Enhancing Distributed Systems with Mechanisms to Cope with Malicious Clients

Technical Report CNDS-2005-4 - December 2005
http://www.dsn.jhu.edu

Yair Amir, Claudiu Danilov, John Lane
Johns Hopkins University
{yairamir, claudiu, johnlane}@cs.jhu.edu

Michal Miskin-Amir
Spread Concepts LLC
michal@spreadconcepts.com

Cristina Nita-Rotaru
Purdue University
crisn@cs.purdue.edu

***Abstract*— In this paper we identify a major security vulnerability in distributed systems: compromised clients under adversarial control can use the system within their authorized access rights and authenticated channels to deliberately insert incorrect data. A significant problem is that when a malicious client insider is discovered, it is hard to quickly assess the scope of the damage, and identify corrupt and suspected updates.**

We propose Accountability Graph, a mechanism that can assist applications in coping and recovering from such attacks. The tool provides accountability enforcement and causality tracking of updates and their dependencies. Upon detection of incorrect data (e.g. by an external intrusion detection mechanism or human assessment), the Accountability Graph will quickly classify all updates in the system as either corrupted, suspected or not affected. The practicality and usefulness of the approach is demonstrated based on the requirements of three different applications: an open source software development project, a military common operation picture application, and a national emergency response system. The Accountability Graph can also be used for risk assessment and vulnerability analysis with respect to the above attack.

I. Introduction

Many distributed services are implemented following a model where a set of servers replicate the service and coordinate their actions to answer client requests while maintaining the consistency of the data. The most basic operations performed by clients are querying the servers or updating data maintained by the servers. Security is a major concern for such systems that often operate over unsecure networks such as the Internet. Significant work conducted in the last several years to develop mechanisms for Byzantine replication [1], [2], [3], access control [4], [5], [6] and intrusion detection [7], [8], [9] provides the support for designing secure distributed services. Specifically, the servers and their operating system are protected against intrusions, corrupted servers are tolerated by running Byzantine replication algorithms, access to resources is tightly enforced by using access control mechanisms, while client actions are monitored by intrusion detection systems.

Although such systems may seem difficult to attack, they overlook that the weakest link is represented by the clients (often communicating with the servers over wireless channels) and the most critical asset is the data itself. Thus, very harmful attacks can come from compromised clients, targeting the data correctness: One or more compromised clients can *use* the system within their authorized parameters to create or inject incorrect inputs or updates to some servers. The (Byzantine) replication algorithms will propagate this information among all servers, corrupting the state of the system so that it will no longer reflect reality. Several observations are important. First, the Byzantine replication protocols running on the servers will replicate data already compromised, so they will not be able to address the attack. Second, these incorrect updates may not be detected immediately, impacting other clients subsequently querying the system and basing their decisions on the erroneous state. This creates a cascading effect in which further created updates are also erroneous because they are based on malicious data. Third, although intrusion detection mechanisms deployed in the system may eventually detect the compromised clients, assessing the extent of the damage and identifying the other components of the system that were affected, or are suspect and need further investigation is very challenging and is not provided by the mechanism mentioned above.

The effect of such an attack can be devastating for applications that are highly dependent on the correctness of their data. For example, in collaborative open-source software development (e.g. Linux), multiple individuals create or augment existing source code. The inherent interdependency between software packages enables a malicious update to one package to significantly impact other components of the system. It is important to identify the packages that may be affected by corrupt code injected into the system, and determine the risk and vulnerabilities associated with it.

Other examples are command and control information systems, such as those used by the military [10] or by emergency response personnel [11]. In such systems, users update the state of the operational situation and make decisions based on it. Correctness of the data is critical, and any misleading information can result in loss of life. A malicious insider can inject authorized yet incorrect information that may mislead honest users and cause them, in turn, to make additional erroneous updates.

Our Focus and Contribution: A major problem with secure distributed systems is that when a malicious client

insider is discovered, it is hard to quickly assess the scope of the damage, and identify corrupt and suspected updates. Therefore, the system is not able to regenerate and recover to a clean state without the effects of these updates. Based on our experience building secure reliable systems, we make the observation that in the best case, this is considered an application-specific issue, and the system infrastructure provides no support in addressing it. Most of the time, this problem is not considered at all. The goal of this work is to raise awareness to this important problem and to show how the distributed infrastructure can assist the application in recovering from such attacks. Our preliminary results based on the requirements of three different applications demonstrate the practicality of our solution.

We propose Accountability Graph, a generic mechanism that provides accountability enforcement and causality tracking of updates and their dependencies in a directed acyclic graph with periodic snapshots. Upon detection of incorrect data, the system traces the data to the corrupt update that generated it, and from that, the Accountability Graph enables us to mark all causally dependent updates as corrupted or suspected. We mark all subsequent updates made by the malicious client that generated the corrupted update as corrupt, and use a standard graph traversal to identify and mark as suspicious all other updates that recursively depend on corrupted updates. No less important, the system is assured that all unmarked updates are not affected by the discovered incorrect data. Our proposed solution can use any intrusion detection mechanism (or human input) that will provide the initial detection. One or several servers forming the underlying distributed service can decide to maintain the graph, the coordination between the servers, including the ordering of the updates, will ensure that the graph looks the same at each server. There is no central authority or point of failure, any server can decide at any time if it will build the graph for events happening in the system.

The contributions of the paper are:

- We identify a significant attack against distributed services mounted by malicious clients that deliberately insert incorrect data through authorized channels.
- We propose a generic mechanism, Accountability Graph, that tracks the dependencies between all of the updates in the system. When notified about compromised clients or corrupt updates by external mechanisms (such as intrusion detection, human assessment, application-specific knowledge), the Accountability Graph can classify data as corrupt, suspect, or not affected.
- We demonstrate the usefulness of our solution in three different applications: an open-source software development project, a military common operation picture application, and a national emergency response system. We show that the overhead associated with our solution is reasonable in these cases.
- We present an additional benefit of the Accountability Graph, namely the ability to conduct risk assessment and vulnerability analysis with respect to the compromised client attack.

