Effective and Efficient Compromise Recovery for Weakly Consistent Replication

Prince Mahajan [*] Ramakrishna Kotla Catherine C. Marshall
Venugopalan Ramasubramanian Thomas L. Rodeheffer Douglas B. Terry Ted Wobber
Microsoft Research, Silicon Valley

Abstract

Weakly consistent replication of data has become increasingly important both for loosely-coupled collections of personal devices and for large-scale infrastructure services. Unfortunately, automatic replication mechanisms are agnostic about the quality of the data they replicate. Inappropriate updates, whether malicious or simply the result of misuse, propagate automatically and quickly. The consequences may not be noticed until days later, when the corrupted data has been fully replicated, thereby deleting or overwriting all traces of the valid data. In this sort of situation, it can be hard or impossible to restore an entire distributed system to a clean state without losing data and disrupting users.

Polygraph is a software layer that extends the functionality of weakly consistent replication systems to support compromise recovery. Its goal is to undo the direct and indirect effects of updates due to a source known after the fact to have been compromised. In restoring a clean replicated state, Polygraph expunges all data due to a compromise or derived from such data, retains as much uncompromised data as possible, and revives valid versions of subsequently compromised data. Our evaluation demonstrates that Polygraph is both effective, retaining uncompromised data, and efficient, re-replicating data only when necessary.

Categories and Subject Descriptors C.2.4 [*Computer-Communication Networks*]: Distributed Systems; D.4.5 [*Operating Systems*]: Reliability

General Terms Algorithms, design, reliability, security

Keywords Compromise recovery, replication

[*] Prince Mahajan is currently affiliated with the Department of Computer Sciences at the University of Texas at Austin.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
EuroSys'09, April 1–3, 2009, Nuremberg, Germany.
Copyright © 2009 ACM 978-1-60558-482-9/09/04. . . $5.00

1. Introduction

Techniques for recovering data from compromises are critically important but noticeably lacking in systems where data is replicated across a growing ecosystem of intermittently connected personal electronic devices (for example, laptops, desktops, cell phones, cameras, PDAs, and music players). Systems employing weakly consistent replication are popular as a means of sharing pictures, calendar entries, work related documents, and other data across multiple devices and meeting the growing expectation of ubiquitous access to this shared data. Weakly consistent replication is also useful for implementing infrastructure services of geographic scale, where connectivity and system size make strong consistency impractical [Vogels 2008]. In either case, compromises resulting from stolen devices [Dvorak 2006], malicious intrusion [CNet 2007], device failures or even accidental deletions [Santry 1999] are problematic.

Updates originating at a compromised device can have a wide variety of effects ranging from simple data loss to widespread data corruption as corrupted data items propagate to other devices. Corruption may compound as unsuspecting users generate new updates based on the corrupted data. To make matters worse, compromises may go undetected for some time [Baker 2006], long enough for all traces of the correct data to be deleted or overwritten. When a lost or stolen device falls into the wrong hands, user-level misbehavior is difficult to prevent; hence, a system must be able to recover data after a compromise is detected.

The goal of recovery is to undo the damage done by a compromised device by removing all the corrupted data items and restoring as much uncorrupted data as possible. The recovery process should impose low network and processing overhead to accommodate the resource and battery constraints of mobile devices. The disruption caused by a compromise should be proportional to the amount of corrupted data injected by the compromised device. For example, a lost cell phone that is found by someone who adds a few phone numbers should not be as disruptive as the theft of a laptop and subsequent erasure of a company's complete customer database.

The problem of compromise recovery in weakly consistent replicated systems is largely unaddressed by previous work. Existing systems typically rely on local [Apple 2007, NetApp 2008] or cloud-based backup [Carbonite 2007, Decho 2007, Symantec 2007]. Backup-based approaches to data recovery suffer from two main limitations. First, these approaches require the system to revert to a precompromise snapshot of its data; uncorrupted data items not backed up before a compromise are lost even if they have not been influenced by the compromised device. Second, recovery proceeds by discarding the local data at replicas and restoring their complete state from a backup server, incurring high network overhead as data is transferred to every device.

Polygraph addresses the challenge of recovering data from compromise by using three fundamental ideas. First, as in more traditional backup-based approaches, Polygraph uses one or more archive sites to record versions of data items that may need to be restored after a compromise is reported. Second, it determines innocent versions that could not possibly have been affected by a compromised replica through the use of a *taint vector* that tracks the devices that influence each version and a *precompromise cut* that summarizes the data known to the archive before the compromise. Third, Polygraph implements replica-local retention by using an *innocence predicate* that distinguishes corrupted data from uncorrupted data at each replica thereby allowing devices to retain uncorrupted data locally.

Together these key mechanisms ensure *effective* recovery by losing as few uncorrupted data items as possible. Through its techniques for reliably detecting innocent versions, Polygraph recovers most data that was not updated or influenced by the compromised replica before its compromise. Moreover, in Polygraph, devices recover as much data as possible from their local store instead of retrieving these data items from the archive server. This replica local-retention approach ensures *efficiency* by minimizing recovery overhead (network bandwidth and processing time).

We implemented Polygraph by extending Cimbiosys, a state-based data replication system [Ramasubramanian 2008]. Evaluation of this implementation over a wide range of input conditions demonstrated that Polygraph is both effective and efficient. For instance, it retained 98% of uncorrupted items for a scenario with moderate propagation delay while a backup-based method retained only 35%. Moreover, in the same scenario, Polygraph consumed ten times less network bandwidth than the backup-based method for restoring uncorrupted device state.

This paper makes the following main contributions:

- We introduce a new recovery mechanism for weakly consistent systems that attains effectiveness and efficiency through influence tracking and replica-local retention.
- We demonstrate our recovery mechanism's practicality by implementing it in an existing state-based data replication system and evaluating its recovery properties.

The rest of this paper proceeds as follows. Section 2 presents the system overview with assumptions, goals, and the key ideas to achieve these goals. The recovery protocol is presented in Section 3 and implementation details in Section 4. We evaluate our system in Section 5. Section 6 extends our model to allow untrusted archives. Section 7 discusses our solution and some potential extensions. Section 8 discusses related work and Section 9 concludes.

2. Overview

Polygraph[1] is a software layer that extends the functionality of weakly consistent replication systems to support compromise recovery. The systems that we target replicate a collection of *items* over a number of devices.

2.1 System model

A *replica* (of a collection) is a copy of some or all of the collection's items on a single device. Each device can store one or more replicas. To simplify the discussion, we assume that there is one replica per device and use the terms device and replica interchangeably.

Devices support applications that can read and update items in the collection. Each update produces a new *version* of an item; item creation and deletion are specialized forms of updates. A new version of an item *supersedes* older versions from which it was derived. Since applications at each replica create new versions independently, the state of the collection as seen at different replicas is only *weakly consistent*. Replicas exchange updates as connectivity permits using pairwise synchronization or some other replication protocol. We use the term *synchronization* to refer to a single, pairwise exchange of items between replicas. We do not rule out other forms of optimistic replication [Saito 2005].

Each item is identified by a unique *itemID* and each replica by a unique *replicaID*. Each version is identified by a *versionID* which is a tuple ⟨*replicaID, versionNumber*⟩; replicaID identifies the replica at which the version was authored. Each replica maintains a monotonically increasing logical time that is used to assign version numbers. Replication protocols in general employ such constructs for replica and version identification, but if no such abstractions are available, the Polygraph implementation can supply them.

Polygraph is designed to minimize architectural changes to the application layer above and the replication protocol below. There are three direct manifestations of this decision. First, we do not mandate any specific granularity for items that can be separately stored or recovered. Instead, how data is mapped to items is determined by applications. For example, if an application manages spreadsheets, it might choose to encode an entire spreadsheet as one item, or it might en-

[1] Polygraph [Wikipedia 2008] is a lie detection technique based on physiological changes caused by the sympathetic nervous system during questioning of a subject. Similary, our Polygraph system allows replicas to recover innocent data by separating them from tainted data.

