Identifying Inter-Component Communication Vulnerabilities in Event-based Systems

Technical Report: USC-CSSE-17-801

Youn Kyu Lee, Daye Nam, and Nenad Medvidovic
Computer Science Department, University of Southern California
941 Bloom Walk, Los Angeles, California, USA 90089
{younkyul, dayenam, neno}@usc.edu

ABSTRACT

Event based systems are flexible, scalable, and adaptable based on its feature of non-determinism in event communication. However, this may yield security vulnerabilities in event communication between components. For example, malicious components can steal sensitive data or manipulate other components in an intended way. This paper introduces *SCUTUM*, a novel technique that automatically detects vulnerable event communication channels from event-based systems by combining static flow analysis and pattern matching. *SCUTUM*'s evaluation demonstrated that it identifies vulnerable event communication channels with higher accuracy than existing techniques from 28 real-world apps and it is applicable to the apps comprising a number of components.

1. INTRODUCTION

Event-based systems (EBS) developed using message-oriented middleware (MOM) platforms are widespread [2–4, 15] The popularity of EBS is originated by its high flexibility, scalability, and adaptability. These advantages are facilitated by its reliance on implicit invocation and implicit concurrency in its event processing. Specifically, in EBS, components may not know the consumers of events they publish, nor do they necessarily know the producers of events they consume. However, this non-determinism in event consumption can introduce inherent security vulnerabilities into a system.

In this paper we target one specific type of security vulnerabilities related to event processing in EBS, *event attacks*. Event attacks are particular type of security attacks caused by non-determinism in EBS's event communication model. Event attacks abuse, incapacitate, and damage the system by launching unintended behaviors or leaking sensitive information through event exchanges. Different types of event attacks have been identified throughout various domains [12–14, 17, 18, 20, 32, 37, 38, 40], such as mobile systems [3] and web applications [4].

Existing security inspection techniques neither focus on event attacks nor correctly detect vulnerabilities across components [11,14, 32]. The inherent characteristics of EBS hamper an accurate analysis of target system. Specifically, existing flow-analysis techniques do not support analysis of implicit invocation between event-clients, which do not explicitly reveal event interfaces. Moreover, for the system that comprises a large number of components, existing flow-analysis techniques are not scalable to analyze it. While a large body of research has been studied on detecting event attacks in Android [12, 18, 21, 24–26, 31, 34, 41, 43], they cannot be directly applied to other types of EBS, since Android operates by its system-specific communication model, APIs, and life-cycles.

To overcome aforementioned challenges and the shortcomings of existing approaches, we designed *SCUTUM* (SeCUrity for evenT-based systems implemented Using MOM platforms), a technique that automatically detects vulnerabilities on event attacks from EBS. *SCUTUM* statically analyzes vulnerabilities by examining control-/data-flow across components as well as event communication patterns in target EBS. *SCUTUM* is distinguished from the existing research because (1) it simultaneously detects multiple types of event attacks, (2) it supports different types of MOM platforms, (3) it extends the detection coverage via a novel combination of static flow analysis and communication pattern matching, and (4) it provides efficient analysis scaling to a number of components.

We have evaluated *SCUTUM*'s ability of identifying vulnerable event communication channels from target EBS. To this end, we created a test benchmark comprising 30 open source testing ground apps from different domains. We also included three open-source event-based applications from different domains [22, 35]. *SCUTUM* was able to identify vulnerable communication channels with high accuracy compared to other state-of-the-art techniques targeting security vulnerabilities. We evaluated *SCUTUM*'s performance by measuring the analysis time on different numbers of apps, and the reduction of overhead. Our results showed that *SCUTUM* was also scalable to the systems comprising a large number of components.

This paper makes three contributions: (1) *SCUTUM*, a technique that accurately finds vulnerable event communication channels from target EBS through a novel combination of data-flow analysis and compositional pattern matching; (2) a full open-source implementation of *SCUTUM*, (3) a novel, open and comprehensive benchmark for EBS vulnerability analyses, (4) a set of evaluations of *SCUTUM* that involve 33 real-world EBSs and compare *SCUTUM* to existing alternatives. Section 2 illustrates event attacks. Section 3 details *SCUTUM* and Section 4 describes evaluation.

2. MOTIVATION

In this section, we introduce different types of event attacks and present a simplified example of an event attack that *SCUTUM* targets.

Event attacks represent the security problems that exploit event-based communication model. The research to date has identified the following types of event attacks:

- *Spoofing*: A malicious component can send an event that spoofs a target component in order to exploit the target's functionality or data.

- *Interception*: A malicious component can intercept an event which is supposed to be sent to other components; and can send back inappropriate replies.

Listing 1: Component `VicClient` of `App1`

```
public class VicClient {
 ...
 Message e = queueSession.createMessage();
 e.setJMSType("TextMessage");
 e.setStringProperty("UserInfo", "private_info");
 sendMessage(e, 1, queueSender);
 ...
}
```

Listing 2: Component `MalClient` of `App1`

```
public class MalClient {
 ...
 public void handleMessage(Message e) {
  String info = e.getStringProperty("UserInfo");
  LeakSensitiveInfo(info);
 }
}
```

- *Eavesdropping*: A malicious component can eavesdrop an event that contains sensitive data which is supposed to be only open to particular components.

