Making Programs Forget: Enforcing Lifetime For Sensitive Data

Jayanthkumar Kannan *Google Inc.*
Gautam Altekar *UC Berkeley*
Petros Maniatis *Intel Labs Berkeley*
Byung-Gon Chun *Intel Labs Berkeley*

Abstract

This paper introduces *guaranteed data lifetime*, a novel system property ensuring that sensitive data cannot be retrieved from a system beyond a specified time. The trivial way to achieve this is to "reboot"; however, this is disruptive from the user's perspective, and may not even eliminate disk copies. We discuss an alternate approach based on *state re-incarnation* where data expiry is completely transparent to the user, and can be used even if the system is not designed *a priori* to provide the property.

1 Introduction

Once a piece of data is given as input to an application, it is difficult to enforce policies on how long such data is retained on the system hosting the application (e.g., in memory, or on disk), if the application is not designed to support such policies *a priori*.

For instance, consider a user who issues a sensitive command in her `bash` session. She desires that this command not be retained in memory or disk, and deletes her command history file. As another example, consider a user who uses Outlook to retrieve a sensitive email, which she immediately deletes after reading. Her intent is to ensure that the data is not retrievable from the computer anymore. In these cases, the user has no guarantee that wiping her history file or deleting an email is sufficient to ensure that no part of her sensitive data is retrievable from memory or on disk; she may have copied it to the clipboard by mistake or she may have copied the data from one email to another or the application may have cached it in memory. Such retention of data long beyond the usage period is very common [6].

A system ensuring that a particular piece of sensitive data given as input to an application is not retrievable from the system after a certain lifetime is said to have the *guaranteed data lifetime* property. This property helps achieve the "least privilege" principle: data should not persist in the system beyond the minimum duration required for its processing. This protects the data from leakage upon system compromise and from other subsequent malicious users of the system.

This problem has been addressed before in the literature; for instance, Perlman et al. [11] addresses cryptographic time-based data *release* policies that control when data is revealed to an application, while Chow et al. [7] and Borders et al. [3] scrub sensitive data *once* the application or user (respectively) are done with it. We aim to achieve the same property under a much tighter interpretation: our goal goes beyond data release to data *in-use* within an application without *any explicit application support*. To the best of our knowledge, this property has not been discussed before in the literature.

This property is easy to achieve if the application supports such an option (e.g., some enterprise email clients support time-based deletion). However, since most applications do not support lifetime enforcement, the problem is much harder to solve in practice. For instance, an application may cache data on disk, or it may use lazy deallocation. This precludes the use of existing approaches (e.g., shredding [7]) that rely on scrubbing memory at de-allocation. Alternatively, one could use virtual machines to isolate the application processing the sensitive data. For instance, the user can initiate a virtual machine that hosts her email client, and then tear down the virtual machine and the application upon session completion [12, 3]. This approach provides coarse granularity, since everything a user does in the temporary VM, including other received emails, is disposed of; it also requires considerable change in user behavior.

In this work, we explore how to obtain this property from a *legacy* application on a *fine* granularity. We envision a system-wide "incognito" mode on a legacy system. This is a hard problem since an application may store input data in memory and on disk; even if one identifies such locations, it is not possible to "zero" out these locations unless the application has explicitly deallocated or deleted them. Doing so would corrupt internal invariants that the application expects from in-memory and on-disk data structures (for instance, if an email is cached, with its length followed by the contents of the email, zeroing out length and contents could crash the program).

We have three main requirements. First, we aim to offer *guaranteed lifetime bounds* irrespective of *any application behavior*. Second, the tool must not interfere with the *correctness* of application integrity; when ensuring the time bound, the application should not crash. Third, we aim to ensure that the user's interaction with her applications should be the same as before; she has a completely *seamless experience*.

Our key observation towards meeting these requirements is the concept of *state equivalence*. We observe that for any program state computed from sensitive user data, there usually exists an equivalent program state not derived from the sensitive data. We say that two program states are equivalent iff the portions of state not derived from sensitive data are the same.