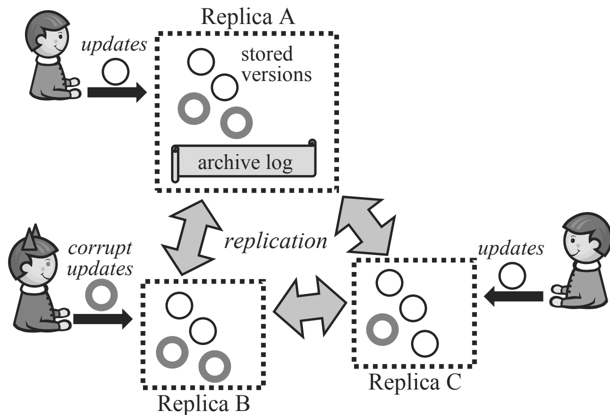

Figure 1. Example Polygraph scenario.

code each row or cell as an item. There are obvious atomicity tradeoffs involved with such a choice. Second, as in all replication systems that offer weak consistency guarantees, concurrent updates from different replicas can conflict. We assume that it is the responsibility of the replication protocol to resolve such conflicts, possibly with application help. Finally, we assume by default that an updated item derives from the previous version present at the updating replica. The application can optionally model different behaviors. For example the contents of an update might be derived from two different parent items or from conflicting versions of the same item.

In Polygraph, at least one replica must serve as an *archive* that records all versions that it sees. Archives provide the information needed to detect innocent versions and restore old versions after a compromise has been detected. Because some replication protocols only disseminate the most recent versions of items, a version that is superseded before fully propagating might never reach an archive.

Figure 1 shows a simple scenario with three replicas, *A*, *B*, and *C*, that we will use for expository purposes. The actual compromise sitations that Polygraph addresses can involve many more devices and stored items. Replica *A* acts as the archive and contains an archive log. Replica *B* has been compromised and has produced corrupted versions that have propagated to the other replicas.

We assume that compromises are detected by an external agent, either a human or an automated system, and that the agent can estimate when the compromise occured. For example, a person may know that his phone was stolen yesterday or an anti-virus scanner may detect corruption that occured subsequent to its previous scan. Our system focuses on mechanisms to recover data once a compromise has been discovered and reported.

2.2 Threat model

Polygraph is designed to recover from malicious user action or simple misuse. Our model is that such events inject *corrupt* updates into the system and thus imply replica compromise. However, we assume that users of a compromised replica cannot tamper with the Polygraph implementation, the underlying replication software, or the operating system. Corrupt updates that are replicated to other devices may serve as the basis for further updates that are thus also corrupted. Eventually, the most recent version of any item in the collection may be corrupt. Versions that are not corrupt are *innocent*.

Although our threat model does not anticipate an attacker who can install a new device operating system or corrupt the existing one, it does address important real-world threats in which the attacker has control of a device but lacks the resources, skill, or intent to subvert its operating system. The integrity of the Polygraph layer can be enforced according to the situation. In some scenarios, the device user interface is sufficiently robust to guarantee system integrity, thus only allowing the user to introduce corrupt data or command sequences. This sort of approach is often used to protect special-purpose computing devices with limited scope for software modification such as cell phones and gaming consoles. Alternatively, like most runtime security enforcement mechanisms, the Polygraph implementation can be protected from corruption arising from buggy or malicious applications. Existing research suggests numerous approaches for providing such protection [England 2003, Saltzer 1974, Wobber 1994].

The replication software or device operating system might well implement its own security framework, and such a framework can prevent further malicious usage once compromise is discovered. For example, smartphone software stacks often implement a “kill-switch” that can disable a device under the control of a service provider or a corporate IT department. Our work is aimed at recovering the non-corrupted state of a replicated system with a particular focus on recovering legitimate state changes that take place between compromise and compromise discovery.

Threats to a replica’s integrity can stem from misuse rather than malice. In addition to addressing malicious threats, Polygraph’s approach is well-suited to address the data corruption introduced by misuse. For example, consider the following scenario:

> *Paul leaves his mobile phone on the coffee table while he’s getting ready for work. His 4 year-old son, anxious to play games he’s seen on phones, starts pressing buttons on his dad’s phone, and in the course of interacting with it, garbles several address book entries. He never finds the games, and senses he’s done something he wasn’t supposed to. So he replaces the phone on the coffee table, and neglects to tell his father about his misadventure. Paul uses the phone all day, adding more contact information, pausing only briefly to wonder why the keypad seems sticky. That evening, Paul synchronizes the phone with his lap-*

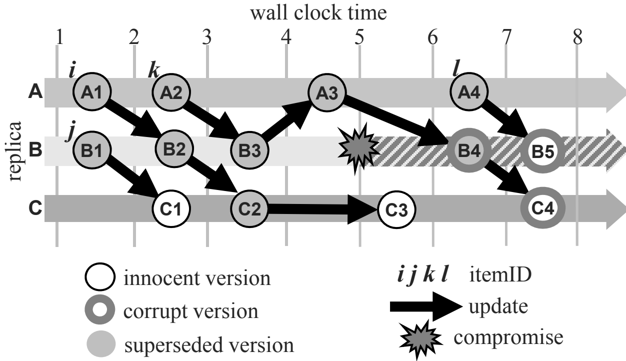

Figure 2. Example update timeline at replicas *A*, *B*, *C*.

top which in turn synchronizes automatically with his work computer, and those of his workplace associates.

This simple scenario illustrates how corrupt updates can spread and remain unnoticed until they have been widely replicated. Such threats are also conceivable in a setting in which data is shared by a large group (tens to hundreds) of people and devices. For example, Microsoft Live Mesh [Microsoft 2008b] replicates data to allow seamless data sharing among many users in a work group or a social network, each with the ability to access and update data from multiple devices. Similarly, Microsoft Active Directory [Microsoft 2008a] uses weakly consistent replication to provide a naming and resource management infrastructure for global-scale networks. In all these settings, the ability to perform timely compromise recovery is vital.

2.3 An example timeline

Figure 2 shows an event timeline for the system of Figure 1. This example timeline will help guide our description of Polygraph in this section and the next. It shows an update history for four items *i*, *j*, *k*, and *l*. This timeline gives a global view of the system, one that may not be perceived by any replica since updated versions propagate asynchronously during replication. (Update propagation is not depicted in the timeline, but is assumed nonetheless.) Note that in this figure, and in Figure 4, the horizontal axis is presented in wall clock time. This time is unrelated to the logical time used for version numbers.

The arrows in Figure 2 indicate a sequence of item updates. For example, item *k* is created as version *A2* and then updated at different replicas to get versions *B3*, *A3*, *B4*, and *C4*. Replica *B* is compromised at time 5, but this is not detected until after time 8. Versions *B4* and *B5* are authored by replica *B* after it was compromised and are corrupt. Version *C4* is also corrupt because it derives from corrupt version *B4*. When the compromise is detected after time 8, versions *A1*, *B1*, *A2*, *B2*, *B3*, *C2*, *A3*, *A4*, and *B4* have been superseded. Assuming the replicas have been communicating frequently, it is likely that all of the superseded versions will have been discarded from the replicas' stores.

2.4 Effective and efficient recovery

A recovery method is *correct* if it ensures that all corrupt versions are removed from the system when the recovery is complete. For example, in Figure 2, corrupt versions *B4*, *B5*, and *C4* must be removed from the system. Correctness could easily be achieved by removing everything and starting afresh, but that would not be a very useful outcome. Rather, a recovery method should also preserve existing innocent versions and recover old innocent versions that were superseded by corrupt versions. We judge a recovery method by how *effective* and how *efficient* it is in accomplishing this task.

