Data Loss Prevention based on data-driven Usage Control

Tobias Wüchner
Technische Universität München
Garching bei München, Germany
tobias.wuechner@in.tum.de

Alexander Pretschner
Technische Universität München
Garching bei München, Germany
alexander.pretschner@in.tum.de

Abstract—**Inadvertent data disclosure by insiders is considered as one of the biggest threats for corporate information security. Data loss prevention systems typically try to cope with this problem by monitoring access to confidential data and preventing their leakage or improper handling. Current solutions in this area, however, often provide limited means to enforce more complex security policies that for instance specify temporal or cardinal constraints on the execution of events. This paper presents UC4Win, a data loss prevention solution for Microsoft Windows operating systems that is based on the concept of data-driven usage control to allow such a fine-grained policy-based protection. UC4Win is capable of detecting and controlling data-loss related events at the level of individual function calls. This is done with function call interposition techniques to intercept application calls to the Windows API in combination with methods to track the flows of confidential data through the system.**

Keywords-**data loss prevention; usage control; microsoft windows security; dynamic data flow tracking**

I. Introduction

As data has become one of the most important business drivers in our modern information-driven society, the protection against disclosure of sensitive data is an important goal for most companies and governmental organizations. This is not only due to the fact that leakage of confidential data may lead to an instant economical loss due to competitive disadvantages, but also may severely damage the reputation of a company and thus also may induce a loss of customers or weakened negotiation positions with other contractors due to damaged trust relationships.

Many security systems that try to cope with this problem mainly focus on protecting against external attacks and attempts to provoke data leaks from outside the companies. Recent studies show, that so-called insider threats are at least a similarly severe problem in terms of disclosure of confidential data. These insider threats do not only manifest themselves in the risk of deliberate data disclosure caused by social engineering or the intentional misuse of privileges by employees. In addition, accidental or inadvertent disclosure of sensitive information by company employees due to non-adherence to company security guidelines or simply due to careless behavior is a most relevant insider threat [1], [2].

Data Loss Prevention systems (DLP) aim to address the risk of data disclosure from insiders (and some malware as well). Their main idea is to observe and control the storage, movement, or handling of confidential data according to specified security policies. A common taxonomy to classify DLP systems is based on their their protection scope [3]: DLP solutions that protect *Data-At-Rest* identify sensitive data in persistent storage locations and, if a security policy specifies this, removes or encrypts them if the location is considered non-trusted; if a DLP solution considers *Data-In-Use*, it monitors and controls the interaction of users and sensitive data according to applied security policies; finally the scope *Data-In-Motion* means that a DLP system monitors and controls the movement of sensitive data at the network level.

Usage control generalizes access control to the future handling of data once access to it has been granted. It can be seen as a natural framework for the idea of Data-In-Use DLP, as it aims at regulating the usage of (confidential) data according to specified security policies and thus also allows to prohibit unwanted events of data leakage. However, it is more expressive with respect to preventing data disclosure than most data loss prevention systems, as allowed or disallowed handling of data can be defined in a more fine granular way. While a typical data loss protection system either allows or disallows, say, copying data from a trusted system to a non-trusted USB stick, a usage control solution could specify that this copying process should only be blocked within business hours or that at maximum five copies are allowed. In sum, usage control offers the possibility for a more fine-granular control of data usages and movements than typical data loss prevention systems, allowing for the enforcement of more complex and context-specific policies, thus establishing a more flexible way of protecting against inadvertent data disclosure. Our paper shows how one usage control framework can be instantiated as a DLP solution.

Problem. The use of data loss prevention solutions is a common way to prevent insider-driven inadvertent information disclosure at endpoint systems [4]. Current data loss prevention solutions are rather limited in terms of flexibility of enforceable security policies [5]. For example, they typically lack possibilities to relate different events of potential data disclosure to each other, or specify temporal or cardinal constraints on them, to allow for a more differentiated enforcement. Moreover, they rarely consider the flow of specified sensitive data items through the system in their decision process, but most often rely on the on-the-fly identification of sensitive data by using

This work was supported by the DFG under grant no. PR 1266/1-1 as part of the Priority Program SPP 1496, “RS^3”

pattern matching or statistical algorithms[6], [7], [8], [9], [10]. With this approach, DLP systems are usually not able to distinguish between individual data items and are thus not able to identify flows of sensitive data if they are encrypted, compressed, or otherwise obfuscated.

Solution. This paper introduces UC4Win[1], a data-driven usage control based data loss prevention solution for Microsoft Windows. UC4Win uses function call interposition techniques to monitor user application calls to the Windows API and match them against special security policies in order to modify or prevent calls that may lead to unwanted flows of confidential data. In order to detect derived data, data flows are tracked.

Contribution. UC4Win is, to our knowledge, the first DLP solution for Microsoft Windows that makes use of data-driven usage control to protect against inadvertent data leakage and that is *independent of concrete data representations*. By exhibiting the concepts of usage control and dynamic data flow tracking, UC4Win is capable of enforcing complex conditional and fine-grained security policies and is able to identify the propagation of individual data items through the system at any point in time. This enables a flexible and non-intrusive protection against inadvertent information disclosure, even in situations where typical content-identification based DLP solutions fail due to obfuscated, encrypted, or compressed data flows.

Organization. After a brief overview of related work in Section II, we define our scope by introducing a set of sample usage scenarios and discuss the resulting requirements in Section III. We clarify the conceptual foundations of this work in Section IV, followed by a discussion of design decisions and the actual design of our prototype in Section V. Finally, we present the results of a security and performance analysis in Section VI and conclude with a discussion in Section VII.

II. Related Work

Recent surveys [11], [3] show a broad spectrum of work that concerns the problem of detecting and preventing loss of sensitive data, often referred to as data loss prevention, information leak prevention, or extrusion prevention. The authors of the Pedigree system [12] propose a taint-based information flow control approach for the network-wide protection against the disclosure of sensitive data. This work focuses on associating taint labels to all processes and files of a system to enforce information-flow based policies on them. The main difference to our work is that Pedigree, even though it performs network-wide tracking of data, provides very limited means to specify and enforce policies more complex than comparing taint marks. The work of Petković et. al. [13] is probably closest to our approach, as they also consider information flows at the file system layer that are caused by function calls. The main difference to our work is that we (1) allow for the specification of temporal and cardinal obligations within our policies and (2) allow representation-independent enforcement and are not limited to the file system level. There are many different commercial data loss prevention solutions, ranging from independent ones focusing on the protection of one particular data state (endpoint, network, storage, etc.) [10] to more sophisticated solutions that integrate protection mechanisms for data-at-rest, data-in-motion, and data-in-use [6], [7], [8], [9]. These solutions mostly focus on detecting sensitive data and reporting events of its potential disclosure, while putting less effort into preventing such events in the first place. According to [11], one of the reasons for that is their missing support for correlating different monitored events, or more general, limited ability to define and enforce complex policies. Our work differs from such commercial solutions as our emphasis is on preventing misuse and disclosure of sensitive data in the first place. We are able to do so by monitoring distinct flows of data items and by correlating different events (in form of temporal, propositional, or cardinal relationships between them) in order to identify and prevent potential data loss.

