# A Software-Hardware Architecture for Self-Protecting Data

Yu-Yuan Chen
Department of Electrical Engineering
Princeton University
Princeton, NJ, USA
yctwo@princeton.edu

Pramod A. Jamkhedkar
Department of Electrical Engineering
Princeton University
Princeton, NJ, USA
pjamkhed@princeton.edu

Ruby B. Lee
Department of Electrical Engineering
Princeton University
Princeton, NJ, USA
rblee@princeton.edu

## ABSTRACT

We propose a software-hardware architecture, DataSafe, that realizes the concept of self-protecting data: data that is protected by a given policy whenever it is accessed by any application – including unvetted third-party applications. Our architecture provides dynamic instantiations of secure data compartments (SDCs), with hardware monitoring of the information flows from the compartment using hardware policy tags associated with the data at runtime. Unbypassable hardware output control prevents confidential information from being leaked out. Unlike previous hardware information flow tracking systems, DataSafe software architecture bridges the semantic gap by supporting flexible, high-level software policies for the data, seamlessly translating these policies to efficient hardware tags at runtime. Applications need not be modified to interface to these software-hardware mechanisms. DataSafe architecture is designed to prevent illegitimate secondary dissemination of protected plaintext data by authorized recipients, to track and protect data derived from sensitive data, and to provide lifetime enforcement of the confidentiality policies associated with the sensitive data.

## Categories and Subject Descriptors

C.0 [**Computer Systems Organization**]: Hardware/software interfaces; D.2.8 [**Operating Systems**]: Security and Protection—*access controls , information flow controls*

## General Terms

Security, Design

## Keywords

information flow tracking, self-protecting data, architecture

## 1. INTRODUCTION

This paper deals with the processing of sensitive data by unvetted applications. We frequently download applications from unknown sources and have to trust that the applications do not do anything harmful. In cloud computing, we frequently use third-party applications, like analytics or management programs, to process our proprietary or high-value data. If we allow these applications to process our confidential or sensitive data, we have to trust that they do not intentionally or inadvertently leak our data.

Allowing third-party applications to process our sensitive data poses several challenges. First, we do not have source code and cannot modify the application program. We only know its advertised functions, but have no idea what the program actually does. We can only execute the program binaries. Second, for a user who is authorized to access the sensitive data using the application in question, how can we ensure that he does not then transmit the data, perhaps transformed or obfuscated, to unauthorized parties? Third, while we do not expect that the applications are outright malicious, we must assume that complex software will very likely have some bugs or security vulnerabilities. How do we increase the confidence that it does what we allow it to do with our sensitive data, and does not leak this out?

We propose a new software-hardware architecture called DataSafe for protecting the confidentiality of data when processed by unvetted applications, e.g., programs of unknown provenance. It is based on the following key insights in response to the challenges identified above. First, the data owner (not the application writer) is the one most motivated to protect the data, and hence will be motivated to make some changes. Hence, in our proposed solution, the data owner must identify the data to be protected and must specify the data protection policy. The application program is unchanged and continues to deal with data only, and is unaware of any policies associated with the data. This gives the added advantage of our solution working with legacy code. The behavior of the application program must be monitored, to track the protected data as the application executes, and to ensure that its protection policy is enforced at all times.

Second, we observe that while an authorized user is allowed to access the data in the context of the application and the current machine (or virtual machine), data confidentiality (beyond this session) is protected as long as any output from the current machine is controlled according to the data's protection policy. Output includes the display, printing, storing to a disk, sending email or sending to the network. Furthermore, any data derived from sensitive data must also be protected. Hence, our DataSafe solution proposes continuous tracking and propagation of tags to identify sensitive data and enforce unbypassable output control.


*CCS'12,* October 16–18, 2012, Raleigh, North Carolina, USA.


Yu-Yuan Chen, Pramod A. Jamkhedkar, and Ruby B. Lee. A Software-Hardware Architecture for Self-Protecting Data.
In Proceedings of the 19th ACM Conference on Computer and Communications Security, October 2012.

DataSafe architecture realizes the concept of *self-protecting data*, data that is protected by its own associated policy, no matter which program, trusted or untrusted, uses that data. The data must be protected throughout its lifetime, including when it is at-rest (i.e., in storage), in-transit, and during execution. The data protection must apply across machines in a distributed computing environment, when used with legacy applications or new unvetted programs, across applications and across the user and operating system transitions. A self-protecting data architecture must ensure that: (1) only authorized users and programs get access to this data (which we call *primary authorization*), (2) authorized users are not allowed to send this data to unauthorized recipients (which we call *secondary dissemination by authorized recipients*), (3) data derived from sensitive data is also controlled by the data's confidentiality policy, and (4) confidentiality policies are enforced throughout the lifetime of the data.

We assume that the first problem of primary authorization can be solved by well-known access control and cryptographic techniques, and will not discuss this further in this paper. Rather, this paper tackles problems (2), (3) and (4). Problem (2), the secondary dissemination by authorized recipients is especially difficult and dangerous, since an authorized recipient of protected information (passing the primary authorization checks) can essentially do anything he/she wants with it in commodity systems.

Secondary dissemination of protected information can be by an authorized user or by an application, and can be either malicious or inadvertent. A malicious user example could be a confidentiality breach by an insider, such as a nurse in a hospital trying to sell the personal information of some celebrity admitted to the hospital whose medical records he or she is authorized to access. An example of inadvertent secondary dissemination of confidential data could be a doctor trying to send his/her family a vacation plan as an attachment, but accidentally attaching some patient's psychiatry record instead. When programs are the culprits rather than users, a malicious, privacy-stealing malware, installed on an authorized user's machine through social-engineering, could send out sensitive information, or a benign application may contain bugs that could be exploited to leak sensitive information. In DataSafe, we enforce unbypassable output control to prevent such breaches by authorized users.

Data derived from sensitive data must also be tracked and protected. An unvetted application program can be *designed* to leak sensitive information. It could transform or obfuscate the sensitive data. For example, a credit card number could be transformed and obfuscated into several paragraphs of text, before being output from the application, so that no sequence of numbers resembling a credit card number can be identified. This requires somehow tracking the information flows from protected data to other variables, registers or memory locations, across applications and system calls, and across combinations of data such as in mashups. In DataSafe, we argue that such continuous tracking of sensitive data, through any number of transformations, requires some form of dynamic information flow tracking.

For confidentiality policies to be enforced throughout the lifetime of the protected data, DataSafe uses encrypted packages to transmit data between DataSafe and non-DataSafe machines in a distributed environment, as illustrated by Figure 1. A data owner wants the sensitive data to be accessed and used by authorized users according to the data's associated security policy. However, authorized users or applications can maliciously or inadvertently compromise the confidentiality of the protected data by distributing (or leaking) the plaintext of the sensitive data to unauthorized users. DataSafe addresses this problem by: (1) controlling the use of data and preventing leakage on a DataSafe machine while data is used by authorized users (Case A), (2) ensuring secure data transfer to both DataSafe and non-DataSafe machines, and in particular that no protected data is ever sent in plaintext outside the machine (Case B), (3) enabling only authorized users to use protected data on DataSafe machines (Case C, D), and (4) preventing any user from accessing protected data (in plaintext) on a non-DataSafe machine (Case E, F). This last case is restrictive, in terms of availability, but provides fail-safe confidentiality protection within the current ecosystem. (With built-in processor security, the idea is that eventually, all future ubiquitous computers will include DataSafe features.)

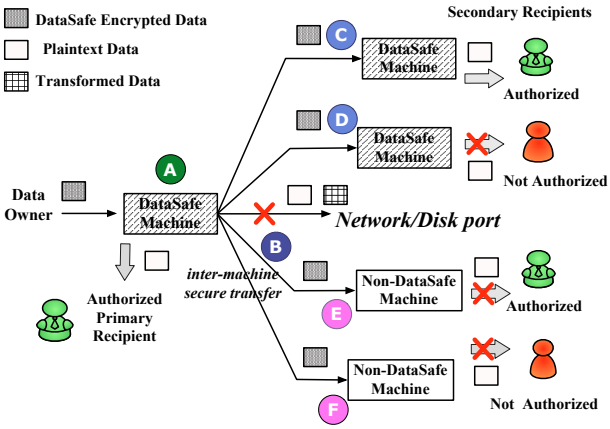



**Figure 1: DataSafe architecture protects data confidentiality across machines (new and legacy) and users (authorized and not authorized).**

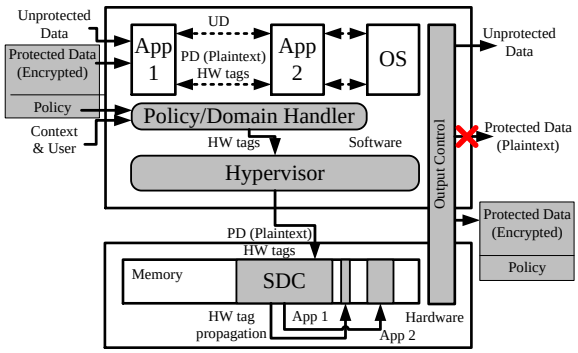



**Figure 2: Software-hardware monitoring of Protected Data (PD) in DataSafe architecture. Unprotected Data (UD) is unchanged. Since the hardware tags of the Protected Data are tracked and propagated at the physical memory level by the hardware, this allows seamless tracking across applications and across application-OS boundaries, as illustrated by the top row of boxes. (Gray indicates DataSafe additions).**

Yu-Yuan Chen, Pramod A. Jamkhedkar, and Ruby B. Lee. A Software-Hardware Architecture for Self-Protecting Data.
In Proceedings of the 19th ACM Conference on Computer and Communications Security, October 2012.