We measure a recovery method's effectiveness by how close it comes to recovering all the most recent innocent versions. For example, in Figure 2, the most effective outcome would be to recover version *C3* of item *i*, *C1* of item *j*, *A3* of item *k*, and *A4* of item *l*. Since *A3* and *A4* are superseded versions, they would most likely have to be recovered from an archive. A recovery method can fail to be ideally effective. For example, a version that is quickly superseded may never reach an archive and consequently there may not exist any record of it.

We measure a recovery method's efficiency by how much bandwidth overhead it imposes. For example, removing all versions from all replicas and then restoring the most recent innocent versions from an archive would be both correct and fairly effective if most updates have reached the archive, but it would consume a large amount of communication bandwidth and disrupt an application's access to data during recovery. The ideal approach removes only corrupt versions and distributes only old innocent versions that need to be recovered from the archive. For example in Figure 2, only versions *A3* and *A4* must be recovered.

2.5 The Polygraph solution

To achieve effective and efficient recovery, Polygraph employs an archive log with *innocent version identification* and *replica-local retention*.

Innocent version identification: Polygraph uses both time- and derivation-based criteria to identify innocent versions. When an archive is notified that replica *B* was compromised at time *t*, it examines its log to determine which versions existed prior to time *t*. Such versions must be innocent, since they were recorded in the archive log prior to the compromise. Any earlier versions from the same authoring replica must also be innocent.

Although the time-based criterion enables versions created before the compromise to be identified as innocent, it does not find innocent versions created after the compromise, such as version *C3* in Figure 2. To determine that such versions are innocent, Polygraph adds a *taint vector* to track the derivational history of each version. Versions of items that were never tainted by a compromised replica are known to be innocent.

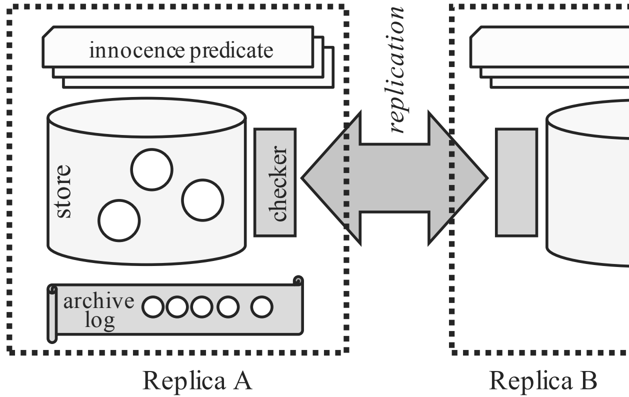

Figure 3. Polygraph software architecture.

Versions that are not identified as innocent are *suspect*. To be correct, Polygraph must suspect all corrupt versions. However, due to practical limitations, some innocent versions may also be suspect.

Replica-local retention: Rather than removing all versions stored at each replica, replica-local retention allows innocent versions to be retained, resulting in an increase in both effectiveness and efficiency. Effectiveness increases because innocent versions that have not made it to the archive can often be recovered from a replica's store. Efficiency increases because locally retained versions do not have to be resent by the archive thereby saving bandwidth.

Replica-local retention is based on the idea of an *innocence predicate* that the archive creates and distributes to each replica. Using this predicate, each replica can determine which of its stored versions is innocent. Only the remaining versions, being suspect, are removed. The replicas also install an innocence predicate as a permanent check against corrupt versions that are still propagating through the system and may be received from other replicas via the replication protocol.

Note that replicas with different local states may retain different versions in a replica-local retention approach. However, weakly consistent replication ensures that the latest version retained at any replica, as well as versions that are restored by the archive, will eventually be received by all other replicas. Thus replicas will eventually converge to a consistent state.

3. Polygraph Design

In this section, we describe Polygraph's architecture, mechanisms, and operation. We then give a correctness argument.

3.1 Architecture

Figure 3 shows Polygraph's software architecture. A replica has a store that holds at minimum the most recent known version of each item and replication software that spreads updates to other replicas. The replica maintains a set of innocence predicates that classify versions as innocent or suspect based on reported compromises. In the case of multiple compromises, to be innocent a version must be classified as innocent by all of the innocence predicates. A checker eliminates suspect versions from the store and prevents them from being received from other replicas.

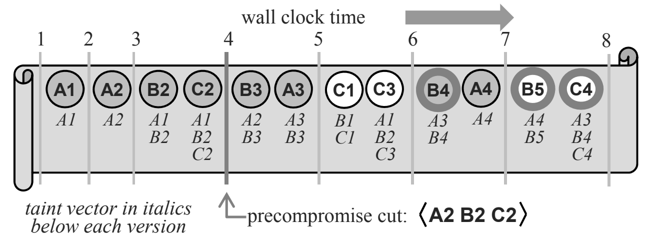

Figure 4. Example archive log for replica A.

An archive is a replica that has an *archive log*. The log is used to resurrect innocent versions that were superseded by suspect versions. To simplify the discussion, we first describe a system with only one archive. Later we extend our solution to support multiple archives.

3.2 Mechanisms

3.2.1 Archive log

The archive log records all versions received or created by the replica serving as the archive along with the time at which they were first seen. Figure 4 shows example contents of the archive log on archival replica *A* corresponding to the timeline of Figure 2. Version *A1* was recorded prior to time 2; *A2* prior to time 3; *B2* and *C2* prior to time 4, and so on. Communication delay between replicas can cause the ordering of updates in the archive log to differ from their ordering in real time. For example, version *B2* and *C1* were created before time 3 in the timeline but *B2* did not reach the log until after time 3 and *C1* until after time 5. Version *B1* never reached the log at all. The log is used for time-based identification and recovery of innocent versions.

3.2.2 Precompromise cut

Polygraph computes a time-based *precompromise cut* on the archive log to identify innocent versions. The precompromise cut contains, for each replica, the latest logical time of any version authored by that replica recorded in the archive log before the time of compromise. This can be represented as a vector with one component per replica.

If we know that a compromise occured after logical time *t* at replica *R*, any version authored at *R* that existed before time *t* must be innocent. Of course, the precise time at which a compromise occurs is not likely to be known, so a conservative estimate should be used. In Figures 2 and 4, replica *B* was actually compromised *at* time 5, but to be conservative, we estimate the compromise happened *after* time 4. Based on this, versions *A1*, *A2*, *B2*, and *C2*, which the archive recorded before time 4, must be innocent. Hence,

the archive computes the precompromise cut as ⟨*A2 B2 C2*⟩ as shown in Figure 4.

However, the archive can also infer the innocence of versions with earlier logical times than those of known innocent versions. Replica *C* must have created version *C1* before it created *C2*, and hence version *C1* must have been created prior to the compromise as well, even though the archive did not record *C1* until after time 4. This is the rationale for the *precompromise cut*.

Some replication systems propagate knowledge about versions separately from the actual contents of the items. In such systems, the archive can also record when each version was first known and use this information in constructing the precompromise cut.

3.2.3 Taint vector

Polygraph uses a derivation-based *taint vector* to identify additional innocent versions that do not fall within the precompromise cut. The taint vector is a compact derivation history that is maintained as metadata for each item. The taint vector records, for each replica, the most recent logical time at which that replica updated the item.

When replica *R* updates an item, Polygraph takes the taint vector from the item's old version and updates the component for *R* to the current logical time of replica *R*. We say that the new version is *tainted* by replica *R*. New items are handled in a similar manner, with the taint vector initialized to zero for all replicas other than the authoring replica. This process guarantees that version creation leaves a traceable influence in the taint vector, thereby ensuring that eventually, when all compromises are detected, all corrupt versions will be found.

When updating an item, an application can optionally remove or add taint if the application semantics warrant it. So, for example, if the application can assert that the previous contents of an item have no influence on the new contents, it may choose to eliminate any taint due to past updates. Similarly, if the application can deduce that multiple different items contributed to the new contents, it can specify that the new taint vector should derive from the union of the contributing items' taint. In any case, the Polygraph layer that creates the taint vector guarantees that the updating replica always leaves its own taint in the taint vector.