The rest of the paper is organized as follows. We present a description of the model considered in this paper in Section II. Section III presents a detailed description of the Accountability Graph. We demonstrate the usefulness and feasibility of our approach for several applications in Section IV. In Section V we present the performance of the Accountability Graph, and in Section VI we survey related work. We conclude the paper in Section VII.

II. System Model

We assume a general message-passing system where one or more servers respond to requests from clients. The requests submitted by clients can be updates (write operations), or queries (read operations). Communication is asynchronous.

We assume that each client has a public and private key pair. Servers know the public keys of all clients that connect to them. Cryptographic techniques such as public-key digital signatures, message authentication codes, and message digests produced by collision-resistant hash functions, are used to provide non-repudiation, message integrity and authentication of messages. We assume that the adversary is computationally bounded such that he cannot subvert these cryptographic techniques.

Clients communicate with the servers using secure channels: the communication is protected from an external adversary by using encryption, all messages are authenticated and carry integrity information which prevents an external adversary from injecting or modifying packets.

The adversary can compromise any number of clients, coordinate the attack, delay communication, modify, delete or replay a message, or simply generate and deliberately send incorrect data. The adversary cannot delay indefinitely correct clients. When a client node is compromised, the adversary has full control over that node, including access to all cryptographic keys stored on the machine.

We assume that there are external mechanisms that can detect that clients were compromised or that they submitted updates containing incorrect data. This can be done by employing tools such as intrusion detection systems, or by having a human review the data offline. The intrusion detection is not instantaneous, i.e. clients can inject several malicious updates before they are detected. Correct clients may use affected data in their decisions (and therefore create incorrect updates themselves) before a malicious update is detected. Thus, the damage caused by a malicious update can affect future updates (not necessarily made by the malicious client), as well as queries that will propagate incorrect information to honest clients.

III. Accountability Graph: Design, Implementation and Expressiveness

Based on the observations formulated in Section I, we believe that there is a need for mechanisms that provide the following:

- Corrupted data isolation: when notified that an update is incorrect, the system can identify the data affected by it,

and provide fast feedback about all other compromised updates.
- Automatic regeneration of non-corrupted states: allow automatic regeneration of a view of the data that includes non-contaminated data, based on initial information about problems that occurred and building knowledge about the extent of the problem.

We address these requirements by building an Accountability Graph, a directed acyclic graph that maintains causal dependency of updates, allowing data classification in the light of a compromised client or incorrect update, and facilitating automatic regeneration of a correct state. We note that the Accountability Graph provides a useful tool to help assess risk assuming that specific participants were compromised at known times, or that specific updates were incorrect. Below we describe our design, with focus on the construction and traversal of the causality graph.

A. Design Overview

To track client updates, we construct the causality graph as follows. Every update in the system is uniquely identified, includes the ID of its client creator, and is signed by that client. Every update also contains the identifier and digital signature [12], [13] of every previous update directly responsible for data on which the new update depends. In addition, we assume a standard causal relationship where every update depends on the previous update created by the same client. The Accountability Graph is maintained such that every update is a node in the graph and there is a directed link from that update to all the updates on which it depends. The dependency information is usually specific to the application. In some cases, dependencies may be introduced by clients themselves. In other cases, dependencies occur based on the flow of the application, while in the most conservative case we may consider that an update introduced by a client depends on all the updates previously reported to that client. In this work we do not make any assumption about the nature of dependencies.

When data is detected as incorrect, it is traced to the corresponding update. Then, by traversing the graph, corrupted and suspected updates are marked. For example, in Figure 1(a), an update of client *C4* is found to be incorrect, and the generating client, *C4*, is presumed malicious. Subsequent updates from that client are marked as corrupt and all the updates that depend on them are marked as suspicious, as shown in Figure 1(b). Notice that the arrows indicate dependency relations (i.e. if update A depends on update B, there is an arrow from A to B). Therefore, the edges in the graph are traversed in the opposite direction of the arrows. The Accountability Graph can now present various views of the system state based on these markings, and the system can regenerate its state as needed based on the updates that are deemed valid. The system can consider only the clean updates, or it can consider both clean and suspected updates.

To limit the memory required for storing the causality graph and the processing required for state regeneration, the system can use periodic posteriori snapshots. Every epoch, for example 12 hours, a snapshot of the system state as of 12 hours ago is calculated and stored. This limits the processing required for state regeneration when bad data is discovered, as calculations are performed from the last valid snapshot (usually the last one). Of course, the length of the actual epoch depends on the rate of the updates in the system.

The pseudocode for the operations used to create and traverse the graph is presented in Algorithm 1. The *SubmitUpdate* function creates a node containing the client's identity and a unique sequence number. It also places the node in an associative container so that it can be found using its identifier. The *AddDependencies* function adds directed edges from all of the nodes in a specified list to the node specified by its identifier. Finally, the function *GetSuspectedUpdates* performs a standard graph traversal, beginning at the specified corrupt node, and marks the other nodes as corrupt, suspect, or not affected.

B. Optimization of the Accountability Graph

In the algorithm described above, when a client submits an update, the Accountability Graph automatically creates an edge connecting the update being added to the previous update submitted by that client. In Figures 1(a) and 1(b), these automatically generated edges correspond to the vertical arrows. Note that FIFO edges are always present in standard network-level causality graphs [14]; they are a conservative approximation to true, application-level, causality. These edges are also vital for our Accountability Graph. When a corrupt update is discovered, we assume that the client that generated this update is malicious and that all subsequent updates submitted by the malicious client are also corrupt. A single traversal beginning at the corrupt update can mark every corrupt and suspected update precisely because of these vertical lines. The FIFO edges link all of the updates that we assume are corrupt. Note that the malicious client cannot alter or remove these edges because it is not responsible for generating them.

