SGX Enforcement of Use-Based Privacy

Eleanor Birrell*†
Pomona College
Claremont, CA
eleanor.birrell@pomona.edu

Anders Gjerdrum‡
UIT The Arctic Univ. of Norway
Tromsø, Norway
anders.t.gjerdrum@uit.no

Robbert van Renesse§
Cornell University
Ithaca, NY
rvr@cs.cornell.edu

Håvard Johansen‡
UIT The Arctic Univ. of Norway
Tromsø, Norway
haavardj@cs.uit.no

Dag Johansen‡
UIT The Arctic Univ. of Norway
Tromsø, Norway
dag@cs.uit.no

Fred B. Schneider†
Cornell University
Ithaca, NY
fbs@cs.cornell.edu

ABSTRACT

Use-based privacy restricts how information may be used, making it well-suited for data collection and data analysis applications in networked information systems. This work investigates the feasibility of enforcing use-based privacy in distributed systems with adversarial service providers. Three architectures that use Intel-SGX are explored: source-based monitoring, delegated monitoring, and inline monitoring. Trade-offs are explored between deployability, performance, and privacy. Source-based monitoring imposes no burden on application developers and supports legacy applications, but 35-62% latency overhead was observed for simple applications. Delegated monitoring offers the best performance against malicious adversaries, whereas inline monitoring provides performance improvements (0-14% latency overhead compared to a baseline application) in an attenuated threat model. These results provide evidence that use-based privacy might be feasible in distributed systems with active adversaries, but the appropriate architecture will depend on the type of application.

KEYWORDS

Use-based privacy; privacy enforcement; SGX

ACM Reference Format:
Eleanor Birrell, Anders Gjerdrum, Robbert van Renesse, Håvard Johansen, Dag Johansen, and Fred B. Schneider. 2018. SGX Enforcement of Use-Based Privacy. In *2018 Workshop on Privacy in the Electronic Society (WPES'18), October 15, 2018, Toronto, ON, Canada.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3267323.3268954

*This work was done while Birrell was a graduate student at Cornell.
†Supported in part by AFOSR grant F9550-16-0250 and NSF grant 1642120.
‡Supported by the Research Council of Norway project numbers 250138, 263248, and 274451.
§Supported in part by NSF CSR 1422544, NIST 60NANB15D327 and 70NANB17H181, and gifts from Huawei, Facebook, and Infosys.

WPES'18, October 15, 2018, Toronto, ON, Canada

ACM ISBN 978-1-4503-5989-4/18/10…$15.00
https://doi.org/10.1145/3267323.3268954

1 INTRODUCTION

Current approaches to privacy in networked information systems are poorly suited to modern applications, where information is collected without user awareness, and data sharing and data analysis are pervasive. Our work explores the feasibility of enforcing an alternate view, sometimes called *use-based privacy* [7, 8, 26], which equates privacy with preventing harmful uses.

Instead of requiring informed consent from data subjects, use-based privacy assumes there has been a societal evaluation that has identified harmful uses. This evaluation presumably will have balanced potential harms and potential benefits of information use—and evaluated the countermeasures in place to prevent potential harms—to determine which uses should be deemed harmful. A system that avoids harmful uses is then considered to be *privacy-compliant*.

Use-based privacy differs from most previous views of privacy in three key ways:

- Use-based privacy policies do not depend on the preferences of the individual data subject but instead focus on collective norms.
- Use-based privacy policies describe how information may be used rather than limiting access or transmission.
- Use-based privacy policies impose restrictions on how both raw data and derived data may be used, and therefore govern information flow through a system.

The first aspect of use-based privacy is philosophically distinct from approaches such as notice and consent [9]—which emphasize informed consent by data subjects—and from technologies like P3P [11] that enable users to express individual preferences. That aspect is instead similar to the philosophy of contextual integrity [1, 27, 28], which defines privacy for personal information relative to an *appropriate context*. Whether a context is appropriate is presumed to be determined by socially-defined *informational norms*, which might depend on time, location, purpose, and/or participating principals. So contextual integrity, like use-based privacy, moves away from user-defined policies and informed consent, focusing instead on elimination of harmful uses (as defined by informational norms). The second and third aspects, however, distinguish use-based privacy from contextual integrity. Contextual integrity ignores how sensitive information is used as it flows through a networked information system and thus ignores derived data; it

focuses on mediating individual communications and evaluating whether each data transmission is authorized. These philosophical differences between use-based privacy and alternate views render existing technical solutions ill-suited for guaranteeing use-based privacy.

To ensure policy-compliance, use-based privacy policies must be enforced whenever a principal uses a value. This can be accomplished by (1) blocking unauthorized uses—*prevention*—or (2) logging all uses—*deterrence through accountability*. Both approaches involve monitoring accesses, and both require the monitor to have the appropriate use-based privacy policy and to be trusted to enforce that policy. The monitor gets the appropriate policy if use-based privacy policies are tied to the data whose use they govern as a *policy tag*; policy tags have been explored previously (e.g., [13, 23, 25, 40]). To guarantee that the monitor is trusted, however, we need some means for monitoring behavior by *service providers*—principals that receive and use data. Existing approaches to monitoring focus exclusively on read/write access control [6, 22] or on systems under the control of a single trusted authority [29, 34] and thus are unsuited for enforcing use-based privacy policies in distributed systems.

Recent developments in trusted hardware—e.g., Intel's Software Guard Extensions (SGX) [10]—offer a new basis for placing trust in a monitor or other program. Using SGX, an untrusted principal can provide a remotely-authenticatable proof or *quote* that attests some program is running or has produced a given output. In this paper, we investigate the feasibility of using Intel's SGX hardware as a root of trust, and we explore how we might leverage that root to implement use-based privacy. An overview of relevant SGX features is given in Section 2.3.

We explore three possible architectures for enforcing use-based privacy in distributed systems with adversarial service providers. In our *source-based monitoring* architecture, *data sources*—trusted principals that store user data—run the monitors. Applications (run by service providers) request data from data sources; only those applications that provide appropriate credentials (e.g., SGX quotes) can gain access to *sensitive* data (data that is limited by policy to particular uses). The source-based monitoring architecture provides strong privacy guarantees. It is also easily deployed; application developers do not need to handle or interpret policies, and enforcement is compatible with legacy applications. However, this architecture exhibits poor performance and incurs significant overhead for many applications. The architecture is described in Section 3 and performance measurements are given.

The performance limitations of source-based monitoring lead us to consider an alternative architecture called *delegated monitoring*, which improves throughput and reduces latency by locating the monitor at the service providers. Delegated monitors act as proxies for local applications and use SGX quotes to prove to a data source that they are instances of a valid monitor; local applications use SGX to locally authenticate with the delegated monitor in order to gain access to data that is limited by policy to particular uses. The architecture provides the same strong privacy guarantees while demonstrating significant performance improvements over source-based monitoring. However, delegated monitoring is less easily deployed than source-based monitoring, because service providers must run a delegated monitor and because local applications must interact with that monitor and store cached policies. This architecture is described in Section 4.

The primary shortcoming of the delegated monitoring architecture is noticeable latency overhead for applications that handle lots of data or that enforce policies for fine-grained data. To eliminate this overhead, we consider a final architecture, *inline monitoring*, which has the monitoring code inlined directly into a monitored service provider application. This final architecture offers the best performance, particularly for applications that handle lots of data or fine-grained policies. However, the architecture imposes a significant burden on application developers—programmers must implement their code with calls to the inlined monitor—and this approach is only able to guarantee privacy compliance in an attenuated threat model. This architecture is described in Section 5.

Given the trade-offs between deployability, performance, and privacy, we believe that the appropriate architecture will depend on the type of application. However, we view our results as positive evidence of the feasibility of enforcing use-based privacy in distributed systems using SGX.

2 BACKGROUND

Values originate from a data source: a data subject or a third-party data store that is trusted by the data subject. Each data source (1) associates an appropriate policy tag (specifying a use-based privacy policy) with each value, thereby creating a *tagged value*, and (2) distributes tagged values only in ways that ensure policy compliance.

Tagged values are received and processed by service providers. Service providers might themselves produce derived values, which must be given associated policy tags. We do not assume that service providers are trusted; they might attempt to use values in a manner that does not comply with the associated use-based privacy policy.