Figure 2 illustrates the key ideas on how DataSafe enables *self-protecting data*. To protect data-at-rest and data-in-transit, DataSafe uses strong encryption to protect the data, while ensuring that only legitimate users get access to the decryption key. For data-during-execution, DataSafe creates a *Secure Data Compartment (SDC)* where untrusted applications can access the data, as they normally would. When data (e.g., a protected file) is first accessed by an application, DataSafe software (Policy/Domain Handler) does a primary authorization check, before translating the data's high-level policy to concise hardware "activity-restricting" tags. The DataSafe hypervisor then creates Secure Data Compartments (SDC), within which sensitive data is decrypted for active use by the application. Each word of the protected data in the SDC is tagged with a hardware activity-restricting tag. From then on, DataSafe hardware automatically tracks the data that initially comes from SDCs, propagating the hardware tags on every processor instruction and memory access. By restricting output activities based on the hardware tags, DataSafe prevents illegitimate secondary dissemination of protected data by authorized recipients, even when the data has been transformed or obfuscated. The hardware tag propagation and output control is done without the knowledge of the applications software, and applies across applications and across application and operating system transitions. We prototype our software-hardware architecture and show that it indeed prevents confidentiality breaches, enforcing the data's confidentiality policy, without requiring any modifications to the third-party applications.

The primary contributions of this paper are:

- A new software-hardware architecture, DataSafe, to realize the concept of Self-Protecting Data. This architecture allows unvetted application programs to use sensitive data while enforcing the data's associated confidentiality policy. In particular, DataSafe prevents secondary dissemination by authorized recipients of sensitive data, protects data derived from sensitive data, and protects sensitive data at-rest, in-transit and during-execution.
- DataSafe architecture is the first that bridges the semantic gap by automatically translating high-level policies expressed in software into hardware tags at runtime, without requiring modification of the application program.
- DataSafe provides efficient, fine-grained runtime hardware enforcement of confidentiality policies, performing derivative data tracking and unbypassable output control for sensitive data, using enhanced dynamic information flow tracking mechanisms.

The rest of the paper is organized as follows. Section 2 compares DataSafe with prior work. Section 3 defines the threat model addressed by the DataSafe architecture. Section 4 describes the software and hardware components of DataSafe architecture. Section 5 presents our prototype implementation. Section 6 provides our test applications and performance analysis, followed by conclusions in Section 7.

## 2. RELATED WORK

We first consider past work on analyzing unvetted applications. We then illustrate the vast past work on both software and hardware information flow tracking proposals. Finally, we describe past work in hardware-enforced security.

First, any software techniques that require access to source code, re-writing the source code or re-compiling the source code, do not apply to our scenarios, since we assume that the user does not have access to the source code of third-party applications. Software methods that analyze binaries or do dynamic binary translations are possible for our scenarios. The BitBlaze project [25] combines static and dynamic analysis for application binaries for various purposes, e.g., spyware analysis [11, 32] and vulnerability discovery [2, 5]. For example, their recent hybrid approach to dynamic information flow tracking [15] can complement what DataSafe does, to provide an even better overall protection system.

Language-based techniques [23] can prevent leaking of information by static type-checking of programs written in languages that can express information flow directly. Programmers can specify the legitimate information flows and policies in the program such that no illegal information flow would be allowed when compiling the program. This static method can be formally verified to be secure. However, unlike DataSafe, it requires access to the source code and re-writing or re-compiling the applications. Also, the programmer is responsible for specifying the data-protection policy, unlike in DataSafe where the data owner specifies the protection policy for his data.

Software solutions involving new operating system designs like HiStar [33] and Asbestos [9, 10, 30] proposed labeling of system objects to control information flow. A process (thread) that has accessed protected data is not allowed to send any data to the network, even if the data sent has no relation at all to the protected data. This coarse-grained information flow protection requires the application to be partitioned into components with different levels of privileges. In contrast, DataSafe's hardware-enforced information flow tracking solution provides fine-grained protection of data at the word level, preventing overly conservative restrictions.

Other software solutions use binary translation [22], or compiler-assisted binary re-writing [29] to change the program, for example, to turn implicit information flows into explicit information flows. We rule out compiler-assisted techniques since we do not have access to source code, but we allow pre-processing of the program binary to instrument the program [15]. However, such software-only information flow tracking approaches may be impractical due to prohibitive performance overhead [22]. For example, to deal with tag assignments and bookkeeping, a single data movement instruction becomes eight instructions after binary translation. A single arithmetic/logic or control flow instruction is replaced by 20 instructions after binary translation. Even with parallel execution of the binary translation [20], the performance overhead is around 1.5X. This is great motivation for DataSafe using hardware information-flow tracking for minimal performance overhead.

We argue that to track transformed protected data efficiently without access to the application's source code, some form of hardware information flow tracking is needed. However, previous hardware solutions are not flexible enough for our purpose. Hardware dynamic information flow tracking solutions include Raksha [7], which can detect both high-level and low-level software vulnerabilities, by programming (i.e., configuring) the Raksha hardware with a small set of four security policies at a time. Thus, only these four vulnerabilities can be detected. In contrast, a novel aspect of our solution is that it allows arbitrary software security policies to be automatically translated by our DataSafe software

**Table 1: Comparison of DataSafe to prior work in (1) expressive high-level policy, (2) automatic translation of software policy to hardware tags, (3) unmodified third-party applications, (4) continuous runtime data tracking, (5) unbypassable output control, and (6) new hardware required.**

| | Exp. High-Level Policy | Auto. Trans. of SW Policy | Runtime Unmod. App. | Runtime Data Tracking | Runtime Output Ctrl. | New HW |
|---|---|---|---|---|---|---|
| Language-based [23] | ✓ | | | | | |
| HiStar [33] | ✓ | | | | ✓ | |
| LIFT [22] | | | ✓ | ✓ | | |
| RIFLE [29] | ✓ | | | ✓ | | ✓ |
| Raksha [7] | ✓(limited) | | ✓ | ✓ | | ✓ |
| GLIFT [27] | | | | ✓ | ✓ | ✓ |
| Bastion [3] | ✓ | | | | | ✓ |
| DataSafe | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |

components to runtime hardware tags, which the DataSafe hardware uses for information tracking and output control.

GLIFT [27] is another hardware DIFT solution that tracks information flow at a much lower hardware level – the gate level. It uses a predicated architecture (implying re-writing or re-compiling applications) which executes all paths of a program to track both explicit and implicit information flow, but at a much higher cost. While a very interesting and potentially promising approach, all the hardware has to be re-designed from the gates up, requiring unproven new hardware design methodologies and tools. Furthermore, unlike DataSafe, the GLIFT protection cannot support chip and machine crossings in a distributed computing environment.

These hardware DIFT solutions either support only a few fixed policies for detecting specific vulnerabilities [7], or require modifying the software [27, 29]. Whenever hardware is used for policy enforcement, there is a semantic gap between the flexibility of policy specification required at the user and domain level, and the restricted actions that can be supported by hardware. We believe we have a solution that bridges this semantic gap. *We believe our solution is the first that supports automatic mapping of flexible software confidentiality policies, associated with the data not the application, to hardware tags suitable for enforcing the data confidentiality policy.* The hardware tags are used for efficient hardware information flow tracking during runtime and for enforcing unbypassable output control. Furthermore, our solution fits in the current software ecosystem, and does not require any changes to the application program.

Other hardware-enabled approaches, which do not implement DIFT techniques, protect sensitive data by requiring access to it via a trusted software component that ensures data confidentiality [3, 4, 8, 16, 17, 18, 19, 24]. These solutions rely on certain trusted components in the application or the underlying operating system to provide data protection. Applications have to be re-written to include a trusted component to access protected data, which is not possible in our scenarios where we do not have access to source code. In contrast, we allow untrusted applications to access our self-protecting data.

The Trusted Platform Module (TPM) [28] is the industry standard for protecting the integrity of a system's software stack, and is also used to protect the encryption/decryption keys which in turn protect the confidentiality of data. However, the TPM, while being able to perform a level of primary authorization by checking the integrity of the software stack, has no control over the decrypted data once the access to the keys has been granted. Our solution prevents this problem of the secondary dissemination of confidential decrypted data. Furthermore, while TPMs can protect software that then protects the data, this approach ties the data with the protected software, whereas our solution provides application-independent data protection.