The algorithm we have specified is simple and computationally efficient, but it suffers from an important problem. Consider what might happen if an honest client's update does not depend on its previous update. For example, in a military Common Operations Picture, a status update, U_s, made by a tank about its fuel, ammunition, and position is actually independent of the last update submitted by this tank. Suppose that a prior update submitted by the tank was dependent on an update made by a malicious client. Then, U_s would be marked as suspect during a graph traversal. In addition, anything with recursive dependencies on U_s would be labeled as suspicious. The result is a high false positive rate.

Before presenting our solution to this problem, we make one important observation: If an update of an honest client is compromised, future updates introduced by that client can be trusted unless they depend on corrupted data themselves. Our modified algorithm automatically adds FIFO edges as described above. However, these edges do not necessarily need to be used. By default, we disable the FIFO edges in the directed acyclic graph so that the graph traversal will not

Algorithm 1 Accountability Graph Operations

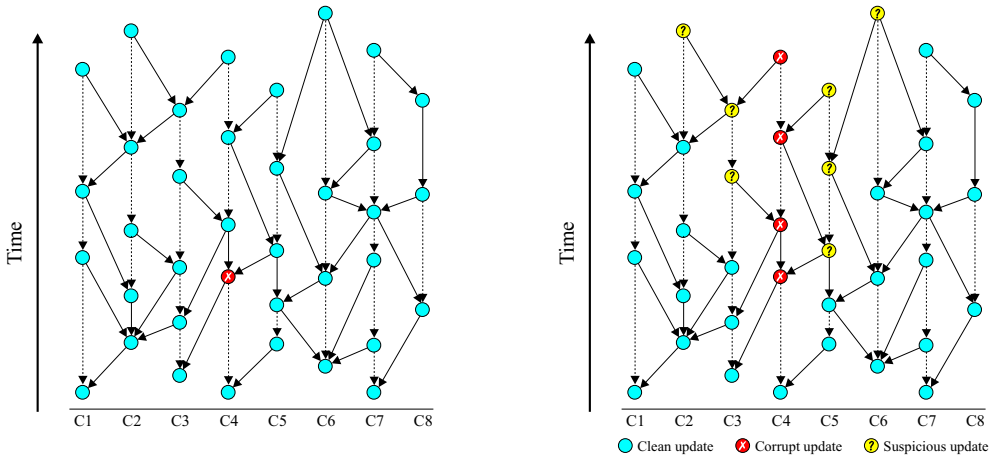
```
Each node (update) maintains:
Node
    Int id // A unique id for this node
    Int client_id // A unique client id
    Node dependents[] // A list of those nodes that are dependent on this node
    Int classification // CORRUPT, SUSPECT, or NOT_AFFECTED

Global Variables:
    Int next_id = 0 //this is used to create the id for the Node
    Int suspects[] //a list to store the ids of suspect nodes
    A Map of all nodes, indexed by the node's id

Int SubmitUpdate( Int client_id )
    next_id = next_id + 1
    let n = new Node with id next_id
    add n to the A-DAG //store the node in a container where it can be accessed via its id
    add the last update of this client to n.dependents
    return next_id

AddDependencies( Int dependent, Int []dependencies )
    let n = node having id of dependent
    for each i in dependencies
        let n_d = node having id of i
        add n to n_d.dependents //this creates a directed edge from n_d to d

Int[] GetSuspectedUpdates( Int corrupt_node )
    clear suspects //remove all suspect updates
    set the classification to NOT_AFFECTED for all nodes
    let next = 0
    let n = Node with id equal to corrupt_node
    add n to suspects
    set n.classification = CORRUPT
    while suspects.size() > next
        let p = Node with id equal to suspects[next]
        next = next + 1
        for each c in p.dependents
            if c.id is not in suspects
                add c.id to suspects
                if c.client_id == n.client_id, set c.classification = CORRUPT
                else, set c.classification = SUSPECT
    return suspects //return the suspects
```



(a) (b)

Fig. 1. Accountability Graph Example

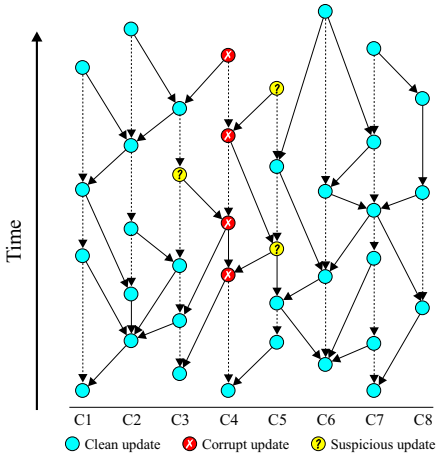


Fig. 2. Removing FIFO Edges

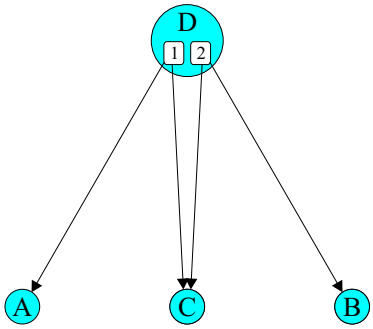


Fig. 3. OR Dependency

use them. When a corrupt update is found, we first enable only those FIFO edges that connect updates made by the malicious client (i.e. the client that generated the corrupt update). Then, a traversal is done as previously described. Figure 2 shows the same scenario as 1(b) with this optimization; now, fewer updates are labeled as suspect. Note that the improved Accountability Graph can express when a client submits an update that really does depend on its previous update. As in the original algorithm, malicious clients are unable to alter these edges because they do not generate them. We believe that this change is important not only because it reduces false positives, but also because it draws attention to the differences between conservative, network-level causality graphs and pruned, application-level causality graphs. By working at the application-level and allowing clients and/or other dependency sources to specify dependency relations, we improve the accuracy and utility of the Accountability Graph.

