Trusted Enforcement of Application-specific Security Policies

Marius Schlegel[a]
TU Ilmenau, Germany

Keywords: Application Security, Security Architecture, Security-Policy-Controlled Applications, Application-specific Security Policies, Application-level Policy Enforcement, Trusted Execution, Intel SGX.

Abstract: While there have been approaches for integrating security policies into operating systems (OSs) for more than two decades, applications often use objects of higher abstraction requiring individual security policies with application-specific semantics. Due to insufficient OS support, current approaches for enforcing application-level policies typically lead to large and complex trusted computing bases rendering tamperproofness and correctness difficult to achieve. To mitigate this problem, we propose the application-level policy enforcement architecture AppSPEAR and a C++ framework for its implementation. The configurable framework enables developers to balance enforcement rigor and costs imposed by different implementation alternatives and to easily tailor an AppSPEAR implementation to individual application requirements. We argue that hardware-based trusted execution environments offer an optimal balance between effectiveness and efficiency of policy protection and enforcement. This claim is substantiated by a practical evaluation based on a medical record system.

1 INTRODUCTION

The rigorous and correct enforcement of application-specific security goals in today's complex software systems is far from being an easy task. In order to protect the confidentiality and integrity of security-critical resources (e. g. customer information or accounting information), such systems increasingly rely on a *security policy*: an automatically enforced set of rules that control evaluating, granting, and revoking access privileges, control tracing, classifying, and confining potential flows of information, or control isolating domains of users and resources.

For almost two decades the idea has been pursued to integrate security policies directly into operating systems (OSs) (Watson, 2013). The control of operations on OS objects (resources described by abstractions such as files, processes, sockets, etc.) is made possible by the extension of standard OSs by mandatory access control (AC) and information flow control mechanisms (Loscocco and Smalley, 2001; Smalley and Craig, 2013; Sze et al., 2014).

At application level, for instance, enterprise resource planning systems typically use role-based AC policies that reflect a company's organizational structure (Alam et al., 2011; Bhatti et al., 2005), workflow management systems use information flow policies (Crampton et al., 2016; Wang and Li, 2010), database systems use label-based AC policies to control access to relations and views (Oracle, 2018; IBM, 2016), (social) information systems use relationship-based AC policies on user data (Rizvi et al., 2015; Fong, 2011), and Big Data and IoT platforms rely on attribute-based AC policies (Gupta et al., 2018; Bezawada et al., 2018). Compared to OSs, the objects used by applications are typically subject of security policies on a higher, application-specific abstraction level. This requires policy rules with application-specific semantics which typically differ significantly from those at OS level and which are also specific for each individual application.

Contemporary OSs do not adequately support application-specific security policies. For this reason, developers often integrate security policies directly into applications which results in large and heterogeneous trusted computing base (TCB) implementations. Due to the close integration of security-relevant functionality and application logic, the identification of an application's TCB perimeter is hard if not impossible, rendering correctness properties difficult to achieve.

To enable precisely defined application TCBs as well as a rigorous and trusted enforcement of individual application security policies, this paper argues for an alternative approach. First, the foundation is laid by strictly separating security-relevant functionality from (potentially untrusted) application logic. Second,

[a] https://orcid.org/0000-0001-6596-2823

Schlegel, M.
Trusted Enforcement of Application-specific Security Policies.
DOI: 10.5220/0010579703430355
In *Proceedings of the 18th International Conference on Security and Cryptography (SECRYPT 2021)*, pages 343-355
ISBN: 978-989-758-524-1

this separation is implemented by an isolation mechanism. Adjusting the strength of isolation and the costs for crossing isolation boundaries, the approach is open for different isolation mechanisms ranging from language-based (e. g. type-safe programming languages and compilers) and OS-based mechanisms (e. g. virtual address spaces via processes) to trusted execution technologies (e. g. Intel SGX) (Shu et al., 2016).

In particular, SGX provides trusted execution environments (TEEs), so-called *enclaves*, which isolate security-sensitive parts of application code and data from the OS kernel, hypervisor, BIOS, and other applications using a hardware-protected memory region (Intel Corporation, 2021). While this significantly reduces the size of application TCB implementations, crossing isolation boundaries typically imposes high costs compared to conventional isolation mechanisms (Weisse et al., 2017). Therefore, we examine its suitability for our approach.