Table 1 illustrates some of the past work and compares them to DataSafe.

DataSafe's self-protecting data has similarities to digital rights management (DRM). Numerous policy models exist for expressing and interpreting usage and DRM policies such as UCON, XrM, ODRL, etc. [13, 21, 31], however these models cannot be used successfully unless they have a trusted computing base to enforce their policies. A key advantage of DataSafe software is that it is policy language agnostic, and can therefore incorporate these policy models when used in different information domains. Furthermore, DataSafe will also enable the use of different applications along with these policy models while utilizing the policy enforcement trusted computing base provided by the DataSafe architecture.

# 3. THREAT MODEL AND ASSUMPTIONS

We assume an attack model in which the main goal of attackers is to steal or leak out sensitive information that an authorized recipient is allowed to access. Attackers can exploit the vulnerabilities within third party applications or the operating system to leak sensitive data. The third party applications are untrusted and may or may not have gone through a vetting process but still may contain bugs or vulnerabilities that can explicitly or inadvertently leak information. We consider malicious software applications that may leak information through transformation of the data. However, covert channels (including implicit information flow which we address in a separate paper) and side channels are out of scope for this paper.

We assume that the hardware computing device is trusted and does not contain any hardware Trojans. Also, DataSafe software components, i.e., the policy handlers and the hypervisor, are trusted and correctly implemented to carry out their functionalities. Secure launch or secure boot technology is employed to launch the hypervisor to ensure boot-time integrity (e.g., Bastion [3], TrustVisor [18] or TPM [28]). The guest operating system running on the hypervisor and the applications running within the guest operating system can be controlled by an attacker.

Our security model does not protect all data that exist on a device. A piece of data can be converted into DataSafe protected data by our architecture, and hence will not be accessible, in plaintext, without the support of DataSafe. All other unprotected data remains unchanged and accessible as usual. We also assume that authorized entities within a domain are authenticated using standard authentication mechanisms such as passphrases, private key tokens or biometrics. We assume that the entities within a domain have

Yu-Yuan Chen, Pramod A. Jamkhedkar, and Ruby B. Lee. A Software-Hardware Architecture for Self-Protecting Data.
In Proceedings of the 19th ACM Conference on Computer and Communications Security, October 2012.

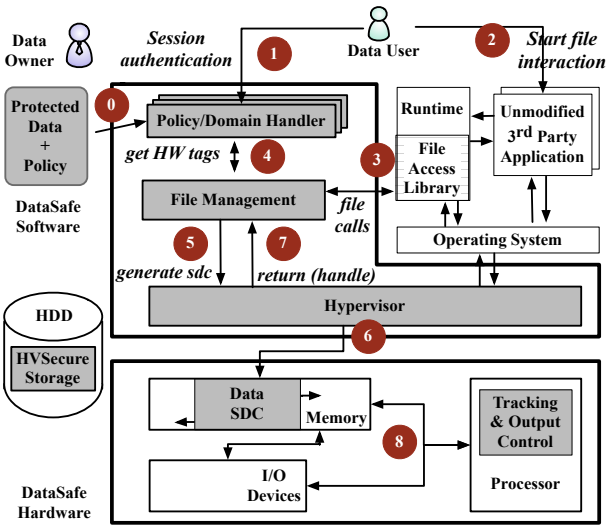



**Figure 3: The software and hardware components of DataSafe. The gray parts are new and trusted DataSafe components, while the striped file access library is modified but untrusted. All other software entities including the unmodified third-party applications and the operating system are assumed to be untrusted.**

access to machines enabled with our hardware and software support if needed. Without proper support, anyone within or outside the domain can only access the encrypted data.

The following threats are out of scope for this paper: (1) denial of service attacks, (2) attacks such as taking a photo of the screen, or human memory, (3) hardware attacks (e.g., memory remanence attack [12]), (4) covert- and side-channel attacks, and (5) data inference mechanisms.

# 4. ARCHITECTURE

We first give an overview of the DataSafe architecture, describing the overall operation of enforcing data confidentiality. We next describe how the DataSafe software components achieve the automatic translation of high-level security policies without having to modify the third-party applications, and then we show how the DataSafe hardware components achieve continuous runtime data tracking with output control.

## 4.1 Overview

DataSafe architecture consists of software and hardware components, as shown in Figure 3. The DataSafe software has the following responsibilities: (1) to translate protected data's high-level security policy into hardware enforceable tags (focusing only on data confidentiality in this paper), (2) to create a secure data compartment (SDC) by associating the tags with the plaintext data in the memory, and (3) to achieve application independence by enabling third party applications to use the data without having to modify them.

The key challenge in the tag generation process is that the hardware tags must accurately reflect the permissions and prohibitions required by the data's security policy. Tags for a given policy are not fixed, but rather they change depending on the context within which the policy is interpreted. In DataSafe software, a policy/domain handler is responsible for translating policies to tags, and the hypervisor is responsible to associate hardware tags with data to create an SDC.

Both the hypervisor and the policy/domain handlers are assumed to be trusted code. The hypervisor maintains its own secure storage (protected by hardware) to store keys and other data structures. The hypervisor is protected by the most-privileged level in the processor and directly protected by the hardware (e.g., as in Bastion [3]). The policy/-domain handler is run in a trusted virtual machine protected by the trusted hypervisor.

### 4.1.1 *DataSafe Operation*

DataSafe operates in four stages – Data Initialization, Setup, Use, Cleanup and Writeback, as explained below.

**Data Initialization.** During the Data Initialization stage, represented by Step 0 in Figure 3, a DataSafe package containing the (encrypted) data to be protected, along with its associated policy, is brought into a DataSafe enabled machine. The details of creation and unpacking of DataSafe packages are explained in Section 5.1.

**Setup.** In the Setup stage, a secure data compartment (SDC) is dynamically created for the data file. An SDC consists of hardware enforceable tags defined over a memory region that contains decrypted data. Hardware tags are generated from the policy associated with the data. Once an SDC is created for a file, users can subsequently use the data file via potentially untrusted applications, while the hardware ensures that the data is used in accordance with the associated policy.

The Setup stage takes place during Steps 1-6, as shown in Figure 3. In Step 1, a user starts a new session by providing his/her credentials, and is authenticated by the policy/domain handler. A session with an authenticated user, data properties and other system or environment properties sets up the context within which the data item is to be used. During the session, the user requests file interaction using a third-party application, as shown in Step 2. The third-party application's request is forwarded to the file management module in Step 3 by the modified file access library of the runtime. In step 4, the file management module requests the policy/domain handler to provide the hardware tags to be set for the file. The policy/domain handler validates the policy associated with the data file taking into consideration the current context (i.e. the user/session properties, data properties and system/environment properties), and generates appropriate hardware tags for the data file.

In Step 5, the file management module requests the hypervisor to create an SDC for the data file with the corresponding hardware tags. In Step 6, the hypervisor decrypts the data file, and creates an SDC for the data file associating the appropriate tags with each word in the SDC. In Step 7, the file handle of the SDC is returned back to the policy/domain handler and the execution is returned back to the application.

**Use.** In the Use stage, the DataSafe hardware tags each word of the protected data in each SDC and persistently tracks and propagates these tags, as shown by Step 8. Once an SDC is set up for a data file, in accordance with the session properties, any third-party application can operate on the protected data as it would on any regular machine. The DataSafe hardware will ensure that only those actions

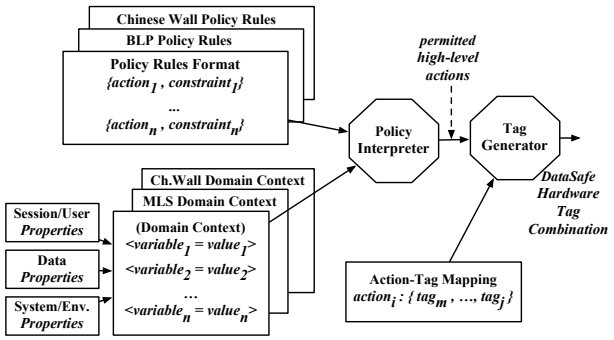



**Figure 4: Translation from high-level policies to hardware tags.**

that are in conformance with the data-specific policy are allowed.

**Cleanup and Writeback.** After the application finishes processing the data, the DataSafe hypervisor re-packages the protected data and the policy if the data was modified or appended to, re-encrypts the protected data, removes the associated tags within the SDC, then deletes the SDC.

## 4.2 Runtime Translation of Expressive Software Policy to Hardware Tags

The two DataSafe software components, the policy/domain handlers and the hypervisor take a high-level policy specified in a policy model, translate the policy into the hardware enforceable tags and create an SDC for the protected data.