2.1 Threat Models

Use-based privacy policies specify use restrictions. Adversaries are service providers that try to use a tagged value in a manner that violates these restrictions. Threat models characterize assumptions about possible service provider behaviors:

> ***Accountable Service Providers.*** Service providers are rational principals that act to optimize some utility function; they might knowingly violate use-based privacy policies under certain circumstances, for example, to increase profits. The utility function gives significant negative weight to being detected in a policy violation, so an accountable service provider will not run code that results in a policy violation that some auditor might detect. It suffices to detect violations in order to guarantee policy compliance by accountable service providers.
>
> ***Malicious Service Providers.*** Service providers here might knowingly violate use-based privacy policies by running code that results in a policy violation—even if that violation might be detected. Such behavior must be prevented. A monitor that implements prevention is needed to enforce policy compliance by malicious service providers.

Accountable service providers are the appropriate threat model if service providers are subject to legal consequences or negative public relations. In other cases (e.g., if service providers can't be reliably identified or if they are irrational), service providers might not conform to the defining assumptions for accountable service providers and should instead be considered malicious.

2.2 Policy Language

We can express use-based privacy as *Avenance policies* [5]. Avenance is a policy language based on reactive information flow specifications [19]; Avenance policies are interpreted as sets of *privacy automata* in which the current state of each automaton gives use authorizations. Syntactically, Avenance policies are JSON encodings that can be parsed as lists of automata.

In the Avenance policy language, authorizations in a state s are specified by conjunctions and disjunctions of *authorization triples*: predicates expressed as triples (I,P,E), where I identifies an invoking principal, P denotes a purpose, and E identifies some executable binary. An executable type E should be used when the authorization depends only on the program binary; purpose type P should be used when the authorization depends on some binary-independent context. I may be defined as a single principal or may be a role, P may be drawn from a hierarchy of purpose labels, and E may be specified by a binary hash or by a type drawn from a hierarchy of program labels. I, P, or E may alternatively be the wildcard $*$, which matches all principals (resp., purposes, executables). *Compound components* I, P, or E are constructed using unions and intersections.

An authorization triple (I, P, E) specifies a predicate that allows a use if the use satisfies all three component sets: I, P, and E. A privacy automaton authorizes a use if the use is authorized by the conjunctions and disjunctions of authorization triples specified by the current state. And an Avenance policy authorizes a use if the use is authorized by all of its privacy automata.

Automata state transitions are associated with *environmental events*—which update the current authorizations associated with a particular value—or *synthesis events*—which define the current authorizations for derived values. These transitions together express *reactive* policies. For example, a user might specify that a derived value (created by combining that user's data with other users' data) may be used for any use, but the raw data may only be used to produce aggregate values. A privacy automata for this policy is shown in Figure 1. Reactive policies are highly expressive, since they can specify how the set of authorized uses changes as data are transformed. However, defining such policies is likely to require careful reasoning about the information flow through various problems; the challenges of defining Avenance policies that instantiate a high-level goal are beyond the scope of this work.

Avenance policies are implemented in Java by the `avenance` package [4] and in C by the library `libav` [3]. Each implementation defines classes (resp. structs) `AvAuthTriple`, `AvState`, `AvRule`, and `AvPolicy`. The class `AvPolicy` (resp. header file `av.h`) defines a public interface for parsing, creating, modifying, and serializing Avenance policies; an excerpt from the Java interface is shown in Figure 2.

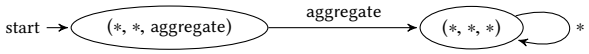


Figure 1: Example Avenance policy.

```
public class AvPolicy {
    public AvPolicy(String p){...}
    public AvPolicy(List<AvRule> rls){...}

    public List<AvRule> getRules(){...}
    public void addRules(List<AvRule> rls){...}

    public Boolean checkPermission(String i, String p,
                                   String e){...}
    public AvPolicy transition(String e){...}
}
```

Figure 2: The interface for the Java implementation of the Avenance policy language.

2.3 Intel SGX

Intel's Secure Guard Extensions (SGX) are an extension to the Intel x86 instruction set architecture. SGX uses chip-specific *hardware keys* to enable the construction of secure execution containers called *enclaves*; each enclave is isolated and supports data sealing, local attestation, and remote attestation.

Enclave Isolation. SGX enclaves provide confidentiality[1] and integrity for programs (and their data) running inside the enclave. This isolation is enforced by *processor reserved memory* set aside during boot. This memory is only accessible to SGX microcode and programs running within enclaves, and it is partitioned into 4k pages, which are collectively referred to as the *enclave page cache* (EPC). Pages in the EPC are exclusively associated with a particular enclave and can only be accessed by that enclave. Information that is paged-out of the EPC into regular DRAM is encrypted under a hardware-derived key.

Data sealing. SGX enclaves are uniquely identified by an SGX Enclave Control Structure, which includes a *measurement*—a 256-bit digest of a cryptographic log recording the build process for the enclave. This measurement is used by the key generation instruction (along with secrets embedded in the SGX chip) to produce hardware-derived sealing keys. Sealing keys for an enclave depend on both the measurement of the enclave and the hardware keys of the chip; sealed data can only be decrypted by the enclave that originally sealed it. Data sealing can provide confidentiality and integrity for audit logs and for tagged values that will be temporarily stored or handled outside the enclave.

Local Attestation. Enclave measurements are also used for local attestation, which allows one enclave to authenticate the program that is running in another enclave. Local attestation (between enclaves) uses a hardware-signed (HMAC'd) copy of the enclave

[1] Side-channel attacks that compromise confidentiality of SGX enclaves have been identified [20, 41]; we assume such attacks cannot undermine the confidentiality of tagged values handled by authorized enclaves.

measurement—the *report*—combined with a Diffie-Hellman key exchange protocol to prove the identity of the program in one enclave to the second enclave. Local attestation is used for local program authentication.

Remote Attestation. SGX implements remote attestation using local attestation together with a pair of dedicated, Intel-authored enclaves: a *provisioning enclave* and a *quoting enclave*. The provisioning enclave requests an attestation key from Intel and stores it sealed under a key that can only be derived by Intel-authored enclaves. The quoting enclave retrieves the attestation key, verifies the measurement using local attestation, and signs the measurement together with an optional message; the resulting signed measurement-message pair, called a *quote*, can be verified by a remote principal using Intel's Attestation Service.

Remote Authentication. Because communications between an application enclave and the quoting enclave are mediated by an untrusted (i.e., non-enclave) application, quotes can be replayed by any program. To mitigate this threat, our remote attestation protocol requires the application enclave to fetch an application secret $\langle s_1, s_2 \rangle$ from the remote server. The server must be able to authenticate valid secrets (in our implementation, $s_2 = H(s_1; k_{DS})$, where k_{DS} is a secret key unique to data source *DS*). An application enclave sets s_2 as the message used during measurement generation and then requests a quote with that measurement, resulting in a quote $q(s_2)$ that contains that message s_2. To perform remote authentication, the application enclave sends the pair $\langle s_1, q(s_2) \rangle$ to the remote server. The server authenticates the secret, authenticates the quote with Intel's Attestation Service, and then uses the authenticated credentials to make an authorization decision.

3 ENFORCEMENT BY SOURCE-BASED MONITORING

The first step in designing an enforcement architecture for use-based privacy is to decide which principal will be trusted to perform the monitoring. Principals are either data sources or service providers; a monitor can be run at either. Since data sources are trusted, it is natural to have data sources run the monitors. In this *source-based monitoring* architecture, SGX can be used to determine which applications (running remotely at a service provider) are authorized to use a given value. Assuming that sensitive values can be processed only by a standard set of data analytics functions[2], a source-based monitor can distinguish between authorized and unauthorized applications and, therefore, can enforce use-based privacy in the presence of malicious service providers. Moreover, with all policy enforcement performed at the data source, application developers do not need to handle policies or explicitly interact with policy mechanisms, and policy enforcement is compatible with legacy applications.

3.1 Designing a Source-based Monitor

Applications run by a service provider are decomposed into an *untrusted app*—run natively—and zero or more *enclave apps*—run inside SGX enclaves. Each app may issue requests $\langle r, x, c \rangle$ to a data source, where r is the type of request (e.g., `GET values`), x is a reference to the requested data (required for requests that retrieve values), and c is a set of credentials. Traditional authentication tokens—e.g., OAuth tokens or signed statements—can attest to the invoker type I and the purpose type P; we use SGX quotes as credentials for the executable type E, as described in Section 2.3. Upon receiving a request, the monitor validates the request: it retrieves the requested values (and their policy tags) from the data store and then constructs a policy-compliant response. This architecture is depicted in Figure 3a, and details (discussed below) are shown in Figure 4.