Specifically, we make the following contributions. (1.) We introduce AppSPEAR – an application-level security policy enforcement architecture providing a functional framework for implementing the reference monitor principles (Anderson, 1972) (§ 2). (2.) Balancing rigor of policy enforcement (*effectiveness*) and isolation/communication costs (*efficiency*), we discuss implementation alternatives for AppSPEAR as well as the consequences of using different isolation mechanisms (§ 3.1). We have cast this spectrum of implementation alternatives into a developer framework, which highlights the developer support provided. Moreover, we describe our experiences with the integration of AppSPEAR into the medical record system OpenMRS using the developer framework (§ 3.2). (3.) We present an evaluation of the AppSPEAR framework based on our implementation showing the practical runtime costs imposed by the different implementation alternatives (§ 4).

2 ARCHITECTURE DESIGN

This section introduces AppSPEAR. Its purpose is to provide a functional framework for the rigorous enforcement of application-specific security policies. In the following, we first discuss fundamental security policy and architecture design principles (§ 2.1). Based on those, we describe our architecture, including its components, their tasks, and interrelationships (§ 2.2).

2.1 Requirements and Design Principles

Application-specific Security Policies. Security policies are usually well-tailored to their respective application domain. While policy rules differ semantically among different classes of policies, namely AC, information flow, and noninterference, they all aim at guarantees towards security properties.

As one of the most prominent policy classes in practice, we focus on AC policies. Furthermore, to support a broad range of different model abstractions (e. g. roles, labels, contexts, risks, relationships, or attributes in general (Ferraiolo et al., 2007; Biswas et al., 2016; Shebaro et al., 2014; Ni et al., 2010; Fong, 2011; Jin et al., 2012)), we consider an AC policy from the perspective of the enforcement mechanisms as a black box of rules. These rules authorize any operation $op \in OP$ (e. g. *read* from or *append* to electronic patient record (EPR) objects) related to a vector of entities $e = (e_i)_{i=1}^{n} \in E^n$ (e. g. application users, EPRs or documents). The interface of an AC policy P can be defined as a semantically neutral *AC function* (ACF) $f_P : E^n \times OP \to \{\text{true}, \text{false}\}$. For example within an EPR management system, appending new checkup results to a patient's EPR could then by requested by $f_P(\langle \text{Alice}, \text{EPR}_{\text{Bob}} \rangle, \text{append})$.

In recent years, research rendered a number of modeling schemes for context-aware AC policies that model adaptive authorization decisions based on physical or logical features, represented by local context variables. Values of such variables may be either computed, e. g. time, date, and resource usage, or perceived by sensors, e. g. temperature, geolocation, and NFC device proximity (Hsu and Ray, 2016; Shebaro et al., 2014). By considering a vector of context values $v = (v_j)_{j=1}^{m} \in V^m$ (e. g. time or geolocation) additionally to f_P, the interface of a context-aware AC policy P' can be defined by an ACF $f_{P'} : E^n \times V^m \times OP \to \{\text{true}, \text{false}\}$. These context values may be input for a risk evaluation metric (as typically used in risk-based AC models) which may calculate a numerical value (Ni et al., 2010). In the simplest case, this value is compared with a risk-threshold to assess the risk as either acceptable (permitting the original access request) or unacceptable (denying the original request).

Reference Monitor Principles. In general, security policies are part of a system's TCB. Subsequently, we consider a TCB from a functional perspective as the set of all functions that are necessary and sufficient for implementing a system's security properties. The part of a software architecture that implements the TCB forms the *security architecture*.

For more than four decades, the reference monitor principles (Anderson, 1972) have provided fundamental guidelines for the design and implementation of security architectures. These principles include three rules requiring a security architecture core (referred to

as *reference monitor*) that is (1.) inevitably involved in any security-related interaction (RM 1, *total mediation*), (2.) protected from unauthorized manipulation (RM 2, *tamperproofness*), and (3.) as small and simple as possible in terms of functional complexity and amount of code (RM 3, *verifiability*).

According to RM 1, it must be guaranteed that any actions on security-relevant objects are inevitably controlled by the policy. In case of security-policy-controlled OSs, this is achieved by calling the policy within the OS kernel services (e. g. process management, filesystem, or I/O subsystem) before the actual execution of any action (e. g. *fork*, *read*, or *send*) on an OS object (i. e. a resource described by abstractions such as processes, files, or sockets) (Smalley et al., 2001). At the application level, RM 1 can be achieved by the separation of any functions relevant to the policy enforcement and the application object management, on the one hand, from functions solely responsible for the application logic, on the other hand. Consequently, any security-relevant object and any function for accessing such an object belong to the TCB. If an application object is accessed by an application logic function, an immediate entry into the TCB takes place. Accordingly, by calling the security policy before any security-relevant object access within the TCB boundaries, the policy cannot be bypassed.