Figure 4 shows the two step process employed by DataSafe: (1) a security policy is interpreted to determine what high-level actions are permitted (policy interpreter), and 2) depending on the high-level actions permitted, the appropriate hardware tags are chosen (tag generator).

DataSafe is designed to be generic, supporting multiple policies languages and policies such as BLP, Chinese Wall, etc. Therefore, policy rules from different policies are first represented in a common policy representation model[1].

A policy is expressed and interpreted in terms of a context, which typically includes information about user properties, data properties and system properties necessary to interpret the policy appropriately, in a given domain context. For example, a BLP policy will require the user's security clearance and the data's security classification, whereas a RBAC policy will require the user's role. The context information is collected and stored in the form of $\{variable, value\}$ pair. The policy and the context information are then fed to the policy interpreter, which determines what high-level actions are permitted by the policy, on the data, under the current context values. If no high-level action is permitted, then access is denied at this point. If this set is non-empty, it means that the user has authorized access to the data, but is expected to use the data only in the manner defined by the permitted action set. The permitted action set is then used to calculate the hardware tags, and to generate an SDC for the data.

**Policy Model.** Our policy model consists of a set of restricted actions $\{ra_1, ra_2, ra_3, ..., ra_n\}$, where each restricted action includes an action associated with a constraint, represented by $ra_i = \{action_i, constraint_i\}$. The *action*, is a high-level action such as "*read*, "*play*", "*view*", etc. The *constraint*, is a first-order predicate formula defined in terms of context variables. A context is defined by a set of variables $\{v_1, v_2, ..., v_n\}$, that represents user, data and system properties. A given constraint evaluates to either $true$ or $false$ based on the values of the context variables. For a given restricted action, $ra_i = \{action_i, constraint_i\}$, if $constraint_i$ evaluates to $true$, then $action_i$ is permitted, otherwise it is not permitted.

[1]While this is not absolutely essential, we use the common form to preclude the need for a separate policy interpretation engine for every policy.

**Table 2: The correspondence between policy-prohibited activities and the hardware tags that restrict that activity.**

| Actions | Category | Restriction | Tag |
|---|---|---|---|
| Edit | Access | No Write to SDC | 0x08 |
| Append | Access | No extensible SDC | 0x10 |
| Read | Access | No read from SDC | 0x20 |
| View | Transient output | No copy to display | 0x01 |
| Send Plain-text | Persistent output | No copy to network | 0x02 |
| Save Plain-text | Persistent output | No copy to disk | 0x04 |

**Permitted Policy-level Actions to HW tags.** For every policy, the semantics of its high-level actions, described within the policy, have a specific interpretation in terms of hardware-level actions. Based on this interpretation, every high-level action maps to a set of hardware tags. At present, the DataSafe prototype supports six hardware tag values, as shown in Column 4 of Table 2, but the architecture can support more tag values. Hardware restriction tags are expressed in the form of a bit vector, where each bit, when set to 1, corresponds to a type of restriction. The hardware tags are restrictive, which means that if a particular tag bit is set, that particular hardware-level action is prohibited. For example, if the tag bit $0x01$ is set for an SDC, the DataSafe Hardware will prevent any application and the OS from copying that SDC's data to the display output. On the other hand, if the policy permits the action "*view*", then tag $0x01$ should <u>not</u> be set. Hence, for a given policy interpretation, the set of tags corresponding to the permitted actions are *not* set, and the rest of the tags are. The tag generation process is independent of whether a policy is attached to a particular datum, or it applies system wide to all data items. Hence, DataSafe can support both mandatory and discretionary access control policies.

DataSafe hardware tags are divided into three categories: (1) Access, (2) Transient Output, and (3) Persistent Output tags. Tags in the Access category, which include *write*, *append* and *read*, prevent in-line modification, appending or reading of an SDC. The tags in the Transient Output category refer to the output devices where the lifetime of the data ends after the data is consumed by the device, e.g., the display or the speaker. The Persistent Output category deals with the output devices where data remain live after being copied to those devices, e.g., network or disk drives. If an application or a user, after gaining authorized access to pro-

**Table 3: BLP policies**

| |
|---|
| **Context Variables:**<br>*sec_clear* ∈ {Top Secret, Secret, Confidential, Unclassified}<br>*sec_class* ∈ {Top Secret, Secret, Confidential, Unclassified} |
| **Action to Tags Map:**<br>*read* ⇒ {No copy to display, No read}<br>*write* ⇒ {No write to SDC, No append to SDC}<br>*leak data* (*implicit*) ⇒ {No copy to disk, No copy to network} |
| **BLP Policy:**<br>$ra_1$ := {*action* := *read*, *constraint* := *sec_class* ≤ *sec_clear*},<br>$ra_2$ = {*action* := *write*, *constraint* := *sec_class* ≥ *sec_clear*} |
| **Use Case1:** *sec_clear* := Secret, *sec_class* := Confidential<br>**Actions permitted:** {*read*}<br>**Actions prohibited:** {*write*, *leak data*}<br>**Tags set:** {No write to SDC, No append to SDC, No copy to disk, No copy to network} |
| **Use Case2:** *sec_clear* := Secret, *sec_class* := Top Secret<br>**Actions permitted:** {*write*}<br>**Actions prohibited:** {*read*, *leak data*}<br>**Tags set:** {No read, No copy to display, No copy to disk, No copy to network} |
| **Action to Tags Map (Extended BLP):**<br>*read* ⇒ {No read}<br>*view* ⇒ {No copy to display}<br>*write* ⇒ {No write to SDC, No append to SDC}<br>*prevent leakage* ⇒ {No copy to disk, No copy to network} |

tected plaintext data, saves the data in plaintext form on a disk drive, or sends the plaintext over the network, the data's confidentiality is permanently lost. Most policies don't explicitly mention the requirement to prevent such activities, but rather assume that the authorized user is trusted not to illegally leak the data out. In order to enforce this critical and implicit assumption, in DataSafe systems, these two tags are always set, for all *confidentially protected* data, for all policies, except for policies that have explicit declassification rules.

### 4.2.1 Bell-LaPadula Policy Example

To give a concrete example of how the policy translation works, we consider a Bell-LaPadula (BLP) policy defined over a multi-level security (MLS) environment. In this system, each user has a *Security Clearance* and each data item has a *Security Classification*. Both properties range over the ordered set {Top Secret > Secret > Confidential > Unclassified}. The BLP policy states: *"A user at a security clearance x can only read data items with security classification y such that $y \leq x$, and can write only to data items with security classification z such that $z \geq x$"*. The representation of this policy in our standard policy model is shown in Table 3.

The context variables *sec_clear* represents *Security Clearance* and *sec_class* represents *Security Classification*. BLP has *read* and *write* as high-level actions, while *leak data* is an implicit action. Each action corresponds to the hardware tags as shown. The BLP policy is the set of restricted actions $\{ra_1, ra_2\}$, where the constraints are expressed as first order formulas over context variables *sec_clear* and *sec_class*. In Use Case 1, action *read* is permitted according to the BLP policy, and hence *read* tags are reset, while *write* and *data leakage* tags are set. In Use Case 2, *write* tags are reset, while *read* and *data leakage* tags are set.

Traditional BLP policy has only two actions *read* and *write*. It does not distinguish between reading some data from the memory and viewing data on the display screen. Now consider a situation where the users are not humans but applications, and certain applications are allowed to read the data, but not allowed to display. For example, an application may be allowed to read and process passwords, but it is not allowed to display the plaintext password on the screen. For such applications, *read* and *view* are two separate high-level actions. In DataSafe such an extended BLP can be supported by introducing a new high-level action and changing the action-tag map as shown in Table 3.

**Table 4: Example entries of the `sdc_list` software structure.**

| ID | Virtual addr | Machine addr | Size | Tag |
|---|---|---|---|---|
| id1 | vaddr1 | maddr1 | size1 | 0x08 |
| id2 | vaddr2 | maddr2 | size2 | 0x1C |

## 4.3 Unmodified Applications

In DataSafe, the confidentiality-protection policy is defined for the data and packaged with the data (see Section 5.1), not defined by a particular application or its programmer. In other words, the data's policy is enforced no matter which application is accessing the data; therefore, applications are agnostic of DataSafe's operation and do not have to be modified to work with DataSafe. Only the file access library in the runtime or the interpreter has to be modified to redirect file calls of the application to the file management module of the DataSafe Software. Furthermore, DataSafe-protected data are protected with the SDCs, where the SDCs are defined at the hardware level, the layer below any software entity.

This is one of the key design features of DataSafe – to define the SDC over the physical machine memory, instead of the virtual memory. This enables us to achieve application independence and cross boundary data protection. Applications access their data through virtual memory. Once an SDC is created in the physical memory, an application can access the data within the SDC by mapping its virtual memory to the SDC in the physical memory. This data can be passed among multiple applications or the OS components.