Per-item taint vectors enable Polygraph to recover some innocent versions that are either created at a uncompromised replica or recorded at the archive after the compromise. In Figure 4 the taint is written below each version. The archive can deduce that version *A4* is innocent because it is not tainted by the compromised replica *B*, even though the version was created after the compromise. Version *A3* is also innocent, but this cannot be proved during recovery because *A3* was tainted by *B* and logged at the archive after the compromise. Here, perfect effectiveness cannot be achieved.

The taint vector can be combined with the precompromise cut as follows. Even a tainted version can be identified

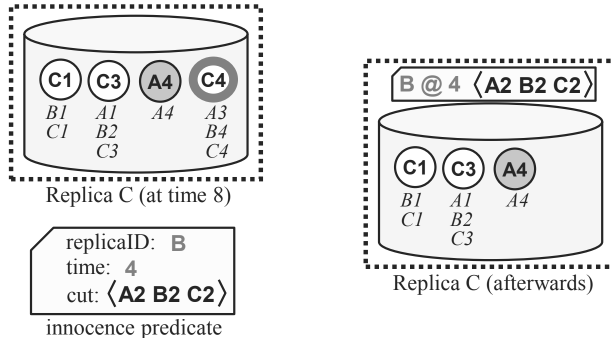

Figure 5. Replica *C* receiving an innocence predicate.

as innocent provided that the taint component for the compromised replica falls within the precompromise cut. For example, version *C3* has taint vector ⟨*A1 B2 C3*⟩. Replica *B* was compromised, but since *B2* is in the precompromise cut, the most recent update to this item by replica *B* preceded the compromise, so version *C3* is innocent.

3.2.4 Innocence predicate

In addition to using the precompromise cut and taint vector to locate innocent versions in the archive log, the archive also packages this information into an *innocence predicate* that is distributed to all replicas so that they can perform *replica-local retention* on the versions in their own stores.

An innocence predicate classifies a version as innocent provided either (1) the version is included in the precompromise cut, (2) the version was not tainted by the compromised replica, or (3) the version's taint vector includes a component for the compromised replica that is included in the precompromise cut. Any other versions are suspect.

Continuing the example of Figures 2 and 4, suppose that at time 8 replica *C*'s store contains the versions *C1*, *C3*, *A4*, and *C4* as illustrated in Figure 5. Note that because of communication delay, in this example replica *C* has not yet received version *B5*, so it still has the superseded version *A4* in its store. Archival replica *A* was notified that replica *B* was compromised after time 4 and it distributed the corresponding innocence predicate shown in the figure. Replica *C* determines that version *C1* is innocent because of the precompromise cut. Version *A4* is innocent because it is not tainted by the compromised replica and version *C3* because its taint component for the compromised replica is *B2*, which is inside the precompromise cut. The remaining version *C4* is suspect and must be removed; an innocent version of item *k* will be recovered from the archive and eventually be repropagated to all replicas.

3.3 Operation

In this section, we describe Polygraph's operation. First, we describe the normal system operation during the replication process. Next, we describe the actions taken to recover from a reported replica compromise. Finally, we describe the ap-

proach for handling multiple replica compromises and multiple archives.

3.3.1 Normal operation

During normal operation, a replica uses the underlying replication layer to exchange versions with other replicas. We assume that it is not practical for all replicas to store all item versions; replicas generally retain only the most recent version of each item although there is no inherent limit. Polygraph guarantees that each item version is properly augmented with a taint vector.

When an item is updated, Polygraph generates the taint vector as explained in Section 3.2.3 and calls the replication layer to update the versionID. Item creation is treated as a special kind of update.

When the replication layer receives versions via communication with other replicas, Polygraph interposes an *innocence check* to ensure that a replica's store remains uncontaminated by suspect versions. All received versions are checked against all previously received innocence predicates. Suspect versions are discarded. This check must be applied persistently because we do not impose a limit on how long it might take a corrupt version to propagate. Implementing such a limit would require global consensus.

In addition to behaving like a normal replica, an archive records each updated or received version in its archive log along with the time at which that version was first seen.

3.3.2 Replica compromise

Recovery from a replica compromise starts when an archive receives a compromise notification. A compromise notification contains the replicaID of the compromised replica and a time after which the compromise occurred.

Upon receiving a compromise notification, an archive scans its log to determine the last version authored by each replica that was received prior to the compromise time. The replicaIDs and logical times of the last precompromise versions for each replica constitute the precompromise cut. The archive constructs an innocence predicate by attaching the precompromise cut to the compromise notification.

The innocence predicate is then distributed to all replicas. On receiving an innocence predicate, each replica, including the archive, scans its store and removes all versions that are classified as suspect. The innocence predicate is also added to the replica's innocence checker to ensure that the replica discards suspect versions received in the future. Note that all replicas need not receive and process innocence predicates at the same time.

Additionally, the archive scans its log a second time, using the innocence predicate to classify each version in the log. If an item has been updated many times, the log is likely to contain many versions of the same item, of which only the most recent innocent one needs to be restored. So, the archive first computes the most recent innocent version of each item and then restores these versions by adding them to its local store. Such restored versions will re-propagate to other replicas using the underlying replication protocol.

3.3.3 Multiple compromises

Different devices may be compromised at different times. In the case of multiple compromises, the archive receives separate compromise notifications for each compromise. These separate notifications are processed by the archive individually, with the archive creating and distributing an innocence predicate for each compromise.

Replicas maintain all of the innocence predicates they have received and combine them when classifying a version as innocent or suspect. Since each predicate corresponds to a different compromise, all of the predicates have to agree that a version is innocent for it to be classified as innocent. Any version suspected by any innocence predicate is classified as suspect. Replicas compute the logical AND of innocence predicates for different compromises to determine whether a received version is innocent.

3.3.4 Multiple notifications

Multiple notifications may be issued for the same compromise. For example, a replica may first be known to be compromised at time t_1 but later it may be discovered that the replica was actually compromised at an earlier time t_2.

In the case of multiple notifications, the most recent compromise notification is used to determine which versions are innocent and hence should be revived. A newer innocence predicate for the same replica's compromise overrides the previous innocence predicate (if any) for the same replica's compromise. The issuing archive's logical time stamp attached to each innocence predicate determines which one is most recent.

Later compromise notifications need not have an earlier compromise time. Thus, a notification issued by mistake can be safely retracted by issuing a "true" innocence predicate that classifies all versions as innocent. Polygraph guarantees correctness as long as the last notification is conservative with respect to the actual compromise time. However, for effectiveness and efficiency, it is best if replicas receive notifications that are earlier in time compared to the prior notifications to ensure that they do not discard innocent versions.

3.3.5 Multiple archives

Polygraph supports multiple autonomous archives that can enforce different archiving policies. For instance, a user's local archive may simply store versions generated by that user, while a company archive may store the documents related to its business.

Besides simplifying the task of deploying archives in a system, multiple archives and different placement policies on these archives offer several advantages.

- Completeness: state-based replication protocols can omit transmission of superseded versions, so having archives

in different places increases the chance of recording a more complete version history, thereby aiding recovery.

- Timeliness: archives placed in different locations may receive versions at different times, increasing the chances of recording a version before a compromise occurs.
- Fault-tolerance: recovery can proceed even if one archive is currently unavailable.

Completeness and timeliness together help improve the effectiveness and efficiency of the system while increased fault-tolerance makes the system more usable in realistic settings.

Each compromise notification is distributed to all the archives. When there are multiple archives, each archive operates independently with regard to recording versions, receiving compromise notifications and generating innocence predicates. An archive can take into account other archives' innocence predicates when scanning its log to restore versions to its local store.

It only remains to describe how each replica combines the multiple innocence predicates generated by the multiple archives for a given compromise. In contrast to the case of multiple compromises in a system with one archive, when there are multiple archives generating innocence predicates for the same compromise, all of the predicates have to agree that a version is suspect for it to be classified as suspect. If even one archive produces an innocence predicate that claims the version is innocent, then the version is properly classified as innocent. Replicas compute the logical OR of innocence predicates that are received from different archives for the same compromise.

