

When an event is dispatched to a unit, the unit may read and/or modify some parts but not others. The unit must then invoke a `release` API call, after which the event dispatcher may deliver the event to other units. Unaltered event parts do not need to have their labels changed. A released event must not cause additional deliveries to units with lower input labels. When multiple units make conflicting modifications to a part, the resulting event will have to contain both versions of the affected part.

3.2 DEFCON architecture

Our DEFCON architecture, that implements the DEFC model, is illustrated in Figure 2. The DEFCON system provides a runtime environment for a set of event processing units that implement the business logic of an event processing application. Units interact with the DEFCON system through API calls. As shown in the figure, the DEFCON system carries out the following tasks:

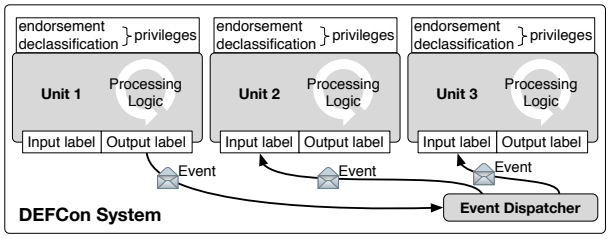

Figure 2: Overview of the DEFCON architecture.

Label/tag management. DEFCON maintains the set of defined tags in the *tag store*. It also keeps track of the input and output labels and privileges for each unit. The tags that make up labels are opaque to units. Units access tags by reference but cannot modify them directly.

Inter-unit communication. DEFCON provides units with a publish/subscribe API to send/receive events. To receive call-backs that provide event references, units register their interests by making subscriptions. An *event dispatcher* sends events to units that have expressed interest previously. This decoupled communication means that the fact that a publish call has succeeded does not convey any information that might violate DEFC (e.g. which units were actually notified).

Unit life-cycle management. DEFCON instantiates and terminates event processing units. Having DEFCON manage units allows it to apply restrictions to the operations that units can do, as described in the next section.

To enforce event flow control, DEFCON must prevent units from communicating directly except through the event dispatcher that can check DEFC constraints. Otherwise a unit with clearance to receive confidential events could avoid the confinement imposed by its label by using a communication channel that is not protected by labels. Therefore each unit must execute within its own *isolate* that prevents it from interacting with other units or components outside of the DEFCON system.

4 Practical, Light-weight Java Isolation

As described in §2.1, a requirement for DEFCON is to prevent unauthorised processing units from communicating with each other, while supporting low latency, high throughput event communication between permitted units. Making units separate OS-level processes achieves isolation but comes at the cost of increased communication latency due to inter-process communication, serialisation of potentially complex event message data and context switching overhead. In §6, we show that this results in higher processing latencies. Therefore, we isolate units executing within the same OS process through the introduction of new mechanisms within the programming language runtime.

We chose Java for our implementation because it is a mature, strongly-typed language that is representative of the languages used to build industrial-strength event processing applications. Processing units are implemented as Java classes, which means that they can communicate efficiently using a shared address space.

We assume that we have access to the Java bytecode of processing units and that they are implemented using the DEFCON API (cf. §5). As a consequence, we can prevent them from using any JDK libraries (e.g. for I/O calls) or Java features (e.g. reflection) that are not strictly necessary for event processing. However, units may still contain bugs that cause them to expose confidential events to other units during regular processing, or they may explicitly try to use events with confidential data as part of their own processing to gain an illicit advantage.

Enforcing isolation between Java objects is not a trivial task because Java was not designed with this need in mind. Even if two Java objects never explicitly shared an object reference, they can exploit a wide range of *covert channels* to exchange information and violate isolation. Covert channels can be classified into storage and timing channels. Storage channels involve objects using unprotected, shared state to exchange data. Therefore we must close storage channels in Java. Since timing channels, which are caused by the modulation of system resources, such as CPU utilisation, are harder to exploit in practice, we ignore them in this work.

There is a large number of existing storage channels in Java, which can be exploited in three fundamental ways: (1) There are about 4,000 *static fields* in the Java Development Kit (JDK) libraries (in OpenJDK 6). For example, a static integer `Thread.threadSeqNum` identifies threads, which can be altered to act as a channel between two classes; (2) Java contains more than 2,000 *native* methods, which may expose global state of the Java virtual machine (JVM) itself. Native methods of standard classes such as `String` and `Object` retrieve data from global, internal data structures of the JVM; and (3) Java has *synchronisation* primitives that enable classes to exchange one bit of information at a time.

Several proposals have been made for achieving isolation in Java. As we explain below, they do not satisfy both of our two main requirements:

Low manual effort. It should be easy to add isolation support to any production JVM, with a minimal number of manual code changes. Many projects have been discontinued due in part to the difficulty of keeping them synchronised with JDK updates;

Efficient inter-isolate communication. The communication mechanism between isolated processing units should allow message passing with low latency and high throughout.

4.1 Existing approaches

Isolation of shared state. Existing approaches to achieving Java isolation involve a great deal of manual work. Modifying production JDKs is a daunting task, while, in comparison, the overall performance of research JDKs is lacking. Certifying a JVM to be free of storage channels would require an exhaustive inspection.

J-Kernel [19] and Joe-E [25] prevent access to global state in an ad hoc way: they restrict user code from defining new classes that contain mutable static fields. For the JDK libraries, they prevent access to classes or methods that are found to expose global state. They achieve this by providing custom proxies to `System`, `File` and other classes.

KaffeOS [4] reports to have manually assessed all of the JDK classes with static fields. Classes were rewritten to remove static fields, re-engineered to be aware of isolates or "reloaded". Reloading unsafe classes in the JVM results in per-isolate instances of static fields. However, this reloading mechanism cannot be applied to classes that are transitively referenced by a shared class, such as `Object`, requiring the manual assessment of a large number of classes.