Once the hardware restriction tags are determined for a given data file, DataSafe associates those tags with the memory region allocated to the file, without having to change how the application accesses the protected data. Such an association is achieved by a secure data compartment (SDC). The DataSafe hypervisor is responsible for the creation, maintenance and deletion of SDCs, and maintains an SDC list as shown in Table 4. An SDC is a logical construct defined over a memory region that needs to be protected, independent of the application. Every SDC has a start memory address, a size, and a tag combination specifying its activity-restricting rules with which the data within the SDC are protected.

SDCs can be defined at different granularities. DataSafe can define different types of SDCs over different parts of the data object. For example, different sections of a document,

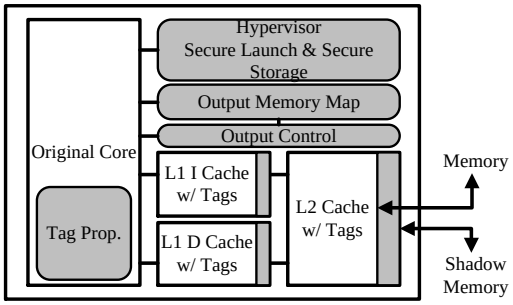



**Figure 5: The DataSafe hardware components (gray).**

different tables in a database, or different parts of a medical record need different types of confidentiality protection.

For applications to access the DataSafe-protected data in an SDC, we modify the application file access library to redirect the access requests from the applications to the policy/-domain handler(s), as shown previously in Figure 3. The modified file access library does not have to be trusted. In case the access request is not redirected by a malicious library for protected data, only encrypted data will be available to the application, which is a data availability issue instead of a confidentiality breach. We describe in more detail about our modified file access library in Section 5.2.2.

## 4.4 Continuous Runtime Data Tracking

In order to provide continuous runtime protection for the protected data (now in plaintext) within an SDC while the application is executing, we use hardware mechanisms to track each word of the protected data throughout the execution of the untrusted application. DataSafe extends each 64-bit data word storage location with a $k$-bit SDC ID and a $j$-bit tag. The shadow memory shown in Figure 5 is a portion of the main memory set aside for storing the tags. It is a part of the hypervisor secure storage, which the DataSafe hardware protects and only allows the hypervisor to access. The hardware tag is set by the hypervisor when an SDC is requested to be set up by the policy handler. Note that only the hypervisor has read/write access to the shadow memory for adding and deleting the tags for the SDCs.

To track and monitor where the protected data resides in the system, we propagate the tags along with the data from within the SDC as it goes outside the SDC to other parts of memory. There are two levels of propagation for the tag bits of an SDC. First, the hardware tag bits are propagated from the shadow memory to the last level on-chip cache, when a cache line is brought from the main memory due to a cache miss. The same tag bits are copied throughout the cache hierarchy, i.e., up to the level-1 data cache. The general purpose registers in the processor are also extended with the ability to propagate the tag bits. On memory load instructions, the tag bits are copied from the level-1 data cache to the destination register.

Each instruction executed in the processor performs tag propagation operations along with its arithmetic or other operations. This way the hardware restriction tags can track sensitive data even if the data has been transformed or encoded by the application. We use the principles of existing information flow tracking techniques[2] [6], where the source tag bits are propagated to the destination register as long as the source register has a nonzero tag bit. In the case where both of the source registers have nonzero tag bits, we take the union of the two tag bits to give the destination register a more stringent policy. For load instructions, the union of the tag of the source address register and the tag of the source memory data is propagated to the tag of the destination register. For store instructions, the union of the tag of the source data register and the tag of the source address register is propagated to the tag of the destination memory address. Thus, the tag propagations for load and store instructions account for the index tag for table lookups. For integer arithmetic and multiply and divide instructions, the tag is a combination of the tag of the first source register, the tag of the second source register, the tag of the condition code register and the tag of other registers if necessary, e.g., the $y$ register[3] for the SPARC architecture. The tag of the condition code register is also updated if the instruction has these side-effects.

If both of the source registers are tagged with the same SDC ID, the destination register is also tagged with this SDC ID. If they are not from the same SDC, we assign a reserved ID tag of $2^k - 1$. Since the resultant data does not belong to either of the two source SDCs, the SDC IDs are not combined; rather a special tag is substituted to indicate that this is an intermediate result.

The tag propagation rules described above handle explicit information flow from the data within an SDC, where the destination operands receive direct information from the source operands. There are also cases where the destination operand receives information from the source operand(s) through a third medium, e.g., the integer condition code or branch instructions. This kind of information flow is implicit but can be exploited to leak information. A vanilla dynamic information flow tracking system without considering such information flow would lead to false-negatives since information could be leaked without being tagged. However, a naive approach that tags any instruction that is dependent on the branch condition's tag may lead to an impractically large amount of false-positives [1, 15]. Such implicit information flows (e.g., where the condition is tagged with restrictions) are a type of software covert channel and covert channels are not in the scope of this paper as stated in our threat model, due to lack of space. (However, we do present our full solutions for implicit information flow in a separate paper.)

## 4.5 Hardware Output Control

DataSafe hardware checks to see whether copying the data to another memory location or output device is allowed, or whether writing to memory locations within the SDC is allowed, according to the hardware tags. In particular, hardware checks if a memory location to be written to is a memory-mapped output device, and enforces output control according to the tag of the word being written.

We introduce a new hardware structure inside the processor: the output memory map, `mem_map`. The `mem_map` is

[2]Special cases such as zeroing a register (e.g., "`xor %eax, %eax`" on x86) are treated differently. For example, the destination tag is cleared in this example.

[3]The $y$ register is used for storing the upper 32-bit result in multiplication and the remainder in division in SPARC architectures.

**Table 5: Example entries of the output memory map `mem_map` hardware structure.**

| Start addr | End addr | Mask |
|---|---|---|
| addr1 | addr2 | Display |
| addr3 | addr4 | Disk |

only accessible to the trusted hypervisor. It stores memory-mapped I/O regions and I/O ports to enable the hardware to know if a memory store instruction is attempting to perform output. It is checked on the destination address of memory store instructions, or any other instructions that write to an output device (e.g., `in` and `out` instructions in x86 architecture), to see if there is a violation of the output policy specified in the tag associated with the data to be written.

Table 5 shows example entries in the `mem_map` hardware structure. The device mask is a bit mask which indicates its functionality e.g., display, speaker, USB storage, NIC, etc. Two devices having the same functionality would have the same mask value. In our DataSafe prototype, the mask is designed to match the activity-restricting bits in the hardware tags, so that it can be easily used by the hardware check logic to determine whether data with a specific tag value can be written to the I/O device.

## 4.6 System Issues

DataSafe's tag propagation is performed by the hardware logic on the *physical* memory; therefore the propagation mechanism is not changed when the protected data is passed between applications, OS components or device drivers.

Direct Memory Access (DMA) data transfers do not need to include the hardware activity-restricting tags, which are runtime tags only and are not stored in persistent storage or transmitted on a network. DataSafe treats DMA regions as output device regions and performs output control to prevent protected data (based on their hardware tags) from being written to these DMA regions. The DataSafe hypervisor also prevents SDCs from being created over allocated DMA regions (and vice versa) so that data in SDCs cannot be over-written by DMA input transfers.

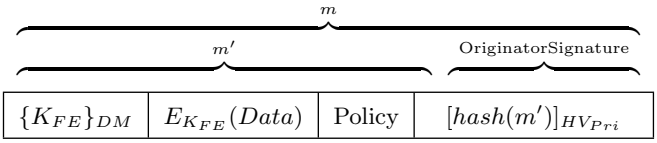
# 5. IMPLEMENTATION

## 5.1 Encrypted Data and Key Management

**Creation and Packaging.** A piece of data can be turned into a piece of DataSafe-protected data on any computing device within the domain that is enabled with DataSafe support. The data owner specifies the confidentiality policy for the data. We describe one implementation of key management for a domain, e.g., a hospital; many other implementations are possible. The format of a piece of DataSafe-protected data is shown in Figure 6. To create DataSafe-protected data that binds the owner-specified policy to the data, the hypervisor first generates a new symmetric key $K_{FE}$, called the file encryption key[4], and uses $K_{FE}$ to encrypt the data. $K_{FE}$ is then encrypted by the domain manager's[5] public encryption key, $K_{DM}$. The trusted DataSafe hypervisor then calculates a cryptographic hash over the encrypted $K_{FE}$, the encrypted data and the owner-specified policy and signs the hash using the its private signing key, $HV_{Pri}$, as the Originator Signature.

[4]Each protected data file has its own random file encryption key.

[5]A domain manager is the administrator or authority that manages the computing devices within a domain and it could be installed on any DataSafe machine.