In the area of usage control, enforcement mechanisms have been instantiated for a wide range of different systems and abstraction levels. This covers implementations for machine languages [14], for operating systems, such as the OpenBSD [15] or the Android operating system [16], [17], for window managers like the X11 windowing system [18], in the context of DRM [19], and for several legacy applications, such as the Firefox web browser [20], the ThunderBird mail client [21], or the OpenOffice word processor [22]. In addition, a multi-layer usage control architecture has been proposed, to combine these solutions in a way that allows data-centric enforcement of usage control requirements by the combination of enforcement mechanisms at multiple layers of abstraction [23].

Our work differs from these instantiations in that we are able to enforce usage control policies at different layers of data abstraction by monitoring and controlling events at only one layer (the Windows API). In contrast to other solutions that rely on combined event processing from monitors at multiple layers of abstraction, we can avoid several technical enforcement problems, e.g. induced by the necessary monitor synchronization. This reduces communication interfaces, which leads to a smaller attack surface, performance improvements, and increased reliability due to reduced system complexity.

Finally, the main difference between our work and the kernel hook based work of Wang et. al. [24] is that they handle simple, binary DRM-related constraints on kernel function calls, while we employ a formal data-driven usage control model to perform representation-independent enforcement of data usage control policies with complex temporal or cardinal constraints.

III. Use Case and Requirements

To understand the scope and goals of UC4Win and to point out the benefits of employing data-driven usage control techniques to prevent inadvertent data leakage, we

[1]A demo video can be found at:
http://www22.in.tum.de/research/distributed-usage-control/

present two sample scenarios that demonstrate the scope of UC4Win. These sample scenarios act as a basis for the user requirements and architectural constraints of UC4Win and were used to align its development process with a real-world context.

A. Scenarios

The scenarios take place in the fictional bank institute "InsecBank". At InsecBank the majority of accounting clerks work in open-plan offices to offer services to the customers. The open-plan offices are open for all customers during business hours. As the business model of InsecBank particularly depends on their reputation, a disclosure of sensitive data would mean a severe loss of reputation, as well as financial losses due to expensive law suits or criminal use of this data. Knowing that inadvertent data disclosure from insiders is one of the biggest information security threats [1], InsecBank decided to employ a usage control system as data loss prevention solution to protect themselves against this kind of threat.

1) Scenario 1: The accounting clerks in the open-plan offices use shared printers, which may be outside their actual line of sight and thus also accessible by customers during business hours. Thus, print-outs that contain sensitive data may be seen or stolen by visitors, as it might happen that clerks do not directly collect their print-outs. To prevent unintended leakage of confidential data via printers, InsecBank may want to enforce usage control policies such as "Customer records in *business_db.xml* must not be printed on shared printers".

2) Scenario 2: Although most of the business processes of InsecBank are supported by a special access-controlled groupware, some tasks still have to be performed outside the groupware environment. At the point where confidential data leaves the controlled groupware environment, it is not protected any longer by security mechanisms. This means that sensitive data may leak via different uncontrolled channels due to file copies, screenshots, or printouts. In order to cope with this kind of data leakage problems, InsecBank wants to enforce policies like "All attempts to copy and paste confidential data from the groupware system to non-trusted application should be blocked", "Screenshots must not be taken whenever they contain confidential data", or "Sensitive data may not be copied to non-trusted external devices."

B. Requirements

Corresponding to our bank setting with employees in different roles (e.g. clerk, manager, system administrator) we distinguish between two different types of users. The majority of users are non-privileged ones without administrator permissions and only restricted possibilities to modify their systems (e.g. install/uninstall software, enable/disable system services, etc.). The other group of users, referred to as privileged ones, has administrator permissions and is thus able to arbitrarily modify the systems. This distinction is necessary, because we do not want non-privileged users to modify the state of UC4Win and only allow managers or administrators to define and deploy usage control policies. This decreases the possibilities for non-privileged users to circumvent the usage control mechanisms.

To allow for a comprehensive and non-intrusive protection against information leakage related insider threats, we want UC4Win to be able to control all system events that have the potential to cause the disclosure of confidential data. As we do not want security policies to be restricted to only specify constraints on specific representations of data (e.g., one specific file), we in addition demand the enforcement to be done independent of the representation of the to-be protected data (e.g., also for any copy of the file, or windows that show parts of that file). Based on the proposed scenarios and the targeted application area of UC4Win, we thus define this set of relevant system events as all events that correspond to: (i) modifications of the file system (e.g. creating, deleting, reading, or writing files); (ii) modifications of the user interface (e.g. displaying data on the screen, taking screenshots); and (iii) communication with peripheral devices (e.g. printing). The focus of UC4Win is on the enforcement of already defined security policies on the usage of sensitive data. Therefore, the identification of sensitive data and the generation of respective security policies is outside this paper's scope.

IV. Background

A. Usage Control

Usage control is a generalization of the concept of access control to what happens to data after access to it has been granted [25]. This is done by enforcing provisions and obligations on its usage, typically specified within usage control policies that state what must and what must not happen to the data upon future usage [26]. Examples for such policies are "delete data after 30 days", "reduce quality of video upon distribution", or "notify owner upon access". The compliance with these policies is then either enforced in a detective way, which means that violations are only detected (and potentially reported) but not prohibited, or in a preventive way, where violations of a policy are prohibited in the first place.

These data items are represented at different layers of abstraction within a system. The data contained inside a picture for example, may be at the same time represented at the file system level (e.g. within picture.jpg), as part of a window (e.g. within the main window of a picture editor), and within a particular area of the memory of the corresponding process (e.g. as part of the editor's heap). We often want to enforce usage control requirements for all representations of a data item; thus we do not want the policy definition to be bound to particular representations.