In order to meet the lifetime bound, at the time of data expiry, we replace the application state with an equivalent one through a process of *state re-incarnation*. We force the program to an equivalent state by rolling back execution to a pristine state and then replaying a modified version of the original sensitive input. The exact perturbation we perform to the input stream depends on our notion of equivalence: we will define this later. This approach meets all our requirements; it does not depend on the application being lifetime-aware, it maintains application integrity, and the user need not modify her interaction with the system. We will now describe our approach and highlight the main challenges in realizing it, followed by preliminary results.

In this work, we assume that the user wishes to wipe the data from her machine only, not from other machines to which her data may have been sent. The distributed case can be handled by suitably tagging network data with markers (as in, distributed information flow control), but it is beyond the scope of this work.

2 Related Work

Our goals differ from existing work in that we aim to erase any in-memory and on-disk traces of sensitive data *after* the data has been seen by the application, *while* the application is still running, and *without* requiring any change in the user behavior. We now discuss work with closely-related objectives. Chow et al. [7] provide secure de-allocation of sensitive data; as soon as any data (or memory pages) is de-allocated by the application (or kernel), it is promptly zeroed out. In contrast, we do not require that the application de-allocate the data before we expire it; thus, we do not rely on timely de-allocation by the application. Borders et al. [3] discuss an approach where any sensitive data is entered by the user into a stand-alone virtual machine which is disposed after use. Though simpler than our approach, it requires the user to interact with two different copies of the application, and any changes made to the "sensitive" VM are lost upon completion. Our system offers an alternative that requires no changes on user behavior. TightLip [13] uses shadow processes to prevent data leakage from an application to external machines. When a process receives a sensitive input, a shadow process is created that receives a scrubbed version of the sensitive input; any differences in visible output (e.g., network communication) between these two processes indicates a possible leakage of confidential information. Though the replay phase of our approach is somewhat similar to TightLip, our goal of guaranteed lifetime is orthogonal to Tightlip's goal of controlling data release. Finally, Perlman et al. [11] presents a file system where policies can be associated with files that are not readable past a certain time. The goal here is that a sender sends a file encrypted using a key that is retrievable from a master server only during a certain time; the system presents a design for the key management required at the server and the corresponding access mechanisms at the recipient. Our system's goal is complementary to these goals; they aim to achieve time-based guarantees on data release, whereas we are concerned with enforcing guarantees on data that has already been released. Our work is also related to Retro [9]; their system aims to "undo" the effects of one specific action to a system (say, a successful network intrusion) by using checkpointing, rollback, and re-execution based on predicates. The main difference from our work is that we wipe all traces of sensitive data from a system in the face of unaware (and not explicitly adversarial) applications.

3 Our Approach: State Re-incarnation

The starting point of our approach is that, since it is not possible to redact data once seen by an application, one has to instead take an application *snapshot* before it receives said sensitive data, and then restore it to this checkpoint once data lifetime has expired. However, in doing so, the application would lose all state, including changes that the user wishes to persist. To prevent this, we log all inputs to the application after it receives the sensitive data, and then initiate *replay*. Logging is necessary to handle the *non-determinism* inherent in applications; for instance, any calls to a random-number generator or file contents (which may change) need to be replayed as such. However, straightforward replay would leak the sensitive data; instead, we perturb the input so that the application no longer sees any sensitive data. Thus the application state at the end of replay cannot include any data derived from the sensitive information. The image of the running application is then replaced with the re-incarnated version from replay.

We note that if the developer can architect her program in a certain fashion, other approaches may be possible (e.g., Micro-rebooting [4]). However, since most applications do not support lifetime enforcement directly, we resort to a more heavy-weight run-time solution. Our run-time approach operates in three stages implemented via an application-level or OS-level framework. For now, we assume that the sensitive data is processed in memory by a single application; we will later discuss the case when multiple applications are involved.

Identifying Sensitive Input: The user needs a mechanism to communicate what part of her input is sensitive. This input may be delivered to the application over any channel (e.g., keyboard, network). We envision two ways to achieve this. The first is meant for keyboard input; the user can prefix and suffix her sensitive input with a specific character sequence. This sequence is intercepted by our framework and is not delivered to the application; our framework can provide visual indication that secure data entry is in progress. The character sequence can also indicate the data lifetime, or a default lifetime can be used. The second method is suited for input from a data source (e.g., a network, or file); a simple data-specific plugin can be used to identify sensitive data. For instance, in an enterprise, a specific SMTP option can be added (by the sender) that indicates the lifetime of an email. A SMTP plugin can determine the sensitivity of a particular network input before delivering it to the application.