- *Confused Deputy*: A malicious component can indirectly access a target component by accessing the other component which can access the target event-client.

- *Collusion*: More than two malicious components can collude to exploit the functionalities or resources of target component.

Listing 1 and 2 illustrate event eavesdropping on an event-based app `App1` which is corrupted to contain a malicious component `MalClient` (depicted in Listing 2). `App1` follows attribute-based event typing [22] and uses Java Message Service [4], a Java MOM API for message-based communication.

Listing 1 shows where `App1`'s vulnerability resides. In this app, events are published without any particular protection. As a consequence, any component can subscribe and eavesdrop the event by declaring the event's attributes. When `VicClient` of `App1` (depicted in Listing 1) publishes an event which contains a sensitive information (i.e., `private_info`), `MalClient` is able to eavesdrop the event and obtain `private_info` by subscribing to the event.

As shown in this example, since event attacks appear to ordinary event interaction, existing malware inspection tools, especially the popular techniques that rely on signature-based detection [10,36,42], may not be able to detect event attacks. Moreover, since publishing and consuming events can be processed via ambiguous interfaces, existing security flow-analysis techniques will not be able to accurately analyze implicit invocations between components. Furthermore, since routing events is performed in an essentially invisible way, it is difficult to expect where the event attacks are actually launched on.

3. SCUTUM

SCUTUM automatically detects vulnerabilities on the five types of event attacks from EBS. Specifically, *SCUTUM* statically analyzes vulnerable event communication channels by examining control-/data-flow across components as well as event communication patterns in target EBS. *SCUTUM* basically resolves four main challenges:

- **Event channel analysis.** In EBS, implicit invocation and ambiguous interfaces hamper extraction of event communication channels via which events are exchanged between components. Specifically, implicit invocation makes it difficult to determine where each event will flow into, and ambiguous interfaces do not explicitly reveal which event will be consumed. Since each MOM platform provides different types of event interfaces, an extensible analysis for extracting event channels is required.

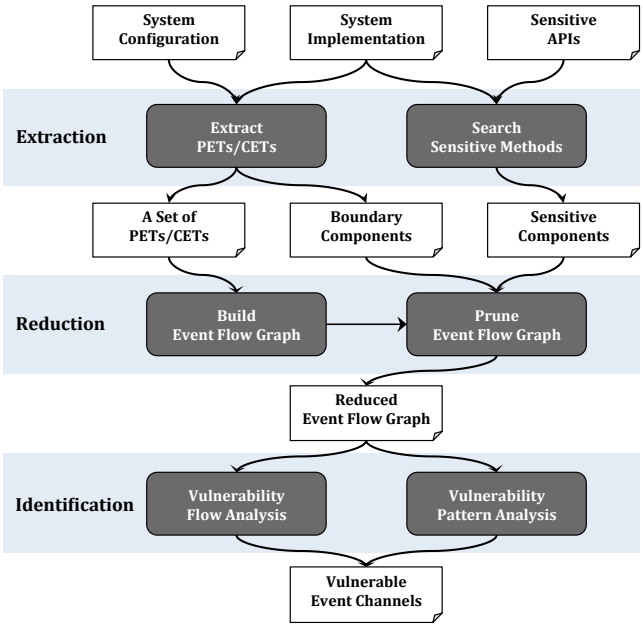

Figure 1: Process for Detection of Security Vulnerabilities in EBS

- **Scalable flow analysis.** To check whether sensitive data leaks or unintended access to sensitive functionality can be launched, control-/data-flow analysis on each component is required. However, for EBSs comprising a large number of components with a number of methods, flow analysis on every component may not be scalable. According to Safi et al. [35], on average, general EBSs contain over 35 methods to be analyzed and analyzing a typical EBS consumes a couple of hours. Although several flow analysis techniques have been proposed for Android apps, considering the fact that mobile platforms limit the size of apps [5], the size and complexity of EBS can be larger than typical mobile apps.

- **Inconstant sensitive APIs.** Analyzing vulnerable flows is dependent on a given set of APIs that may handle private data or sensitive functionality. While prior work has categorized a set of sensitive Android APIs based on the supervised machine-learning approach [33], the set is not equally valid in other EBS domains since each system may use different types of APIs.

- **Inconstant trust boundaries.** Trust boundaries between components are determined by each component's trust level (i.e., components in the same trust boundary have the same trust level), and event attacks are launched across different trust boundaries. While some EBSs use a constant type of trust boundary (i.e., an application in Android), EBSs may have different types of trust boundaries depending on their system configuration.