Sun's MVM [10] and I-JVM [17] avoid manual examination of static fields by transparently replicating all of them per isolate. The JVM is modified to keep replicated copies of static fields per isolate. It also tracks which isolate is currently executing, making corresponding replicas visible to that isolate. MVM is the only project that reports to have attempted a complete assessment of the native methods that can expose global state. The cost of repeating this process for each new JVM release is considerable and, since MVM was completed only on Solaris/SPARC and is no longer maintained, reproducing it without detailed knowledge of JVM internals is hard.

Inter-isolate communication. MVM (similar to `.NET` AppDomains [33]) uses a separate heap space per isolate, which requires serialisation of objects exchanged between isolates. Incommunicado [30] improves MVM's inter-isolate communication by using deep-copying in place of serialisation. These approaches limit the performance of event processing applications because they require message passing to copy data. As we show in §6, this nullifies many of the performance advantages of sharing an address space between isolates.

Efficient inter-isolate communication is supported by KaffeOS and I-JVM, which allow objects to be shared between isolates. However, this is not appropriate for enforcing event flow control because once two isolates have established a shared object, the system can no longer separate them when their labels change. J-Kernel and JX [18] provide an approach better suited to DEFC: they use indirection through a proxy for objects created in dif-

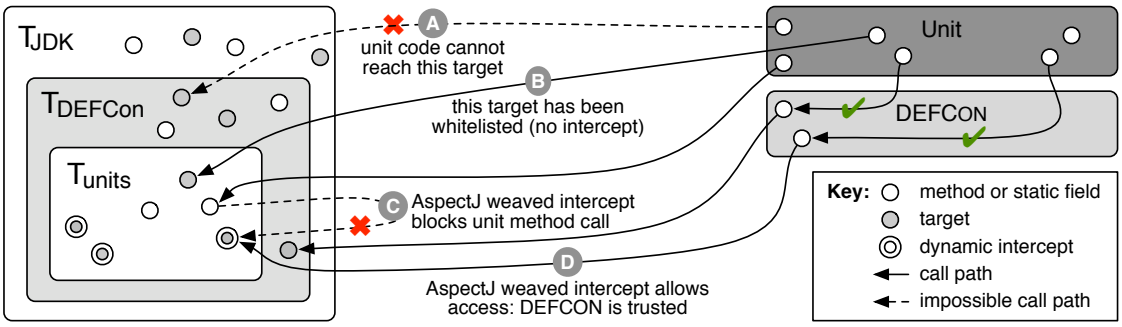

Figure 3: Illustrating our isolation enforcement between units using a combination of static white-listing and dynamic intercepts.

ferent isolates. However, their synchronous invocation model is at odds with decoupled event processing, which requires fast unidirectional communication.

4.2 Our isolation methodology

We describe a practical methodology for achieving Java isolation that provides fast, safe inter-isolate communication, while being easy to apply to new JDK versions. It does not require changes to the JVM or exhaustive code analysis.

We achieve efficient communication between isolates using message passing. Units do not have references to each other, only to objects controlled by DEFCON. For objects exchanged through events, we want to provide the semantics of passing objects by value, and exploit the single address space to avoid data copying. Our performance requirements preclude deep-copying of messages. Additionally, shared state is unacceptable because it violates isolation. Thus, we only allow units to exchange immutable objects, leaving it to units to perform copying only when needed.

We developed tools that help in the analysis of dangerous JDK *targets*: static fields, native methods and synchronisation primitives that could be used by units to communicate covertly. We were able to secure OpenJDK 6 in four days by manually inspecting only 52 targets (15 native methods, 27 static fields, and 10 synchronisation targets), without any modifications to the JVM.

As we illustrate in Figure 3, we divide potentially dangerous targets into three sets, T_{DEFCon}, T_{units} and T_{JDK}: a set of targets in the JDK only used by the DEFCON implementation (T_{DEFCon}), targets used by processing units (T_{units}), and targets used by neither (T_{JDK}). T_{units} was based on the event processing units that form the implementation of our trading platform described in §6.

Static dependency analysis. Targets not used at all (T_{JDK}), such as AWT/Swing classes, can be eliminated from the JDK without further impact. As a first step, we trim any classes that are not used by the DEFCON implementation or the event processing units of our financial scenario. This resulted in a subset of the JDK containing more than 2,000 used targets ($T_{\text{DEFCon}} \cup T_{\text{units}}$)—approximately 20% of the full JDK.

A significant proportion of these targets are only accessed by the DEFCON system (T_{DEFCon}) because they are not useful to units for processing events. Typically, (non-malicious) units use classes from the `java.lang` and `java.util` packages and have little reason to directly access classes from packages, such as `java.lang.reflect` or `java.security`. Thus we define a custom class loader that constrains the JDK classes that units can access to a *white-list*—e.g. precluding calls such as the one labelled 'A' in Figure 3.

However, restricting the set of classes alone does not prevent transitive access to dangerous targets. When the custom class loader permits the resolution of a white-listed JDK class, the loading of the class is delegated to the JVM bootstrap class loader. If the class contains references to other JDK classes, they are directly resolved by the JVM bootstrap classloader and therefore cannot be controlled.

Reachability analysis. In order to address the problem discussed above, a static analysis tool computes all targets that are transitively reachable from classes specified in the custom class loader white-list, i.e. T_{units} targets. This analysis enumerates possible method-to-method execution paths. The reachability analysis must cover code paths that involve dynamic method dispatch; a call to a given signature in the bytecode could execute code from any compatible subtype. Although the previous dependency analysis reduces the number of false positives in this phase, T_{units} still has 1,200 dangerous targets reachable from `java.lang`—approximately 320 native methods and 900 static fields.

Heuristic-based white-listing. Some of the targets in T_{units} can be declared safe using simple heuristics:

- We can white-list the 66 static fields and 20 native methods from the `Unsafe` class. This class provides direct access to JVM memory and is guarded by the Java Security Framework. Any access to it from user code would be a critical JVM bug.

- Some final static fields classified as immutable, such as strings or boxed primitive types, can be shared because they are constants.
- The use of some private static fields can be determined to be safe: vectors of constants and primitive fields that are not declared "final" but are only written once.