To construct a privacy-compliant response to a request for data, the monitor invokes an *authentication layer* to authenticate the *request credentials* and determine the *use type*—an authorization triple (I, P, E)—for the application that issued the request. We consider two possible approaches. In a *prevention-based* monitor, the authentication layer compares the authenticated credentials to a whitelist of known credentials in order to determine the use type. This results in a monitor that enforces privacy compliance with malicious adversaries. Note that a prevention-based monitor is implicitly assumed to know the functionality of all enclave apps (and their quotes) in advance, and the pre-determined mapping between quotes and use types is assumed to be error-free. In a *detection-based* monitor, the authentication layer creates a log entry—including an identifier for the service provider, the authenticated credentials, and the claimed use type—and then accepts the claimed use type. Because a service provider could lie about the use type, a detection-based monitor does not guarantee privacy compliance by malicious adversaries. Observe, however, that the audit log ensures that incorrect use types can be detected after the fact, so a detection-based monitor is sufficient to guarantee privacy compliance in the presence of accountable adversaries.

After determining the use type, the monitor retrieves the requested values (and their policy tags) from the data store. It then invokes a *authorization layer*, which compares the use type to the use-based privacy policy defined by the policy tags—which defines authorized use types—and constructs a policy-compliant response. The details of how this response is constructed are implementation-specific and are discussed in Section 3.2.

Since use-based privacy expresses restrictions on how derived values may be used, the monitor is also responsible for computing derived policies and associating them with derived values. To do so, the monitor maintains a *taint store* that maps applications to the Avenance policy(s) of the values that that application has received. Each time the monitor sends values to an application, it adds the corresponding policies to the taint store entry for that application. When the monitor receives a new value x from an application, it first invokes the authentication layer to authenticate the request credentials and determine the use type (I, P, E) and the application identifier *aid*. It then looks up the policy(s) ρ associated with *aid* in the taint store, invokes the transition triggered by the executable type E(*aid*) to produce a derived policy ρ', constructs a tagged value $\langle x, \rho' \rangle$, and stores the new tagged value in the data store.

[2]The popularity of common data analytics packages including Scipy and Scikit-learn provides evidence in favor of this assumption. Nonetheless, if future work disproves this assumption, the enforcement mechanisms discussed in this work will continue to provide privacy guarantees in the presence of accountable service providers.

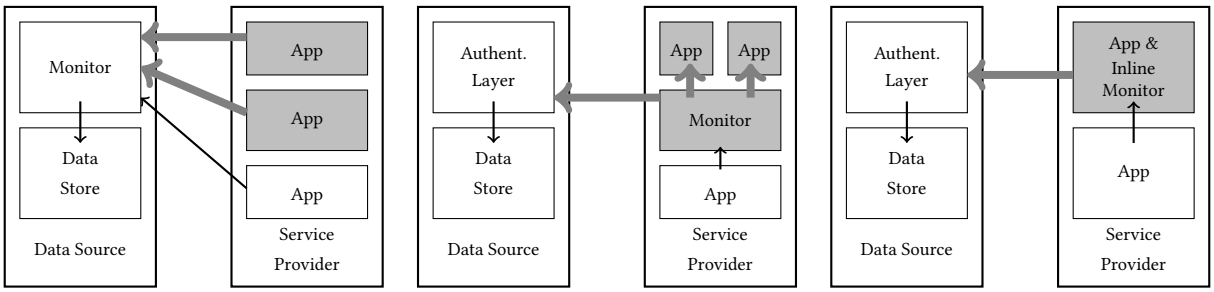


Figure 3: An overview of the different architecture designs. The direction of each arrow indicates which principal instigates communication between two components of the system. SGX enclaves are shown in gray; wide gray arrows indicate that a program has authenticated using an SGX report (local attestation) or quote (remote attestation).

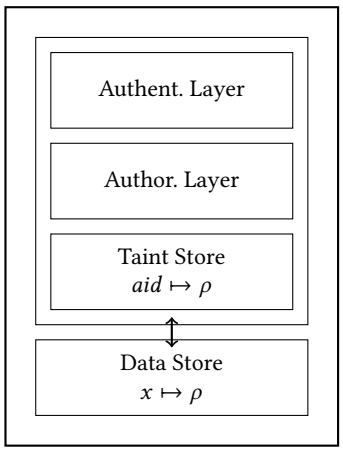


Figure 4: Detailed design for a data source that implements source-based monitoring.

3.2 Implementation of Source-based Monitoring

We implemented a data source that instantiates source-based monitoring on an existing mobile health platform called Ohmage [35, 39]. Ohmage is an open-source system designed to facilitate distributed data collection and analysis for health studies and applications—an ideal candidate for use-based privacy. It has been used for dozens of real-world studies and also serves as the backend for several production applications. Ohmage is designed with a classic three-tiered architecture comprising a back-end database, a server component implemented in the Spring Boot framework [38], and a family of front-end mobile applications. Our data source is implemented in 7634 lines of Java on top of the existing Ohmage server.

Data Store. The backend of Ohmage is a secure, Open mHealth-compliant data store that can be accessed through an API. The data store operates on *datapoints*, each comprised of header information (id, schema, time, source) and a JSON-encoded body; datapoints are classified by *schema*. The API allows operations for storing and retrieving datapoints: `GET datapoints/{id}`, `GET datapoints`, `POST datapoints`, and `GET datapoints/scope`. We extend the Ohmage data store to store tagged values and enforce policy tags by storing values as datapoints in Ohmage and storing tagged policies in a local MySQL database.

Policy Association. Our implementation supports both discretionary (data-subject defined) policies and mandatory (admin defined) policies through a new `POST policy` API call, which allows data subjects to modify the policy for their own datapoints and allows admins to modify the mandatory policies applied to all stored datapoints. The API has operations to modify the policy for a single specified datapoint or update the set of *preference rules*—policies that apply to all future incoming datapoints that match the specified schema. Requests to store datapoints can also specify an existing policy using the optional HTTP header `AvPolicy`.

Policy Granularity. Avenance policies could be associated with atomic values (e.g., integers) or with structured values (e.g., health records) under control of a single principal; policies could also be associated with aggregate objects containing information about many different users. Our data source enforces policies at the granularity of individual datapoints—in which case a request for multiple datapoints returns only the authorized datapoints—or at the granularity of datasets—in which a request for multiple datapoints is authorized only if all requested datapoints are authorized. The granularity can be configured at runtime.

Policy Enforcement. To ensure privacy compliance, our data source only accepts requests received over a TLS connection and accompanied by request credentials. Credentials might include an OAuth token, a purpose label, and/or an SGX quote. OAuth credentials are authenticated by the Ohmage authentication service and then used to lookup the service provider identity *spid*—a unique identifier associated with an OAuth client secret. Purpose labels are not authenticated; they are instead interpreted as credentials of the form "U says P" for the user U defined by the OAuth token and some purpose type P. SGX quotes are authenticated with the Intel Attestation Service and then cached; the quote is also used to define the application id *aid*. Finally, the data source determines

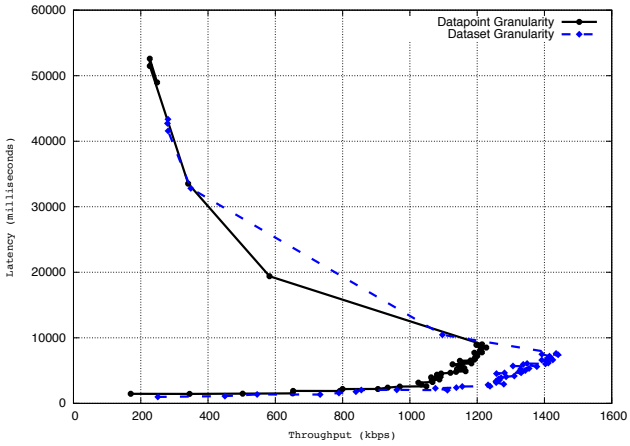


Figure 5: Latency and throughput of a data source with source-based monitoring as a function of the number of concurrent requests, ranging from 1 to 50 concurrent requests. The solid black line shows performance when the data source enforces datapoint-granularity policies and the dashed blue line shows performance for dataset granularity.

the executable type: if configured for prevention-based enforcement it compares the quote to a whitelist of trusted enclaves, if configured for detection-based enforcement it accepts the claimed enclave type after logging the request. The enforcement mode is configured at runtime; if the mapping between credentials and use types is not known in advance for all users and all applications, the monitor should be configured for detection-based monitoring. The data source then performs monitoring as described in Section 3.1.

3.3 Evaluating Source-based Monitoring

We deployed our data source with source-based monitoring on Amazon EC2 T2.small instance with an Intel Xeon E5-2676 2.4 GHz CPU and 2GB of memory running Ubuntu 14.04 LTS (kernel version 3.13.0).