C. Enriching the expressiveness of the Accountability Graph

The Accountability Graph construction algorithm described above has limited expressiveness; the integrity of an update depends on the integrity of all of its dependencies. If C depends on the set $\{A, B\}$, then C is suspect if A or B is suspect. This means that the integrity of C depends on A AND B. Expressing dependencies with only AND operators does not always adequately capture a dependency relation. For example, consider an application where two motion sensors, S_a and S_b, cover the same area. Suppose that both sensors make an update stating that there is motion in this area. An administrator receives these two updates and makes an update, D, that dispatches a security guard to the area. The integrity of D depends on the integrity of S_a OR S_b. The administrator is conservative and therefore would have dispatched the guard even if only one of the sensors reported motion. The original algorithm cannot express this dependency.

We increase the expressive power and, thereby, the accuracy of our dependency graph by introducing an OR operator. The

basic idea is shown in Figure 3, which depicts that D depends on ($A\ OR\ B$) $AND\ C$. Note that node D contains two numbered cells and that dependency arrows originate from these cells. During a graph traversal, both numbered cells in D must be visited before D is marked as suspect. If only A is suspect, then only cell 1 will be visited. Similarly, if only B is suspect, then only cell 2 will be visited. However, if C is suspect, then both cell 1 and 2 will be visited. In general, we can modify our algorithm so that it can express all dependency relations written using linear combinations of OR and AND operations.

The OR operator enables the Accountability Graph to express dependencies on redundant sources of information. In the above example, the motion sensors provide the same information. If a client bases an update on both sensors, the update remains unaffected even if one of the sensors was corrupt. This is important because fault tolerance is commonly improved by using redundancy. Therefore, the OR operator increases the usefulness of the Accountability Graph as an offline analysis tool. When the Accountability Graph is used to assess system vulnerabilities, the OR operator can be used to show the benefits of strategically placed redundancy.

IV. Case Studies

To demonstrate the feasibility of the Accountability Graph for real applications, we consider three applications for which we believe our mechanisms can offer great benefits. These applications are drawn from: open-source software projects, network-centric warfare applications, and information access to national emergency systems.

A. Collaborative Open-Source Software Projects

Many applications today rely on open source software projects, such as Debian [15], Red Hat [16], Apache [17], and Gnome [18]. Such projects are collaborative, distributed over several machines, and involve many participants. For example, the Debian project has over 1000 registered developers managing about 10000 software packages supporting 12 different platforms. If a critical machine is compromised, all packages that passed through it during creation are suspect. If a package is compromised, any other packages that use it are compromised. If a client is untrustworthy, all packages that client was involved in are suspect. Unfortunately such incidents are a reality: in 2001 [19], the public server used by the Apache Software Foundation to provide the source code repository, binary distribution, web services, and public mailing lists was compromised; in 2001 a developer introduced a Trojan horse in one of the Debian packages, while more recently in 2003 Debian was again in the news [20] when four servers were compromised, one of them hosting security updates; Gnome also was the target of an attack in 2004 [21].

We chose to use Red Hat software packages (RPMs) as a representative open source software project because RPMs are widely used and because the Red Hat Package Manager contains tools for extracting dependency information. Each RPM package contains a set of capabilities such as programs, libraries and data, and may require other capabilities to already be installed on the system. The fifteen Red Hat distributions using RPMs contain 52,984 software capabilities, spanning 7 years of development. The distributions we considered, the number of RPMs, and the number of software capabilities in each distribution are presented in Table I.

Version	Number of RPMs	Number of Capabilities
RedHat 4.2	458	632
RedHat 5.0	482	693
RedHat 5.1	523	789
RedHat 5.2	573	891
RedHat 6.0	645	1523
RedHat 6.1	718	1691
RedHat 6.2	743	2049
RedHat 7.0	865	2113
RedHat 7.1	1016	2918
RedHat 7.2	1231	3862
RedHat 7.3	1438	5715
RedHat 8.0	1472	6432
RedHat 9	1402	7128
RedHat Fedora 1	1466	7754
RedHat Fedora 2	1619	8804
TOTAL	14651	52984

TABLE I
RED HAT DISTRIBUTIONS

Each package provides one or more software capabilities such as programs, libraries, or data. Some capabilities have many thousands of directly or recursively dependent capabilities resulting in very complex dependency relationships. If one of the capabilities is corrupt, it can potentially affect all of the capabilities that depend on it. It is very difficult and time consuming to manually analyze such a system and determine the set of capabilities that may be affected by a corrupt capability. The Accountability Graph automates this task and thus can be extremely useful when corrupt software is discovered and distribution administrators want to promptly determine the extent of the possible damage.