To enforce usage control requirements on a real system, they have to be first translated into a more formal and machine-readable format. With the Obligation Specification Language (OSL) [27], a general-purpose usage control policy language, one can specify obligations and provisions at the level of events. Such policies consist of two parts. The event declaration part defines all events that can potentially occur within the context of a particular system. The second part of such a policy contains one or more preventive or detective usage control mechanisms. These mechanisms are defined as event-condition-action (ECA) rules and specify obligations and provisions, based on the previously defined events. Listing 1 depicts the abstract syntax of these ECA rules [17].

Listing 1: Abstract OSL Policy Syntax

```
Policy::= eventDeclaration,
          {PreventiveMechanism | DetectiveMechanism}+;

PreventiveMechanism ::= Event, Condition,
                        AuthorizationAction,
                        {ExecuteAction};
DetectiveMechanism ::=  Event, Condition,
                        {ExecuteAction};

Event ::= actionName, {paramMatch};
paramMatch ::= paramName, value, [type];
type ::= containerUsage | dataUsage;
Condition ::= PL | TL;

PL ::=  true | false | xPathEval(𝕊) | eventMatch |
not(PL) | and(PL,PL) | or(PL,PL) | implies(PL,PL);

TL ::=  PL | not(TL) | and(TL,TL) | or (TL,TL)
| implies(TL,TL) | since(TL,PL) | always(TL)
| before(ℕ,TL) | during(ℕ,TL) | within(ℕ,TL)
| replim(ℕ,ℕ,ℕ,PL) | repmax(ℕ,PL) | repsince(ℕ,PL,TL);

AuthorizationAction ::= allow | inhibit | {Modifier};
Modifier ::= paramName, value;
ExecuteAction ::= notify | execute;
```

The formal semantics of the OSL language and the question on how to derive implementation-level policies are outside the scope of this paper. At this point we thus only discuss the essential ideas. For a detailed discussion please refer to the available literature [27], [28], [29].

In order to verify the compliance of a particular event with a usage control policy, the event name and its parameters are first matched against the respective *Event* part of the policy. If the event matches, the *Condition* part of the policy is evaluated. The *Condition* part consists of propositional or past temporal logic formulas that may contain nested logical, cardinal, temporal, or XPath expressions. The always(a) operator of OSL corresponds to the always operator of LTL, the since(a,b) evaluates to true if b has always been true since a was true the first time, and the before(n,a) evaluates to true if a was true n time steps before. The OSL operators during(n,a) and within(n,a) are generalizations of the LTL next operator, where during(n,a) is true if a was always true within the last n time steps, and within(n,a) if a was at least one time true within the last n steps. The OSL cardinal operators can entirely be modeled within LTL, nevertheless they are explicitly defined to reduce complexity. The replim(l,m,n,a) operator evaluates to true if a was true at least l times and at most m times within the last n time steps, while repmax(n,a) is true if a was true at most n times in the past. Finally repsince(n,a,b) is true, if a has been true at most n times since b evaluated to true for the first time. If the *Condition* part evaluates to true, the event execution is, according to the *AuthorizationAction* part of the policy, either allowed, inhibited, or the event has to be modified prior to its execution. Also additional actions may be executed, if the *ExecuteAction* part of the policy specifies it. With the full expressiveness of linear temporal logic together with macros for cardinalities and xPath expressions, OSL allows to specify fine-grained usage control policies at the level of events. To increase the expressiveness even more and to allow to specify policies on events whose parameters are related to data instead of concrete representations, the type flag in the policy can be set to dataUsage instead of containerUsage. This marks a specific parameter to represent a data item and thus to be considered in the data flow tracking.

B. Dynamic Data Flow Tracking

The representations of data in our context are runtime-dependent due to continuous data flows through the system, triggered by the execution of events. To formally model these data flows we employed a generic transition system based dynamic data flow model, as introduced in [18], [15]. Its main idea is the representation of data flows between different data containers within a system, modeled by the tuple $(C, D, P, A, F, R, \Sigma, i)$ and initiated by the execution of system events. C refers to the set of all containers in the system that may contain data (e.g. files, processes, windows, or the clipboard) and can be seen as sources and sinks of all data flows. The set D contains all abstract to-be protected data items (e.g. customer records, a song, or a picture), independent of their concrete representation. The set of principals P denotes all active entities of the system that actually trigger the execution of events (in our context these are the user processes). The set of actions A models all events that potentially cause a flow of data (e.g. reading or writing a file, copying data to the clipboard, or taking a screenshot). The naming set F consists of identifiers that uniquely address the containers of set C (e.g. file/window handles, or process IDs). A system state $\sigma \in \Sigma$ is defined by three mappings: the storage mapping s of type $(C \rightarrow 2^D)$ that models the containment relation between data and containers; the alias mapping l of type $(C \rightarrow 2^C)$ that models the relationship between different containers; and the naming relation f of type $(P \times F \rightarrow C)$ that uniquely identifies the containers. Therefore the set of all possible states of the system is defined by $\Sigma := (C \rightarrow 2^D) \times (C \rightarrow 2^C) \times (P \times F \rightarrow C)$, where the initial state i consists of three empty mappings. Finally the transition relation R contains all state transitions, defined by $R \subseteq \Sigma \times P \times A \times \Sigma$, that are initiated by actions, triggered by principals.

The semantics of this system are modeled via traces, where a trace maps abstract points in time to a set of states. For a more detailed discussion of the syntax and semantics of the employed model please refer to [18], [15].