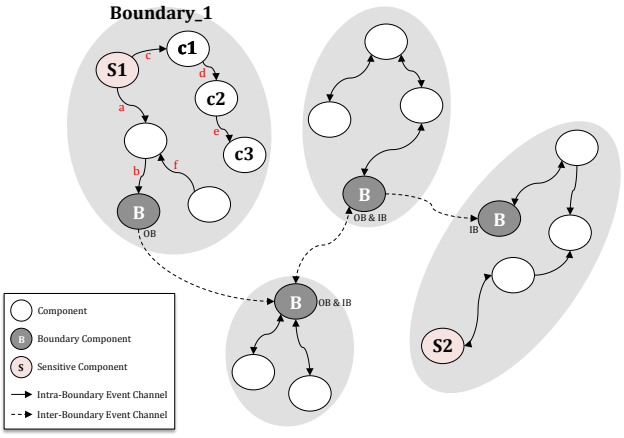

Figure 2: An Event-Flow Graph

The inputs to *SCUTUM* are target EBS's (1) implementation, (2) configuration, and (3) a list of sensitive APIs. The configuration includes the information of underlying MOM platform (i.e., the list of methods for event communication and the base class for events) and trust boundaries. Note that each trust boundary comprises at least one component. As depicted in Figure 1, *SCUTUM*'s analysis is divided into three distinct phases: *Extraction*, *Reduction*, and *Identification*. In the remainder of this section, we discuss each of the three phases in detail.

3.1 Extraction

In this phase, *SCUTUM* inpsects target system's implementation and extracts two types of information that will be used in the later phases: (1) published event types (PET) and consumed event types (CET) accessed by each component, which can be used to infer event communication channels between components. To enable this, *SCUTUM* leverages Eos [22], a technique that statically extracts event types and their attributes based on the characteristics of underlying MOM platform. *SCUTUM* extracts every component's PET and CET along with corresponding attributes from the system implementation. In Listing 1, an example of PET published at line 5 is $e_5 = \{(\text{“}UserInfo\text{”};\text{“}private\_info\text{”})\}$. (2) locations where the sensitive APIs are accessed or called. For each method in a given list of sensitive APIs, *SCUTUM* identifies the component where the method is called along with its location in the system implementation. In case when a list of sensitive APIs are not provided, *SCUTUM* returns the location of every method in each component.

3.2 Reduction

To identify vulnerable event communication channels in EBS, *SCUTUM* considers both inter-component flows (e.g., event channels between components) and intra-component flows (e.g, control or data flows within a component) by combining extracted event types with each component's control-flow graph (CFG). However, for large-scale EBS, generating and traversing every component's CFG will not be scalable. To address this, *SCUTUM* employs an event flow graph (EFG) which provides a macro perspective of target system (see Figure 2), and reduces the number of components that should be analyzed by preprocessing EFG.

In an EFG, components are connected by the edges that represent event communication channels between pairs of components. Each edge has a direction that an event is being sent to. *SCUTUM* connects the edges by matching PET and CET of every pair of components [22, 35]. Once an EFG is constructed, for each component where a sensitive API is called, *SCUTUM* checks if the sensitive API is reachable from or to its event interfaces, i.e., consuming event interface (CEI) and publishing event interface (PEI), via its call graph (CG). If yes, *SCUTUM* labels the corresponding component as a *sensitive component* (see Figure 2). *SCUTUM* then labels the components that form an event communication channel across trust boundaries as *boundary component*s. Specifically, if a component's PEI for an event communication across trust boundaries is reachable from its CEI (or sensitive API) via CG, *SCUTUM* sets its attribute as *outflow-boundary* (*OB*). Reversely, if a component's CEI for an event communication across trust boundaries is reachable to its PEI (or sensitive API) via CG, it is set to be *inflow-boundary* (*IB*). In case when a boundary component contains both types of flows, it can have two attributes (i.e., *OB* and *IB*).

Algorithm 1: *Pruning Event Flow Graph*

Input: $G \Leftarrow$ an EFG
Output: $ReducedG \Leftarrow$ a reduced EFG
1 Let S_G be a set of sensitive components in G
2 Let O_G be a set of outflow-boundary components in G
3 Let I_G be a set of inflow-boundary components in G
4 Let SC be a set of components
5 **foreach** $c \in G$ **do**
6 **if** $(c \in S_G)$ **then**
7 $SC+ = TraverseEventChannels(c, O_G)$
8 **else if** $(c \in I_G)$ **then**
9 $SC+ = TraverseEventChannels(c, S_G)$
10 $ReducedG = G.remove(c \notin SC)$