Another tool white-lists according to the above heuristics, reducing the number of dangerous targets to approximately 500 static fields and 300 native methods. Such cases are represented in Figure 3 by the call labelled 'B'.

Automatic runtime injection. To secure targets in T_{units} left after the preceding static analysis stage, we would have to duplicate unsafe static fields and manually assess native methods for covert communication channels, as done by other JVM isolation projects. In contrast to these projects, we wanted to avoid any JVM source code modification and to minimise the number of native JDK methods that needed to be checked.

For this reason, we employ aspect-oriented programming (AOP) [21]: by modifying JDK code in a programmatic way, we can duplicate static fields without changing the JVM and inject access checks to protect the execution of native methods. We employ the MAJOR/FERRARI framework [40] because it can manipulate JDK bytecode, as well as our own code, using the AspectJ language. We specify *pointcuts* to intercept all targets left after our static analysis, as follows:

Native methods: When access to a native target is as part of a call to the DEFCON API (described in §5), we can consider it safe by assuming the API is correctly designed (call 'D' in Figure 3). Otherwise we raise a security exception (call 'C').

Static fields: When a static field can be cloned without creating references that are shared with the original, we do an on-demand deep copy and create a per-unit reference. This occurs on a `get` access for most types, but can be deferred to the time of a `set` method for primitive or constant types. If field copying is not possible, we raise a security exception.

Manual white-listing. In this way, we automatically close JDK covert storage channels without changes to the JVM. However, before running the units in our financial scenario, we had to manually check 15 native methods and 27 static fields, which were intercepted and raised security exceptions. Below are a few examples of manually white-listed targets with a brief justification:

`java.lang.Object.hashCode`: This effect of this method is equivalent to reading a constant field.

`java.lang.Object.getClass`: Since `Class` objects are unique and constant, this method essentially retrieves a constant static field.

`java.lang.Double.longBitsToDouble`: This method does not access any JVM state.

`java.lang.System.security`: This target is safe because the reference to the security manager is protected from modification by units.

While the above methodology results in safe isolation, intercepting targets adds an overhead. We therefore profile the execution paths of units to identify frequently encountered targets that may be white-listed manually. During this profiling, we discovered 15 additional frequently-accessed targets (6 static fields and 9 native methods) that we were able to white-list.

4.3 Restricting synchronisation channels

As explained in §3.2, the DEFCON system must ensure that references held by one unit cannot escape to another unit. To avoid serialisation or deep-copying and to prevent the establishment of unrestricted shared state, units are limited to exchanging immutable objects whose references can be shared safely. However, every Java object, even if it is immutable, has a piece of modifiable information: its synchronisation lock. The lock is modified by `synchronized` blocks and by `wait` and `notify` calls.

This need to control synchronisation on shared objects also closes a further Java-specific channel due to the "interning" of strings. A string that has been interned is guaranteed to have a unique reference, common with all other strings of the same value in the JVM. This lets reference comparison (==) replace the more expensive `equals` method.

Previous proposals [10, 17] to avoid synchronisation on shared objects such as interned `String`s and `Class`es provide a copy per isolate. This would defeat the purpose of our message passing scheme that uses shared objects with the intent of avoiding copying them.

Automatic runtime injection. Instead we allow units to synchronise only on types that are guaranteed to never be shared with other units. This is indicated by the type in question implementing our `NeverShared` tagging interface. A type T can implement `NeverShared` as long as (a) the DEFCON system prevents instances of T being put into events, (b) no (white-listed) native method can return the same instance of T to two different units, and (c) no static field of type T is white-listed as being safe. Neither `Class` nor `String` objects satisfy these requirements and thus units cannot synchronise on them.

Units can instead make their own types for synchronisation that implement `NeverShared`. If a type is statically known to implement `NeverShared`, then synchronisation happens with no runtime overhead. Otherwise AOP will be used to inject a runtime type check: if this check fails and the attempt to synchronise comes from a unit, a security exception is raised.

DEFCON API call	Description
createEvent() $\rightarrow e$	Creates a new event e.
addPart(e, S, I, *name*, *data*)	Adds to event e a new part *name* containing *data* with label (S, I).
delPart(e, S, I, *name*)	Removes from event e part *name* with label (S, I)
readPart(e, *name*) $\rightarrow$ *(label, data)**	Returns the data in part *name* of event e. If there are multiple visible parts with the same *name*, all are returned. $S_p \subseteq S_u^{in}$ and $I_p \subseteq I_u^{in}$ must hold for every part returned to the unit.
attachPrivilegeToPart(e, *name*, S, I, t, p)	Attaches a privilege p over a tag t to part *name* with label (S,I) to create a privilege-carrying event for delegation (cf. §3.1.5). The call succeeds if the caller has t^{pauth}.
cloneEvent(e, S, I) $\rightarrow e'$	Creates a new instance e' of an existing event e. All the tags in the caller's output confidentiality label are attached to each part's label and only the caller's output integrity tags are maintained on each cloned part. This precludes DEFC violations based on observing the number of received events.
publish(e)	Publishes a new event e. Events without parts are dropped.
release(e)	Releases an event e (cf. §3.1.6).
subscribe(*filter*) $\rightarrow s$	Subscribes to events with a non-empty *filter*, creating a subscription s. The *filter* is an expression over the name and data of event parts. For an event to match, $S_p \subseteq S_u^{in}$ and $I_p \subseteq I_u^{in}$ must hold for each part in the filter at the time of matching.
subscribeManaged(*handler*, *filter*)$\rightarrow s$	Declares a *managed subscription* s that enables a unit to process multiple tags without contaminating its state permanently. DEFCON then creates and reuses separate unit instances with contaminations appropriate for the processing of incoming events. Units with managed subscriptions are similar to Asbestos' event processes [13].
getEvent() $\rightarrow (e, s)$	Blocks the caller until an incoming event e matches one of the unit's subscriptions s.
instantiateUnit(u', S, I, $O_{u'}^{p}$, $O_{u'}^{\text{pauth}}$)	Instantiates a new unit u' at a given label (S, I), as long as it can delegate privileges to the new unit. The new unit inherits the caller's contamination.
changeOutLabel($\langle$S|I$\rangle$, $\langle$add|del$\rangle$, t)	Adds/removes tag t to/from a unit's output label $(S_u^{\text{out}}, I_u^{\text{out}})$ independently of the input label $(S_u^{\text{in}}, I_u^{\text{in}})$. The unit can then declassify/endorse parts with tag t (cf. §3.1.4).
changeInOutLabel($\langle$S|I$\rangle$, $\langle$add|del$\rangle$, t)	Adds/removes tag t to/from a unit's input label and output label.