V. Design

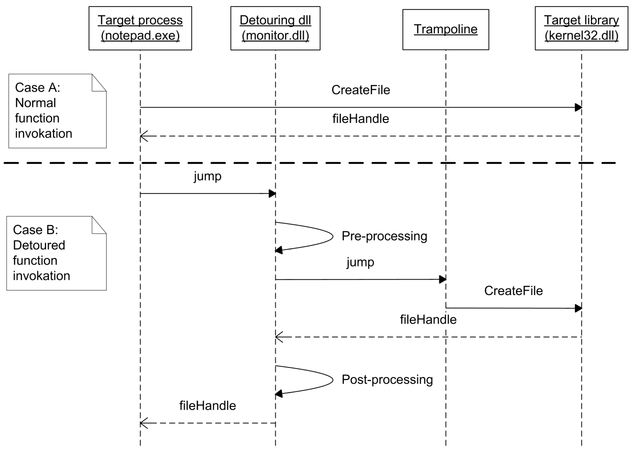
A. Options for Monitoring

All NT kernel based Windows systems consist of a user mode and a kernel mode part. In principle all processes whose execution is triggered by non-privileged users are executed in the user mode part and have to make use of the Windows API if they want to invoke low-level kernel functionality. This for example includes reading or writing to files, creating or manipulating windows, or invoking network functionality. As a consequence, all user related actions could be intercepted at both system layers, which enables a wide range of possibilities to deploy our event monitors. Although a deployment within the kernel mode layer is thinkable, this option suffers from several drawbacks. First of all, the Windows NT kernel itself and the corresponding Native API are only sparsely documented, which complicates modifications like the integration of a usage control component. A more compelling argument is that we might lose the ability to enforce more high-level policies that, for example, specify constraints on user interface events. The reason for this is that the Native API's purpose is to provide low-level functionality and thus, in contrast to the Windows API, lacks high-level functions for tasks like user interface manipulation.

As our focus is on the protection against inadvertent disclosure by normal system users, which boils down to preventing user processes from invoking functions that might lead to unwanted data flows, the deployment at the user mode layer is the more reasonable choice. Consequently, the interception of user application calls to the Windows API was the natural way of archiving this goal. There are different ways to realize the interception of function calls from to the Windows API. One of the simpler ones is the modification of source code or binaries of all to-be monitored programs to allow to intercept and control all relevant function calls. We did not choose this solution, as it would require us to modify all programs in the targeted context which might not all be a-priori known and their modification in many cases problematic due to missing source code. The main drawback of this solution is its necessary restriction to a fixed set of user applications, which is unacceptable for a generic DLP solution.

The second option is the direct replacement of the Windows API library files with custom ones that include our monitoring functionality. The replacement libraries could then conditionally forward the calls to the original Windows API library files. While this solution does not limit the enforcement to a particular set of applications, it suffers from limitations like conflicts with integrity protection mechanisms (e.g. the Windows File Protection).

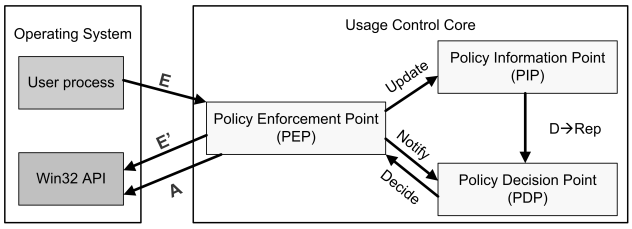
The third option is the use of function call interposition techniques. They are based on runtime manipulations of calls to external functions within the memory of the target process to reroute the calls to a custom detouring library. This library can then perform arbitrary computations and thus implement the monitoring functionality before it invokes the original function call destination and returns to the calling process. Figure 1 depicts this process for the example of a CreateFile Windows API function call. The upper half of the picture illustrates the respective function call without and the lower half with function call interposition. The main benefit of this technique is that it neither requires the replacement of system libraries nor


Target process (notepad.exe)
Detouring dll (monitor.dll)
Trampoline
Target library (kernel32.dll)
Case A: Normal function invokation
CreateFile
fileHandle
jump
Case B: Detoured function invokation
Pre-processing
jump
CreateFile
fileHandle
Post-processing
fileHandle
Figure 1: Function Call Interposition

any static modifications of the involved applications. Also, the interception can be completely done at runtime which allows an efficient and flexible monitoring by targeting the interception to the relevant processes and functions. Based on these considerations we chose function call interposition for the event monitoring of UC4Win.

B. Design

UC4Win is an instantiation of a generic representation-independent usage control architecture [28] and consists of three main components, depicted in Figure 2: the Policy Enforcement Point (PEP), responsible of intercepting events and enforcing usage restrictions on them; the Policy Decision Point (PDP), in charge of deciding about the admission of the intercepted events; and the Policy Information Point (PIP) that implements the data flow model and maintains the connection between data and its representations within the system.


Operating System
User process
Win32 API
Usage Control Core
Policy Enforcement Point (PEP)
Policy Information Point (PIP)
Policy Decision Point (PDP)
E
E'
A
Update
Notify
Decide
D→Rep
Figure 2: Conceptual View

Starting with the interception of a user-mode application call to one of the relevant Windows API functions (*E*), the PEP temporarily blocks its execution and forwards the event to the PDP, including the parameter values of the intercepted function call (*Notify*). The PDP then decides whether to allow, inhibit, or modify the execution (*Decide*), by matching it against the corresponding usage control policies. This evaluation step may require the PDP to communicate with the PIP, whenever one of the deployed policies makes restrictions on the usage of all representations of a data item (e.g. all file copies or windows containing data from *business_db.xml*) rather than on only one particular representation of it (e.g. only *business_db.xml* itself). In these cases, the PDP queries the PIP for all representations of that data item ($D \to Rep$) to assess if the intended function call violates usage constraints on one of these representations. Based on the PDP's decision, the PEP then either unblocks and continues the function call execution (*Allow: E*), modifies some of its parameter values (*Modify:* $E \to E'$), or completely stops the further execution (*Inhibit: E*). Optionally, if a policy demands it, the PEP may execute additional actions (A) (e.g. a user notification). In cases where the function call was executed, the PEP notifies the PIP about that event (*Update*), which then in turn makes the corresponding changes to the internal data flow model, based on the data flow semantics of the respective function call.

C. Instantiation of the Data Flow Model

To track the flow of data induced by the execution of user-mode process calls to the Windows API, we adopted the dynamic data flow model introduced in Section IV-B to our Microsoft Windows specific context. Due to our focus on user-mode process calls to the Windows API, we instantiate the set of principals P as the set of active user-mode processes, identified by their unique process ID $P := PID$. The set of containers C is instantiated as the set of all windows, processes, files, output devices, and the clipboard that may contain data. Therefore it is defined as $C := C_{Windows} \cup C_{Files} \cup C_{Devices} \cup C_{Clipboard} \cup m_p$, where m_p denotes the process memory container of process p. Also we instantiate the set of names F as all unique identifies for these containers, namely the set of window, device and file handles, as well as (absolute) file names, with $F := F_{wHandle} \cup F_{fName} \cup F_{fHandle} \cup F_{dHandle} \cup F_{dName}$. We define the set of actions A as all Windows API functions that potentially induce a flow of data from or to the identified containers. For the sake of simplicity, neither the set of containers C nor the set of actions A is complete. We hence restrict the modeling to the following subset. In general, it is a daunting task to model the complete Windows API with thousands of documented functions. In practice, it is also not necessary to do so to cover the common cases of inadvertent information disclosure. The semantics of these actions, expressing the way their execution modifies the state of the data flow model, are defined as follows. For the sake of simplicity, the signatures of the actions do not exactly correspond to the real Windows API function and are either modified or used in an abstracted manner, whenever parameters have no significance in terms of the induced data flows.