SCUTUM prunes event communication channels that are not vulnerable to event attacks by implementing Algorithm 1 on EFG. For example, within the trust boundary *Boundary_1* in Figure 2, two different events from component *S1* (i.e., *a* and *c*) may initiate subsequent event communication *b* and *d*-*e*, respectively. Considering the fact that event attacks exploit (1) event communication across trust boundaries and (2) event communication that flows into or from sensitive APIs, event communication channels *c*, *d*, and *e* are not essentially vulnerable to event attacks, because they are not involved in the event communication across trust boundaries. To increase scalability in subsequent flow analyses, *SCUTUM* excludes those event channels (i.e., *c*, *d*, and *e*) and corresponding components (i.e., *c1*, *c2*, and *c3*). Consequently, a pruned graph can reduce the overhead of subsequent flow analyses. For example, in Figure 2, the event communication channels *c*, *d*, and *e* will be pruned since it does not involve any boundary component. This enables avoiding unnecessary intra-component flow analysis for the components involved in those channels. Specifically, considering the complexity of analysis, Algorithm 1 takes $O(N)$ where N is the number of components in EFG, and traversing CFG requires $O(n)$ time where n is the number of methods in CFG [19]. Accordingly, while traversing CFG after pruning takes $O(N) + (N-x)m \times O(n)$ where x is the number of reduced components and m is the average number of methods in each component, traversing CFG without pruning takes $Nm \times O(n)$. Considering the fact that $m \geq 1$ and the average number of component (=16.66) is smaller than the average number of methods (=35) in general EBS [22, 35], if Algorithm 1 prunes at least one component, traversing CFG after pruning always takes less analysis time than that without pruning.

Algorithm 1 iterates over *c* in *G* (lines 5-9) and considers two different cases in which event communication channels are vulnerable to event attacks: (1) The first case is for a set of event communication channels directed from a sensitive component to an outflow-boundary component (lines 6-7). If *c* is a sensitive com-

Algorithm 2: *Identification of Vulnerabilities*

Input: $G \Leftarrow$ an EFG
Output: $VulCh \Leftarrow$ a set of vulnerable event channels
1 Let S_G be a set of sensitive components in G
2 Let O_G be a set of outflow-boundary components in G
3 Let I_G be a set of inflow-boundary components in G
4 Let SM_c be a set of sensitive methods in a component $c \in G$
5 Let E_G be a set of connected event channels between components $x \in S_G$ and $y \in O_G$ or between $x \in S_G$ and $y \in I_G$
6 **foreach** $e \in E_G$ **do**
7 $c_1 \Leftarrow e.head$
8 $c_2 \Leftarrow e.tail$
9 **if** $((c_1 \in S_G) \wedge (c_1 \in O_G))$ *or* $((c_1 \in S_G) \wedge (c_2 \in O_G))$ **then**
10 **foreach** $s \in SM_{c_1}$ **do**
11 **if** $identifyFlow(c_1, s, PEI_{c_1}, \text{“}Out\text{”}, e.remove(c_1))$ **then**
12 $VulCh$ += e
13 **if** $((c_2 \in S_G) \wedge (c_2 \in I_G))$ *or* $((c_2 \in S_G) \wedge (c_1 \in I_G))$ **then**
14 **foreach** $s \in SM_{c_2}$ **do**
15 **if** $identifyFlow(c_2, s, CEI_{c_2}, \text{“}In\text{”}, e.remove(c_2))$ **then**
16 $VulCh$ += e
17 return $VulCh$

ponent, Algorithm 1 calls $TraverseEventChannels(n, A)$ (line 7), a method that identifies all sets of event communication channels (i.e., a set of connected edges) directed from n to each component in A, where n is a component and A is a set of components. Traversing G is performed until no connected edge exists or an edge forms a loop. For example, from the graph in Figure 2, the set $a \Rightarrow b$ will be returned, but the set $c \Rightarrow d \Rightarrow e$ will not. Whenever Algorithm 1 identifies the set of edges from c to any component in O_G, it returns the involved components to *SC*, a set of components. (2) The second case is for a set of event channels from an inflow-boundary component (lines 8-9) to a sensitive component. Algorithm 1 traverses the edges from c to any component in S_G and returns the corresponding identified components to *SC*. Once every component in G is examined, Algorithm 1 removes the components that are not belonging to *SC* and their corresponding edges from G. Finally, the pruned graph *ReducedG* is returned (line 10).

3.3 Identification

In this phase, *SCUTUM* identifies vulnerable event channels by implementing Algorithm 2 on the pruned EFG. The output is a set of vulnerable event channels.

Algorithm 2 iterates over each connected event channel between $x \in S_G$ and $y \in O_G$ or between $x \in S_G$ and $y \in I_G$ (i.e., e in E_G), which directs from sensitive component to boundary component or reverse (lines 6-16). Two different cases are considered depending on the direction of e:

(1) The first case is for e whose direction is from a sensitive component to an outflow-boundary component (lines 9-12). Algorithm 2 checks if c_1 (=the starting component of e) is both sensitive and outflow-boundary component, or if c_1 is a sensitive component and c_2 (=the last component of e) is an outflow-boundary component. If yes, for each sensitive method s in c_1, Algorithm 2 checks if intra-component flows exist between s and event publishing interfaces in c_1 (=PEI_{c_1}) by calling $identifyFlow$.