Moreover, the precise delineation of the TCB paves the way for protecting the integrity of an application security policy from unauthorized manipulation (cf. RM 2). The functional separation is implemented by isolation mechanisms (Shu et al., 2016), which, depending on the assumed attacker model and the degree of tamperproofness required, range from language-based isolation (e. g. policy execution in a class instance separated by type-safety-checking compilers) to OS-based isolation (e. g. policy execution in a separate virtual address space) and virtualization techniques (e. g. policy execution in a virtual machine), as well as hardware-based isolation (e. g. policy execution in an SGX enclave) to total physical separation (e. g. policy execution on a dedicated server).

By adapting RM 3, it generally holds that the smaller a TCB's functional perimeter and the lower its complexity, the better a TCB can be analyzed regarding correctness properties. A precise functional perimeter reduces TCB size and complexity because the functions solely associated with the application logic are no longer part of the TCB. Furthermore, to facilitate the verifiability of certain parts of the TCB, in particular security policies, w. r. t. model properties, such as dynamic state reachability in dynamic AC models (Stoller et al., 2011; Tripunitara and Li, 2013; Schlegel and Amthor, 2020; Schlegel and Amthor, 2021), an additional separation within the TCB boundaries of policy-specific from policy-independent TCB functions should be anchored within the architecture.

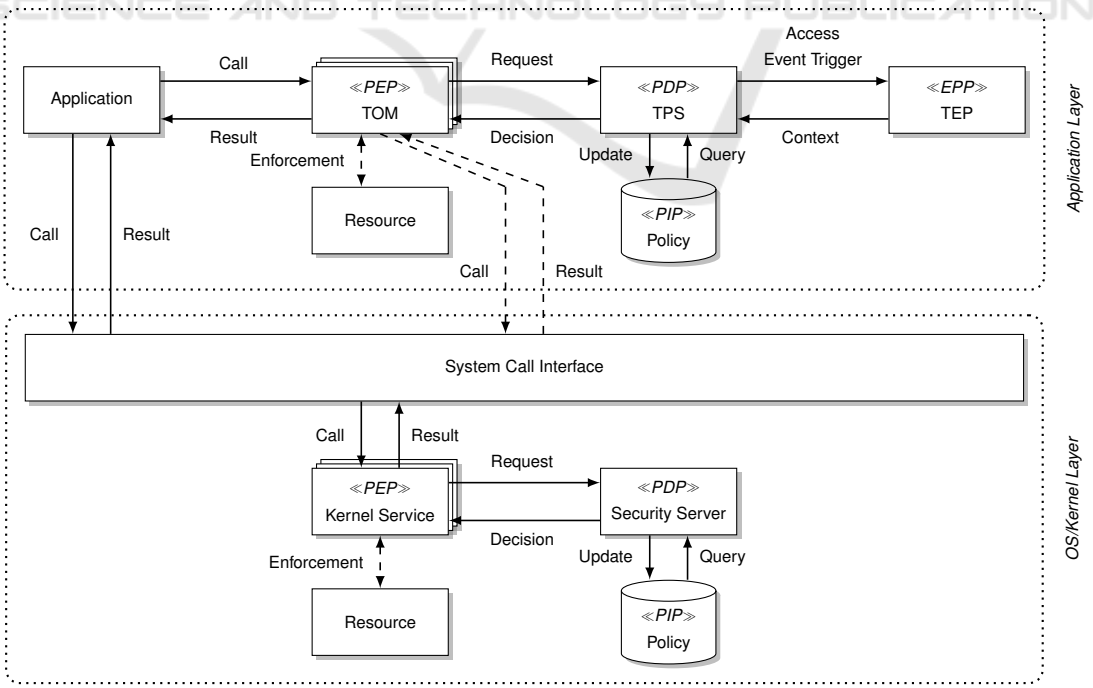

Figure 1: Functional architecture for mutually independent policy enforcement on application layer (AppSPEAR) and OS layer (SELinux/Flask).