Checkpoints/Logs: Once sensitive input is identified, the framework initiates a checkpoint of the application receiving the input. This includes all pertinent application state, including memory, registers, execution state of any threads (e.g., stack). Further, all subsequent inputs to the application, whether sensitive or not, are logged. This can be achieved by logging return values at the syscall level using, for instance, binary translation.

Perturbed Replay: The key part of the system concerns actions initiated when the lifetime of a sensitive input expires. If the application that received the sensitive input has terminated, then no action need be initiated (we are assuming that the memory belonging to the process is scrubbed out immediately). Otherwise, the system restores the application to the checkpoint corresponding to data entry. It then replays all subsequent inputs after which the system resumes. The goal is to ensure that the application is left in a *equivalent* state; we discuss this issue in detail subsequently.

This approach can be extended to multiple data items. Multiple checkpoints are maintained when each data item is seen by the application; during replay at the expiry of a data item D, the checkpoints of subsequent data items D' received by the application are updated.

4 Challenges

State re-incarnation raises several interesting questions: *fidelity*, *pervasiveness*, *containment*, and *overhead*.

Fidelity: The key question regarding perturbed replay is that application state at the completion of replay should be concordant with the user's expectations. While the constraint from the secrecy aspect is clear (an adversary cannot retrieve the sensitive data from the application state), it is difficult to define exactly what the user's expectations correspond to. Thus, the notion of state equivalence is not a precise one.

Consider the case where Outlook obtains a sensitive email over the network; our mechanism takes a suitable checkpoint and records subsequent input. The user reads the email, and then navigates (say, using a down arrow) to the next (non-sensitive) email and deletes it. During replay, the sensitive email is suitably perturbed; however, we need to ensure that the navigation keystroke is interpreted in the same "context" as during the original run. If the cursor points to a different email upon replay, then a different email would be deleted, and thus the user's expectations would be violated. The general problem is that inputs to the applications subsequent to the sensitive input from the user or the network only make sense in the context of the sensitive data. We now consider options for perturbing the input stream, and examine the impact on secrecy and fidelity:
Replay with Omission: This is the simplest approach one can envision; sensitive input is omitted during replay. While this approach guarantees secrecy (since the application never sees sensitive input in replay), the fidelity of this approach is poor. In the example of Outlook navigation, if the sensitive email is never seen by the application in replay, an incorrect email may be deleted.
Replay with Substitution: In this method, the sensitive input is replaced with non-confidential boiler-plate input (e.g., : "This is a sensitive email that has expired.") during replay. This clearly provides secrecy. With respect to fidelity, while this works in the email navigation example, there are several counter-examples. For instance, if the original email differs in length from the boiler-plate input, replay may deviate considerably.
Replay With Consistent Substitution: This approach is similar to replay with substitution, with one difference: the substituted input is such that the application follows exactly the same control flow as before. The instructions

executed by the application can be classified as data manipulation instructions (e.g., *mov ax, bx*) or unconditional and conditional jumps (e.g., *jnz address*). We compute the substituted input such that all conditional jumps play out in the same way as before using standard symbolic execution and constraint solving (e.g., as in ODR [1]). It is possible to improve on this by marking what portion of the sensitive data has already been deleted by the system; such data need not be re-computed.

Regarding fidelity, note that though the execution path remains the same, the output may be different. For instance, if the sensitive email is written to disk, during replay, the substituted input will end up on on disk instead. Also note that, secrecy and high fidelity can be contradictory requirements; the computed input may reveal some information about the sensitive input (e.g., typically, processing depends on input length; so the perturbed input would reveal the length of the sensitive input). This trade-off is fundamental: restoring the application to any sort of meaningful state requires the substituted input to reveal some information. In practice, this information may be fairly low; in the typical processing of a message in an application's data plane, we expect few conditional jumps based on the contents of the email. Thus, the number of constraints that need to be solved (each of which reveals some information about the sensitive input due to comparison operations) will be minimal in practice. Further, methods to estimate this leakage by considering all conditional jumps using a conservative upper-bound are known (e.g., Castro et al. [5]). Thus, a threshold can be set on the information leakage by the user; if this threshold is exceeded, the system can request the user to shutdown the application as a fallback.