As depicted in Algorithm 3, *identiFlow* checks if a given component contains an intra-component flow between given two methods. If $flag$ is “Out”, it inspects every node in the control-flow graph of m_1 and m_2, and checks if a node in m_2 is dependent on a node in m_1 (line 4-12). If yes, the corresponding node (=c) is appended to an event flow f. As long as a node exists in a given set of event channels e, Algorithm 3 recursively checks connected intra-component

Algorithm 3: *identifyFlow*

Input: $c \Leftarrow$ a component, $m_1, m_2 \Leftarrow$ a method, $flag \Leftarrow$ a flag, $e \Leftarrow$ a set of event channels
Output: $f \Leftarrow$ an event flow
1 Let $Nodes(m)$ be a set of nodes in the CFG of m
2 Let $ConsumeM(c_1, n_1, c_2)$ be $r \in CEI_{c_2}$ that receives e, where $e \in PET_{c_1}$ is dependent on a node n_1
3 Let $PublishM(c_1, n_1, c_2)$ be $r \in PEI_{c_2}$ that publishes e, where a node n_1 is dependent on $e \in CET_{c_1}$
4 **if** $flag =$ “*Out*” **then**
5 **if** $u \in Nodes(m_2)$ *is directly or transitively control or data dependent on* $v \in Nodes(m_1)$ **then**
6 $f += c$
7 **if** $e \neq \{\}$ **then**
8 $n \Leftarrow e.head$
9 $r = ConsumeM(c, u, n)$
10 $identifyFlow(n, r, PEI_n, \text{“}Out\text{”}, e.remove(n))$
11 **else**
12 return *true*
13 **if** $flag =$ “*In*” **then**
14 **if** $u \in Nodes(m_1)$ *is directly or transitively control or data dependent on* $v \in Nodes(m_2)$ **then**
15 $f += c$
16 **if** $e \neq \{\}$ **then**
17 $n \Leftarrow e.tail$
18 $r = PublishM(c, v, n)$
19 $identifyFlow(n, r, CEI_n, \text{“}In\text{”}, e.remove(n))$
20 **else**
21 return *true*

flows exists between *PEI* and *CEI* of subsequent nodes in e (lines 7-10). For the reverse case when $flag$ is “In”, Algorithm 3 checks the nodes from m_1 to m_2 and recursively identifies connected intra-component flows (lines 13-21) until every node in e is examined.

(2) The second case is for e whose direction is from an inflow-boundary component to a sensitive component (lines 13-16). If c_2 (=the last node of e) is both sensitive and inflow-boundary component, or if c_2 is a sensitive component and c_1 (=the first component of e) is an inflow-boundary component, for each sensitive method s in c_2, Algorithm 2 inspects intra-component flows between s and event consuming interfaces in c_2 (=CEI_{c_2}) by calling $identifyFlow$.

After identifying vulnerable flows, *SCUTUM* performs pattern analysis on event communication channels in EFG based on previously identified compositional patterns [18]. *SCUTUM* considers four different patterns as follows: (c: a component, T: a trust boundary, $\Rightarrow$: an event communication channel from left to right)

1. Given components $c_1, c_2, \in T_1, c_3 \in T_2$ and $c_1 \neq c_2 \neq c_3$, if $(c_3 \Rightarrow c_2) \wedge (c_1 \Rightarrow c_2)$, $c_3 \Rightarrow c_2$ can be *spoofing*.

2. Given components $c_1, c_2, \in T_1, c_3 \in T_2$, and $c_1 \neq c_2 \neq c_3$, if $(c_1 \Rightarrow c_3) \wedge (c_1 \Rightarrow c_2)$, $c_1 \Rightarrow c_3$ can be *interception* or *eavesdropping*.

3. Given components $c_1 \in T_1, c_2, c_3 \in T_2$ and $c_1 \neq c_2 \neq c_3$, if $(c_1 \Rightarrow c_2) \wedge (c_2 \Rightarrow c_3) \wedge \neg(c_1 \Rightarrow c_3)$, $c_1 \Rightarrow c_2 \Rightarrow c_3$ can be *confused deputy*.

4. Given components $c_1, c_2, \in T_1, c_3 \in T_2$ and $c_1 \neq c_2 \neq c_3$, if $(c_1 \Rightarrow c_2) \wedge (c_2 \Rightarrow c_3) \wedge \neg(c_1 \Rightarrow c_3)$, $c_1 \Rightarrow c_2 \Rightarrow c_3$ can be *collusion*.

The patterns are based on the assumption that an event communication channel within boundary is intended access, but an event communication channel across boundary may be unintended access

from a malicious component. If a set of event communication channels matches any of those patterns, *SCUTUM* labels the channels as vulnerable. Finally, *SCUTUM* returns vulnerable event communication channels by combining the vulnerable flows and channels.

4. EVALUATION

We have empirically evaluated *SCUTUM* in terms of its accuracy, applicability, and performance in detecting vulnerabilities from EBS.

4.1 Accuracy

We evaluated *SCUTUM*'s accuracy in identifying vulnerabilities by comparing its results against those of *Xanitizer* [9] and *Owasp Orizon* [7], state-of-the-art tools for detecting security vulnerabilities in Java apps.