To model the information state changes, we need some additional notation to update the corresponding functions. For any mapping $m : S \to T$ and variable $x \in X \subseteq S$, we define $m[x \leftarrow expr]_{x \in X} = m'$ with $m' : S \to T$ such that $m'(y) = expr$ if $y \in X$ and $m'(y) = m(y)$ otherwise. We apply function composition for multiple updates on disjoint sets, with simultaneous and atomic replacements (the semicolon is syntactic sugar): $m[x_1 \leftarrow expr_{x_1}; \ldots; x_n \leftarrow expr_{x_n}]_{x_1 \in X_1, \ldots, x_n \in X_n} = m[x_n \leftarrow expr_{x_n}]_{x_n \in X_n} \circ \ldots \circ m[x_1 \leftarrow expr_{x_1}]_{x_1 \in X_1}$.

l^* will denote the transitive reflexive closure of alias function l.

CreateFile: The CreateFile Windows API function is used to open a file, specified via its absolute file name, and returns a handle to this file upon execution. Therefore it has to be invoked prior to any write or read operation on that file.

$$\begin{aligned}
&\forall s \in \left[C \to 2^D\right], \forall l \in \left[C \to 2^C\right], \forall f \in [P \times F \to C],\\
&\forall p \in P, \forall fn \in F_{fName}, \forall fh \in F_{fHandle}:\\
&(\,(s, l, f)\,, p, CreateFile\,(fh, fn)\,,\\
&(s, l, f\,[(p, fh) \leftarrow f(p, fn)]\,)) \in R
\end{aligned}$$

ReadFile: The ReadFile function is used to read data from a file, identified via the passed file handle. Upon execution, there is a data flow from the file to the memory of the calling process. Therefore we model a data flow from the file container to the process memory m_p and all referenced containers (e.g. windows that belong to that process), indexed by the transitive reflexive closure l^* of the alias function of m_p. As we in general cannot know where the sensitive data within the file is located, we have to do a coarse-grained estimation, assuming that all data flows into the process memory.

$$\begin{aligned}
&\forall s \in \left[C \to 2^D\right], \forall l \in \left[C \to 2^C\right], \forall f \in [P \times F \to C],\\
&\forall p \in P, \forall fh \in F_{fHandle}:\\
&(\,(s, l, f)\,, p, ReadFile\,(fh)\,,\\
&(s[t \leftarrow s(f(p, fh)) \cup s(t)]_{t \in l^*(m_p)}, l, f)) \in R
\end{aligned}$$

WriteFile: The WriteFile function is used to write data to a file, identified via the passed file handle. Again we have to conservatively model the data flows, as we cannot asses which parts of the process memory are written to the file.

$$\begin{aligned}
&\forall s \in \left[C \to 2^D\right], \forall l \in \left[C \to 2^C\right], \forall f \in [P \times F \to C],\\
&\forall p \in P, \forall fh \in F_{fHandle}:\\
&(\,(s, l, f)\,, p, WriteFile\,(fh)\,,\\
&(s[f(p, fn) \leftarrow s(f(p, fn)) \cup s(m_p)], l, f)) \in R
\end{aligned}$$

CreateWindow: The CreateWindow function has to be invoked to create a new window to assemble the user interface of a process. As any change to the process memory may indirectly induce a flow of data to its windows, we model this flow by a directed alias relation between process memory m_p and all its windows. We have to do this over-approximation in the alias modeling, as we cannot deduce at the level of function calls to which particular window the data has flown. For the same reason we also have to assume that the complete process data has flown into a new window container upon creation.

$$\begin{aligned}&\forall s\in\left[C\rightarrow 2^{D}\right],\forall l\in\left[C\rightarrow 2^{C}\right],\forall f\in[P{\times}F\rightarrow C],\\&\forall p\in P,\forall wh\in F_{wHandle}:\\&(\,(s,l,f)\,,p,CreateWindow\,(wh)\,,\\&(s\left[f\,(p,wh)\leftarrow s(m_{p})\right],\\&l\,[m_p\leftarrow l\,(m_p)\cup f\,(p,wh)]\,,f)\in R\end{aligned}$$

TakeScreenshot: There exists a variety of functions to create a dump of all currently visible windows. To cover all of them we introduce the abstract TakeScreenshot function that models their common principle of transferring a dump of the content of all visible windows to the clipboard. Conceptually it is triggered whenever one of the various methods of taking a screenshot is executed, and results in a data flow from all visible windows, indexed by the set of window handles wh_{vis}, to the clipboard.

$$\begin{aligned}&\forall s\in\left[C\rightarrow 2^{D}\right],\forall l\in\left[C\rightarrow 2^{C}\right],\forall f\in[P{\times}F\rightarrow C],\\&\forall p\in P,\forall c\in C_{clipboard},\forall wh_{vis}\subseteq C_{windows}:\\&((s,l,f),p,TakeScreenshot(wh_{vis}),\\&(s[c\leftarrow\bigcup_{t\in wh_{vis}}s(f(p,t))],l,f)\in R\end{aligned}$$

SetClipboardData: The SetClipboardData function allows to transfer arbitrary data to the system clipboard. Thus, it leads to a flow of data from the calling process memory container m_p to the clipboard container $c\in C_{clipboard}$. We again have to take a coarse estimation, assuming that all data within a process container flows into the clipboard container. The clipboard only contains one data item per time, so its content gets replaced upon function call.

$$\begin{aligned}&\forall s\in\left[C\rightarrow 2^{D}\right],\forall l\in\left[C\rightarrow 2^{C}\right],\forall f\in[P{\times}F\rightarrow C],\\&\forall p\in P,\forall c\in C_{clipboard}:\\&(\,(s,l,f)\,,p,SetClipboardData\,(c)\,,\\&(s\left[c\leftarrow s(m_{p})\right],l,f)\in R\end{aligned}$$