$A \rightarrow B : m, Cert_A$
$B \rightarrow DM : \{K_{FE}\}_{DM}$
$DM \rightarrow B : \{K_{FE}\}_{HV\_B}$

**Figure 6: Encrypted DataSafe package for storage and for transmission between machines: the originator ($A$), the receiver ($B$) and the domain manager ($DM$), with respective DataSafe hypervisors on $A$ and $B$ denoted as $HV\_A$ and $HV\_B$. $[x]_{HV}$ denotes a private key signature or decryption operation by $HV$, while $\{x\}$ denotes a public-key verification or encryption operation. $Cert_A$ denotes the public key certificate of $A$ that is signed by the domain manager.**

**Transfer.** Once a DataSafe self-protecting data package is created, it can be moved to any DataSafe enabled computing device within the domain for use. In a non DataSafe-enabled machine, only encrypted data can be accessed.

**Unpacking.** When an authorized recipient receives a piece of DataSafe-protected data and accesses it with an application, the policy/domain handler validates the data and the policy, and retrieves the file encryption key $K_{FE}$. Validation of the data and the policy is done by verifying that the originator signature was signed by a trusted hypervisor within the domain. A hash is re-calculated and compared with the decrypted hash in the signature, to ensure that the data, the policy and the encrypted file encryption key have not been tampered with.

Since the file encryption key $K_{FE}$ is encrypted with the domain manager's public encryption key, the policy/domain handler follows a secure protocol to retrieve the file encryption key. The domain manager ensures that the requesting hypervisor is not on the revocation list; otherwise the request is denied. In DataSafe, public-key crypto is used for system identification and non-repudiation to protect smaller-size items such as the $K_{FE}$, and efficient symmetric-key crypto is used for protecting the larger data content. Since the $K_{FE}$ is encrypted, it is stored on the user's machine in the normal unsecured storage, whereas the hypervisor's private signing key, $HV_{Sign}$, and the domain manager's secret decryption key are stored in their respective DataSafe machine's hypervisor secure storage (See Figure 3). Note that since the $K_{FE}$ is encrypted using the domain manager's public encryption key, no key exchange between different DataSafe systems is required. Only individual communication with the domain manager is needed (Figure 6). To prevent the domain manager from becoming a bottleneck or a single point of failure, multiple or backup key management servers can be installed on other DataSafe machines to provide enhanced data availability.

**Redistribution and Declassification.** An authorized

Table 6: The policy/domain handler API.

| API Call | Description |
|---|---|
| open_file | Open an existing DataSafe protected file. |
| close_file | Close an open DataSafe protected file. |
| read_file | Read from an open DataSafe protected file. |
| write_file | Write to an open DataSafe protected file. |

user can access the DataSafe protected material in plaintext, and also pass on the original DataSafe encrypted package (signed by the originator) to another machine. If he transforms the protected data and wants to pass this modified data to another machine, he has to re-package it (as described for packaging above) and sign with his own trusted hypervisor's private key.

Some data items may get declassified to be used on non-DataSafe devices. Declassification is done by the Domain/Policy Handler while the data is not in use (not loaded into memory) by any application, and thus precludes the need to un-tag the data. This allows for authorized declassification by trusted software components – by decrypting the data, and dissociating any policy associated with it. Once declassified, such data can be treated as data that can be used on any device.

## 5.2 DataSafe Software

### 5.2.1 Policy Handler

The policy/domain handler is primarily responsible for hardware tag generation from the high-level policy. It is also responsible for setting up the context, which includes maintaining the values for user properties, data properties, and system/environment properties. Since both these responsibilities are specific to a particular information domain, we have a separate policy/domain handler for each domain. At present, we have implemented a policy/domain handler for Multi-level Security systems that supports BLP and Biba policies, a policy handler for the Chinese Wall policy, one for Clark-Wilson, and one for medical information systems. In all policy/domain handlers, policies are represented in the standard policy model using the XML format. New policies can be specified in XML and interpreted directly by the policy interpreter. Each policy/domain handler maintains a separate database for storing user and data properties. All policy handlers share a common policy interpreter, which is possible since all policies are represented in a standard form.

### 5.2.2 File Management Module

For the prototype implementation, DataSafe software has a separate file management module that provides a file management API for accessing DataSafe-protected files and provides file handling functions, as shown in Table 6. The file management module loads the encrypted file into the memory, and forwards the file access request to the policy/domain handler, which translates the policy associated with the file into hardware tags, and requests the hypervisor to set up SDCs for the file.

Currently, the file management module supports file handling functions for Ruby-based applications. We have modified the Ruby Interpreter to redirect file handling calls to the file management module. This file management module provides a file handle to the Ruby Interpreter, which it subsequently uses for file operations. If a file attempts to obtain untagged data by bypassing the redirection of file calls, it only ends up getting encrypted content. Similar file management modules for non-interpreted languages such as C is under development with a modified C-library (libc) for redirected protected file access.

Table 7: The new hypercalls.

| Semantic | Description |
|---|---|
| sdc_add( addr, size) | Adds a new SDC protecting policy-encoded data starting at virtual address addr with size size |
| sdc_del(sdcid) | Deletes an existing SDC with ID = sdcid |
| sdc_extend( sdcid, size) | Extends an existing SDC with ID = sdcid, with contents of size size |

### 5.2.3 Hypervisor

The hypervisor is responsible for the instantiations of SDCs, the management of domain-specific secret keys and the provision of environment properties for context generation.To manage the SDCs, the hypervisor keeps a software structure, called the active SDC list, sdc_list, which stores a list of active SDCs for all policy handlers.

Table 7 shows the new hypercalls introduced to support the SDCs: sdc_add, sdc_del and sdc_extend. Hypercalls for context generations and others are omitted. The sdc_add hypercall is called when the policy/domain handler requests a new SDC. The sdc_del is called later to delete an SDC. The sdc_extend is used when the high-level policy allows for appending to the protected data, where the size of a SDC is adjusted to include appended data.

## 5.3 DataSafe Prototype

Our prototype implementation builds upon the open source processor and cache hardware and the hypervisor in the OpenSPARC platform. The current prototype is implemented in the Legion simulator of the OpenSPARC platform. This simulates an industrial-grade OpenSPARC T1 Niagara processor with 256 MB of memory, running the UltraSPARC Hypervisor with Ubuntu 7.10. We utilize the load_from/store_to alternate address space (ldxa and stxa) instructions in the SPARC architecture to access our new hardware structure, mem_map, at the same time limiting the access to only hyperprivileged software.

The open source hypervisor in the OpenSPARC platform is modified and extended with the functionality to support secure data compartments (SDCs). Our new hypercall routines are implemented in SPARC assembly and the SDC-specific functions are implemented using the C language. The policy/domain handler is implemented in the Ruby language and the policies are expressed in XML format.

# 6. ANALYSIS

This section evaluates the security, performance and cost of the DataSafe architecture.

## 6.1 Security Tests

We tested our prototype with several experiments.

Table 8: A summary of experimental results.

| # | Test Case | Attacks | Res. |
|---|---|---|---|
| ***SW to HW Tags*** | | | |
| 1 | BLP | read, write, output ctrl. | ✓ |
| 2 | Chinese Wall | read, write, output ctrl. | ✓ |
| 3 | Hospital Policy | Nurse Attack | ✓ |
| | | Doctor Attack | ✓ |
| ***Application Independence*** | | | |
| 4 | Editor (Ruco) | read, write, output ctrl., transformation | ✓ |
| 5 | Search (Grepper) | read, write, output ctrl., transformation, fine-grained control | ✓ |
| 6 | Text Transformation (HikiDoc) | password leak (allow read but no display) | ✓ |

#### 6.1.1 Support for high-level policies

We first test the support for high-level policies, automatically mapped into hardware tag generation at runtime. We tested three different types of policies: a multi-level security policy using the BLP policy (explained in Section 4.2.1), a multi-lateral security policy using the Chinese Wall policy, and our own concocted hospital policy.

All these policies were first expressed in the DataSafe policy model in an XML format. The policies were then interpreted using the DataSafe policy interpreter and hardware tags were generated under different contexts. For each policy, we tested the read/display, write and output control. With the hospital policy we tested the scenarios of a malicious nurse leaking out private information, and the accidental leak of psychiatric data through email by a doctor (discussed in the Introduction).

#### 6.1.2 Application Independence

Next, we tested DataSafe's capability to support *unmodified* third party applications, using three applications, Ruco, Grepper and HikiDoc, downloaded from RubyForge. All three are Ruby-based applications. Ruco is a lightweight text editor, Grepper provides the same functions as the "grep" command-line utility for searching plain-text data sets for lines matching a regular expression, and HikiDoc reads text files and converts them to HTML documents. We were able to run all the three applications on DataSafe, *unmodified*.

The experiments with the Ruco editor include basic read-/display and write control. In addition we modified Ruco to test illegal saving of plaintext on the disk, either with or without data transformation. A similar set of experiments were carried out with the Grepper application. In addition, with Grepper we tested fine-grained tracking by creating SDCs with different tags and sizes over different parts of a file – DataSafe could successfully track the data and enforce fine-grained output control of sensitive data.