4.1.1 Experimental Setup

Since existing test benchmarks for Java apps [6, 16, 23] are not targeting EBSs as well as event attacks, we have created a test benchmark for evaluating effectiveness of security analysis tools for EBSs. Our benchmark mainly targets two representative types of MOM platforms, (1) Java Message Service (JMS) [4], the widely adopted Java-oriented middleware for exchanging messages between components, and (2) Prism-MW [1], an extensible middleware platform that enables efficient implementation, deployment, and execution of distributed software systems. The benchmark comprises 20 distinct event-based apps (10 for each MOM platform) each of which implements a single type of event attack. In this benchmark, all five types of event attacks (recall Section 2) are implemented, and each app had the sole purpose of performing an event attack that exploits a target component. To ensure that every event attack can be actually launched, two graduate students at USC manually inspected the code of each app. By running each app on a PC, we also confirmed that the attacks are successfully launched at runtime. The benchmark also comprises 5 "trick" apps containing vulnerable but unreachable components, whose identification would be a false warning. This yielded a total of 25 event-based apps containing 20 vulnerable event communication channels. The benchmark is distributed in `https://github.com/namdy0429/SCUTUM`.

4.1.2 Evaluation of SCUTUM

We evaluated *SCUTUM* for accuracy in identifying vulnerabilities as compared to *Xanitizer* and *Owasp Orizon*. *Xanitizer* statically detects security vulnerabilities such as injections and privacy leaks in Java apps by using taint-flow analysis. *Owasp Orizon* also statically analyzes insecure code patterns from Java apps by using pattern matching [8].

We ran the three tools on our test benchmark and measured their (1) *precision*, i.e., identified vulnerabilities that were actually vulnerable to event attacks, and (2) *recall*, i.e., the ratio of identified to all vulnerabilities. *SCUTUM* detected vulnerabilities, specifically vulnerable event communication channels, with 100% precision and recall and correctly ignored all "trick" cases with unreachable vulnerable components. However, both *Xanitizer* and *Owasp Orizon* were unable to find any of the vulnerabilities on event attacks from the benchmark. Specifically, *Xanitizer* did not return any result. While *Owasp Orizon* reports some security warnings such as *"potential dangerous keyword in the method"*, those are not directly related to vulnerabilities on event attacks. This is primarily because those tools are not targeting event attacks and not supporting inter-component flow analysis.

4.2 Applicability

Table 1: Subject Systems and Results

Name	Type	Event Mechanism	SLOC	No. of Identified Vulnerabilities
Dradel	Software IDE [30]	c2.fw [29]	11K	12
ERS	Crisis Response [27]	Prism-MW [28]	7K	11
KLAX	Arcade Game [39]	c2.fw [29], Java events	5K	2

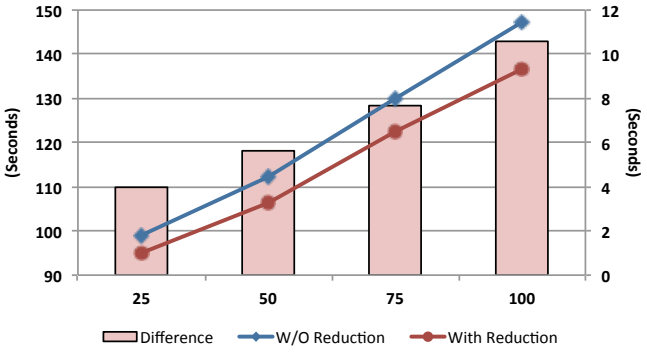

Figure 3: *SCUTUM*'s Performance on Different Number of Components

To assess if *SCUTUM* is applicable to real-world EBSs, from the test suite which have been used in prior research [22], we selected three subject EBSs that are implemented using MOM platforms. All subject systems are implemented in Java, but are from different application domains, of different sizes, and use different underlying mechanisms for consuming events as shown in Table 1.

Since the list of sensitive methods for the subject systems was not provided, we ran *SCUTUM* with the setting that every 'setter' or 'getter' method is a sensitive method and each component has different trust boundary. According to the well-known sensitive method list for Android [33], 81% of sensitive methods are setters or getters (getters: 97%, setters: 65%), which implies that getters and setters are more likely to be sensitive to security attacks compared to other methods. However, it is important to note that it does not necessarily mean that all getters and setters are always sensitive methods.