The integrity of a software capability in some specific distribution depends on the integrity of the same software capability in all older distributions. From one distribution to the next, capabilities evolve slowly and source code added to one version is generally present in many subsequent versions. Suppose that a malicious programmer added a vulnerability to the source code of the encryption capability libcrypt.so.1 in RedHat 5.0. Clearly, anyone using a capability that directly or recursively depended on this version of libcrypt.so.1 could have been affected. It is also possible that the malicious code has propagated to subsequent versions of libcrypt.so.1 in RedHat 5.1 through Fedora 2. Therefore, when a compromise of libcrypt.so.1 in RedHat 5.0 is found, it is important to obtain a list of all software packages that depend on this version or on any subsequent version. Note that because libcrypt.so.1 is a shared library, fixing all versions of libcrypt.so.1 will produce a properly functioning system assuming that static linking was not used. However, we are also concerned with finding any capability that may have been vulnerable during the time

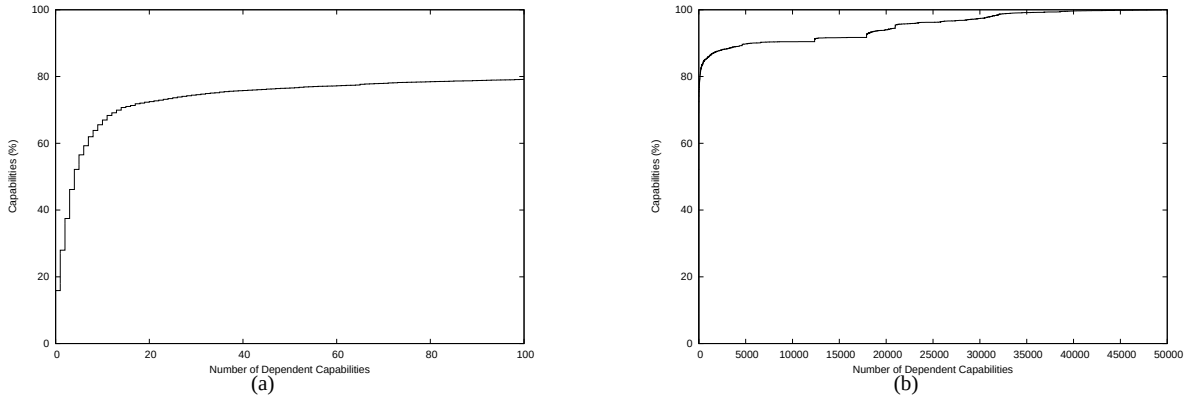


Fig. 4. CDFs of the percentage of capabilities (on the Y axis) having less than the specified number of dependent capabilities (on the X axis).

that the encryption library was corrupt. In this example, all capabilities that depend on libcrypt.so.1 may have stored data that is insecure and therefore should be examined.

Below, we analyze the dependency graph generated based on the RPM packages. Each RPM provides a set of software capabilities. Commonly, the provided set contains only one capability. Other RPMs provide several capabilities comprised of applications, shared libraries, and other software related files. An RPM may also require a set of capabilities. The capabilities that are provided depend on the capabilities that are required. Using these sets, we created a mapping from each capability to its dependencies. More precisely, in the Accountability Graph, software capabilities represent nodes, edges are drawn between the provided capabilities and those that they require, and, even though not specified in the RPM system, developers who create software packages are considered clients. For each distribution, we constructed a list of tuples having the form: (C_n, dependencies of C_n). Then, we added all of these tuples to one graph. Within each distribution, there are many dependencies and some of these reflect recursive relationships. A distribution sometimes included more than one version of a capability. When this occurred, we determined a causal order based on version numbering and made a dependency chain so that the newest version recursively depended on all prior versions. We linked the oldest version of a capability in Distribution n to the newest version in Distribution n-1.

After constructing the dependency graph, we ran a traversal beginning at each software capability and retrieved a list of all dependent capabilities. Figures 4(a) and 4(b) show cumulative distribution function (CDF) plots of the percentage of capabilities (on the Y axis) having less than the specified number of dependent capabilities (on the X axis). The figures differ only in the range shown on the y axis. We see that about 65% of capabilities have less than 20 other capabilities that depend on them. However, 20% have over 100 dependent capabilities, and furthermore, about 10% of the capabilities have over 15000 dependent capabilities. Capabilities having the greatest number of dependents include: *"libc.so.5"* (48288 dependents), *"filesystem = 1.3-1"* (48901 dependents), *"setup = 1.7-2"* (48915 dependents) all from RedHat 4.2, and *"/sbin/ldconfig"* (49441 dependents) from RedHat 5.0.

The RPM dependency graph described above shows that, generally, a corrupt capability affects a relatively small number of other capabilities. However, some capabilities can affect many others. Therefore, the Accountability Graph can be used to identify the capabilities that would cause the most damage if they were corrupted; these capabilities represent system-wide vulnerabilities. Also note that the complex recursive dependency relations and sometimes large number of dependents make it very difficult to manually identify suspect capabilities when a corrupt capability is discovered. We believe that this case study illustrates the usefulness of the Accountability Graph both as an offline analysis tool and as an online damage assessment tool.

B. A Military Common Operation Picture Application

The military Common Operation Picture (COP) application provides a current view of the battle space shared by all friendly forces, and enables planning and coordination of the forces. The information provided by the COP may include the location of friendly and enemy units, current level of available supplies and ammunition in each unit, location of natural and man made obstacles, currently executed tactical plans and future possible plans for the different units, etc. Authentication and access control strictly determines who is allowed to view or update different parts of the operational picture. Participants constantly monitor the COP, modify their plans, and issue commands as the situation progresses.

Such applications depend heavily on the fact that information provided to the system is correct. The following scenario illustrates this problem: An update, coming from a compromised intelligence officer computer, that updates the location of an enemy unit to be 3 km south of where it actually is. This update will be accepted by the system because the intelligence officer is authenticated and is authorized to make it. A logistics officer that needs to re-supply a friendly unit, notices the location of the enemy unit according to the COP, and plots a path that will avoid the enemy. This path is also updated into the COP. The unit that is being supplied selects a

location to meet with the logistics convoy based on the updated path. In parallel, a friendly commando unit plans to attack the enemy unit. Once it is discovered that the enemy unit location is incorrect, the Accountability Graph can quickly mark the plans of the logistics convoy, the supplied unit and the commando unit. These plans were dependent directly or indirectly on the incorrect update and will have to be re-evaluated.