To evaluate performance, we measured latency and throughput of the data source responding to a `GET datapoints/scope` request for 500 datapoints. We tested the performance as the data source handled between 1 and 50 concurrent requests. As shown in Figure 5, implementation choices—for example, whether policies were associated with values at the granularity of individual datapoints or for the full dataset—did impact the performance. But all implementations overloaded at a relatively low load (less than 50 simultaneous requests), after which throughput collapsed and latency drastically increased.

We also measured the performance of source-based monitoring for a common use case [5]: user preferences, privacy regulations, and/or corporate privacy policies restrict uses for raw data but allow derived values (e.g., anonymized values, encrypted values, or aggregated values) to be used more liberally. One such policy is depicted in Figure 1. A privacy-compliant service provider might first request the raw data, generate the derived values, and then use the derived values.

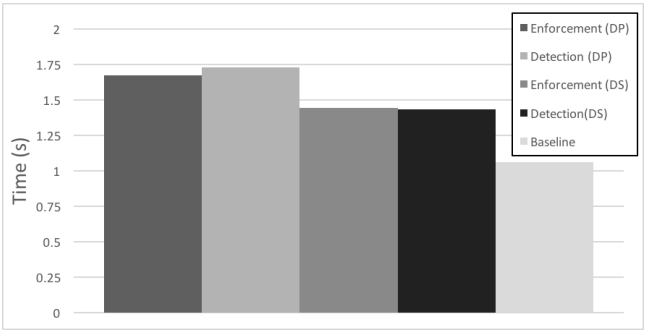


Figure 6: Performance of the PMSys averaging function with source-based monitoring.

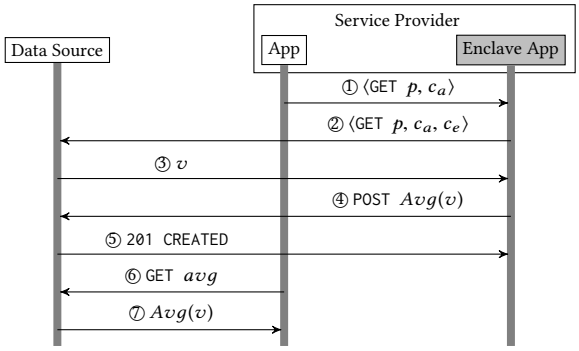


Figure 7: Protocol for the PMSys averaging function in a source-based monitoring architecture.

To evaluate performance for this use case, we ported one such application, called PMSys [32, 33], to run on the source-based monitoring architecture. PMSys is a mobile and web-based application developed jointly at Simula Research Laboratory and UIT The Arctic University of Norway that performs physiological evaluation and training-load personalization for soccer players. PMSys collects data about player mood, sleep patterns, physical fitness, and injuries and displays aggregate statistics (including average) to authorized coaches. These data are subject to a contractually-defined privacy policy negotiated with the players (who all are members of elite clubs and national teams in Norway, Sweden, and Denmark) and to relevant national and EU privacy laws that restrict data use and data sharing.

We measured the end-to-end latency of the PMSys averaging function on synthetic data matching the PMSys data collected for one month.[3] To eliminate network bottlenecks, we ran our data source on a dedicated Amazon EC2 R4.large instance with an Intel Xeon dual-core E5-2686 2.3 GHz CPU and 15GB of memory running Ubuntu 14.04 LTS (kernel version 3.13.0). We deployed the application on an OptiPlex 5040 with an SGX-enabled Intel Core i5-6500 3.20 GHz CPU and 16GB of memory running Ubuntu 14.04 LTS (kernel version 4.4.0). As shown in Figure 6, this averaging function

[3]Using actual data from the production system would have been incompatible with the existing terms of service and Norwegian data protection laws.

experiences 35-62% overhead compared to a baseline averaging function with no policy enforcement. However, poor performance is unsurprising given the number of round-trips required; as shown in Figure 7, this averaging function requires three round trips to the server in order to enable the source-based monitor to mediate access to the derived (average) value. Note that for the experiments with datapoint (DP) granularity, the overhead due to logging causes detection-mode to be more expensive than enforcement mode—so that design choice would be reasonable only in cases where predetermining the use type of some apps is infeasible. Dataset (DS) granularity generates shorter log entries, reducing the overhead for enclave exit and log writing and thereby rendering the performance difference between detection-mode and enforcement-mode negligible.

4 ENFORCEMENT BY DELEGATED MONITORING

To mitigate the throughput bottleneck imposed by the monitor in a source-based monitoring architecture, we turn to an alternative design. In a *delegated monitoring* architecture, service providers run the monitors in dedicated SGX enclaves, which enables each monitor to authenticate itself to the data source. We assume that there will only be a small number of implementations of delegated monitors, so a data source can whitelist the credentials for delegated monitors to ensure that tagged values are shared only with valid instances of a delegated monitor. SGX is used here also to locally determine which applications run by the service provider are authorized to use values, and it is used to provide confidentiality and integrity for tagged values handled by a delegated monitor. As before, we assume that sensitive values can only be processed by a standard set of data analytics functions, so a delegated monitor can distinguish between authorized and unauthorized applications and, therefore, can enforce use-based privacy in the presence of malicious service providers.

A delegated monitoring architecture requires each service provider to run a monitor. Because each monitor is responsible for mediating the requests from just one service provider, the delegated monitoring architecture eliminates the performance bottleneck incurred by a source-based monitor. This architecture also offers an opportunity to mitigate the second performance drawback of source-based monitoring: the number of round-trips required for a typical application. In the source-based architecture, it is necessary to send all derived values to the data source, because the monitor (run by the data source) needs to mediate all requests, including requests for derived values. Because a delegated monitor is run locally by a service provider, those round trips are no longer necessary. Instead policies pertaining to derived values can be determined by the local monitor, and derived tagged values can be cached locally using SGX sealing. This design improves performance at the cost of introducing a burden on application developers, who must now handle tagged values and must modify any legacy applications.

4.1 Designing a Delegated Monitor

Delegated monitors run by a service provider act as a proxy for untrusted applications: they issue requests to a data source and they mediate messages to and from enclave applications. This architecture is depicted in Figure 3b, and an example sequence of interactions is depicted in Figure 8. The design of the delegated monitor is the same as the source-based monitor design depicted in Figure 4.

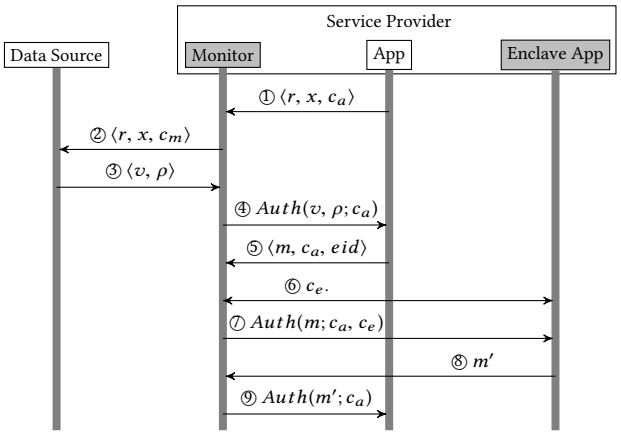


Figure 8: Example interactions with a delegated monitor.

Delegated monitors accept requests $\langle r, x, c_a \rangle$ from untrusted applications, where r is the type of request (e.g., `GET values`), x is a reference to the requested data (required for requests that retrieve values), c_a is a set of invoker credentials (e.g., message ① in Figure 8). A monitor then replaces the credentials c_a with a set of monitor credentials c_m and issues the modified request ② to a data source in the form $\langle r, x, c_m \rangle$. Upon receiving the request, the data source authentication layer checks the monitor credentials and then issues a response ③. After a monitor receives a response from a data source, it mediates the response to enforce policy compliance. If the response contains no tagged values (e.g., an acknowledgment), then it forwards the response to the untrusted application. If the response contains tagged values $\langle v, \rho \rangle$, then the monitor invokes an authorization layer, which compares the use type of the untrusted application (I, P, *null*)[4]—determined by the internal authentication layer from the application credentials c_a—to the use-based privacy policy defined by the policy tags and constructs a policy-compliant response ④ $Auth(v, \rho; c_a)$. The details of how this response is constructed depend on the granularity of the policy tags returned by the data source, but the monitor forwards authorized values to the untrusted application in plaintext and encrypts all other values using an SGX sealing key.[5]

Delegated monitors also mediate messages between untrusted applications and trusted applications. Communication is always initiated by an untrusted application, which sends a message $\langle m, c_a, eid \rangle$ where m is either a set of tagged values or sealed tagged values, c_a

[4]Note that the use type cannot define an executable type because untrusted applications do not run inside SGX enclaves and therefore cannot produce the necessary credential—a quote—for an executable type E.