For each subject system, *SCUTUM* flagged potential vulnerabilities (specifically, vulnerable event communication channels) on event attacks as depicted in Table 1 (*Dradel*: 12, *ERS*: 11, *KLAX*: 2). We manually inspected each system's code to check if the identified event channels are indeed vulnerable to event attacks. On average, its precision was 85.67% (*Dradel*: 75%, *ERS*: 82%, *KLAX*: 100%). Every false positive was caused by its inaccuracy in identifying control-/data-flow between sensitive methods and event interfaces. *Xanitizer* reported 83 security warnings, such as *"may expose internal representation by returning reference to mutable object"*, and *"IO Stream Resource Leaks"* (*Dradel*: 6, *ERS*: 62, *KLAX*: 15), and 7 of them were related to event attacks. *Owasp Orizon* returned 13 implementation bugs such as *"empty catch detected"* and *"found potential dangerous keyword"* (*Dradel*: 9, *ERS*: 1, *KLAX*: 3), however, none of them was related to the vulnerabilities on event attacks. Although *SCUTUM* outperformed *Xanitizer* and *Owasp Orizon* in our evaluation, it is important to note that *Xanitizer* and *Owasp Orizon* detected additional vulnerabilities that *SCUTUM* did not.

4.3 Performance

To evaluate the performance of *SCUTUM*, we tested it on the event-based apps comprising different numbers of components. We created four different apps by adding different number of components (i.e., 25, 50, 75, and 100, respectively) to an app which was

randomly selected from our benchmark. The added components are designed to have at least two event communication channels with other components and contain one method. None of the added components are involved in vulnerable event communication channels so that they can be pruned in *Reduction* phase. We ran each app on a PC with an Intel dual-core i5 2.7GHz CPU and 4GB RAM.

On average, *SCUTUM*'s analysis took 115.04*s* (*Extraction* phase: 69.02*s*, *Reduction* and *Identification* phase: 46.02*s*). To evaluate the effectiveness of pruning EFG, we also measured *SCUTUM*'s analysis time for each app 'without' *Reduction* phase. As shown in Figure 3, as the number of components increased, the difference of analysis time between 'with' and 'without' *Reduction* phase also increased. Considering the fact that the added components are designed to have a simplified structure (i.e., two event channels and one method), the difference can dramatically increase for real-world EBS containing the components with higher complexity.

5. REFERENCES

[1] Prism-MW - Architectural Middleware for Mobile and Embedded Systems. http://sunset.usc.edu/~softarch/Prism/, 2001. [Online; accessed July 18, 2017].

[2] Gartner says worldwide pc, tablet and mobile phone combined shipments to reach 2.4 billion units in 2013. http://www.gartner.com/newsroom/id/2408515, 2013. [Online; accessed April 15, 2015].

[3] Android Open Source Project. https://source.android.com, 2015. [Online; accessed August 16, 2016].

[4] Java Message Service (JMS). http://www.oracle.com/technetwork/java/jms/index.html, 2016. [Online; accessed September 16, 2016].

[5] APK Expansion Files | Android Developers. https://developer.android.com/google/play/expansion-files.html, 2017. [Online; accessed August 15, 2017].

[6] Owasp Benchmark. https://www.owasp.org/index.php/Benchmark, 2017.

[7] Owasp Orizon. https://www.owasp.org/index.php/Category:OWASP_Orizon_Project, 2017.

[8] Owasp Top10 2013. https://www.owasp.org/index.php/Top_10_2013, 2017.

[9] Xanitizer. https://www.rigs-it.net/index.php/product.html, 2017.

[10] F. Anjum, D. Subhadrabandhu, and S. Sarkar. Signature based intrusion detection for wireless ad-hoc networks: A comparative study of various routing protocols. In *Vehicular Technology Conference (VTC)*, volume 3, pages 2152–2156. IEEE, 2003.

[11] J. Bacon, K. Moody, and W. Yao. A Model of OASIS Role-based Access Control and Its Support for Active Security. *ACM Trans. Inf. Syst. Secur.*, 5(4):492–540, Nov. 2002.

[12] H. Bagheri, A. Sadeghi, J. Garcia, and S. Malek. COVERT: Compositional Analysis of Android Inter-App Permission Leakage. *IEEE Transactions on Software Engineering*, 41(9):866–886, 2015.

[13] H. Bagheri, A. Sadeghi, R. Jabbarvand, and S. Malek. Automated Dynamic Enforcement of Synthesized Security Policies in Android. Technical Report GMU-CS-TR-2015-5, George Mason University, 2015.

[14] A. Belokosztolszki, D. M. Eyers, P. R. Pietzuch, J. Bacon, and K. Moody. Role-Based Access Control for Publish/Subscribe Middleware Architectures. In *Proceedings of the 2nd International Workshop on Distributed Event-Based Systems (DEBS)*, pages 1–8, 2003.

[15] F. Biscotti et al. Market Share: AIM and Portal Software, Worldwide, 2009. *Gartner Market Research Report*, 2010.

[16] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA)*.

[17] S. Bugiel, L. Davi, R. Dmitrienko, and T. Fischer. Towards Taming Privilege-Escalation Attacks on Android. In *NDSS*, 2012.

[18] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing Inter-Application Communication in Android. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2011.

[19] K. D. Cooper, T. J. Harvey, and K. Kennedy. A simple, fast dominance algorithm. *Software Practice & Experience*, 4(1-10):1–8, 2001.