In the case of multiple archives and multiple compromises, each innocence predicate contains the replicaID and compromise time of the compromised replica. This information enables receiving replicas to combine the various innocence predicates by first ORing the innocence predicates for the same compromise and then ANDing the resulting predicates for multiple compromises.

3.4 Correctness

In this section, we argue that Polygraph ensures correctness: eventually all corrupt versions are removed from all uncompromised replicas.

First, we define the following terms to facilitate the description. Subscript R refers to replica R's component in a taint vector or a precompromise cut. Let $TC(v)$ be the taint vector of version v. So $TC_R(v)$ is R's component in the taint vector. Similarly, let $LT_R(t)$ be the logical time of replica R at time t. Let $CN(A,B,t,t')$ denote a compromise notification delivered to archive A claiming that replica B was compromised after time t when it was actually compromised at time $t' > t$. Let $PC(A,B,t,t')$ be the precompromise cut generated by archive A in response to $CN(A,B,t,t')$ and let $PC_R(A,B,t,t')$ be replica R's component in that cut. Let $IP(A,B,t,t')$ be the innocence predicate generated by archive A in response to $CN(A,B,t,t')$.

Lemma 1: If a compromised replica B creates a version v at time $t'' \geq t'$, where t' is the actual compromise time, then $TC_B(v) \geq LT_B(t')$.

This says that versions authored by a compromised replica B after its compromise will have a taint vector whose B component is at least as large as B's logical time at the time of the compromise. This follows trivially from the way the taint vector is computed.

Lemma 2: If an uncompromised replica C creates a version v derived from a version w then $TC_R(v) = TC_R(w)$ for all replicas $R \neq C$.

This says that an updated version copies taint from the previous version. This follows from the way taint vectors are generated.

Lemma 3: If $PC(A,B,t,t')$ is a precompromise cut generated by archive A and the compromised replica B creates a version v at time $t'' \geq t'$ then $PC_B(A,B,t,t') < LT_B(t'')$.

This says that a version generated by a compromised replica after its compromise will not be included in the precompromise cut. Since $t' > t$ it follows that $t'' > t$. Hence, the archive A could not have received version v before time t.

Lemma 4: If v is a corrupt version then eventually archive A generates an innocence predicate $IP(A,B,t,t')$ that classifies v as suspect.

Since v is corrupt, there must be some compromised replica B that last updated it after being compromised. Let t' be the time at which B was compromised. After this compromise is detected, eventually archive A generates an innocence predicate $IP(A,B,t,t')$. Lemma 4 follows from Lemmas 1-3 and the way the innocence predicate is used to classify suspect versions.

Lemma 5: If v is a corrupt version then eventually all uncompromised replicas classify v as suspect.

This follows from Lemma 4, the way multiple compromises are composed, and the fact that all uncompromised replicas eventually receive all innocence predicates.

Theorem: All uncompromised replicas eventually remove all corrupt versions.

This follows from Lemma 5 and the fact that an archive will not restore a suspect version. QED

4. Implementation

We have implemented Polygraph by extending the Cimbiosys [Ramasubramanian 2008] weakly consistent replication system. Our implementation is written in C# and uses the Microsoft SQL Server 2005 Express Edition for storing the backup log and the compromise notifications. We also cache the compromise notifications in memory for efficient access. The archive executes queries on the SQL server to generate the precompromise cut and to obtain the most recent innocent versions for each item.

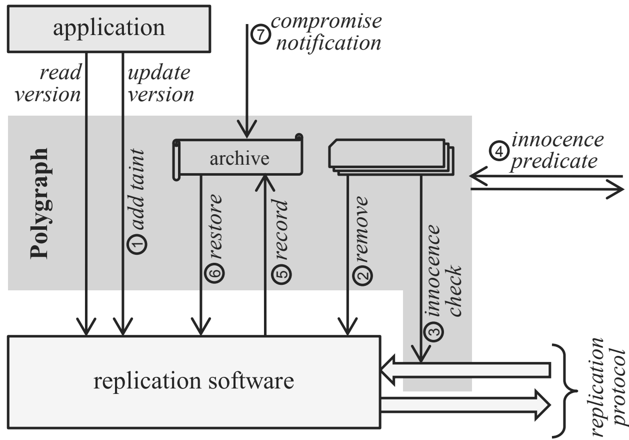

Figure 6. Polygraph extends existing replication software.

Figure 6 shows how Polygraph extends the replication software layer. The numbers in the list below correspond to the numbered item in Figure 6. The first four extensions apply to all replicas. The last three extensions only apply to replicas that contain an archive.

1. Polygraph adds a taint vector to each version and records the logical time in the authoring replica's component when an application updates a version.
2. Whenever the set of innocence predicates changes, Polygraph removes all suspect versions from the replica.
3. Polygraph ensures that suspect versions do not leak back into the replica via replication from other replicas.
4. Polygraph distributes innocence predicates to all replicas.
5. Polygraph records all versions in an archive log.
6. Whenever the set of innocence predicates changes, Polygraph ensures that the most recent innocent version of every item has been restored from the archive log.
7. A compromise notification causes an archive to scan its log to determine the precompromise cut and create a corresponding innocence predicate.

Polygraph implements taint vectors by adding them to each version's metadata. The application supplies the old taint vector to which Polygraph adds the taint component for the authoring replica. In our prototype, a taint vector is a vector of versionIDs with one component per replica in the system. Taint vectors, therefore, consume a small amount of additional storage and also require somewhat extra bandwidth during network transmission. A versionID is 20 bytes long comprising a 16-byte replicaID and a 4-byte logical time. More compact representations are possible.

Innocence predicates are roughly the same size as taint vectors. Given that any practical system should have few compromises, the overhead associated with storing and transmitting innocence predicates should be negligible. Similarly, the computational overhead involved in performing predicate checks involves only three simple comparisons, and is also negligible.

Cimbiosys is a knowledge-driven, state-based replication system that summarizes the current state of each replica's store in a compact knowledge representation based on version vectors [Ramasubramanian 2008]. A replica's knowledge succinctly records the set of versions that are stored by the replica or that are known to have been superseded by other versions. During synchronization with another device, a replica is only sent versions that are not already included in its knowledge. When suspect versions are removed from a replica, the replica's knowledge must be updated appropriately. In particular, versions that were previously known to have been superseded but are now the most recent innocent version of an item must be dropped from the replica's knowledge so that the replica will reacquire such versions during future synchronizations. In our implementation, when a new innocence predicate is received, the replica's knowledge is reduced so that it contains only those innocent versions that remain in its store. Fortunately, Cimbiosys' support for content-based partial replication requires a similar notion of knowledge retraction to cope with changing filters, so it was not difficult to implement Polygraph on the replication mechanisms provided by Cimbiosys.

Our implementation distributes innocence predicates by exploiting the replication layer. A new item is created for each innocence predicate and the replication software spreads it to all replicas.

We implemented the archive as a separate application using a special Polygraph API to restore versions and add innocence predicates to the replica and to get callbacks when new versions are received. The archive exposes an API for receiving compromise notifications. The API also supports archive log truncation. To truncate the log we must retain at least the most recent version of every item. And, of course, if the log is truncated at time t, it is only possible to recover from compromises that occur subsequent to t. Thus, one could imagine implementing a policy that allows recovery only within a fixed window into the past.

5. Evaluation

In this section, we present a quantitative evaluation of Polygraph's performance in terms of effectiveness and efficiency. Our evaluation answers the following key questions:

- How effective and efficient is compromise recovery in Polygraph compared to backup-based approaches?
- How much does each technique employed by Polygraph contribute to its effectiveness and efficiency?
- What effect does propagation delay have on Polygraph's effectiveness and efficiency?