Table 1: Description of the DEFCON API available to event processing units. Note that due to contamination independence S and I in API calls may be transparently changed by the system: $S' = S \cup S_u^{\text{out}}$ and $I' = I \cap I_u^{\text{out}}$

Manual inspection. JDK methods that synchronise on locks cannot safely be accessed from units. For example, `Classloader.loadClass()` and many `StringBuffer` methods are synchronised. However, both are types that are never shared, i.e. they satisfy the above three requirements. Instead of modifying them in the JDK source-code, we transformed them to implement `NeverShared` through an aspect that is applied before the interception aspect.

5 DEFCON API

We built a DEFCON prototype system in Java that implements the DEFC model and enforces isolation as described in §4. The API calls that units may use to interact with the DEFCON system are described in Table 1.

Contamination independence. Most of the calls do not impose restrictions on the caller, yet they are safe because of a unit's contamination. Calls such as `addPart()`, which adds a new part to an event (cf. Table 1), should not fail if a unit is unable to write at the requested contamination level because units may not be aware of their initial contamination. Instead DEFCON guarantees that any tags present in the unit's current output label are attached transparently to generated parts. For example, a unit with a label $S_u^{\text{out}} = \{d\}$ that invokes `addPart` with label $S = \{t\}$ causes that part to be labelled $S' = \{d, t\}$. This highlights an important property of the API: *contamination independence*. It allows a unit to be sandboxed by instantiating it at a higher contamination level that it is unaware of. All of its input and output will be affected by this initial contamination.

Freezing shared objects. Most of the API calls receive or return potentially mutable objects. References to these objects may not be communicated to other units since changes to their state cannot be controlled. In particular, this applies to objects representing event parts and labels.

The `addPart()` call allows a unit to include objects of various types in a part. For immutable types, making shared references is safe. However, this is not true for mutable types (e.g. `Date`) or collection types that support adding multiple objects to a part (e.g. `HashMap<Date>`). To avoid the cost of serialising and copying such types during event dispatching, DEFCON limits contents of event parts to a subset of types. These types must be either immutable or extend a package-private `Freezable` base class.

For `Freezable` objects, mutating operations are disallowed after a call to `freeze()` has been made. This incurs the overhead of checking an `isFrozen` flag on each mutating operation. For collection classes, a call to `freeze()` must efficiently freeze all contained objects. To avoid iterating through collections, each `Freezable` object that is attached to a `Freezable` collection has a reference to the collection's `isFrozen` flag. This makes `freeze()` a constant time operation. The overhead of mutating operations on a `Freezable` object is linear with the number of collections the object is part of. A similar approach is used to make the `Label` type immutable.

6 Evaluation

The goal of our experimental evaluation is to demonstrate the practicality of the DEFC model in high-performance event processing. We describe the implementation and evaluation of a simple financial stock trading platform built using DEFCON. We compare our implementation to *Marketcetera* [3], a popular open source trading platform written in Java. Although only one of few open source offerings in a space dominated by proprietary solutions, Marketcetera is gaining momentum by providing performance comparable to proprietary systems [35] and features such as rapid strategy development, complex event processing and interaction with various exchanges.

We quantify event processing performance using two metrics: *event throughput*, the number of events processed per unit time, and *event latency*, the delay that events experience when being processed. We also measure the overhead of our DEFC approach on event-driven applications in terms of processing performance and memory consumption.

DEFCON isolates the trading strategies of traders, thus allowing multiple strategies to be hosted on the same machine as the stock market feed. Instead, Marketcetera uses multiple JVMs, one per client, to isolate trading strategies, limiting the scalability of a similar deployment to a smaller number of traders. DEFCON achieves low event processing latency and high throughput, while scaling to 10 times the number of traders compared to Marketcetera. As explained in §2.1, this makes co-location affordable to more traders because a single machine can securely serve a larger number of trading strategies.

6.1 Financial trading scenario

We adopt a classic trading algorithm called *pairs trade* [39]. It is based on the observation that changes in the stock prices of established companies in the same industry are frequently correlated. Traders use this to predict the stock price for the immediate future and gain a return. The performance of the pairs trade algorithm depends heavily on the latency of trades: co-located traders' orders will increase the price of the cheapest stock and decrease the most expensive one, thus limiting the profit margins for remote traders.

Marketcetera implementation. We implemented the pairs trading strategy as a Strategy Agent in Marketcetera 1.5.0. Strategy Agents host one or more strategies of the same client. For isolation, a separate JVM is created for each client's Strategy Agent. Since third-party libraries may leak client data that they receive, each client must vouch for their strategy's implementation or fully trust its developer. To use brokering services, Strategy Agents communicate with an Order Routing Service (ORS) that forwards requests to an exchange. We extended the ORS to provide local brokering facilities and a corresponding market data feed to the Strategy Agents.

DEFCON implementation. DEFCON allows competing traders to execute latency-sensitive operations in the same JVM. The confidentiality guarantees provided by the DEFC model enable concurrent execution of proprietary trading algorithms without fear of disclosure. As illustrated in Figure 4, our trading platform has the following processing units:

Trader units encapsulate traders' strategies for buying and selling stocks using pairs trading.