With HikiDoc we tested a scenario for authorized read but prohibited display. In this scenario, simulating "password leak" attacks, the HikiDoc application takes two files as input: 1) text file (to be converted to HTML), and 2) a file containing passwords for user authentication. The program is supposed to read the password file for authentication, but not leak the password out. We inserted a malicious piece of code in the application which transforms the password into a code, and then distributes the code at predefined locations in the HTML file. The attacker can then retrieve the code parts from the HTML file, assemble the code, and reverse the transformations to get the original password. DataSafe could track the transformed pieces of a password and prevent their display.

In all these applications, the data read from the file is passed through different Ruby libraries, the Ruby Interpreter, and the operating system, before being displayed. In addition, the data is processed in different formats before being output in a presentable form. Tests on these applications show that DataSafe is application independent, can continuously track protected data after multiple transformations and can do this across multiple applications in the user space, and across the user-OS divide.

#### 6.1.3 Continuous Data Tracking and Output Control

Apart from testing policy support and application independence, the experiments above also test the capability of DataSafe to enforce SDCs and hardware activity restricting tags. This includes the capability to track protected data in a fine grained manner across applications and OS, and to enforce output control only on that data which is tagged with such a restriction. The insight we derived from the above tests is that a more comprehensive, yet quick, coverage can perhaps be achieved by just a small set of synthetic test cases which represent different classes of attacks that can leak protected data, as shown in Table 9. In each test case, programs were run on the DataSafe machine (DS column), and on an existing non-DataSafe machine (nDS column). For each test case, the sensitive data files were protected by a policy to prohibit the test case scenario.

Test cases 1-5 of Table 9 test the output control capabilities of DataSafe based on output port types. In these cases, SDCs were created to prevent *edit, append, save, send over the network, and display.* Test cases 6-8 represent data transformation attacks by a single program. In these cases, a test program reads and transforms the data multiple times, and then tries to send the data out on one of the output ports (i.e. disk, network and display). Test cases 9-11 represent cross program attacks, where data is read by Program 1 (P1) and passed on to Program 2 (P2) which carries out the attack. Test cases 12-14 represent transformation and cross program combined attacks. In these test cases, data is read by Program 1(P1) and transformed multiple times, and then the transformed data is sent to Program 2 (P2), which carries out the attack. In test case 15, different parts of a file were protected by SDCs with different protection tags. DataSafe was able to prevent different attacks targeting each of these protected segments. *In all the test cases, the attack succeeded in the existing machine (nDS), but DataSafe (DS) was successful in defeating the attack.*

### 6.2 Performance and Cost

Since DataSafe is a software-hardware architectural solution, its advantages come at the cost of changes in both hardware and software. These costs are in two distinct phases: 1) the *Setup* (and Termination), carried out by DataSafe software, incurs performance costs in the redirection of file calls and setting up of SDCs, and 2) the *Operation* phase, carried out by DataSafe hardware, incurs performance costs due to

**Table 9: Test cases for illegal secondary dissemination and transformation tested for DataSafe (DS) and non-DataSafe (nDS) machines. "F" represents a file, and "P" represents a program. "✗" means attack failed (good), and "✓" means attack succeeded (bad).**

| No. | Test Case | DS | nDS |
|---|---|---|---|
| | ***Output Control*** | | |
| 1 | edit [F1, P1] | ✗ | ✓ |
| 2 | append[F1, P1] | ✗ | ✓ |
| 3 | read[F1, P1] ; save[F1, P1] | ✗ | ✓ |
| 4 | read[F1, P1] ; send[F1, P1] | ✗ | ✓ |
| 5 | read[F1, P1] ; display[F1, P1] | ✗ | ✓ |
| | ***Transformations*** | | |
| 6 | read[F1, P1] ; transform[F1, P1] ; save[F1, P1] | ✗ | ✓ |
| 7 | read[F1, P1] ; transform[F1, P1] ; send[F1, P1] | ✗ | ✓ |
| 8 | read[F1, P1] ; transform[F1, P1] ; display[F1, P1] | ✗ | ✓ |
| | ***Cross-Program*** | | |
| 9 | read[F1, P1] \| save[F2, P2] | ✗ | ✓ |
| 10 | read[F1, P1] \| send[F2, P2] | ✗ | ✓ |
| 11 | read[F1, P1] \| display[F2, P2] | ✗ | ✓ |
| | ***Transformations and Cross Program*** | | |
| 12 | read[F1, P1] ; transform[F1, P1] \| save[F2, P2] | ✗ | ✓ |
| 13 | read[F1, P1] ; transform[F1, P1] \| send[F2, P2] | ✗ | ✓ |
| 14 | read[F1, P1] ; transform[F1, P1] \| display[F2, P2] | ✗ | ✓ |
| 15 | ***Fine-grained Transformation and Tracking*** | ✗ | ✓ |

information flow tracking and output control. We analyze the cost of these changes separately, and then discuss the end-to-end cost of running third party applications.

### 6.2.1 Software Performance

Table 10 shows the costs incurred for file operations *open, add_sdc read, write, delete_sdc* and *close*. The overhead of *open* is due to file access redirection and the setting up of memory mapped regions which does not take place in non-DataSafe machines. The cost of adding and deleting SDCs on DataSafe is small compared to the other operations. These performance costs are the same for any file size.

In contrast, we actually achieve better performance during the Operation phase for *read* and *write* operations in DataSafe because of the use of memory mapped file operations. These performance gains are directly proportional to the file size (shown for reading or writing a 2.5MB file in Table 10). Hence, as the file size increases, the performance costs of *open* and *close* get amortized leading to better results. This is verified by the total application execution times of different file sizes, shown in Table 11. As the file size increases, the relative performance cost of DataSafe decreases. For a reasonable file size of 2.5MB, the performance cost of DataSafe is only about 5%.

### 6.2.2 Hardware Performance

We now evaluate the hardware performance overhead during the Operation phase. The hardware tags can be added to the existing processor datapaths by extending the widths of the registers, buses and caches (as shown in Figure 5). Alternately, as shown in Figure 7, they can be a separate

**Table 10: Performance costs of DataSafe software operations vs. non-DataSafe (in cycles on the Legion simulator).**

| Operation | non-DataSafe | DataSafe |
|---|---|---|
| *open* | 117521.4 | 341109.8 |
| *add_sdc* | N/A | 10177 |
| *read* | 9016594 | 2847026 |
| *write* | 2847026 | 1659347 |
| *delete_sdc* | N/A | 3976 |
| *close* | 22076.4 | 278525 |

**Table 11: Performance cost (in seconds) of running Hikidoc application on increasing file sizes.**

| App | 0.5 MB | | 2.5 MB | |
|---|---|---|---|---|
| | non-DS | DS | non-DS | DS |
| Hikidoc | 0.53 | 0.67 (26.42%) | 3.49 | 3.68 (5.44%) |

and parallel "tag datapath". This clearly shows that the tag propagation logic is done in parallel with the instruction execution, hence the hardware tag propagation does not incur runtime overhead, as also found in [7].

Since all tag propagation operations can be done in parallel, the only source of hardware runtime overhead involves the output checking of memory store instructions. However, memory stores are not on the critical path, as opposed to memory loads, and normally stores are delayed waiting in the store buffer queue for an unused cache access cycle. Hence, the output checking can be performed while the memory store instruction sits in the store buffer or the memory write buffer. Output control involves checking against the `mem_map` structure, similar to the operation of a victim buffer [14] or a small fully associative cache, with a different comparator design. The comparator for a victim buffer is testing for *equality*, whereas we test for *inequality*. Our hardware FPGA implementation of inequality versus equality comparators showed that they had comparable latency. Therefore, the net effect of performing output checking on store instructions is equivalent to adding a one cycle delay for store instructions waiting in the store buffer queue. Hence, the output checking has no discernible impact on the overall processor bandwidth (in Instructions executed Per Cycle, IPC).

### 6.2.3 Storage Overhead and Complexity

The software complexity of DataSafe amounts to a total of 50% increase in the hypervisor code base, about half of which was for a suite of encryption/decryption routines for both asymmetric and symmetric crypto and cryptographic hashing algorithms (Table 12). Each `sdc_list` entry takes up about 26 bytes of memory space, considering a full 64-bit address space. The total storage overhead incurred by the `sdc_list` varies according to the number of entries in the `sdc_list`. In our prototype implementation 20 entries are typically used, amounting to around half a kilobyte of storage overhead.