GetClipboardData: The GetClipboardData function is used to fetch data from the system clipboard. It is modeled by the transfer of the complete clipboard content of $c\in C_{clipboard}$ to the calling process memory container and all its aliased containers $l^{*}(m_p)$.

$$\begin{aligned}&\forall s\in\left[C\rightarrow 2^{D}\right],\forall l\in\left[C\rightarrow 2^{C}\right],\forall f\in[P{\times}F\rightarrow C],\\&\forall p\in P,\forall c\in C_{clipboard}:\\&(\,(s,l,f)\,,p,GetClipboardData\,(c)\,,\\&(s\,[t\leftarrow s(t)\cup s(c)]_{t\in l^{*}(m_p)},l,f)\in R\end{aligned}$$

CreateDC: The CreateDC function is used to retrieve a handle to a graphical output device (e.g. a printer) and thus is invoked if a process starts a printing job. Therefore, whenever a call to the CreateDC function is made, we assume that all data within the memory container m_p of the calling process flows into the output device container, addressed by its unique device name.

$$\begin{aligned}&\forall s\in\left[C\rightarrow 2^{D}\right],\forall l\in\left[C\rightarrow 2^{C}\right],\forall f\in[P{\times}F\rightarrow C],\\&\forall p\in P,\forall dn\in F_{dName},\forall dc\in F_{dHandle}:\\&((s,l,f),p,CreateDC(dn,dh),\\&(s[f(p,dn)\leftarrow s(f(p,dn))\cup s(m_p)],\\&l,f[(p,dh)\leftarrow f(p,dn)]))\in R\end{aligned}$$

With this set of data flow semantics we are able to model the influences of all relevant Windows API function calls on the data flow system state. All functions within the scope of our system that we did not explicitly model here boil down to one or more of the introduced functions. The *recv()* and *send()* WinSock network functions for example have the same data flow semantics as the ReadFile and the WriteFile functions.

D. Policy Enforcement

Coming back to our initial setting and scenarios, we now demonstrate how UC4Win enforces usage control policies at runtime. Listing 2 shows the (slightly simplified) XML representation of the ECA rule that models the usage control policy "Customer records in *business_db.xml* must not be printed on shared printers" from our first scenario. As we want this usage control policy to apply to all representations of the data in *business_db.xml*, we need to ensure that all data flows that origin from *business_db.xml* are tracked. This is ensured by using the *type="dataUsage"* flag of the *FileName* parameter within the trigger part of the mechanism. The PDP recognizes this parameter to represent the initial representation of a data item. Upon policy deployment, the PIP is notified to track any data flow from this initial container. At runtime, if an event is to be executed on a specific container, the PIP will be queried if this container contains the data inititally represented in file *business_db.xml*.

Listing 2: Example Policy

```
<preventiveMechanism name="BlockLeak">
  <trigger action="CreateDC" index="ALL" isTry="true">
    <a:paramMatch name="lpszDriver"
    value="winspool"/>
    <a:paramMatch name="lpszDevice"
    value="SharedPrinter"/>
    <a:paramMatch name="FileName"
    value="business_db.xml" type="dataUsage"/>
  </trigger>
  <condition> <True/> </condition>
  <authorizationAction>
    <inhibit/>
  </authorizationAction>
  <action name="notify">
    <a:parameter name="title" value="UC Alert!" />
    <a:parameter name="msg"
    value="Printing confidential data not allowed." />
  </action>
</preventiveMechanism>
```

Figure 3 depicts the basic steps of this data initialization and policy enforcement process. If for example a wordpad process wants to print data from *business_db.xml*, it invokes the CreateDC function prior to starting a new printing job. This invocation is intercepted and blocked by the PEP, which forwards it to the PDP. The PDP then successfully matches the intercepted event against the mechanism, because the file name parameter references the *business_db.xml* file and the printer name parameter indicates a shared printer as target device. As the *Condition* part of the mechanism evaluates to true, the PDP returns with an *Inhibit* response to the PEP (as specified in the *AuthorizationAction* part of the policy) which then blocks the printing attempt and triggers a user notification (as specified in the *Action* part). If, after that, data is for example copied via the clipboard from the wordpad process to another, say notepad process, the corresponding data flow (originating from *business_db.xml*) is recognized. In case the notepad process then tries to print this data, the policy also matches on the corresponding event invocation so that the printing is inhibited as well.

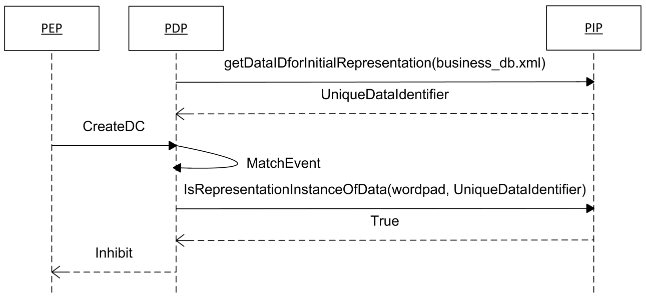

PEP
PDP
PIP
getDataIDforInitialRepresentation(business_db.xml)
UniqueDataIdentifier
CreateDC
MatchEvent
IsRepresentationInstanceOfData(wordpad, UniqueDataIdentifier)
True
Inhibit

Figure 3: Policy Enforcement Workflow

With this policy we thus effectively prevented user attempts to print data from *business_db.xml* on shared printers. Due to space limitations, we did not make use of the full expressiveness of the used policy language in this example mechanism. For instance, we did not specify any temporal or cardinal constraints within the *Condition* parts of the policies. However, by doing so, we could easily reduce the strictness of our policies by, say, allowing a maximum of three print-outs of data from *business_db.xml*. This might make sense in situations where we want to lessen the level of security in favor of a reduced interference with user workflows. We see this flexibility in the policy definition and enforcement as one of the main benefits and contributions of UC4Win in comparison to other DLP solutions. For more complex policy examples please refer to [17]. At this point please note that the expressiveness of the proposed policy language also allows us to specify much more differentiated reactions on policy violations. Instead of simply denying the execution of such events we for example could notify the user of the potential security breach to increase his awareness while at the same time minimizing interference with workflows.