[5]This design eliminates unnecessary round-trips to the data source by caching encrypted copies of tagged values with any application that is not authorized to use those values. This caching might violate a use-based privacy policy unless we interpret policies as allowing encrypted copies of tagged values to be used by any principal in any way. We consider such an interpretation consistent with existing user preferences and legal requirements.

is a set of invoker credentials, and *eid* is an enclave application (e.g., message ⑤ in Figure 8). The monitor authenticates the invoker credentials to determine the invoker type I and purpose P, and then authenticates the enclave application *eid* ⑥ and determines the executable type E. It then invokes the authorization layer, which compares the use type (I, P, E) to the use-based privacy policy defined by the (decrypted, if necessary) policy tags, constructs a policy compliant message ⑦ $Auth(m; c_a, c_e)$ (using SGX sealing, if necessary), and forwards the resulting message to enclave *eid*. It also updates the taint store entry for *eid* to include the policies for any tagged values sent to *eid* in plaintext. When the monitor receives a response—a set of values ⑧ m'—it looks up the policy ρ associated with *eid* in the taint store, invokes the transition triggered by the executable type E to produce a derived policy ρ', and constructs a new set of tagged values from m' and ρ'. Finally, it invokes the decision engine to determine whether ρ' authorizes the untrusted application (I, P, *null*), constructs a policy compliant response ⑨ $Auth(m'; c_a)$ (using SGX sealing, if necessary), and forwards that response to the untrusted application.

4.2 Implementation of Delegated Monitoring

Data Source. We modified our data source to work in concert with delegated monitoring. It retains the same data store and policy association API as in the source-based monitoring architecture, and it, too, can be configured to construct tagged values at either datapoint granularity or dataset granularity.

Instead of mediating requests to enforce privacy compliance, the modified data source uses an authentication layer to only accept requests over TLS from delegated monitors. The data source authentication layer authenticates credentials $\langle s_1, q, e\rangle$ as describe in Section 2.3 and determines the use type E using the same authentication mechanism—either prevention-based or detection-based—as the source-based monitor in Section 3. If the requester successfully authenticates as a delegated monitor—denoted by the executable type E = `policyrm`—the data source returns the requested tagged values.

Delegated Monitor. We implemented a delegated monitor in 1149 lines of C/C++ that runs as a dedicated SGX enclave. On initialization, the monitor establishes its credentials $\langle s_1, q, e\rangle$ for use in remote program authentication, as described in Section 2.3: it retrieves an application secret (s_1, s_2) from the data source, generates a quote q with message s_2, and defines E = `policyrm`. All subsequent requests to the data source are sent over TLS using a version of the `mbedtls` client ported to run inside an SGX enclave [42]; these request include the monitor credentials as a message header.

Policy Granularity. Like the source-based monitoring implementation, our implementation of delegated monitoring supports policy tags at two different granularities: individual datapoints and datasets.

Policy Enforcement. For efficiency, our delegated monitor exclusively implements prevention-based monitoring; it determines use types (I, P, E) by comparing invoker and enclave credentials to a whitelist of known types.

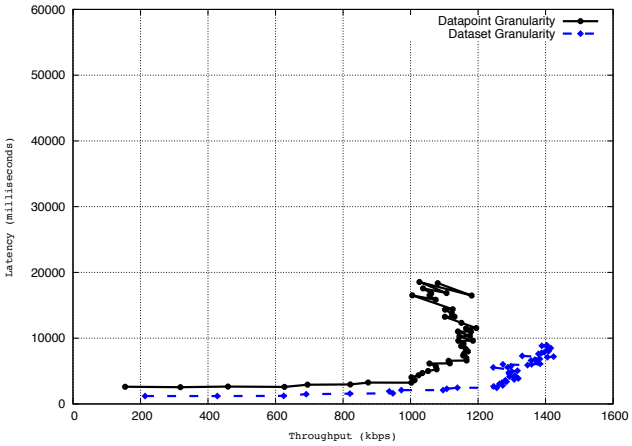


Figure 9: Latency and throughput of a data source with client-side monitoring (delegated monitoring or inline monitoring).

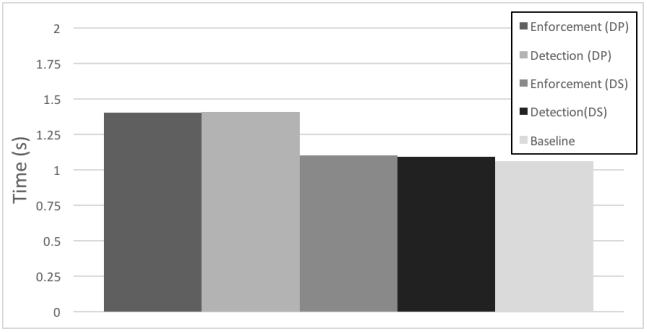


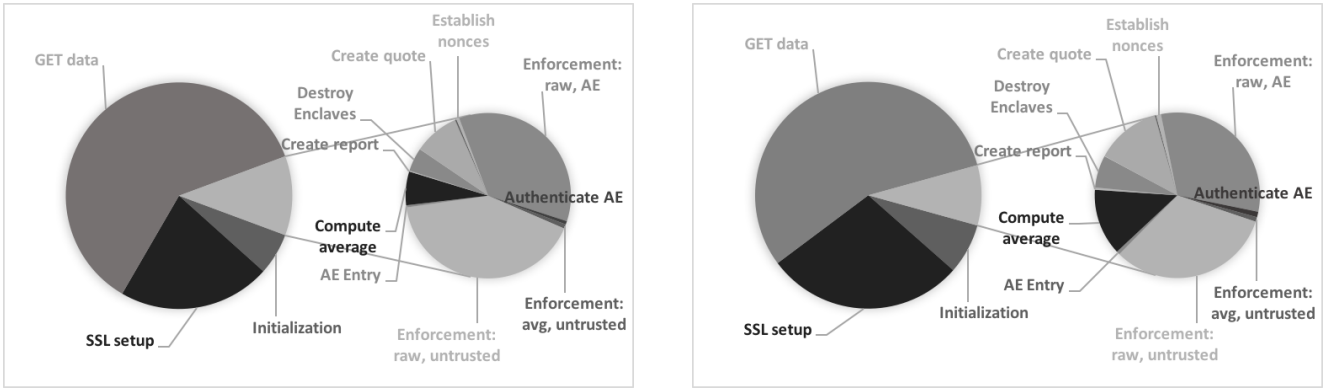
Figure 10: Performance of the PMSys averaging function with delegated monitoring.

4.3 Evaluating Delegated Monitoring

We deployed our data source with delegated monitoring on the same Amazon EC2 instances and the same local client that we used to evaluate source-based monitoring.

We evaluated the performance of the data source in the delegated monitoring architecture by reproducing the latency and throughput experiment we ran for the source-based monitoring architecture. The simplified authentication layer run by the data source eliminates the throughput bottleneck incurred by a source-based monitor; this improved performance is evident for both datapoint granularity and dataset granularity (Figure 9). Observe that implementing policy association at the granularity of datasets results in a moderate increase in throughput and a significant decrease in latency, as compared to the datapoint-granularity implementation.

To evaluate the performance of the delegated monitor for the common aggregate-then-use case, we ported the PMSys application—which requests values, computes the average in an SGX enclave, and then uses the average in an untrusted application—to run in the delegated monitoring architecture. The reduced number of



(a) Datapoint granularity **(b) Dataset granualarity**

Figure 11: Breakdown of the latency of the PMSys averaging function with delegated monitoring.

round trips significantly improves the performance of the averaging function, as compared to the source-based monitoring architecture (Figure 10); there is a 3% overhead for dataset-granularity and the overhead is cut in half for datapoint-granularity enforcement. Significant components of the remaining overhead are due to the cost of sealing cached values (Figure 11). The majority of the latency is due to enclave initialization, SSL negotiation, and fetching the raw data; these costs are fixed. The majority of the remaining latency can be attributed to the cost of enforcing policy compliance when caching raw data with the untrusted application (which requires sealing the data) and when transferring cached, raw data to the averaging enclave (which requires unsealing the data). This cost is likely to increase for applications that handle more data (much of the difference in latency between datapoint and dataset mode is due to the increase space required to store policies at the granularity of individual datapoints).

5 INLINE MONITORING

To eliminate the latency overhead imposed by sealing cached tagged values, we propose yet a third design. In an *inline monitoring* architecture, the service provider performs monitoring inline with a monitored application. The inline monitor provides an API of monitor calls, and each service provider augments their application code with appropriate calls to that API. The inline architecture enables applications to process tagged values within a single enclave, eliminating the need to seal cached values, but it introduces a significant burden on application developers, who must now instrument their code with monitor calls.