[20] L. Fiege, M. Mezini, G. Mühl, and A. P. Buchmann. Engineering Event-Based Systems with Scopes. In *Proceedings of the 16th European Conference on Object-Oriented Programming (ECOOP)*, pages 309–333, 2002.

[21] A. P. Fuchs, A. Chaudhuri, and J. S. Foster. SCanDroid: Automated Security Certification of Android Applications. Technical report, Dept. of Computer Science, University of Maryland, College Park, 2009.

[22] J. Garcia, D. Popescu, G. Safi, W. G. J. Halfond, and N. Medvidovic. Identifying Message Flow in Distributed Event-Based Systems. In *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, pages 367–377, 2013.

[23] S. Heckman and L. Williams. On Establishing a Benchmark for Evaluating Static Analysis Alert Prioritization and Classification Techniques. In *Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 41–50, 2008.

[24] W. Klieber, L. Flynn, A. Bhosale, L. Jia, and L. Bauer. Android Taint Flow Analysis for App Sets. In *Proceedings of the 3rd International Workshop on the State of the Art in Java Program Analysis (SOAP)*, 2014.

[25] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. Mcdaniel. In *Proceedings of the 37th International Conference on Software Engineering (ICSE)*, pages 280–291.

[26] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang. Chex: Statically Vetting Android Apps for Component Hijacking Vulnerabilities. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2012.

[27] S. Malek, N. Medvidovic, and M. Mikic-Rakic. An extensible framework for improving a distributed software system's deployment architecture. *IEEE Transactions on Software Engineering*, 38(1):73–100, 2012.

[28] S. Malek, M. Mikic-Rakic, and N. Medvidovic. A style-aware architectural middleware for resource-constrained, distributed systems. *IEEE Transactions on Software Engineering*, 31(3):256–272, 2005.

[29] N. Medvidovic, E. M. Dashofy, and R. N. Taylor. The role of middleware in architecture-based software development. *International Journal of Software Engineering and Knowledge Engineering*, 13(04):367–393, 2003.

[30] N. Medvidovic, D. S. Rosenblum, and R. N. Taylor. A language and environment for architecture-based software development and evolution. In *Software Engineering, 1999. Proceedings of the 1999 International Conference on*, pages 44–53. IEEE, 1999.

[31] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel. Semantically Rich Application-Centric Security in Android. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2009.

[32] L. I. W. Pesonen, D. M. Eyers, and J. Bacon. Encryption-Enforced Access Control in Dynamic Multi-Domain Publish/Subscribe Networks. In *Proceedings of the Inaugural International Conference on Distributed Event-based Systems (DEBS)*, pages 104–115, 2007.

[33] S. Rasthofer, S. Arzt, and E. Bodden. A Machine-Learning Approach for Classifying and Categorizing Android Sources and Sinks. In *NDSS*, 2014.

[34] S. Rasthofer, S. Arzt, E. Lovat, and E. Bodden. DroidForce: Enforcing Complex, Data-centric, System-wide Policies in Android. In *Proceedings of the 9th International Conference on Availability, Reliability, and Security (ARES)*, 2014.

[35] G. Safi, A. Shahbazian, W. G. Halfond, and N. Medvidovic. Detecting Event Anomalies in Event-Based Systems. In *Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 25–37, 2015.

[36] M. K. Sahu, M. Ahirwar, and A. Hemlata. A review of malware detection based on pattern matching technique. *Int. J. of Computer Science and Information Technologies (IJCSIT)*, 5(1):944–947, 2014.

[37] B. Shand, P. Pietzuch, I. Papagiannis, K. Moody, M. Migliavacca, D. Eyers, and J. Bacon. Security policy and information sharing in distributed event-based systems. *Reasoning in Event-Based Distributed Systems*, pages 151–172, 2011.

[38] M. Srivatsa, L. Liu, and A. Iyengar. EventGuard: A System Architecture for Securing Publish-Subscribe Networks. *ACM Transactions on Computer Systems (TOCS)*, 29(4):10:1–10:40, 2011.

[39] R. N. Taylor, N. Medvidovic, K. M. Anderson, E. J. Whitehead, J. E. Robbins, K. A. Nies, P. Oreizy, and D. L. Dubrow. A component-and message-based architectural style for gui software. *IEEE Transactions on Software Engineering*, 22(6):390–406, 1996.

[40] S. J. Templeton and K. E. Levitt. Detecting Spoofed Packets. In *DARPA Information Survivability Conference and Exposition Proceedings*, volume 1, pages 164–175, 2003.

[41] X. Wang, K. Sun, Y. Wang, and J. Jing. DeepDroid: Dynamically Enforcing Enterprise Policy on Android Devices. In *NDSS*, 2015.

[42] H. Wu, S. Schwab, and R. L. Peckham. Signature based network intrusion detection system and method, Sept. 2008. US Patent 7,424,744.

[43] R. Xu, H. Saïdi, and R. Anderson. Aurasium: Practical Policy Enforcement for Android Applications. In *Proceedings of the 21st USENIX Conference on Security Symposium (USENIX)*, 2012.