Pair Monitor units provide pairs trading as a service since it is used by all traders in our system. Based on a stock pair and an investment threshold, it sends events to traders when the expected price difference occurs.

A **Local Broker unit** enables traders to clear their orders locally, without the need to involve the stock exchange, by matching traders' bid/ask orders.

A **Stock Exchange unit** is responsible for the communication with the stock exchange. In its simplest form, it is the source of events regarding trades that occur there.

A **Regulator unit** samples a subset of local trades on behalf of a regulatory body. It may verify that the volume of a trader's trades has not exceeded a given quota.

DEFCON operation. We describe the operation of our trading platform using steps 1–9 in Figure 4.

Step 1: A Trader unit declares its interest in a given pair of stocks. The published event is protected by a unique trader tag t_1, owned by Trader 1. Since the selection of stocks and the parameters of the pairs trade are sensitive information that must be protected from disclosure, Trader 1 delegates the t_1^+ privilege only to the corresponding Pair Monitor unit. This Pair Monitor uses this privilege to learn about the pair of symbols to monitor. All its output will only be visible to the Trader 1 that owns t_1.

Step 2: The Pair Monitor issues two tick event subscriptions: one for each of the two symbols that it has to mon-

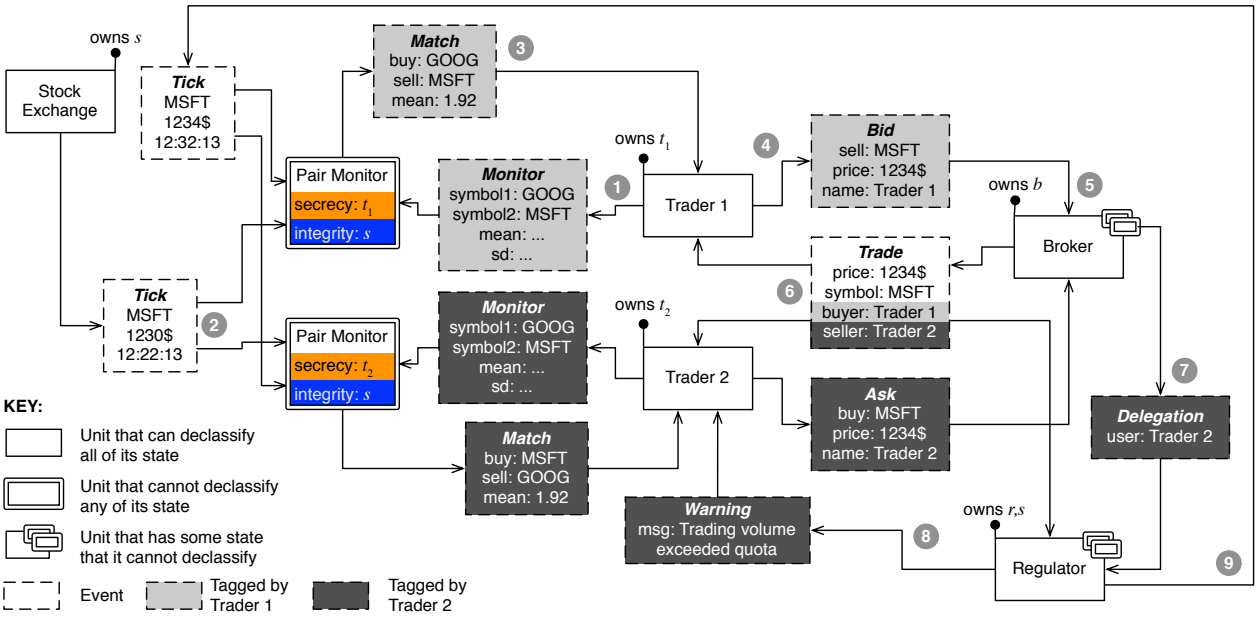

Figure 4: Workflow of the implementation of our stock trading platform in DEFCON, highlighting the DEFC aspects.

itor. Pair Monitor units are always instantiated with read integrity s and are thus only able to perceive events published by the Stock Exchange unit that owns s.

Step 3: Once a tick event is published with an adequate price, the Pair Monitor sends an event to the Trader. This event is tagged with t_1 and Trader 1 is the only unit with the necessary confidentiality read label to receive it.

Step 4: Trader 1 may decide to sell stocks using the local brokering facilities. A bid order is generated with the offered price and the issuing Trader's details. The issuing Trader has three different security requirements for a published order: (1) it must convey the details of the order to the Broker to be matched against other orders; (2) no other unit, apart from the Broker, should be able to associate the order with the issuing Trader; (3) no unit should be able to correlate two orders by the same Trader. A competing Trader correlating orders may learn the underlying trading strategy. In addition, a Trader does not trust the Broker not to reveal information about trades. To capture these security requirements, the first part of the bid, price/symbol, is protected by a broker tag b while the second, name, is protected both by b and by a randomly-generated tag t_r used only for this order. The Broker has b^+ and b^-. The first part also carries the privilege t_r^+, allowing only the Broker to see the Trader's name as long as it accepts the additional contamination.

Step 5: The Broker can read the first part and declassify it. It has to use a *managed subscription* (cf. §5) to learn the Trader's identity. Once a corresponding ask order is received, the Broker matches it and completes a trade.

Step 6: The first part of the trade event is declassified and visible publicly. The two additional parts with the sensitive identities of the Traders are protected individually by unique tags. Each Trader can identify its own trades while DEFCON guarantees that no other unit can do the same.

Step 7: A Regulator may intercept trades to verify that they are compliant with trading rules. If it observes a suspicious trade, it uses a managed subscription to receive t_r^+ over the unique tag t_r that protects the Trader's identity. Since this privilege is only needed for suspicious trades, the Regulator receives it from the Broker on-demand. The delegation event is only possible as long as $t_r^{+\text{auth}}$ was included in the second part of the bid order in step 4.

Steps 8+9: With this privilege, the Regulator can communicate a warning to the Trader. The Regulator owns the integrity tag s and is able to republish the local trade as a valid stock tick perceivable by the Pair Monitors.