We evaluated five compromise-recovery techniques—two backup-based methods and three Polygraph variants—

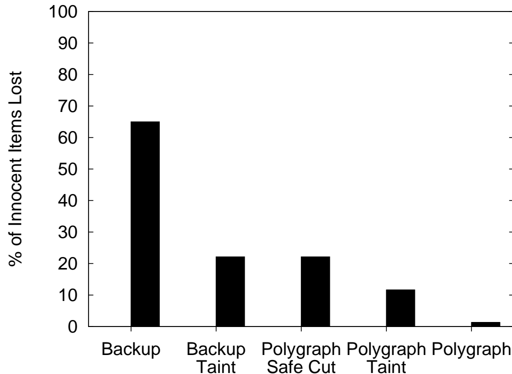

Figure 7. Effectiveness: items lost during recovery.

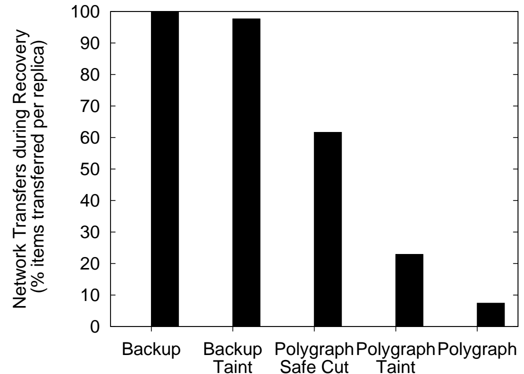

Figure 8. Efficiency: network overhead during recovery.

on the Cimbiosys replication platform [Ramasubramanian 2008]. The first technique represents typical off-the-shelf solutions available today to backup and recover data. In this technique, denoted as *Backup*, the archive logs each item it receives and recovers from a compromise by discarding all items logged after the reported compromise time. The second technique we evaluate is a custom backup technique that takes advantage of taint-indicating metadata typically maintained by weakly-consistent replication systems. In this technique, denoted as *Backup Taint*, the archive discards only those items logged after the reported compromise time that also carry the taint of the compromised replica. Note that for Backup Taint, innocent items that have not reached the archive cannot be recovered.

While the above backup-based techniques perform recovery only at the archive—discarding all local copies of the items on the replicas—Polygraph and its variants enable replicas to retain innocent items with the help of the innocence predicate produced at the archive. The first Polygraph variant, called *Polygraph Safe Cut*, detects innocence by using the precompromise cut defined in Section 3.2.2, which is a vector composed of the latest version number known to the archive for each replica. Replicas discard those items whose version numbers do not fall into the precompromise cut. The second Polygraph variant, called *Polygraph Taint*, instead uses taint to detect innocence. That is, replicas discard all items updated by the compromised replica, even if the update occurred before the replica was compromised. Finally, the technique called *Polygraph* uses both the taint vector and the precompromise cut in the innocence predicate and represents the complete Polygraph technique detailed in Section 3. Note that Polygraph not only retains all the versions that are retained by both *Backup* and *Backup Taint* at the archive but also additional versions at other replicas.

5.1 Setup

We performed experiments by running multiple instantiations of Cimbiosys on the same computer. Each experiment had 10 replicas and an archive. We are in effect emulating multiple devices on a single machine to quantify efficiency and effectiveness of Polygraph under different operating conditions. The replicas and the archive form a fully replicated system, in which each replica synchronizes and exchanges updates with the archive or another randomly chosen replica periodically. At a fixed time during the experiment called the *compromise time*, we set a randomly picked replica to act in a compromised manner by injecting corrupted data as updates to existing items in the system. At a later time, we notify the archive about the compromised replica and the correct compromise time.

The experimental workload is organized as a sequence of four phases:

1. *Initialization phase*: We insert a total of 1000 items at randomly chosen replicas (100 each on average) at the start of the experiment. We then let the replicas synchronize until the inserted items are fully replicated.
2. *Precompromise update phase*: We make a total of 1000 updates. For each update, we choose a random replica and a random item. The replicas continue to synchronize in the background. The rate of synchronization is an experimental parameter.
3. *Postcompromise update phase*: After the compromise, we perform another 1000 updates in the same manner as in the previous phase while synchronizations continue in the background (at the same rate).
4. *Compromise recovery phase*: Finally, the archive initiates compromise recovery using one of the five experimental techniques.

For each experiment in this section, we report the average value of ten executions, each with a different seed for the random number generator.

5.2 Effectiveness

We first examine the effectiveness of different recovery techniques. We measure effectiveness as the percentage of innocent items lost in the entire replication system. We consider

an item as lost if its latest version was not retained at the archive or at any uncompromised replica at the end of the recovery phase. Another way to judge effectiveness is to measure the gap (i.e., the number of discarded versions) between the latest version of the item and version retained in the system. This alternative metric shows the same general trends, and so we omit it from the discussion.

Figure 7 shows the percentage of innocent items lost while using different recovery techniques for the moderate synchronization rate scenario. In this experiment, Backup loses a substantial percentage (65%) of innocent items while Polygraph loses a very small percentage (1.3%). Since we perform updates at random, some items (about 35%) never get updated while some items get updated more than once. Backup discards all the items logged after the reported compromise and therefore loses all the item versions generated in the postcompromise update phase. Using taint information improves the effectiveness of the Backup scheme substantially by limiting lost items to about 22%.

Among the Polygraph variants, Polygraph Safe Cut has about the same effectiveness as Backup Taint. This result is expected because all versions covered by the precompromise cut were already present at the archive and local retention of some of these versions did not add to the effectiveness. Polygraph Taint, on the other hand, surpasses Backup Taint demonstrating the benefits of local recovery. It loses only about 11% innocent items compared to about 22% for Backup Taint due to the local retention of untainted innocent versions that had not been logged at the archive by the time of compromise detection.

Finally, Polygraph enables the local retention of even more innocent items. It retains items updated by the compromised replica before the compromise time by combining the taint vector recorded on the items with the precompromise cut in the innocence predicate.

5.3 Efficiency

Next, we examine the efficiency of the recovery techniques in terms of the network overhead involved in re-replicating the innocent items that replicas discarded and the superseded versions that the archive reintroduced during recovery. Figure 8 shows the per replica recovery overhead as the percentage of items transferred over the network for the moderate synchronization rate scenario.

Both backup-based techniques have close to 100% recovery overhead; each erases the local store completely and has to rebuild it using the network. The recovery overhead in Figure 8 for Backup Taint is slightly lower than 100% because we did not count replication overhead for items whose latest version was not present already in the local store. The Polygraph variants on the other hand have lower recovery overhead as they retain innocent items in the local store using the precompromise cut, or the taint vector, or both to detect innocence. All schemes consume some bandwidth to replicate superseded versions reintroduced by the archive.

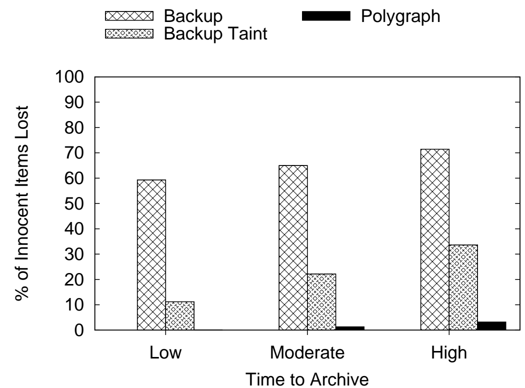

Figure 9. Effectiveness vs. propagation delay.

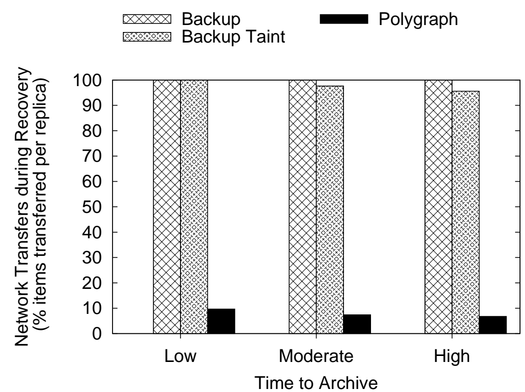

Figure 10. Efficiency vs. propagation delay.