To ensure policy compliance, a data source must send tagged values only to correctly-inlined applications running inside SGX enclaves. With many correctly-inlined applications, a data source cannot be expected to maintain a database identifying all. Prevention-based monitoring—in which the data source authentication layer maintains a whitelist of authorized enclaves—is therefore infeasible.[6] Instead, we focus on detection-based monitoring, and we design and implement an inline monitor that will ensure privacy compliance by accountable service providers. Note that this design effectively places trust in application developers; incorrect annotations due to developer errors might result in policy violations that will only be detected after the fact.

5.1 Designing an Inline Monitor

An inline monitor should handle policies for tagged values, and it should provide an API with calls for storing polices, enforcing policies, and generating policies for derived values. We therefore designed an inline monitoring library that enables service providers to add policy monitoring code to existing enclave applications.

On initialization, the monitor creates a *policy store*, which stores tagged values; tagged values can subsequently be added to or deleted from the policy store. The monitor automatically computes policies for derived values based on program annotations, which label the executable type `E` of the function that generates the derived value, and adds derived tagged values to the policy store. Observe that there is no authentication of the executable type `E`; however, an application is only considered to be correctly-inlined if all derivation functions are annotated with the correct executable type `E`.

The inline monitor provides monitor calls that should be used to label sections of code that implement particular executable types and annotations that should be invoked when tagged values are used. When a tagged value is used, the inline monitor enforces the associated policy; the details are implementation-specific, but might use either prevention-based or detection-based enforcement. Again, there is no assurance that these labels are correct, but an application is only considered to be correctly-inlined if all uses are correctly labeled.

To use the inline monitor, a service provider adds monitor calls to their application code. Correctly-monitored code must initialize the monitor, must add all tagged values to the policy store, must

[6]The preceding architectures do not have this constraint. In either a source-based monitoring architecture or a delegated monitoring architecture, all service providers might use a common set of data analysis enclave applications to manipulate tagged values; in a delegated monitoring architecture, the data source must also authenticate the delegated monitor, but each service provider runs an instance of the same (or one of a small number of) monitor enclave, so prevention-based enforcement is a feasible option.

polstore * init_polstore(int *m*, char **l*)	Initialize a polstore with enforcement mode *m* and logfile name *l*.
int store_policy(polstore **s*, void **v*, char **p*)	Create a polstore entry in *s* for the value at location *v* and associate it with policy serialized as *p*.
pol * retrieve_policy(polstore **s*, void **v*)	Return the policy from *s* associated with the value at location *v*.
int delete_policy(polstore **s*, void **v*)	Delete the entry associated with *v* from polstore *s*.
int check_policy(polstore **s*, void **v*, char **i*, char **p*, char **e*)	Return a boolean indicating whether the use (i, p, e) is currently permitted by the policy associated with *v*.
void change_use(polstore **s*, char **i*, char **p*, char **e*, int *b*)	Add (if $b = 1$) or remove (if $b = 0$) use type (i, p, e) from the set of current uses for polstore *s*.
void *use(polstore **s*, void **v*)	Use the value *v* for the current use(s) defined in polstore *s*.
int trans(polstore **s*, char **i*, char **p*, char **t*, int *n*, void **ins*[], void **o*)	Use the *n* values *ins*[] for use (i, p, t), where *t* is a transitions type, and associated the derived policy with the output stored in *o*.

Figure 12: Monitoring API for our inline monitor implementation.

correctly label all sections of code according to their use type, and must label all uses of tagged values. To receive values from a data source, an application must perform remote program authentication and must provide credentials that authenticate the service provider. The data source authentication layer is identical to that used in the delegated monitoring architecture when configured for detection-based monitoring: it authenticates the credentials, creates a new log entry, and then returns the requested tagged values.

5.2 An Implementation of Inline Monitoring

We implemented an inline monitor as a C library in 2701 lines of code; it can be compiled to run inside SGX enclaves. The full API supported by our implementation is given in Figure 12.

Inline Monitor. The inline monitor implements the policy store as an in-memory list; it uses the memory address as an identifier for a value and maps addresses to policies. Tagged values can be added or removed from the policy store using API calls store_policy and remove_policy. Derived values should be added to the policy store using the transition call trans, which automatically defines the derived policy based on the declared inputs and synthesis event and which associates the derived policy with the derived value in the policy store.

When a tagged value is used, that use should be accompanied with a monitor call use that will enforce privacy compliance. This enforcement can be prevention-based or detection-based; details are discussed below. Authorization decisions are determined by the current set of use types. Uses are labeled using change_use to mark the beginning and end of code segments that implement a particular use and use use to indicate when particular tagged values are used.

The inline monitor also includes a check_policy call; a policy-compliant application that uses the inline monitor in prevention mode should call check_policy immediately prior to any call to use and only proceed if the use is authorized.

To write a new log entry, the monitor encrypts the log entry using SGX sealing and then exits the application enclave to write the encrypted entry to the logfile.

Policy Granularity. Our inline monitor can be deployed to enforce privacy compliance at any level of granularity. The application developer may choose what granularity to add tagged values to the policy store.

Policy Enforcement. An inline monitor can either prevent unauthorized uses—using the program annotations as use types—or simply log all interactions with the monitor; our implementation supports both and can be configured using a compiler flag. Since the data source implements detection-based monitoring, this is an implementation choice that can be configured for performance optimization or to minimize programmer burden; it has no effect on the privacy guarantees provided.

When the monitor is configured with logging, it generates a secure audit log that contains a record for each invocation of a monitor call that affects the state of the monitor—store_policy, delete_policy, trans, and change_use—and each time enforcement occurs—each invocation of use. A record contains the monitor call, the arguments to the monitor call, and a record id (a counter that is increased with each record). Each entry is encrypted using SGX sealing and then written to a logfile stored in the local file system. Log records cannot be modified because SGX sealing ensures integrity, and the counter ensures that log records cannot be deleted. Note that an auditor uses the application to retrieve (and unseal) the audit log—the correctness of this function is ensured because the data source logs the application quote—and the retrieval function includes the current counter value, so truncations of the audit log can also be detected.

5.3 Evaluating Inline Monitoring

The inlined applications are compatible with the data source implemented for delegated monitoring, so the data source exhibits the same performance shown in Figure 9.

To evaluate the performance of the inline monitor, we ran a series of microbenchmarks that evaluate the costs of various library calls. These results are shown in Figure 13. We find that the detection-based implementation has higher latency due to the additional cost of encrypting log entries with SGX sealing, exiting the enclave,

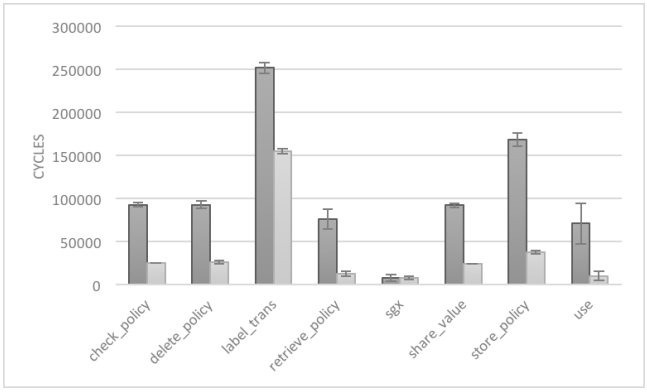


Figure 13: Performance of inline monitoring library calls. Results with logging are in dark gray; results with prevention-based enforcement are in light gray.

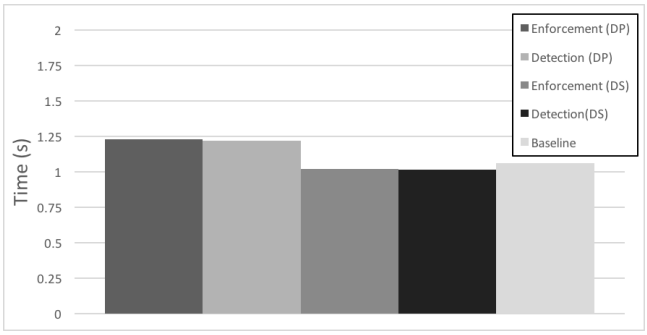


Figure 14: Performance of the PMSys averaging function with inline monitoring.

and writing the log entries to the logfile. Note that the delegated monitor and the inline monitor use the same implementation of common functions—e.g., computing policies for derived values—to facilitate comparison.