6.2 Experimental results

We evaluated our system with a synthetic workload of stock tick events that was derived from traces of trades made on the London Stock Exchange. In our workload, we selected the tick prices so that they triggered the pairs trading algorithm for each pair once every 10 ticks. This approach both generated a significant order load and also allowed us to avoid the issue of choosing suitably correlated pairs from real market data. Since the main bottleneck was the filtering that occurred between the Stock Exchange and the Pair Monitor units, the tick rate achieved only caused transient queuing in the system.

We varied the number of Traders on the platform. Each Trader monitors a single symbol pair that was chosen ac-

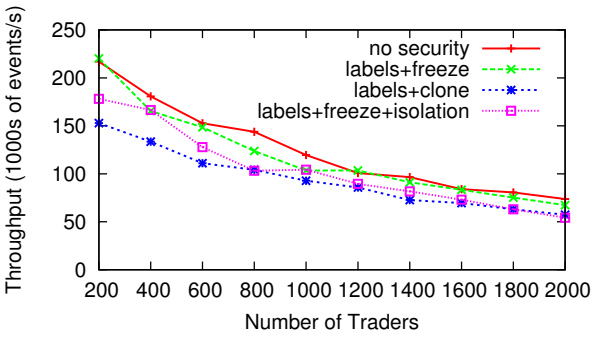

Figure 5: Maximum supported event rate in DEFCON as a function of the number of traders.

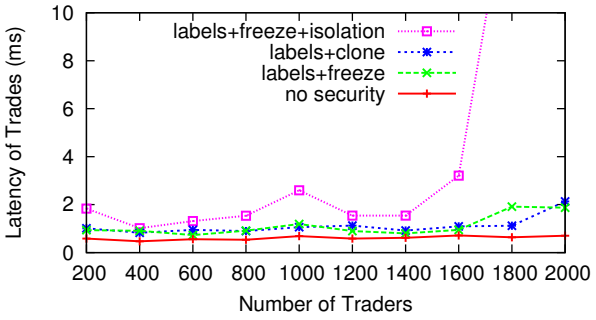

Figure 6: Event processing latency in DEFCON as a function of the number of traders.

cording to a Zipf distribution. This emulated the fact that some symbol pairs are well known to be correlated and, as a result, the majority of Traders monitor their prices. All tests were run on a dual processor Intel Xeon E5540 2.53 GHz machine with a maximum of 1 GiB heap memory using Sun's Hotspot JVM, version 1.6.0_16.

DEFCON performance. To explore the limits of our DEFCON deployment, we had the Stock Exchange unit replay tick event traces as quickly as possible, while measuring the achieved throughput every 100 ms. Figure 5 shows the median throughput when increasing the number of Traders in the system. In the simplest case without security (*no security*), the system performance

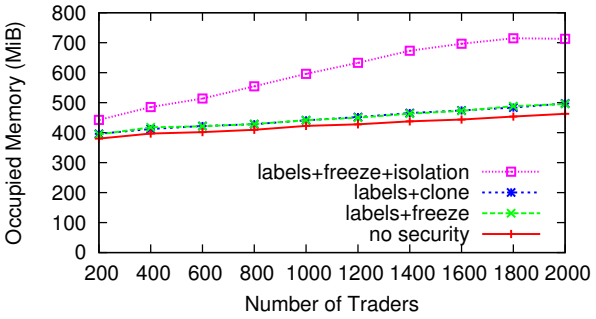

Figure 7: Amount of used memory in DEFCON as a function of the number of traders.

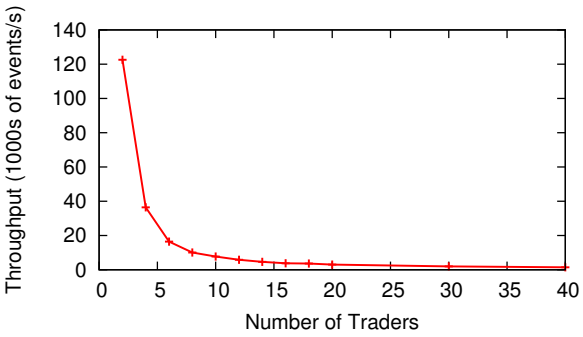

Figure 8: Maximum supported event rate in Marketcetera as a function of the number of traders.

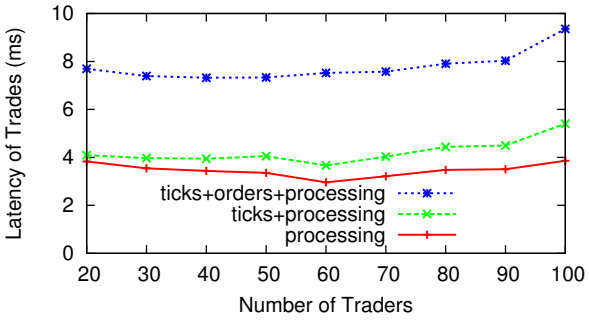

Figure 9: Event processing latency in Marketcetera broken down into individual contributions.

ranges from 220,000 events per second with 200 Traders to 75,000 events with 2,000 Traders. (Note that the Stock Exchange unit in our implementation is single-threaded.) The overhead of introducing labels and freezable objects (*labels+freeze*) is within the error margin, while the overhead of cloning (*labels+clone*) is around 30%, even with the simple data structures of our financial application. The overhead of adding isolation (*labels+freeze+isolation*) is around 20%, staying constant with the number of Traders.

Next we measured latency as the time difference between when a trade event is produced by the Broker and the time when the originating tick event occurred. This includes the processing time of the Stock Exchange, Pair Monitor, Trader and Broker units. In Figure 6, we plot the 70th percentile of latencies, again increasing the number of Traders. We ignore higher latency percentiles because they are affected by the characteristics of the workload and the operation of the Java garbage collector. Spikes in trading activity, as commonly found when markets open, result in transient congestion in the Broker and thus queueing of events. Short periodic activations of the garbage collector preempt processing threads for about 20 ms and increase the latency of individual events.