In this application, the clients are all the participants authorized to update any part of the common operation picture state. The nodes of the Accountability Graph are the updates to the state, and the edges of the graph refer to the past updates that influenced the decision to make the dependent update.

The Common Operation Picture application is not large compared with current hardware capabilities, and allows storing all the updates throughout the duration of a military engagement (a few months time). To explore the feasibility of the Accountability Graph to support this kind of application with adequate performance, we need to estimate the size of the graph. If we consider tracking about 5000 units, each causing the generation of about one update per minute, then over 12 hours this scenario generates about 3,600,000 updates. A snapshot of the COP state, taken every 12 hours, limits the required calculation for traversing the Accountability Graph to this number of nodes. Our experiments provided in Section V indicate that the Accountability Graph can provide an answer in matter of seconds for a dependency graph of this size.

C. Information Access for National Emergency Response Systems

The Clinicians' Biodefense Network (CBN) [22], [23] is a nationwide Internet-based information exchange system designed specifically for use by US-based clinicians in the aftermath of a bioterrorist attack. The network is managed and operated by the Center for Biosecurity of the University of Pittsburgh Medical Center. CBN was designed to facilitate communication and timely exchange of accurate and precise information among clinicians in the event of bioterrorism, and provide practitioners around the country with clinically oriented information quickly enough to guide decision-making

The design of the CBN envisioned communication between the network editorial staff, network contributors, and many thousands of network subscribers. The network data contributors are highly trusted clinicians and prominent experts who provide critical clinical information to the network, and comment on information provided by other data contributors. Several hundred clinical experts and leaders were expected to participate as data providers. The number of information updates during an emergency situation could reach several thousand per day.

The Clinicians' Biodefense Network architecture was designed to employ state of the art security mechanisms. Clinical leaders, experts, and network administrators need specific credentials in order to provide information to the network.

The network must provide accurate, correct and timely information. The potential exists for malicious users who impersonate a network content contributor, e.g. by stealing a password or finding other ways to infiltrate the network, to provide misleading data that could be dangerous and potentially life-threatening. For example, during a Biodefense attack, an update can incorrectly report identified cases in the wrong locations, in order to create confusion and hamper the response to legitimate cases. Further updates by other data providers can be based on this misleading data, and must also be identified and reevaluated once the malicious update is detected.

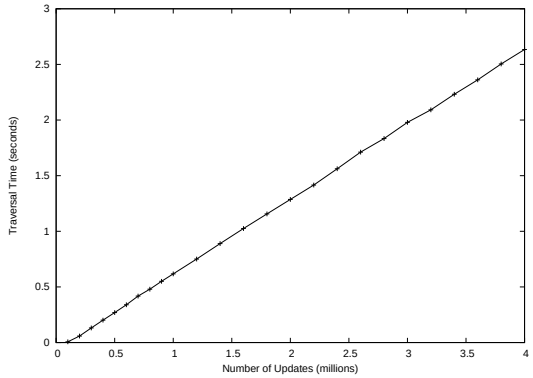


Fig. 5. Traversal time as a function of the number of updates.

In this application, the clients are the clinical leaders and experts who provide data to the network. The nodes of the accountability graph are the messages sent on the network. The edges of the graph refer to the past messages that this message is in response to.

The type of scenarios described in the Common Operation Picture operation exist here, although the size of the CBN state and overall number of nodes in the graph is considerably smaller.

V. Performance of the Accountability Graph

The Accountability Graph keeps track of all updates introduced into the system and their dependency on other updates. It may appear that this mechanism introduces an unacceptably high overhead due to storing and processing the updates. The goal of this section is to analyze the resulted overhead and to show that for many applications it does not affect the performance significantly.

Intuitively, there are several factors that can affect the time it takes to build or traverse the graph: the number of updates (nodes) in the graph, the number of clients, the depth of the graph, and the number of dependencies (edges) in the graph. We implemented a simple data structure in C++ using the STL library and conducted several experiments to evaluate its performance as a function of these factors.

Experiment Set-up: In the experiments presented below we make no assumption about the end application, and consider a random graph with the following structure: The Accountability Graph is constructed based on time-slices in which updates are committed. Each update has a corresponding node in the graph. The nodes are organized in a two

dimensional matrix having a column for each client and a row for each time slice. The graph was built and traversed on a Intel Pentium IV 2.8GHz computer with 1GB RAM.

To initialize the graph, all clients submit one update at time t_0. These updates are not dependent on any other update. At every subsequent time-slice, all clients submit one more update. Thus, at t_1 and after, a client's update depends on a constant number of nodes in prior time-slices. This defines the number of dependencies of that particular update.

In the experiments below we constrain the number of rows on which an update can depend. This is because in practice, only a snapshot of the dependents will be preserved. Note that the number of rows depended upon is an upper bound. For example, if this number is 20, an update submitted in time-slice t_{100} can depend on any update submitted between t_{80} and t_{99}, inclusive. Dependencies are selected randomly within the defined range of time-slices with uniform probability.

Number of updates: Figure 5 shows traversal time as a function of the number of updates in the graph. The graph traversal starts at the first time-slice, on the first update of the client having an identifier equaling $N/2$, where N is the number of clients. For this experiment we fixed the number of clients to 20, the number of dependencies to 5 and the number of prior time-slices in which nodes can have dependencies to 20. We observe that the traversal time grows linearly with the number of updates, and that for about 4 million updates, the traversal time takes less than 3 seconds. We believe that several seconds response time for marking potentially affected updates is reasonably fast to serve current applications. For example, a distributed replicated system that can handle around 80 updates per second, with a snapshot taken every 12 hours will accumulate about 3.5 million updates between snapshots. Building a 4 million dependency graph as described above took less than 20 seconds, and the data structure occupied about 225 MB of memory.