To evaluate the performance of the inline monitor for the common aggregate-then-use case, we ported the PMSys application—which requests values, computes the average in an SGX enclave, and then uses the average in an untrusted application—to run in the inline monitoring architecture in prevention mode. As shown in Figure 14, inline monitoring is within the error margin of the baseline system for dataset-granularity enforcement—small differences are due to various uncontrollable sources of variance introduced by Amazon EC2 instances—and it offers significantly improved performance (14% overhead) with datapoint granularity. However, this performance comes at the cost of increase the burden on application developers and attenuated privacy guarantees.

6 RELATED WORK

Use-based Privacy. Use-based privacy was first introduced by Cate [7] as a solution to the shortcomings of "notice and consent" and the underlying guidelines—the Fair Information Practice Principles [9]—which defined acceptable standards for handling sensitive data. Observing that users rarely make use of either opt-ins or opt-outs and typically don't make informed decisions about access to their data, Cate proposed a new approach. His work explored the legal and philosophical implications of use-based privacy; the feasibility of a technical regime for expressing or enforcing use-based privacy was not addressed.

The Avenance policy language [5] expresses use-based privacy as summarized in Section 2.2. Previous implementations of the Avenance language [3, 4] provide detection-based enforcement, but those privacy guarantees depend on trusting service providers to deploy the enforcement mechanism.

Alternate Privacy Regimes. Many systems have been developed with the goal of expressing and enforcing privacy. However, none were intended to enforce use-based privacy. Alternate approaches either focus exclusively on private information transmission rather than controlling usage as information flows through a networked information system (e.g., [14, 27, 29, 31]) or fail to exhibit all key attributes required for use-based privacy (e.g., [15, 37]).

Contextual Integrity [27] is a philosophical approach to privacy that has been formalized as a logic for reasoning about privacy [1]. Because contextual integrity defines privacy relative to socially-determined informational norms, contextual integrity can be interpreted as a special case of use-based privacy that focuses on data collection and data sharing. Transmissions are authorized when they occur in an appropriate context, as determined by social norms. The emphasis on a societal determination of acceptable or non-harmful uses (rather than informed consent or data minimization) is closely aligned with the philosophy of use-based privacy. However, the exclusive focus on data transmission, and the lack of restrictions on derived values, render existing enforcement mechanisms inapplicable for use-based privacy.

Differential privacy [14] classifies a response to a database query as a privacy violation unless the algorithm used to generate the response satisfies a specific statistical property (viz., ε-differential privacy). This definition has been formalized and implemented as an extensible platform for privacy-preserving data analysis [24]. However, differential privacy, like contextual integrity, focuses exclusively on defining authorized transmissions. This approach does not support general policy synthesis for derived values, and it does not include environmental events, sticky policies, or obligations. So like contextual integrity, mechanisms for enforcing differential privacy cannot be used to enforce use-based privacy.

Datta et al. [12] propose an alternative approach termed *use privacy*, which restricts the use of protected information types and their *proxies*—correlated and causally related data types. Although there is no support for reactive policies, the restrictions on proxy use fulfill a similar role in limiting how information (not just values) can be uses. Their work develops an algorithm for detecting proxy use in data-driven systems (e.g., machine learning systems) and for eliminating "inappropriate" proxy uses. Although general use-based privacy policies are beyond the scope of this work, their approach effectively restricts information use by a single centralized system.

Note that it is tricky to compare the performance of mechanisms that are intended to achieve different goals. Therefore, we have not undertaken comparisons of our architectures with implementations of these alternate privacy regimes.

Use-based Authorization Regimes. Several existing projects define languages for expressing restrictions on how data are used, and can therefore be viewed as partially implementing the requirements of use-based privacy. However, none of these regimes fully support use-based privacy, and none implement policy enforcement in a distributed system with adversarial service providers.

Usage Control (UCON) [29, 30] is an extension of traditional access control models (e.g., discretionary access control, mandatory access control, role-based access control) that enables continuity of access decisions. Here access control decisions are re-evaluated after the context (e.g., subject roles, time, number of previous accesses) changes. UCON was a reaction to increased networking and data sharing within a diverse ecosystem of devices, and it can be viewed as the first technical approach to use-based authorizations. Initial UCON systems enforced policies on a single system; later versions introduced distributed usage control [6, 16, 34], but assumed that all systems were run by trusted principals.

An alternative approach was outlined by Petković et al. [31], who consider a restricted form of use-based privacy, which they call *purpose control*. Their work creates an audit log of service provider actions and then detects policy violations by checking whether the audit trail is a valid execution of the organizational process—modeled as a formula in the Calculus of Orchestration of Web Services (COWS)—for a permitted purpose. This work does not consider prevention-based enforcement or enforcement in the presence of adversarial service providers.

Legalease [37] is a privacy policy language that implicitly supports policies encoded as domain-specific attributes. For example, a Legalease policy might say, "DENY DataType IP Address, UseForPurpose Advertising EXCEPT ALLOW DataType IPAddress:Truncated", which asserts that the full IP address may not be used for advertising. Many use-based policies can be encoded in Legalease by defining appropriate attributes. Legalese is deployed in Grok, a policy compliance system for Bing that automatically maps code-level elements to attributes and enforces policies using compile-time information flow analysis. However, Grok assumes that the entire system is under the control of a single, trusted principal.

The Thoth policy language [15] specifies data use policies comprising confidentiality, integrity, and declassification policies, each defining principals that are authorized and under what conditions. Although policies are designed to be expressed at a lower level than under our approach, Thoth's conditions are sufficiently flexible to capture policies that depend on who, what, or why as well as temporal, discretionary, autocratic, and jurisdictional policies. Thoth is implemented as a kernel-level compliance layer for enforcing data use policies in data retrieval systems, but it assumes that the enforcement layer is deployed by a trusted principal.

Lonet [18] is a system for expressing and enforcing security policies for shared data using isolated containers. Lonet policies—which are associated with data files and defined as metadata—are expressed as automata; states specify the set of authorized users and declare event-driven obligatory meta-code, and state transitions specify how to derive policies for derived values depending on the type of program that produces the derived value. Lonet implements a reference monitor that enforces these policies, but security depends on trusting service providers to deploy the enforcement mechanism.

Policy Enforcement with SGX. SGX offers a new basis for placing trust in a monitor or other program, so it is a natural tool for enabling policy enforcement in distributed systems where service providers are operated by untrusted principals. Several previous projects have explored related ideas, but, to the best of our knowledge, there are no prior systems that use SGX to guarantee privacy.

Haven [2] uses SGX to create a shielded execution environment, allowing unmodified Windows application binaries to be hosted inside SGX enabled enclaves. Applications then interface with a library version of the Windows operating system running entirely inside the enclave, reducing the dependencies on the underlying system. Moreover, Haven implements a shielding module for interfacing with components outside of the enclave, which provides access to, among other things, an encrypted and integrity protected file system. While the design of Haven places the entire OS inside an enclave—allowing for applications to be securely monitored by existing enforcement mechanisms—our approach yields a smaller trusted computing base. Our work also supports privacy enforcement in distributed systems.

VC3 [36] is a system for trustworthy data analytics in the cloud; it is a MapReduce framework that uses SGX to protect sensitive data. VC3 enforces confidentiality and integrity for code and data, and it enforces verifiability of code execution; it does not support enforcement for high-level policies or for use-based privacy.

Ryoan [17] is a distributed sandbox for performing computations on sensitive data. Ryoan uses SGX enclaves to protect data confidentiality and integrity from malicious service providers; it does not support enforcement for high-level policies or for use-based privacy.

Glamdring [21] is a framework for enforcing data confidentiality by automatically partitioning applications into untrusted and enclave apps and adding runtime monitoring. Although the policy language is limited—data is either secret or public—Glamdring requires only a small number of manual annotations (sensitive labels on data), thereby minimizing developer burden and facilitating deployment.

7 CONCLUSION

Use-based privacy offers an appealing approach to enhancing privacy in distributed systems that require data sharing. But successful enforcement depends on a trustworthy monitor and having a basis for trust in applications. In this work, we investigate the feasibility of using Intel SGX as a root of trust to enforce such policies in the presence of an active adversary. The natural, source-based monitoring architecture enables privacy enforcement against malicious adversaries with minimal effort for application developers, but it brings significant performance overhead. So we explore two alternative architectures—delegated monitoring and inline monitoring—that offer improved performance and that demonstrate a trade-off between deployability, performance, and privacy. We find that a delegated monitoring architecture provides the best performance for enforcing privacy against malicious adversaries, but that an inline monitoring architecture provides performance improvements—particularly for applications that handle more data or require finer-grained policies—with attenuated privacy guarantees. Given the

Architecture	Privacy Guarantees	Performance	Deployability
Source-based	malicious adversaries (✓)	poor (-)	no programmer burden (✓)
Delegated	malicious adversaries (✓)	moderate (~)	some policy handling (~)
Inline	accountable adversaries (~)	good (✓)	significant annotations (-)

Figure 15: Tradeoffs between different monitoring architectures. ✓indicates goals that are fully met , ~ indicates goals that are partially met, – indicates the architecture failed goals.

trade-offs between deployability, performance, and privacy (summarized in Figure 15), we believe that the appropriate architecture will depend on the type of application. However, we view our results as positive evidence of the feasibility of enforcing use-based privacy policies in a decentralized, adversarial ecosystem.