Figure 6, shows that the latency without security (*no security*) is about 0.5 ms independently of the number of Traders. Introducing label checks into the system sees la-

tencies rise to approximately 1 ms without and 2 ms with isolation. This behaviour continues up to 1,500 Traders after which the system becomes overloaded.

In Figure 7, we measured the memory consumption in the above experiment. Of the total memory consumed, about 300 MiB is used to cache tick events. We observe that while the overhead of (*labels+freeze*) is minimal compared to the base case, the weaving framework incurs an overhead of 50 MiB for 200 Traders, and up to 200 MiB for 2,000 Traders.

Marketcetera performance. We compare the results from DEFCON with the performance of Marketcetera. The median throughput of Marketcetera is shown in Figure 8. Although the event rate for only 2 Traders is high, the system does not scale well. (Note the lower number of Traders compared to Figure 5.) With just 10 active Traders, the throughput falls below 10,000 events per second. This is mostly due to Strategy Agents filtering market data individually as the platform does not support centralised market data filtering. Memory consumption is also significantly higher in Marketcetera. Starting from 2 GiB for 20 Traders, the used memory across all JVMs reaches 6 GiB for 100 Traders. Without multiple JVMs allocating memory, DEFCON manages to support 1,500 Traders using less than 1 GiB of heap space.

For measuring latency in Marketcetera, we chose a low rate of 1,000 events per second in the stock feed with a small number of Traders. This reduces the CPU load and allows us to draw conclusions about latency while not being affected by scheduling phenomena. In Figure 9, we show the 70^{th} percentile of trades' latencies in Marketcetera, measured at the broker, when increasing the number of Traders. (Figure 6 shows the corresponding result for DEFCON but again note the lower number of Marketcetera Traders.) Latency in Marketcetera is around 8 ms. The plot breaks this total latency down into its individual contributions: the time to filter unwanted events and execute the pairs trading algorithm (*processing*), the time for tick propagation from the Market Feed to the Strategy Agents (*ticks+processing*) and order propagation from Strategy Agents to the ORS (*ticks+orders+processing*). When we introduced 100 Traders, the increasing cost of communication across JVMs surpassed the actual processing latency. In contrast, DEFCON is able to provide latency at around 1 ms for significantly more Traders. We believe that this is because DEFCON tick propagation uses our event dispatching mechanism, which does not involve communication across JVMs.

Security comparison. In DEFCON, the pairs trading algorithm is a service that each Trader may decide to use. DEFC guarantees that the unit that implements the algorithm does not have the ability to leak traders' choices. Marketcetera requires that each Strategy implementation be fully-trusted. As a result, it does not support third-party services that Strategies may use. Moreover, DEFCON enables the efficient reuse of a single trading strategy across multiple users; Marketcetera instead requires a new JVM each time. Code reuse is particularly beneficial when traders belong to the same organisation such as a small hedge fund. The Marketcetera Broker can, deliberately or involuntarily, leak a user's trades or orders to other parties. In contrast, the DEFCON Broker is prevented by DEFC from correlating two clients' orders. The Broker's developers can guarantee that no bugs may result in data leaks that violate the Traders' confidentiality. In addition, a regulatory service can only reliably be integrated into Marketcetera by its original developers. Instead, DEFCON can support a Regulator unit without concern that the additional code may damage the security properties of the system.

7 Conclusions

High-performance event processing applications, for example as found in algorithmic stock trading, need strong information security without sacrificing performance. We presented DEFCON: an event processing system that enforces *decentralised event flow control* (DEFC). This model meets the particular security needs of event processing by providing mandatory protection of event data from bugs and intentional leaks. DEFCON relies on isolation at the programming language level and we described a practical methodology for achieving isolation in Java with low manual effort. By isolating processing units running in the same address space, we tried to strike an optimal balance between the need for isolation and efficient inter-isolate communication. Our evaluation shows the practicality of our solution when compared to an open source trading platform.

In future work, we plan to investigate issues in a distributed system build from a set of DEFCON nodes. We also want to explore additional techniques for isolation in Java, such as dynamic recompilation of classes, and approaches that facilitate provably secure hand-off of Java references, such as Kilim [36]. Although we do not address denial-of-service attacks in this work, we believe that thanks to our message passing paradigm it is possible to use common profiling techniques from aspect-oriented programming for resource accounting [40]. In related work, we have begun to investigate how the DEFC model can be applied to functional languages such as Erlang that naturally support isolation between components through message-passing [31].

Acknowledgements

We thank Oli Bage, Eva Kalyvianaki, Cristian Cadar and Jonathan Ledlie for comments on earlier paper drafts.

We also thank our anonymous reviewers and our shepherd, Nickolai Zeldovich, for their guidance. This work was supported by grants EP/F042469 and EP/F044216 ("SmartFlow: Extendable Event-Based Middleware") from the UK Engineering and Physical Sciences Research Council (EPSRC).

References

[1] AITE GROUP. Market data infrastructure challenges, April 2009. www.aitegroup.com/reports/200904222.php.

[2] Progress Apama Website. www.progress.com/apama.

[3] ASAY, M. Marketcetera gives hedge funds cloud-based trading. *Cnet News* (2009).

[4] BACK, G., HSIEH, W. C., AND LEPREAU, J. Processes in KaffeOS: Isolation, resource management, and sharing in Java. In *OSDI'00* (San Diego, CA, USA), USENIX, pp. 23–23.

[5] BEERI, C., EYAL, A., MILO, T., AND PILBERG, A. Monitoring business processes with queries. In *VLDB'07* (Vienna, Austria).

[6] BELL, D. E., AND LA PADULA, L. J. Secure computer systems: Mathematical foundations and model. Tech. Rep. M74-244, The MITRE Corp., Bedford, MA, USA, May 1973.

[7] Betfair website. www.betfair.com.