Number of clients: Figure 6 shows the traversal time and the number of nodes traversed, as a function of the number of clients submitting updates, in a scenario where the number of nodes is 4 million, and there are 5 dependencies per node. New updates can depend on the updates in previous 100 time-slices. The number of nodes visited decreases as the number of clients increases. Note that because the number of updates is fixed, the number of time-slices decreases as the number of clients increases. One node is visited in time slice t_0. Approximately 5 nodes in t_1 are dependent on this node. In t_2, approximately 5^2 nodes are dependent, recursively on the original node in t_0. This trend continues (at a decreasing rate because of overlapping dependents) until all nodes in a time-slice are marked as suspect. As the number of clients increases, it takes a larger number of time-slices before all clients in each subsequent time-slice are marked as suspect, and therefore, fewer nodes are traversed as the number of clients increases.

The traversal time is initially large because of CPU cache misses, since dependent nodes are spread across a range of memory proportional to the number of clients. It then increases slightly as the number of clients increases to $10,000$ due to an increase in cache misses. Then it decreases because the number of nodes traversed decreases.

Number of dependencies: Figure 7 shows the traversal time and the number of nodes traversed as a function of the number of dependencies, when the number of nodes, clients, and dependency time-slices are fixed. We consider a graph with 4 million nodes, $100,000$ clients, and 20 dependency time-slices. It can be noted that at 1, 2, 3 and 4 dependencies, the traversal time and number of nodes traversed are small. The number of nodes traversed increases rapidly thereafter. The traversal time is approximately linear with the number of dependencies.

Summary: Our experiments show that factors that affect the performance of the Accountability Graph are number of compromised clients, number of dependencies, and the number of updates. For all of the scenarios we considered, and without performing any application-specific optimizations, the traversal time was less than 8 seconds. The number of dependencies and the number of dependencies per update seemed to be the most influential factors in increasing the traversal time.

VI. Related Work

In this section we summarize related work in several research areas in distributed systems that relate to the problem we present in this paper. We note that our work is complementary to the work presented below.

a) Directed Acyclic Graphs in Operating and Distributed Systems: The core mechanism of our tool is building a causality graph. Directed acyclic graphs (DAGs) were previously used in operating systems and distributed systems. For example, the Time Warp Operating System [24] maintains two "wave fronts" of computation. The front wave front represents speculative computation that is hazarded by rollback. The rear wave front bounds the roll back. Computations on the rear wave front line depend only on prior computations that are committed. Computations between the two wave fronts are tracked using causal dependency tracking (a dependency DAG), and can be canceled.

In the distributed systems field, the Trans protocol [14] and the Transis system [25] also use a DAG in order to ensure reliable delivery of multicast messages using non-reliable multicast. This DAG is maintained on the fly and updated accordingly as messages are delivered by all of the members of the group. Strom and Yemini [26] replace synchronization by causal dependency tracking in order to overcome benign process failures in distributed systems.

b) Intrusion Detection Systems: The security community has developed many mechanisms that can detect malicious users after they have penetrated a computer system. The large body of intrusion detection literature [9] testifies to our assertion that malicious intruders will sometimes gain access to even the best secured systems. We provide a way for to assess and cope with these penetrations, while intrusion detection systems provide ways to detect them. When a malicious, yet authorized, intruder is detected, systems typically alert an

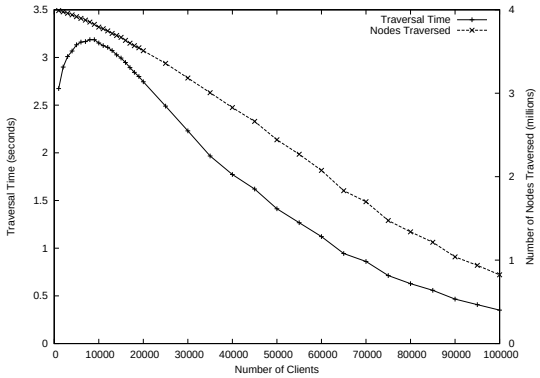


Fig. 6. Traversal time as a function of the number of clients.

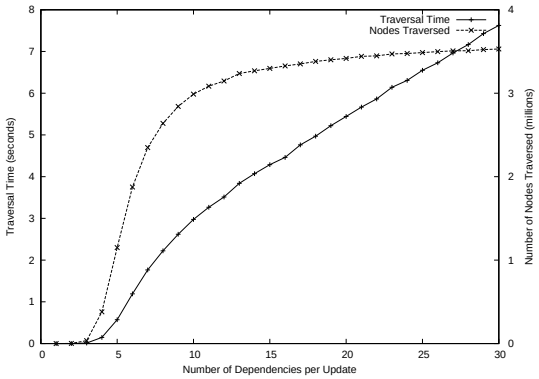


Fig. 7. Traversal time as a function of the number dependencies

administrator and may trigger an automated damage mitigation response. Traditionally, intrusion detection researchers have taken two main approaches: misuse detection and anomaly detection. Misuse detection focuses on identifying user behavior that matches a specific attack signature [7]. Anomaly detection [8] focuses on finding user behavior that deviates from normal system use. We want to emphasize that intrusion detection strategies are more than merely complementary to the Accountability Graph. Intrusion detection forms an important component of our solution since a malicious user must first be detected before the damage created by that user can be mitigated.