2.2 AppSPEAR Design

An important representative of security architectures that implements the reference monitor principles is the Flask security architecture (Spencer et al., 1999). Motivated by enabling the flexible interchangeability of security policies, the policy logic, referred to as *policy decision point* (PDP), is separated from the policy enforcement, distributed over *policy enforcement points* (PEPs). This key idea has been adopted by SELinux (Loscocco and Smalley, 2001), extending the Linux kernel by mandatory AC, and followed by a wide range of OSs (Watson, 2013). The SELinux architecture comprises PEPs located in the OS kernel services (referred to as *object managers*) and a singular PDP (referred to as *security server*) providing an OS security policy runtime environment (see Fig. 1).

Comparing Flask's/SELinux' objective of rigorous and flexible policy enforcement on OS level with ours for the application level, we argue to adapt these principles. The general idea is to instantiate Flask/SELinux-alike architecture components for each policy-controlled application which *complement* the architecture (and policy) at the kernel level. This idea is illustrated in Fig. 1. Moreover, we consider state of the art in the enforcement of context-/attribute-based AC policies as represented by the Policy Machine (Ferraiolo et al., 2015) and its successor, Next-Generation Access Control (NGAC) (Ferraiolo et al., 2016).[1]

Trusted Object Managers. In order to implement our architecture at the application level according to the discussed design principles, in general, a separation between security-relevant functionality and security-irrelevant application logic is required. *Trusted object managers* (TOMs) are architecture components to rigorously isolate trusted and non-trusted application parts and comprise all functions which are required for establishing authenticity, integrity, and confidentiality of application objects.

Consequently, a TOM provides an encapsulated object access interface for actions such as reading or modifying security-relevant application objects (e. g. patient record objects or customer record objects). This interface is implemented by local or remote procedures, functions, or methods. Inspired by object-oriented software design, a generic TOM provides a generic object abstraction and abstract basic functions for its management, such as create, read, write, and destroy. For a concrete application, type-specific TOMs have to be derived from that generic TOM, which implement the abstract object and functions according to the type of object. Hence, a separate TOM is implemented for each object type analogous to the OS kernel services acting OS object managers.

When an application requests to perform a particular action on a TOM-managed object, the corresponding application security policy is immediately involved through the interface between a TOM and the *trusted policy server* (TPS). In this way, (1.) any security-relevant subject-object interaction is inevitably controlled by the policy and (2.) decisions can be enforced tamperproof, because the TOMs are part of the TCB and, thus, are isolated from the untrusted application logic. Finally, depending on the granted or denied access, the corresponding output of the object access is returned to the calling application logic function.

For many applications, it is practically inevitable to use OS objects in addition to the application-specific ones. Actions on OS objects are controlled by the OS security policy, if present. Scattering application policy rules on both OS and application layers would contradict the TCB properties we aim for. Furthermore, if there is no OS policy present, such accesses would not be controlled at all. To maintain the responsibility of application policies and to not require an OS policy for each object type an application uses, a simple wrapper for OS objects in form of a separate TOM is established. To reduce development effort, these may be reused (or generated automatically) based on a reference implementation. These TOMs may also be used by other TOMs to control accesses to OS objects and system calls at the application level.

In order to ensure the authenticity of application objects appearing as policy entities, TOMs are responsible for managing entity identifications (comparable with SELinux security identifiers) and for assigning them correctly as well as irrevocably to their implementations as runtime or persistent application objects such that they can be clearly identified at any time.

If we isolate architectural components from each other according to the design principles discussed in § 2.1, the communication between these components requires the crossing of isolation boundaries, which in turn causes certain costs depending on the type of isolation (e. g. IPC when isolating via virtual address spaces). To reduce those costs, for instance for the communication of TOMs and the TPS, a caching mechanism may be used for policy decisions, similar to the SELinux access vector cache. However, it should be noted in particular that when implementing a stateful application policy, it may be necessary to explicitly manage cache consistency by invalidating cached access decisions which may become invalid due to policy state changes.

[1] Since we focus on policy enforcement, we omit a *policy administration point* as an entry point for administration.

Trusted Policy Server. The TPS represents AppSPEAR's PDP. Its main task is to provide a policy runtime environment including data structures that represent any policy abstractions and components. Because of the wide range of different modeling schemes for application policies, it is not reasonable to limit the TPS to specific implementations. In contrast, providing the TPS with a wide range of model implementations would lead to a universal range of functions, which counteracts the principle of a small TCB perimeter.