Each of these perturbation options adhere to various notions of state equivalence; we believe that perturbed replay with consistent substitution offers the best trade-off with respect to fidelity and secrecy. We plan to evaluate this by experimenting with typical applications (e.g., web browsers, email clients) and use scenarios (e.g., entering password, processing sensitive emails).

Pervasiveness: The second challenge in our approach is to handle all possible ways in which sensitive data may linger in a system. Apart from the application receiving the sensitive data, OS modules may store copies of the data as well. Application input is typically processed by the OS (e.g., device drivers, the network stack, the keyboard buffering system) before it is sent to the application. The same applies for screen or network output. These software modules can buffer this information; either intentionally (e.g., to maintain some history for various statistics) or accidentally (e.g., maintaining a static buffer for keyboard events is typical; sensitive data may linger in such buffers).

To address such data retention in the OS, our approach can be applied at the OS level. One option is to use a virtual machine monitor (VMM); our framework can reside in the VMM to record all inputs and outputs to the OS. Of course, this begs the question of how to ensure that no data is retained by the newly introduced VMM layer: we argue that a VMM is typically simple enough so that its code can be audited to rule out such leaks. An alternative is to interpose our framework on the VMM; for instance, allow the rollback of certain portions of the VMM's memory. We believe that relying on a VMM that does not buffer data or offers an explicit API to control buffering to be a much simpler option.

Containment: The application of interest may communicate with other entities: processes on the same machine or remote machines. This means that: (a) outputs sent to these entities may leak information. (b) inputs by these entities may only apply to the original execution.

Regarding outputs to processes on the same machine, when using an *application-level replay* mechanism, one can recursively apply the notion of checkpointing, logging, and replay. For outputs related to sensitive data, the recipient processes can be checkpointed. We use dynamic taint tracking (e.g., Newsome et al. [10]) for the latter. During replay with consistent substitution, any constraints implied by the execution of the recipient process are also considered in computing the substitution input. This issue does not arise for a *VMM-level replay* mechanism; in that case, the framework sees only a single "blob" of execution, which includes the OS and all applications within it.

The issue that arises in both application-level replay and VMM-level replay is data sent to *remote* machines. In this case, there are two alternatives we plan to explore. First, if the remote machine is within the enterprise, we can first verify that lifetime enforcement is supported on that machine (using standard attestation techniques), and then convey these guarantees remotely. Alternatively, if the machine is external, outputs can either be delayed until the lifetime expires and the output based on the perturbed input can be computed. This works if the application does not need to receive a response in order to proceed; in this case, we can simply notify the user that her data is being released externally, and the choice to permit or disallow this is up to the user.

Regarding inputs from other entities to the application, the fidelity issue arises. For instance, when the password is sent by the browser to an external website, the website may return a page corresponding to successful authentication. Such inputs can be replayed as such. It is possible however that the external machine mirrors the sensitive data; in this case, taint tracking is "lost" because the external machine is not under our sphere of influence. However, we believe that such occasions of mirroring are rare; to handle exceptions, it is possible to add protocol level tainting rules to ensure that such taints are propagated as well. One can, for instance, identify matching requests and responses [2] and suitably modify the response from the web server during replay in accordance with the perturbed request. Alternatively, such external inputs may simply be ignored without affecting replay; the application may not rely on using the external input. We plan to examine the effect of this in our experiments with typical applications.