VI. Evaluation

A. Security

In order to allow an effective protection against information disclosure, a DLP solution must not be easily modified, disabled, or circumvented. The purpose of this security analysis is therefore to identify potential vulnerabilities from an attacker's point of view and assess their impact on the overall security of UC4Win. As our intention is the protection against inadvertent data disclosure in a business context, we focus the analysis on potential ways of circumventing or disabling our system that could be exploited by normal users without malicious intention. Our attacker model is thus that of a normal employee that unintentionally discloses sensitive information, due to careless behavior and non-adherence to security guidelines. Additional assumptions of our security analysis are: i) the attacker does not have administrator privileges; ii) the underlying Windows operating system is free of vulnerabilities; iii) the policy storage is tamper-proof and cannot be modified in any way by non-privileged users. The first and most fundamental assumption i) is in our opinion justified, as it is not typical for normal employees in a business context to be given administrator privileges. We are aware that assumption ii) is very strong as Windows can not be considered completely fault free, still it has to be taken as faults at this level always lead to an insecure system. Assumption iii) can be easily ensured with standard Windows access control mechanisms.

Attacks on the Policy Enforcement: The most obvious way of circumventing the enforcement of UC4Win is to interfere with its function call interposition mechanisms. We only intercept a limited subset of the complete Windows API, therefore an easy way to do so is to use functions that are not intercepted by UC4Win. As the chosen selection of intercepted functions covers the Windows API call behavior of common Windows programs (e.g. Microsoft Office), we do not see this as a high risk. We could cope with this threat by explicitly specifying all allowed processes with a known set of potential API calls (via a respective usage control policy). The second option to circumvent the enforcement of UC4Win is the manipulation of the interception mechanism itself. Possibilities include the manipulation of the memory of the monitored processes to remove the injected function re-routing mechanisms, or to launch a Man-in-the-Middle attack on the communication between the monitored processes and the PEP. Both attacks are not feasible under the taken assumptions as the manipulation of the process memory requires elevated privileges and the communication between the different UC4Win components is secured via encrypted IPC channels.

Attacks on the Policy Evaluation: UC4Win in theory could also be circumvented by manipulating the used security policies. Because assumption iii) demands that the physical data storage (the UC4Win policy folder in the file system) is protected against manipulation by normal users, the naive modification of the persistent policies can be ruled out. By deploying malicious policies to the PDP, an attacker could still circumvent the data protection mechanisms of UC4Win. As the policy management functionality of UC4Win is only accessible with administrator permissions, this attack is also not feasible within our context. The manipulation of the policies at runtime within the PDP's memory is again not possible due to the inability to modify the process memory without elevated privileges. Finally, the data item initialization phase between PDP and PIP could be manipulated in order to prevent the effective policy evaluation. Again, we assume the attack to not be feasible due to the used encrypted IPC communication.

Attacks on the Availability: The most obvious way to attack the availability of UC4Win to disable its protection is the usage of Denial-of-Service attacks. Flooding its components with a high load of events by artificially generating Windows API calls can significantly slow down the UC4Win process itself. In extreme cases this would cause the operating system to kill the UC4Win process if it exceeds the maximum response time. To cope with this problem, a system running UC4Win could be configured in a way that it reboots whenever a UC4Win process exits unexpectedly. While this countermeasure reduces the threat of sensitive data disclosure, it is not very practicable due to its negative effects on the system stability.

B. *Performance*

As a significant performance overhead of a DLP solution induces a bad user experience, we analyzed the performance of UC4Win, measuring its behavior in terms of relative computation time overhead. This measurements were done under different configurations: i) without UC4Win; ii) UC4Win without deployed policies; iii) UC4Win with deployed policies. To get an idea of the performance under realistic conditions and in extreme situation of stress, we conducted two distinct types of performance tests. For the first test type we specified a security policy that was triggered whenever data was read from or written to a specific file. To simulate a realistic user behavior we used a macro script that opened this file, added some characters, and saved it again. For this test type, the time for the execution of the complete macro was measured. The second test type was a typical stress test where we step-wise increased the pressure on UC4Win by artificially generating CreateFile events in 1000 function call packages. Here, the time for the execution of each function call package was measured. To minimize environmental influences on the measurements, we conducted all performance tests within a virtual machine installation of Windows 7 SP1 x86 with 4 GByte of RAM and a 2 GHz single-core CPU, using the same virtual machine snapshot for all tests and configurations. All tests were executed 100 times and their arithmetic mean used for the further calculations. Table I depicts the results of both test types, showing the relative computation time overhead of the tests with respect to the non-UC configuration i). The small measured overhead of the macro test can be explained by the fact that the amount of time that was spent for the monitored function calls themselves, compared to the overall execution time of the macro, was relatively small. As the stress test almost entirely consists of intercepted events, the measured performance overhead is correspondingly much higher. The big step between the measurements of configuration i) and ii), compared to the small step between ii) and iii) indicates that the main overhead stems from the function call interposition mechanism.

Table I: Performance Evaluation results

	i) NoUC	ii) UC - Pol	iii) UC + Pol
Macro	100.00%	117.32%	122.12
Stress	100,00%	525,97%	579,05

While the overhead under stress conditions may appear prohibitive, the overhead in the "normal" use case can be considered acceptable, because it is barely noticeable. In other words, the overhead cannot be considered a show stopper for our DLP solution.

VII. Discussion and Conclusion

In this paper we introduced UC4Win, a data loss prevention solution for Microsoft Windows. Because this solution instantiates a more general usage control framework, it exhibits the expressiveness of data-driven usage control to allow for a fine-grained enforcement of security policies against data loss. Technically, our solution works on the basis of Windows API function calls. By using function call interposition we are able to intercept these calls, evaluate their DLP policy compliance, and block or modify them upon detected policy violations. To allow a comprehensive representation-independent data leakage prevention we incorporated a dynamic data flow model to track flows of sensitive data through the system. In contrast to other DLP solutions that mainly work on the identification of sensitive data instead of tracking their propagation, we are able to identify individual data items at any point in time, regardless of their current representation. This enables us to precisely specify and enforce policies on individual data items, which allows a very fine-grained data leakage prevention. Apart from data leakage, caused by internals, UC4Win also provides means to detect and prevent data disclosure by malware, as both happen through the same intercepted API calls.