Here, the approach is to support exclusively the present policy and to provide only functions necessary for that policy. Therefore, a policy implementation is specifically tailored to its application while maintaining the interfaces for TOM-to-TPS and TPS-to-TOM communication. To reduce the implementation effort, developer support is provided in the form of TPS libraries implementing the data structures and functionality of frequently used security models. Beyond that, we aim to extend developer support even further: By providing a convenient policy specification language (Amthor and Schlegel, 2020) and a compiler to generate policy-specific TPSs, the effort is reduced to the specification of policy rules.

Finally, to deal with application and system failures, crashes, or reboots, the TPS comprises functions for the persistent and secure storage of the policy's state. Depending on the required guarantees, different strategies (e. g. logging diffs, complete state backups, etc.) are applied at different points in time (e. g. fixed times, after each state change, etc.).

Trusted Event Processor. The *trusted event processor* (TEP) is an optional add-on to provide context-based security policies information about the physical and logical context of a system according to their interface (cf. $f_{P'}$, § 2.1). The TEP may be implemented based on asynchronous triggers instead of function calls, which may originate from local hard- and software components, e. g. GPS, clock, or temperature sensors, typically implemented by an OS interrupt mechanism. Any access decision of the TPS may also trigger an event that is needed for logging and auditing.

3 ARCHITECTURE IMPLEMENTATION

AppSPEAR provides a functional framework for implementing the reference monitor principles at the application layer. The flexibility of AppSPEAR enables the separation of its architecture components and their implementation by means of isolation mechanisms in a variety of ways. § 3.1 discusses multiple architecture instantiation alternatives and their individual characteristics regarding application TCB size. Each variant has, on the one hand, a certain degree of possible rigor in terms of TCB isolation (*effectiveness*) and, on the other hand, certain costs in terms of required resources and communication effort for crossing isolation boundaries (*efficiency*). To implement these instantiation alternatives, a selection of isolation mechanisms and developer support provided by a configurable developer framework are discussed. § 3.2 then describes a practical case study integrating AppSPEAR into the electronic medical record system OpenMRS.

3.1 Balancing Effectiveness and Efficiency

Architecture Instantiation. AppSPEAR's design provides different alternatives for instantiating and implementing its components. Fig. 2 shows reasonable variants illustrating boundaries at which the architecture components may be isolated from one another. Since the TEP realizes parts of the implementation of context-based security policies (e. g. risk analysis and estimation), TPS and TEP are explicitly not isolated from each other.

If a large application TCB comprising the potentially untrusted application logic as well as all AppSPEAR components (see Fig. 2a), then the implementation is both in terms of strictness and costs at the level of application-integrated security policies. From a qualitative point of view, AppSPEAR enables a structured software engineering of applications to be equipped with individual security policies.

Beyond this simple low-cost but low-quality implementation, AppSPEAR supports several alternatives with greater effectiveness. Taking advantage of the functional encapsulation of the TPS and the TEP, the security policy and its runtime environment can be implemented isolated from the rest of the application (see Fig. 2b). This paves the way for a tamperproof – naturally depending on the strength of the used isolation mechanism – and analyzable policy implementation. In quantitative terms, compared with the previous alternative, the application TCB comprises still the same set of functions, but here, the policy integrity is protected by more effective isolation.

A further qualitative improvement of the effectiveness is achieved by also isolating the TOMs as part of the application TCB from the untrusted application (see Fig. 2c). The isolation boundary between the TOMs and the TPS/TEP is moved between the untrusted application logic and the TOMs. This re-

sults in a smaller application TCB perimeter because only security-relevant functions are part of the TCB. From a conceptual perspective, the costs are comparable to the second variant (see Fig. 2b) due to a single isolation border and an equally frequent crossing of the TCB boundary. Nevertheless, the software engineering effort that is required for existing software architectures to create the technical prerequisites, i. e. a well-defined interface between untrusted application logic and TOMs, is also relevant when considering the isolation of TOMs. Since many applications typically use a database system and access stored object data, there is often already a functionally separated object management as intended by TOMs, which puts the costs into perspective at this point.