In addition to lowering network traffic, the local retention enabled by Polygraph minimizes disruption. The items that are retained can continue to be accessed and updated by users. This improvement is valuable when network connections are weak and recovering a complete item store takes a long time.

5.4 Impact of propagation delay

Finally, we examine how the speed with which items propagate affects the performance of Polygraph. We control propagation delay by varying the ratio of updates to synchronizations. We ran tests with three different settings: *low* (0.1 updates per sync), *moderate* (5 updates per sync), and *high* (100 updates per sync). Figures 9 and 10 respectively show the efficiency and effectiveness of Polygraph compared to Backup and Backup Taint for these three parameter settings.

Figure 9 shows that propagation delay has a significant impact on the effectiveness of compromise recovery. For all techniques, the percentage of lost innocent items increases with propagation delay. With slow item propagation, more items generated before the compromise fail to reach the archive until after the compromise time. Backup-based techniques discard all these items.

On the other hand, the innocence predicate check allows Polygraph to retain many of the innocent items lost in backup-based techniques. In Figure 9, the relative effectiveness of Polygraph over the backup-based techniques increases with propagation delay. The effectiveness of Polygraph itself shows a small but noticeable decrease. This decrease is due to the diminished accuracy of the innocence predicate. Longer propagation delay means that the information used by the archive to generate innocence predicates is more out-of-date and therefore less accurate.

Slower propagation also reduces the recovery overhead for re-replication. Items that are discarded due to a strict innocence check, and then re-propagated suffer a double replication cost. Delayed propagation reduces the chance that an item will incur this cost. Figure 10 highlights this inverse effect of propagation delay on recovery overhead.

5.5 Summary

Overall, our evaluation highlights three major points. First, Polygraph provides effective compromise recovery. It retains substantially more innocent items compared to Backup, 98% vs. 35% in our experiments with a moderate propagation delay. Second, Polygraph is efficient. Accurate innocence checking eliminates the need to re-replicate innocent items already present in the local store, resulting in a greater than ten-fold reduction in recovery overhead. Finally, Polygraph's improved performance is even more crucial in common scenarios where archival backups do not happen promptly, but rather at the user's convenience and when good network connections are available.

6. Untrusted Archives

Our system model so far has assumed that archives are trusted. This section relaxes that assumption and presumes that archives can also be compromised.

If an archive can be compromised, replicas must be able to determine whether an innocence predicate is legitimate. To implement such a check, innocence predicates can include a compromise notification signed by a trusted third party. The signature indicates that the archive in question is trusted to direct recovery actions related to the specific compromise. For example, if compromise detection is performed by a human, the notification can be signed by credentials specific to that person.

An additional problem is that a compromised archive cannot be trusted to faithfully replay versions from its log. One practical technique for preventing archives from manufacturing bogus versions is to require authoring replicas to cryptographically sign versions that they generate. In fact, Cimbiosys does exactly this in its replication protocol implementation. To ensure faithful log replay, such protocol signatures must be represented in the log.

Unfortunately, a corrupted archive whose compromise has not been discovered can still produce an innocence predicate with an incorrect precompromise cut. A precompromise cut can be exploited by a compromised archive in two ways; it can generate an overly conservative or an overly liberal precompromise cut.

A overly conservative precompromise cut excludes versions known by the archive to be innocent thereby causing replicas to remove innocent versions. Although a conservative precompromise cut sacrifices effectiveness and efficiency by removing recoverable innocent versions, it doesn't affect correctness as all suspect versions are still removed. Effectiveness and efficiency concerns can be addressed by the use of multiple archives as described in Section 3.3.5. In particular, the use of multiple archives over the same content can help avoid denial-of-service failures due to the spurious generation of an overly conservative precompromise cut.

Alternatively, the compromised archive may generate a overly liberal precompromise cut that includes corrupted versions, thereby sacrificing correctness. To address this problem, replicas revoke all innocence predicates that are issued by archives that are known to be compromised. On receiving an innocence predicate indicating an archive compromise, replicas ignore all (past or future) innocence predicates created by the compromised archive and re-apply the remaining predicates to their local store and innocence checker. The re-application removes any versions that have become suspect due to the revocation of innocence predicates, thereby ensuring that only versions known to be innocent by uncompromised archives remain in the system.

7. Discussion

In this section, we discuss additional aspects of Polygraph as well as some possible extensions.

Compromise discovery. Our system relies on the ability of an external agent to accurately detect compromises. Although that is a strong requirement, it is hard to imagine making any progress towards compromise recovery without it. Furthermore, in many scenarios involving known theft or malicious acts or where accurate forensics are possible, it is reasonable to assume that compromise can not only be discovered but pinpointed precisely.

Polygraph guarantees to discard any corrupt versions as long as the external agent provides a conservative estimate of the compromise time that is equal to or earlier than the actual compromise time. If the reported compromise time is later than the actual compromise time, our system may retain some corrupt versions after the recovery. The percentage of corrupt versions that remain in the system depends on various factors such as the difference between the actual compromise time and the reported compromise time, the number of versions that are injected into the system by the compromised replica, the fraction of corrupt versions that reach the archive, and clock skew between the archive and the external agent.

Retaining suspect versions. We can improve effectiveness and efficiency if replicas retain versions that fail innocence predicate checks instead of discarding them immediately. This can help a replica recover local versions when the archive corrects a too-conservative innocence predicate with a later, more-accurate one.

Strengthening the threat model. If malicious attacks can tamper with the replication and Polygraph layers of replica nodes, there are additional considerations.

First, a replica may attempt to forge items so they appear to be authored by other replicas. As in the case of untrusted archives, this threat can be eliminated if replicas sign all data that they create. Items without valid signatures are then invalid. The keys that make such signatures can be certified and revoked by a trusted third party in the usual fashion.

Second, we can no longer assume that the Polygraph layer will reliably represent the local replica in the taint vector of every item it updates. Instead we can add a *taint check* at all partner replicas to ensure that each received item contains taint that at minimum reflects the item's versionID. So, for example, if replica *B* creates version *B5*, the receiver checks that *B5* is included in the received taint vector. The taint check is implemented by adding it to the checker module described in Figure 3.

Third, a corrupt replica can manufacture updates that appear to be from the past in logical time. To deal with this threat, the trusted third party can revoke the corrupt replica's key and issue a replacement key to the archive. The archive uses the new key to re-author versions authored by the compromised replica prior to the time of compromise. All versions signed by a revoked key are then treated as suspect. This will, of course, reduce effectiveness by eliminating potentially innocent versions and reduce efficiency by forcing all re-authored updates to be fetched from the archive. The use of Forward Secure Signatures [Cronin 2003] might provide a better approach to this problem by restricting the ability to make signatures that appear to be from the past.

Finally, replication metadata on a compromised replica may be corrupted. For example, the replica's knowledge can be modified to indicate that the replica knows of versions that it has not yet received, thereby obstructing eventual convergence. Detecting and recovering from such protocol-specific attacks requires new techniques that are outside the scope of this paper.

8. Related Work

There has been much prior work in the area of weakly consistent replication. Systems such as Bayou [Petersen 1997], PRACTI [Belaramani 2006], Ensemblue [Peek 2006], Cimbiosys [Ramasubramanian 2008], and WinFS [Novik 2006] all support replication in the context of intermittent, peer-to-peer communication and are well represented in the literature. Unfortunately, none of these systems deals explicitly with recovery from corruption. Moreover, it is possible that Polygraph could be layered on top of some of the other systems above in addition to Cimbiosys.