For the DataSafe hardware, the main cost comes from the

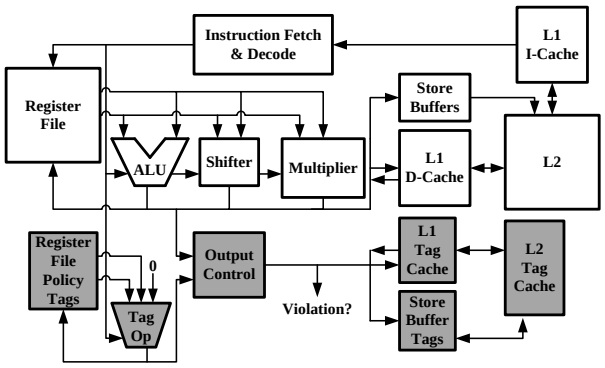



**Figure 7: One possible implementation of the DataSafe information flow tracking processor architecture. White boxes are existing components while grey boxes are new.**

**Table 12: The complexity of DataSafe's software and hardware modules in terms of source lines of code (SLOC).**

| | **Ruby** | **C** |
|---|---|---|
| Policy/Domain handler | 1197 | 207 |
| **Software** | **SPARC Assembly** | **C** |
| Base hypervisor | 37874 | 35066 |
| DataSafe hypervisor | 41657 | 51959 |
| Crypto | 0 | 16045 |
| **Hardware** | **SPARC Assembly** | **C** |
| Base OpenSPARC | 1050 | 51395 |
| DataSafe hardware | 0 | 3317 |

cache and memory overhead for storing the tags. For a 10-bit tag per 64-bit word used in our prototype, the storage overhead is 15.6% for the shadow memory, on-chip caches and the register file. Existing techniques for more efficient tag management [26] can be applied to reduce storage overhead. The tag storage includes 4 of the 6 new (grey) CPU components in Figure 7. The remaining 2 components (Tag Operation and Output Control) discussed above, are low complexity components.

# 7. CONCLUSION

We presented the DataSafe architecture for realizing the concept of self-protecting data. DataSafe enables owners of sensitive data to define a security policy for their encrypted data, then allow authorized users and third-party applications to decrypt and use this data, with the assurance that the data's confidentiality policy will be enforced and plaintext data will be prevented from leaking out of these authorized use sessions. Data is protected even if transformed and obfuscated, across applications and user-system transitions. Data is also protected when at-rest or in-transit by encrypted, policy-attached, DataSafe packages.

DataSafe hardware uses our enhanced dynamic information flow tracking (DIFT) mechanisms to persistently track and propagate data in-use, and to perform unbypassable output control to prevent leaking of confidential data. Because this is done in hardware, performance overhead is minimal. However, unlike previous hardware DIFT solutions, DataSafe's key novelty is in seamlessly supporting flexible security policies expressed in software, bridging the semantic gap between software flexibility and efficient hardware-enforced policies. DataSafe is also application independent, thus supporting both legacy and new but unvetted applications. This is often a practical necessity, since users have no means to modify third-party program executables. But more importantly, it provides the separation of data protection from applications, which we feel is the right architectural abstraction.

Self-protecting data, with unmodified legacy applications, may seem an unreachable goal, but we hope to have shown that it may be possible if we are willing to consider new hardware enhancements with a small trusted software base. We hope that DataSafe provides the architectural foundation over which multi-domain, multi-policy, end-to-end self-protecting data solutions can be further researched for distributed systems.

# 8. ACKNOWLEDGEMENTS

This work was supported in part by NSF CCF-0917134. We thank the anonymous reviewers and our shepherd, Radu Sion, for their comments which have helped to improve this paper. We also thank Si Chen for helping with the software performance measurements.

# 9. REFERENCES

[1] M. I. Al-Saleh and J. R. Crandall. On information flow for intrusion detection: what if accurate full-system dynamic information flow tracking was possible? In *Proceedings of the Workshop on New Security Paradigms*, pages 17–32, 2010.

[2] D. Brumley, J. Caballero, Z. Liang, J. Newsome, and D. Song. Towards automatic discovery of deviations in binary implementations with applications to error detection and fingerprint generation. In *Proceedings of USENIX Security Symposium*, pages 15:1–15:16, 2007.

[3] D. Champagne and R. B. Lee. Scalable architectural support for trusted software. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture*, pages 1–12, 2010.

[4] S. Chhabra, B. Rogers, Y. Solihin, and M. Prvulovic. Secureme: a hardware-software approach to full system security. In *Proceedings of the International Conference on Supercomputing*, pages 108–119, 2011.

[5] C. Y. Cho, D. Babić, P. Poosankam, K. Z. Chen, E. X. Wu, and D. Song. Mace: model-inference-assisted concolic exploration for protocol and vulnerability discovery. In *Proceedings of USENIX Security Symposium*, pages 10–10, 2011.

[6] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum. Understanding data lifetime via whole system simulation. In *Proceedings of USENIX Security Symposium*, pages 22–22, 2004.

[7] M. Dalton, H. Kannan, and C. Kozyrakis. Raksha: a flexible information flow architecture for software security. In *Proceedings of the ACM/IEEE International Symposium on Computer Architecture*, pages 482–493, 2007.

[8] J. S. Dwoskin and R. B. Lee. Hardware-rooted trust for secure key management and transient trust. In

*Proceedings of the ACM Conference on Computer and Communications Security*, pages 389–400, 2007.

[9] P. Efstathopoulos and E. Kohler. Manageable fine-grained information flow. In *Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems*, pages 301–313, 2008.

[10] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazières, F. Kaashoek, and R. Morris. Labels and event processes in the asbestos operating system. In *Proceedings of the ACM Symposium on Operating Systems Principles*, pages 17–30, 2005.

[11] M. Egele, C. Kruegel, E. Kirda, H. Yin, and D. Song. Dynamic spyware analysis. In *Proceedings of the USENIX Annual Technical Conference*, pages 18:1–18:14, 2007.

[12] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten. Lest we remember: cold boot attacks on encryption keys. In *Proceedings of USENIX Security Symposium*, pages 45–60, 2008.

[13] R. Iannella. Open digital rights language (ODRL), Version 0.5, Aug. 2000. `odrl.net/ODRL-05.pdf`.

[14] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the ACM/IEEE International Symposium on Computer Architecture*, pages 364–373, 1990.

[15] M. G. Kang, S. McCamant, P. Poosankam, and D. Song. DTA++: Dynamic taint analysis with targeted control-flow propagation. In *Proceedings of the Network and Distributed System Security Symposium*, 2011.

[16] R. B. Lee, P. C. S. Kwan, J. P. McGregor, J. Dwoskin, and Z. Wang. Architecture for protecting critical secrets in microprocessors. In *Proceedings of the ACM/IEEE International Symposium on Computer Architecture*, pages 2–13, 2005.

[17] D. Lie, C. A. Thekkath, and M. Horowitz. Implementing an untrusted operating system on trusted hardware. In *Proceedings of the ACM Symposium on Operating Systems Principles*, pages 178–192, 2003.

[18] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig. TrustVisor: Efficient TCB Reduction and Attestation. In *Proceedings of the IEEE Security and Privacy*, pages 143–158, 2010.

[19] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: an execution infrastructure for TCB minimization. In *Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems*, pages 315–328, 2008.

[20] V. Nagarajan, H.-S. Kim, Y. Wu, and R. Gupta. Dynamic information flow tracking on multicores. In *Workshop on Interaction between Compilers and Computer Architectures*, 2008.

[21] J. Park and R. Sandhu. The UCON$_{ABC}$ usage control model. *ACM Trans. Inf. Syst. Secur.*, 7(1):128–174, 2004.

[22] F. Qin, C. Wang, Z. Li, H.-s. Kim, Y. Zhou, and Y. Wu. Lift: A low-overhead practical information flow tracking system for detecting security attacks. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*, pages 135–148, 2006.

[23] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *Selected Areas in Communications, IEEE Journal on*, 21(1):5–19, Jan. 2003.

[24] L. Singaravelu, C. Pu, H. Härtig, and C. Helmuth. Reducing tcb complexity for security-sensitive applications: three case studies. In *Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems*, pages 161–174, 2006.

[25] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, N. James, P. Poosankam, and P. Saxena. BitBlaze: A new approach to computer security via binary analysis. In *Proceedings of the International Conference on Information Systems Security. Keynote invited paper.*, pages 1–25, 2008.

[26] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 85–96, 2004.

[27] M. Tiwari, H. M. Wassel, B. Mazloom, S. Mysore, F. T. Chong, and T. Sherwood. Complete information flow tracking from the gates up. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 109–120, 2009.

[28] Trusted Computing Group. Trusted Platform Module. https://www.trustedcomputinggroup.org/home.

[29] N. Vachharajani, M. J. Bridges, J. Chang, R. Rangan, G. Ottoni, J. A. Blome, G. A. Reis, M. Vachharajani, and D. I. August. RIFLE: An Architectural Framework for User-Centric Information-Flow Security. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*, pages 243–254, 2004.

[30] S. Vandebogart, P. Efstathopoulos, E. Kohler, M. Krohn, C. Frey, D. Ziegler, F. Kaashoek, R. Morris, and D. Mazières. Labels and event processes in the asbestos operating system. *ACM Trans. Comput. Syst.*, 25, December 2007.

[31] XrML 2.0 technical overview, version 1.0, March 2002. `www.xrml.org/reference/XrMLTechnicalOverviewV1.pdf`.

[32] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: capturing system-wide information flow for malware detection and analysis. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 116–127, 2007.

[33] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in histar. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, pages 263–278, 2006.