Overhead: The final challenge is in ensuring that the system does not adversely impact performance. Since our target here is client-side systems (as opposed to heavily loaded servers), we do have some leeway in terms of overhead. The overhead of identifying sensitive data is minimal for both dynamic methods (e.g., keyboard markers) and configuration-based methods (e.g., relying on marked email). The overhead of checkpointing is also one-time and incurred only at the receipt of sensitive information. Logging is more of a burden since it is required at all times subsequent to the receipt of sensitive information. We believe this can be done with an overhead of about 3X which may be reasonable for client-side systems especially with the advent of multi-core systems (this estimate is from Practical Data Confinement [8] when the volume of sensitive data is 20% of total data). The overhead of replay involves both the computation of suitable substitution input as well as replaying past input. The main advantage here is that replay can be initiated in parallel with the usage of the application; the replay can operate on a shadow image of the application. Even if the data lifetime is on the order of days, we expect that typical applications will not process old data frequently; thus, the overhead of logging and replay would be proportional to the number of instructions that operate on tainted data, rather than data lifetime.

We evaluated this overhead using an application-level logging and replay mechanism based on binary translation (Valgrind). To obtain preliminary feasibility results, we focus currently on logging; logging is active continuously, while perturbed replay is invoked only when lifetime expires. We ran `bash` using a harness and consider all keyboard input as sensitive. We typed in a command at the prompt, and measured the overhead.

Our harness uses binary translation to replace every instruction with an instrumented version that propagates taint suitably. Because of translation, during invocation of `bash`, there is an initial delay (5 seconds), after which performance impact during interactive use is not noticeable. We typed in a single command and measured the number of instructions executed. Out of a total of $33,481$ instructions, only 266 of them ever process any tainted input. This bodes well since the overhead of constraint solving and replay is determined by the number of instructions operating on tainting data. These 266 instructions generated 9730 constraints (each instruction may be executed multiple times, for instance, in a loop). Results in previous work (e.g., ODR [1]) for comparable number of constraints indicate that replay with consistent substitution holds promise.

5 Conclusion

We introduce a novel approach to ensure that sensitive data is seamlessly "scrubbed" once its lifetime expires. Our approach ensures this without any application support and without impacting application correctness or the user experience. This approach raises a number of challenges; we believe that the issue of fidelity will be the hardest to handle, and hope to evaluate it through experiences with real applications.

References

[1] ALTEKAR, G., AND STOICA, I. ODR: Output-Deterministic Replay for Multicore Debugging. In *Proc. SOSP* (2009).

[2] BARHAM, P., DONNELLY, A., ISAACS, R., AND MORTIER, R. Using Magpie for Request Extraction and Workload Modelling. In *Proc. OSDI* (2004).

[3] BORDERS, K., V, E., WEELE, E., LAU, B., AND PRAKASH, A. Protecting Confidential Data on Personal Computers with Storage Capsules. In *Proc. Usenix Security* (2009).

[4] CANDEA, G., KAWAMOTO, S., FUJIKI, Y., FRIEDMAN, G., AND FOX, A. Microreboot — A technique for cheap recovery. In *Proc. OSDI* (2004).

[5] CASTRO, M., COSTA, M., AND MARTIN, J.-P. Better bug reporting with better privacy. In *Proc. ASPLOS* (2008).

[6] CHOW, J., PFAFF, B., GARFINKEL, T., CHRISTOPHER, K., AND ROSENBLUM, M. Understanding Data Lifetime via Whole System Simulation. In *Proc. USENIX Security* (2004).

[7] CHOW, J., PFAFF, B., GARFINKEL, T., AND ROSENBLUM, M. Shredding your garbage: reducing data lifetime through secure deallocation. In *Proc. USENIX Security* (2005).

[8] ERMOLINSKIY, A., KATTI, S., SHENKER, S., FOWLER, L., AND MCCAULEY, M. Practical Data Confinement, In Submission.

[9] KIM, T., WANG, X., ZELDOVICH, N., AND KAASHOEK, F. Intrusion Recovery Using Selective Re-execution. In *Proc. OSDI* (2010).

[10] NEWSOME, J., AND SONG, D. Dynamic taint analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Proc. NDSS* (2005).

[11] PERLMAN, R. File System Design with Assured Delete. In *Proc. SISW* (2005).

[12] QubesOS: Architecture. http://qubes-os.org/Architecture.html.

[13] YUMEREFENDI, A. R., MICKLE, B., AND COX, L. P. TightLip: Keeping Applications From Spilling The Beans. In *Proc. NSDI* (2007).