While the concepts behind UC4Win in our opinion effectively prevent data disclosure at the level of Windows API calls, our prototype is still work-in-progress and thus has certain limitations. First of all its data loss prevention effectiveness almost entirely depends on the quality of the used security policies and the precise definition of the to-be protected data. This is in particular crucial as UC4Win itself is, in contrast to many other DLP solutions, not able to automatically identify sensitive data. In addition to that, the definition of such security policies is not a trivial task, as the used temporal logic based policy language is far from being intuitively usable by non-experts. Nevertheless, there have recently been promising advances in the research on easing this definition task [29]. Besides these policy based issues, the necessary over-approximations in the dynamic data flow tracking in some cases lead to label creep problems in a way that non-sensitive data is mistakenly considered sensitive. This problem could at least partially be solved by the employment of heuristics and declassification strategies, which is matter of current research. The conducted security analysis indicates, that although UC4Win is not easy to circumvent under the taken assumptions, it cannot be considered secure in a more general case. This means that UC4Win might not be able to withstand sophisticated attacks, and thus may not be suitable to defend against data disclosure by malicious attackers such as hackers.

REFERENCES

[1] "Insider risk management: A framework approach to internal security." www.rsa.com/document.asp?doc_id=10388, last accessed April 2012.

[2] M. McCormick, "Data theft: A prototypical insider threat," in *Insider Attack and Cyber Security*, vol. 39 of *Advances in Information Security*, pp. 53–68, Springer US, 2008.

[3] M. B. Salem, S. Hershkop, and S. J. Stolfo, "A survey of insider attack detection research," in *Insider Attack and Cyber Security*, vol. 39 of *Advances in Information Security*, pp. 69–90, Springer US, 2008.

[4] S. Liu and R. Kuhn, "Data loss prevention," *IT Professional*, vol. 12, no. 2, pp. 10–13, 2010.

[5] H. Balinsky, D. Perez, and S. Simske, "System call interception framework for data leak prevention," in *Proc. of EDOC'11*, pp. 139 –148, 29 2011-sept. 2 2011.

[6] "Mcafee total protection for data loss prevention." http://www.mcafee.com/us/resources/solution-briefs/sb-total-protection-for-dlp.pdf, last accessed May 2012.

[7] "Sophos data protection suite." http://www.sophos.com/en-us/medialibrary/PDFs/factsheets/sophosdataprotectionsuitedsna.pdf, last accessed May 2012.

[8] "Rsa data loss prevention suite." http://www.rsa.com/products/DLP/sb/9104_DLPST_SB_0311.pdf, last accessed May 2012.

[9] "Mydlp." http://www.mydlp.com/, last accessed May 2012.

[10] "Symantec data loss prevention for endpoint." http://www.symantec.com/content/en/us/enterprise/fact_sheets/b-dlp_for_endpoint_DS_21189146.en-us.pdf, last accessed May 2012.

[11] A. Shabtai, Y. Elovici, L. Rokach, A. Shabtai, Y. Elovici, and L. Rokach, "A taxonomy of data leakage prevention solutions," in *A Survey of Data Leakage Detection and Prevention Solutions*, SpringerBriefs in Computer Science, pp. 11–15, Springer US, 2012.

[12] M. B. T. Yogesh Mundada, Anirudh Ramachandran and N. Feamster, "Practical dataleak prevention for legacy applications in enterprise networks," tech. rep., Georgia Institute of Technology, 2011.

[13] M. Petkovic, M. Popovic, I. Basicevic, and D. Saric, "A host based method for data leak protection by tracking sensitive data flow," in *Proc. of ECBS'12*, pp. 267–274, april 2012.

[14] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Orm, S. Okasaka, N. Narula, N. Fullagar, and G. Inc, "Native client: A sandbox for portable, untrusted x86 native code," in *Proc. of SP'07*, 2007.

[15] M. Harvan and A. Pretschner, "State-based Usage Control Enforcement with Data Flow Tracking using System Call Interposition," in *Proc. of NSS'09*, pp. 373–380, 2009.

[16] G. Bai, L. Gu, T. Feng, Y. Guo, and X. Chen, "Context-aware usage control for android," in *SecureComm'10*, pp. 326–343, 2010.

[17] D. Feth and A. Pretschner, "Flexible data-driven security for android," in *To appear in Proc. of SERE'12*, 2012.

[18] A. Pretschner, M. Büchler, M. Harvan, C. Schaefer, and T. Walter, "Usage control enforcement with data flow tracking for x11," in *Proc. of STM'09*, pp. 124–137, 2009.

[19] "Adobe livecycle rights management es." http://www.adobe.com/products/livecycle/rightsmanagement/indepth.html, last accessed April 2012.

[20] P. Kumari, A. Pretschner, J. Peschla, and J.-M. Kuhn, "Distributed data usage control for web applications: a social network implementation," in *Proc. of CODASPY'11*, pp. 85–96, 2011.

[21] M. Lörscher, "Data usage control for the thunderbird mail client," Master's thesis, University of Kaiserslautern, 2012.

[22] C. Schaefer, T. Walter, A. Pretschner, and M. Harvan, "Usage control policy enforcement in OpenOffice.org and information flow," in *Proc. of ISSA'09*, p. 393, 2009.

[23] E. Lovat and A. Pretschner, "Data-centric multi-layer usage control enforcement: A social network example," in *Proc. of SACMAT'11*, pp. 151–152, 2011.

[24] Y. Wang, Y. Shen, and J. Pan, "Usage control based on windows kernel hook," in *Proc. of ICIMT'09*, pp. 264 – 267, 2009.

[25] J. Park and R. Sandhu, "The UCON ABC usage control model," *ACM Trans. Inf. Syst. Secur.*, vol. 7, no. 1, pp. 128–174, 2004.

[26] A. Pretschner, M. Hilty, D. Basin, C. Schaefer, and T. Walter, "Mechanisms for Usage Control," in *Proc. of ASIACCS'08*, pp. 240–245, 2008.

[27] M. Hilty, A. Pretschner, D. Basin, C. Schaefer, and T. Walter, "A policy language for distributed usage control," in *Computer Security ESORICS 2007*, vol. 4734 of *Lecture Notes in Computer Science*, pp. 531–546, Springer Berlin / Heidelberg, 2007.

[28] A. Pretschner, E. Lovat, and M. Büchler, "Representation-independent data usage control," in *Proc. of STM'11*, 2011.

[29] P. Kumari and A. Pretschner, "Deriving implementation-level policies for usage control enforcement," in *Proceedings of the second ACM conference on Data and Application Security and Privacy*, Proc. of CODASPY'12, (New York, NY, USA), pp. 83–94, ACM, 2012.