Spreitzer et al. [Spreitzer 1997] discuss various countermeasures to reduce the impact of server corruption in similar systems. They present a protocol that prevents servers from modifying or fabricating data as it propagates. Furthermore, they detect other malicious server behavior through client auditing. However, unlike Polygraph, this work does not entertain the possibility that the originating client (the data source) might be corrupted.

Commercial products [NetApp 2008, Decho 2007, Apple 2007, Carbonite 2007] as well as research systems [Peterson 2005, Santry 1999] already support the recovery of versioned data. These systems preserve superseded versions of data to help recover from malicious software, operator error, and (in some cases) disk failures and natural disaster. However, to recover from corruption in such systems, the user must either manually distinguish corrupt data from uncorrupted data, or roll-back to a state that existed at a known good time. Polygraph extends the time-based rollback approach to effectively and efficiently recover the state of a dynamic distributed system.

Lomet et al. [Lomet 2006] propose a rollback facility for database systems that can undo tainted transactions while allowing untainted ones to proceed. This work assumes a centralized database that supports rollback as well as a complete transaction log. We do the same for a distributed system, but with incomplete logs and asynchronous nodes.

The concept of taint has appeared in many different contexts such as buffer-overflow attack detection [Newsome 2005], language-based privacy [Myers 2000], distributed information flow control [Myers 1997], and operating systems [Zeldovich 2006]. As in these systems, we use taint to reflect the possibility of corruption along a data path, and as in many of the systems above, we provide a means for removing taint in a discretionary fashion. However, we present the first system to use taint to facilitate recovery amongst weakly consistent replicas.

9. Conclusion

Polygraph is a robust solution for recovering from data corruption in weakly consistent replicated storage systems. Upon notification of a device compromise, Polygraph ensures that 1) all data items corrupted by the compromised device are removed from the system and replaced with older, uncorrupted versions when available and 2) all innocent items not updated by the compromised device are retained so that the rest of the system experiences minimal disruption. To reduce unnecessary bandwidth consumption, devices retain as many innocent items locally as possible. We have implemented Polygraph on top of a real replication platform. Our implementation can handle multiple compromises and enables each device to recover independently at the most suitable opportunity.

Acknowledgments

We would like to thank our shepherd, Patrick Goldsack, and the anonymous reviewers for their insightful feedback on this paper. We also thank Úlfar Erlingsson for helping us rationalize our threat model.

References

[Apple 2007] Apple Inc. Time Machine. `http://www.apple.com/macosx/features/timemachine.html`, 2007.

[Baker 2006] M. Baker, M. Shah, D. S. H. Rosenthal, M. Roussopoulos, P. Maniatis, T.J. Giuli, and P. Bungale. A fresh look at the reliability of long-term digital storage. In *Proc. of the ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys '06)*, pages 221–234, 2006.

[Belaramani 2006] N. Belaramani, M. Dahlin, L. Gao, A. Nayate, A. Venkataramani, P. Yalagandula, and J. Zheng. PRACTI replication. In *Proc. of the USENIX Symposium on Networked Systems Design and Implementation (NSDI '06)*, pages 59–72, May 2006.

[Carbonite 2007] Carbonite, Inc. Remote backup online. `http://www.carbonite.com`, 2007.

[CNet 2007] CNet.com. Biz travelers beware: Airport ad-hoc hot spots could be dangerous. `http://news.cnet.com/8301-10784_3-9888021-7.html`, 2007.

[Cronin 2003] E. Cronin, S. Jamin, T. Malkin, and P. McDaniel. On the performance, feasibility, and use of forward-secure signatures. In *Proc. of ACM Conference on Computer and Communications Security*, pages 131–144, 2003.

[Decho 2007] Decho Corporation. Online backup, data backup, and remote backup solutions. `http://www.mozy.com`, 2007.

[Dvorak 2006] P. Dvorak. Spike in laptop thefts stirs jitters over data. *The Washington Post*, June 22, 2006.

[England 2003] P. England, B. Lampson, J. Manferdelli, M. Peinado, and B. Willman. A trusted open platform. *IEEE Computer*, 7(36):55–62, 2003.

[Lomet 2006] David Lomet, Zografoula Vagena, and Roger Barga. Recovery from "bad" user transactions. In *SIGMOD '06: Proc. of the ACM SIGMOD International Conference on Management of Data*, pages 337–346, 2006.

[Microsoft 2008a] Microsoft Corporation. About Active Directory Domain Services. `http://msdn.microsoft.com/en-us/library/aa772142(VS.85).aspx`, 2008.

[Microsoft 2008b] Microsoft Corporation. Live Mesh. `https://www.mesh.com`, 2008.

[Myers 1997] A. Myers and B. Liskov. A decentralized model for information flow control. In *Proc. of the ACM Symposium on Operating Systems Principles (SOSP '97)*, pages 129–142, October 1997.

[Myers 2000] A. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering Methodology*, 9(4), 2000.

[NetApp 2008] NetApp, Inc. Snap vault. `http://www.netapp.com/us/products/protection-software/snapvault.html`, 2008.

[Newsome 2005] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proc. of the Network and Distributed System Security Symposium*, February 2005.

[Novik 2006] L. Novik, I. Hudis, D. Terry, S. Anand, V. Jhaveri, A. Shah, and Y. Wu. Peer-to-peer replication in WinFS. Technical Report MSR-TR-2006-78, Microsoft Research, June 2006.

[Peek 2006] D. Peek and J. Flinn. Ensemblue: Integrating distributed storage and consumer electronics. In *Proc. of the Symposium on Operating Systems Design and Implementation (OSDI '06)*, pages 219–232, November 2006.

[Petersen 1997] K. Petersen, M. Spreitzer, D. Terry, M. Theimer, and A. Demers. Flexible update propagation for weakly consistent replication. In *Proc. of the ACM Symposium on Operating Systems Principles (SOSP '97)*, pages 288–301, October 1997.

[Peterson 2005] Z. Peterson and R. Burns. Ext3cow: A time-shifting file system for regulatory compliance. *ACM Transactions on Storage*, 1(2):190–212, May 2005.

[Ramasubramanian 2008] V. Ramasubramanian, T. Rodeheffer, D. Terry, M. Walraed-Sullivan, T. Wobber, C. Marshall, and A. Vahdat. Cimbiosys: A platform for content-based partial replication. Technical Report MSR-TR-2008-116, Microsoft Research, 2008. To appear in *Proc. of the USENIX Symposium on Networked Systems Design and Implementation (NSDI '09).*

[Saito 2005] Y. Saito and M. Shapiro. Optimistic replication. *ACM Computing Surveys*, 37(1):42–81, 2005.

[Saltzer 1974] J. Saltzer. Protection and the control of information sharing in Multics. *Communications of the ACM*, 17(7):388–402, 1974.

[Santry 1999] D. S. Santry, M. J. Feeley, N. C. Hutchinson, A. C. Veitch, R. W. Carton, and J. Ofir. Deciding when to forget in the Elephant file system. In *Proc. of the ACM Symposium on Operating Systems Principles (SOSP '99)*, pages 110–123, December 1999.

[Spreitzer 1997] M. Spreitzer, M. Theimer, K. Petersen, A. Demers, and D. Terry. Dealing with server corruption in weakly consistent, replicated data systems. In *Proc. of the ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom '97)*, pages 234–240, September 1997.

[Symantec 2007] Symantec Corporation. Symantec Backup. `http://www.symantec.com`, 2007.

[Vogels 2008] W. Vogels. Eventually consistent. *Queue*, 6(6):14–19, 2008.

[Wikipedia 2008] Wikipedia. Polygraph. `http://en.wikipedia.org/wiki/Polygraph`, January 2008.

[Wobber 1994] E. Wobber, M. Abadi, M. Burrows, and B. Lampson. Authentication in the Taos Operating System. *ACM Transactions on Computer Systems*, 12(1):3–32, February 1994.

[Zeldovich 2006] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *Proc. of the Symposium on Operating Systems Design and Implementation (OSDI '06)*, pages 263–278, November 2006.