REFERENCES

[1] Adam Barth, Anupam Datta, John C. Mitchell, and Helen Nissenbaum. Privacy and contextual integrity: Framework and applications. In *IEEE Symposium on Security and Privacy*, pages 184–198, 2006.

[2] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding applications from an untrusted cloud with haven. In *Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation*, pages 267–283. USENIX Association, 2014.

[3] Eleanor Birrell. Avenance middleware. https://bitbucket.org/cornell-ebirrell/av-middleware, 2018.

[4] Eleanor Birrell. Avenance package. https://bitbucket.org/cornell-ebirrell/pol-server, 2018.

[5] Eleanor Birrell and Fred B. Schneider. A reactive approach to use-based privacy. Technical Report 54843, Cornell University, Computing and Information Science, November 2017.

[6] Laurent Bussard, Gregory Neven, and F.-S. Preiss. Downstream usage control. In *IEEE Internation Symposium on Policies for Distributed Systems and Networks (POLICY)*, pages 22–29, 2010.

[7] Fred Cate. Principles for protecting privacy. *Cato Journal*, 22:33–57, 2002.

[8] Fred Cate, Peter Cullen, and Viktor Mayer-Schönberger. Data protection principles for the 21st century. Oxford Internet Institute, 2013.

[9] Federal Trade Commission et al. Fair information practice principles. *last modified June*, 25, 2007.

[10] Intel Corp. Intel software guard extensions (Intel SGX). https://software.intel.com/sites/default/files/332680-002.pdf, June 2015.

[11] Lorrie Cranor, Marc Langheinrich, Massimo Marchiori, Martin Presler-Marshall, and Joseph Reagle. The platform for privacy preferences 1.0 (P3P 1. 0) specification. *W3C recommendation*, 16, 2002.

[12] Anupam Datta, Matthew Fredrikson, Gihyuk Ko, Piotr Mardziel, and Shayak Sen. Use privacy in data-driven systems: Theory and experiments with machine learnt programs. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1193–1210. ACM, 2017.

[13] Dorothy E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, 1976.

[14] Cynthia Dwork. Differential privacy. In *33rd International Colloquium on Automata, Languages and Programming, part II (ICALP)*, volume 4052, pages 1–12, Venice, Italy, July 2006. Springer Verlag.

[15] Eslam Elnikety, Aastha Mehta, Anjo Vahldiek-Oberwagner, Deepak Garg, and Peter Druschel. Thoth: Comprehensive policy compliance in data retrieval systems. In *USENIX Security Symposium*, pages 637–654, 2016.

[16] M. Hilty, A. Pretschner, D. Basin, C. Schaefer, and T. Walter. A policy language for distributed usage control. In Joachim Biskup and Javier López, editors, *12th European Symposium On Research In Computer Security (ESORICS)*, volume 4734 of *Lecture Notes in Computer Science*, pages 531–546, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[17] Tyler Hunt, Zhiting Zhu, Yuanzhong Xu, Simon Peter, and Emmett Witchel. Ryoan: A distributed sandbox for untrusted computation on secret data. In *OSDI*, pages 533–549, 2016.

[18] Håvard D Johansen, Eleanor Birrell, Robbert Van Renesse, Fred B. Schneider, Magnus Stenhaug, and Dag Johansen. Enforcing privacy policies with meta-code. In *Proceedings of the 6th Asia-Pacific Workshop on Systems*, 2015.

[19] Elisavet Kozyri, Owen Arden, Andrew C. Myers, and Fred B. Schneider. JRIF: Reactive information flow control for Java. Technical Report 41194, Cornell University, Computing and Information Science, February 2016.

[20] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 557–574, Vancouver, BC, 2017. USENIX Association.

[21] Joshua Lind, Christian Priebe, Divya Muthukumaran, Dan O'Keeffe, Pierre-Louis Aublin, Florian Kelbert, Tobias Reiher, David Goltzsche, David Eyers, Rüdiger Kapitza, Christof Fetzer, and Peter Pietzuch. Glamdring: Automatic application partitioning for intel SGX. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 285–298, Santa Clara, CA, 2017. USENIX Association.

[22] Markus Lorch, Seth Proctor, Rebekah Lepro, Dennis Kafura, and Sumit Shah. First experiences using XACML for access control in distributed systems. In *Proceedings of the 2003 ACM workshop on XML security*, pages 25–37. ACM, 2003.

[23] Petros Maniatis, Devdatta Akhawe, Kevin Fall, Elaine Shi, Stephen McCamant, and Dawn Song. Do you know where your data are? Secure data capsules for deployable data protection. In *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems*, 2011.

[24] Frank McSherry. Privacy integrated queries: An extensible platform for privacy-preserving data analysis. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, pages 19–30. ACM, 2009.

[25] Marco Casassa Mont, Siani Pearson, and Pete Bramhall. Towards accountable management of identity and privacy: Sticky policies and enforceable tracing services. In *Proceedings of the 14th IEEE International Workshop on Database and Expert Systems Applications*, pages 377–382, 2003.

[26] Craig Mundie. Privacy pragmatism: Focus on data use, not data collection. *Foreign Aff.*, 93:28, 2014.

[27] Helen Nissenbaum. *Privacy in Context: Technology, Policy, and the Integrity of Social Life*. Stanford University Press, 2009.

[28] Helen Nissenbaum. A contextual approach to privacy online. *Daedalus*, 140(4):32–48, 2011.

[29] Jaehong Park and Ravi Sandhu. Towards usage control models: Beyond traditional access control. In *Proceedings of the Seventh ACM Symposium on Access Control Models and Technologies*, SACMAT '02, pages 57–64, 2002.

[30] Jaehong Park and Ravi Sandhu. The UCONABC usage control model. *ACM Trans. Inf. Syst. Secur.*, 7(1):128–174, February 2004.

[31] Milan Petkovic, Davide Prandi, and Nicola Zannone. Purpose control: Did you process the data for the intended purpose? *Secure Data Management*, 6933:145–168, 2011.

[32] Svein A. Pettersen, HÃ¥vard D. Johansen, Ivan A. M. Baptista, PÃ¥l Halvorsen, and Dag Johansen. Quantified soccer using positional data: A case study. *Frontiers in Physiology*, 9:866, 2018.

[33] PMSys. http://forzasys.com/pmsys.html.

[34] Alexander Pretschner, Manuel Hilty, and David Basin. Distributed usage control. *Communications of the ACM*, 49(9):39–44, 2006.

[35] N. Ramanathan, F. Alquaddoomi, H. Falaki, D. George, C. K. Hsieh, J. Jenkins, C. Ketcham, B. Longstaff, J. Ooms, J. Selsky, H. Tangmunarunkit, and D. Estrin. Ohmage: An open mobile system for activity and experience sampling. In *2012 6th International Conference on Pervasive Computing Technologies for Healthcare (PervasiveHealth) and Workshops*, pages 203–204, May 2012.

[36] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. VC3: Trustworthy data analytics in the cloud using SGX. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 38–54. IEEE, 2015.

[37] Shayak Sen, Saikat Guha, Anupam Datta, Sriram K. Rajamani, Janice Tsai, and Jeannette M. Wing. Bootstrapping privacy compliance in big data systems. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (Oakland)*, 2014.

[38] Spring. Spring boot framework. https://projects.spring.io/spring-boot/, December 2017.

[39] Hongsuda Tangmunarunkit, Cheng-Kang Hsieh, Brent Longstaff, S Nolen, John Jenkins, Cameron Ketcham, Joshua Selsky, Faisal Alquaddoomi, Dony George, Jinha Kang, et al. Ohmage: A general and extensible end-to-end participatory sensing platform. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 6(3):38, 2015.

[40] Jennifer Widom. Trio: A system for integrated management of data, accuracy, and lineage. Technical report, Stanford InfoLab, 2004.

[41] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 640–656. IEEE, 2015.

[42] Fan Zhang. mbedtls-SGX. https://github.com/bl4ck5un/mbedtls-SGX.