[8] BREWER, D., AND NASH, M. The Chinese Wall security policy. In *IEEE Symposium on Security and Privacy* (Oakland, CA, USA, May 1989).

[9] CLARK, J. Still the need for speed. *Waters* (2008).

[10] CZAJKOWSKI, G., AND DAYNES, L. Multitasking without compromise: A virtual machine evolution. In *OOPSLA'01*.

[11] DEPARTMENT OF DEFENSE. Trusted computer system evaluation criteria (Orange Book), 1983.

[12] DUHIGG, C. Stock traders find speed pays, in milliseconds. *The New York Times* (2009).

[13] EFSTATHOPOULOS, P., KROHN, M., VANDEBOGART, S., ET AL. Labels and event processes in the Asbestos Operating System. In *SOSP '05* (Brighton, UK), ACM, pp. 17–30.

[14] Esper website. esper.codehaus.org.

[15] FINANCIAL SERVICE AUTHORITY. Press releases, 1997-2009. available at www.fsa.gov.uk.

[16] FLATLEY, R. Check your speed. *The Trade News* (2007).

[17] GEOFFRAY, N., THOMAS, G., MULLER, G., ET AL. I-JVM: a Java virtual machine for component isolation in OSGi. In *Dependable Systems and Networks (DSN)* (Estoril, Portugal, April 2009), p. 10.

[18] GOLM, M., FELSER, M., WAWERSICH, C., AND KLEINÖDER, J. The JX Operating System. In *USENIX ATC'02* (Monteray, CA, USA), pp. 45–58.

[19] HAWBLITZEL, C., CHANG, C.-C., CZAJKOWSKI, G., HU, D., AND VON EICKEN, T. Implementing multiple protection domains in Java. In *USENIX ATC'98* (New Orleans, LA, USA).

[20] IATI, R. The real story of trading software espionage. *Advanced Trading* (2009).

[21] KICZALES, G., LAMPING, J., AND OTHERS, A. M. Aspect-oriented programming. In *ECOOP'97* (Jyvskyl, Finland).

[22] KROHN, M., YIP, A., BRODSKY, M., ET AL. Information flow control for standard OS abstractions. In *SOSP'07* (Stevenson, WA, USA), ACM, pp. 321–334.

[23] LONDON STOCK EXCHANGE. Hosting capacity increases fivefold. Press Release, November 2009.

[24] LUCKHAM, D. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley, 2002.

[25] METTLER, A., WAGNER, D., AND CLOSE, T. Joe-E: A security-oriented subset of Java. In *Network and Distributed System Security (NDSS)* (Dan Diego, CA, USA, 2010), Internet Society.

[26] MÜHL, G., FIEGE, L., AND PIETZUCH, P. *Distributed Event-Based Systems*. Springer-Verlag, 2006.

[27] MYERS, A., AND LISKOV, B. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology 9*, 4 (2000), 410–442.

[28] MYERS, A. C. JFlow: Practical mostly-static information flow control. In *POPL'99* (San Antonio, TX, USA).

[29] NAIR, S., SIMPSON, P., ET AL. A virtual machine based information flow control system for policy enforcement. *Electronic Notes in Theoretical Computer Science 197*, 1 (2008), 3–16.

[30] PALACZ, K., VITEK, J., CZAJKOWSKI, G., AND DAYNAS, L. Incommunicado: efficient communication for isolates. In *OOPSLA'02* (Seattle, WA, USA), ACM, pp. 262–274.

[31] PAPAGIANNIS, I., MIGLIAVACCA, M., EYERS, D. M., SHAND, B., BACON, J., AND PIETZUCH, P. Enforcing user privacy in web applications using Erlang. In *Web 2.0 Security and Privacy (W2SP)* (Oakland, CA, USA, 2010), IEEE.

[32] ROY, I., PORTER, D. E., BOND, M. D., MCKINLEY, K. S., AND WITCHEL, E. Laminar: practical fine-grained decentralized information flow control. In *PLDI'09* (Dublin, Ireland), ACM.

[33] SCHANZER, E. Performance considerations for run-time technologies in the .NET framework. msdn.microsoft.com/en-us/library/ms973838.aspx, Aug 2001.

[34] SCHMERKEN, I. Skyrocketing market data message rates leading trading firms to consider hardware acceleration. *Wall Street & Technology* (2007).

[35] SHOEMAKER, K. Marketcetera 1.5 release note. *Ostatic* (2009).

[36] SRINIVASAN, S., AND MYCROFT, A. Kilim: Isolation-typed actors for Java. In *ECOOP'08* (Paphos, Cyprus).

[37] STONEBRAKER, M., ÇETINTEMEL, U., AND ZDONIK, S. The 8 requirements of real-time stream processing. *SIGMOD Rec. 34*, 4 (2005), 42–47.

[38] THOME, B., GAWLICK, D., AND PRATT, M. Event processing with an Oracle database. In *SIGMOD'05* (Baltimore, MD), ACM.

[39] VIDYAMURTHY, G. *Pairs Trading: Quantitative Methods and Analysis*. Wiley Finance, 2004.

[40] VILLAZÓN, A., BINDER, W., AND MORET, P. Aspect weaving in standard Java class libraries. In *Principles and Practice of Programming in Java (PPPJ)* (Modena, Italy, 2008), ACM.

[41] YIP, A., WANG, X., ZELDOVICH, N., AND KAASHOEK, M. F. Improving application security with data flow assertions. In *SOSP'09* (Big Sky, MT, USA), ACM, pp. 291–304.

[42] ZELDOVICH, N., BOYD-WICKIZER, S., AND MAZIÈRES, D. Securing distributed systems with information flow control. In *NSDI'08* (San Francisco, CA, USA), pp. 293–308.

[43] ZELDOVICH, N., KOHLER, E., ET AL. Making information flow explicit in HiStar. In *OSDI'06* (Seattle, WA, USA), USENIX Association, pp. 19–19.

[44] ZENDRIAN, A. Don't be afraid of the dark pools. *Forbes* (May 2009).