The BackTracker Tool by King and Chen [27] was designed to help system administrators analyze intrusions to their operating system. Working backward from a detection point such as a suspicious file or process, BackTracker identifies the events and objects that could have affected that detection point and displays chains of events in a dependency graph. The system administrator can focus the detective work on those chains of events in order to understand how the intruder gained access to the system. Our work, in comparison, assumes that the malicious clients (insiders) are authorized to access the system and simply use the application to create or inject incorrect inputs or updates. We are concerned with quickly identifying what part of the current application state is corrupt, suspected, and (no less important) not affected. The use of dependency graphs in BackTracker, together with our past directed acyclic graph work in Transis [25] to efficiently track causality, inspired our Accountability Graph approach to cope with malicious clients.

c) Byzantine-Resilient Replication: The first practical work to solve replication while withstanding Byzantine failures is the work of Castro and Liskov [1]. Their algorithm requires a number of $3f + 1$ servers in order to tolerate f faults and asks the client to wait for $f + 1$ identical answers out of $2f + 1$ answers in order to make sure that it received a correct answer. The work is fundamentally based on Byzantine Consensus, for which a good overview can be found in [28]. The Byzantine replication provides protection against malicious servers, and does not address the malicious clients problem.

VII. Conclusions

In this paper we identified a significant attack against distributed systems, mounted by malicious clients that deliberately insert incorrect data into the system using authorized channels. We proposed a generic mechanism, Accountability Graph, that tracks the dependencies between all of the updates in the system, and that can classify data as corrupt, suspect, or not affected, giving the ability to conduct risk assessment and vulnerability analysis with respect to the compromised client attack. We demonstrated the usefulness of our solution in three different applications, and we showed that the overhead associated with our solution is reasonable in these cases.

References

[1] M. Castro and B. Liskov, "Practical byzantine fault tolerance and proactive recovery," *ACM Trans. Comput. Syst.*, vol. 20, no. 4, pp. 398–461, 2002.

[2] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin, "Separating agreement from execution for byzantine fault-tolerant services," in *SOSP*, 2003.

[3] D. Malkhi and M. Reiter, "Byzantine quorum systems," *Journal of Distributed Computing*, vol. 11, no. 4, pp. 203–213, 1998.

[4] S. Osborn, "Database security integration using role-based access control," in *IFIP WG11.3 Working Conference on Database Security*, August 2000.

[5] S. De, C. Eastman, , and C. Farkas, "Secure access control in a multi-user database," in *ESRI User Conference*, 2002.

[6] S. Rizvi, A. Mendelzon, S. Sudarshan, and P. Roy, "Extending query rewriting techniques for fi ne-grained access control," in *SIGMOD 2004*, 2004.

[7] "Using decision trees to improve signature-based intrusion detection," in *RAID 2003*, 2003.

[8] Y. Xie, H.-A. Kim, D. R. O'Hallaron, M. K. Reiter, and H. Zhang, "Seurat: A pointillist approach to anomaly detection.," in *RAID*, pp. 238–257, 2004.

[9] E. Jonsson, A. Valdes, and M. Almgren, eds., *Recent Advances in Intrusion Detection: 7th International Symposium, RAID 2004, Sophia Antipolis, France, September 15-17, 2004. Proceedings*, vol. 3224 of *Lecture Notes in Computer Science*, Springer, 2004.

[10] C. Wilson, "Network Centric Warfare: Background and Oversight Issues for Congress; CRS Report for Congress," June 2004.

[11] R. Wible, "Update report on alliance for building regulatory reform in the digital age," June 2004. http://www.astronaut3.com/ncsbcs/content/html/update_report.htm.

[12] *Digital Signature Standard (DSS)*. No. FIPS 186-2, National Institute for Standards and Technology (NIST), 2000. http://csrc.nist.gov/publications/fi ps/fi ps186-2/fi ps186-2.pdf.

[13] R. L. Rivest, A. Shamir, and L. M. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, vol. 21, pp. 120–126, Feb. 1978.

[14] P. M. Melliar-Smith, L. E. Moser, and V. Agrawala, "Broadcast protocols for distributed systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, pp. 17–25, January 1990.
[15] "The debian linux distribution." http://www.debian.org/.
[16] "The redhat linux distribution." http://www.redhat.com/.
[17] "Apache software foundation," 1999-2004. http://www.apache.org/.
[18] "Gnome foundation," 2003. http://www.gnome.org/.
[19] "Apache software foundation server compromised, resecured," May 2001. http://seclists.org/lists/bugtraq/2001/May/0350.html.
[20] "Some debian project machines compromised," November 2003. http://www.debian.org/News/2003/20031121.
[21] C. Franklin, "Gnome servers attacked putting the penguin on guard," April 2004. http://www.networkcomputing.com/showitem.jhtml?articleID=18900826.
[22] "Clinicians' biodefense network," 2001. http://www.upmc-cbn.org.
[23] L. Randovich, "Hopkins Center to Launch Clinicians' Biodefense Network," *Biodefense Quarterly, A publication of the Johns Hopkins Center for Civilian Biodefense Network*, vol. 4, no. 2, 2002.
[24] D. Jefferson, B. Beckman, F. Wieland, L. Blume, and M. Diloreto, "Time warp operating system," in *Proceedings of the eleventh ACM Symposium on Operating systems principles*, pp. 77–93, ACM Press, 1987.
[25] Y. Amir, D. Dolev, S. Kramer, and D. Malki, "Transis: A communication sub-system for high availability," in *In Proceedings of the 22nd Annual International Symposium on Fault Tolerant Computing*, pp. 76–84, July 1992.
[26] R. Strom and S. Yemini, "Optimistic recovery in distributed systems," *ACM Transactions on Computer Systems (TOCS)*, vol. 3, August 1985.
[27] S. T. King and P. M. Chen, "Backtracking Intrusions," in *Symposium on Operationg System Principles (SOSP)*, pp. 223–236, October 2003.
[28] M. J. Fischer, "The consensus problem in unreliable distributed systems (a brief survey)," in *Fundamentals of Computation Theory*, pp. 127–140, 1983.