The example of an application utilizing a database already shows that due to the size and complexity of today's application TCBs and, consequently, the high verification effort, not the entire TCB can always be proven to be correctly implemented using conventional methods. Analogous to the principle of separate server processes in microkernel-based OSs, the TPS can be isolated as an essential part of the TCB from the rest of the TCB (see Fig. 2d) for security and robustness reasons (e. g. preventing unintended changes to the policy data structures due to reference errors). Thus, correctness properties of the security policy can be analyzed easier by formal models and methods (e. g. security properties such as right proliferation (safety) in dynamic AC models). This qualitative improvement also implies additional costs, since isolation mechanisms are used at two borders and these have to be overcome in a controlled manner for each policy request.

Isolation Mechanisms. In order to get beyond the weak security guarantees of application-integrated security policies (cf. Fig. 2a), harder measures are necessary. Based on the classification of isolation mechanisms in (Shu et al., 2016), we select and discuss the implementation of AppSPEAR (cf. Fig. 2b – 2d) by means of two mechanisms that go beyond purely software-based, intra-application-enforced isolation.

At the OS level, the virtual memory management provides one of the most fundamental isolation mechanisms. Each process has its own private virtual address space so that the memory of executed programs is isolated. This allows not only to allocate memory resources as optimally as possible according to processes' needs, but also to avoid the propagation of errors, faults, and failures to other processes or the entire system. Furthermore, even if a process is compromised, the adversary cannot breach the security of other processes without extensive efforts. By isolating the AppSPEAR components from the potentially untrusted application logic and each other using virtual address spaces via processes as in Fig. 2b – 2d, vulnerabilities of the untrusted application part can no longer affect the trusted parts. To enable communication across process boundaries, communication mechanisms controlled by the OS are necessary, such as local domain sockets, messages queues, pipes etc.

Beyond that, Intel SGX enables applications to protect private code and data from privileged system software such as the OS kernel, hypervisor, and BIOS as well as other applications. To achieve this, SGX uses protected TEEs called enclaves. An enclave is a protected area within an application process's address space. To meet integrity and confidentiality require-

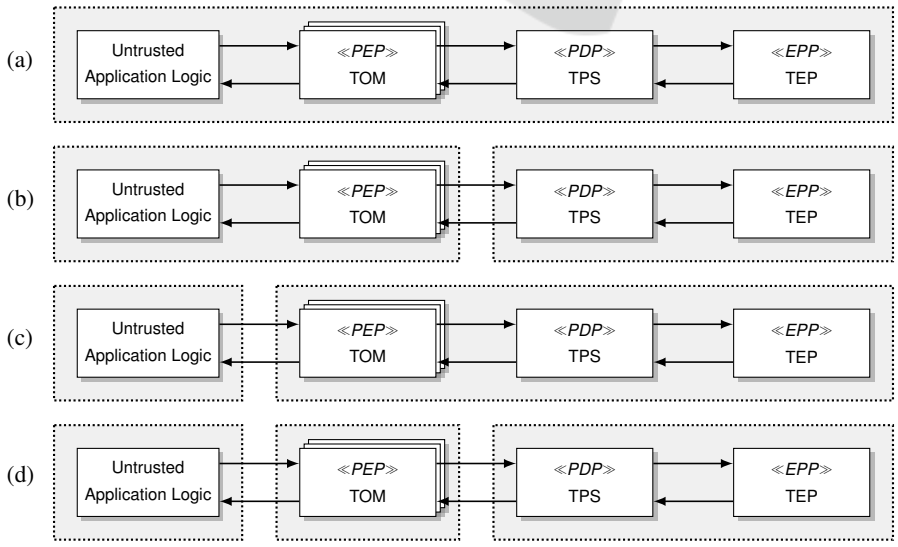

Figure 2: AppSPEAR instantiations with different separation/isolation of the architecture components.

ments, protected application code is loaded into an enclave after measuring using hardware-based attestation, and enclave data is automatically encrypted when leaving the CPU package into memory. Consequently, this design significantly reduces the TCB to only code executed inside the enclave (as well as the CPU, which in the end must always be trusted).

Since enclave memory cannot be read directly from outside of an enclave, data, which needs to be passed between trusted and untrusted application parts, has to be copied explicitly from and to an enclave. The SGX SDK provides mechanisms to create corresponding bridge functions, *ecalls*, which dispatch enclave entry calls to corresponding functions inside the enclave defining an enclave's interface. Corresponding functions that reside in the untrusted application part are called *ocalls* and invoked inside the enclave to request services outside of the enclave (e. g. system calls).

Software Framework and Developer Support. The implementation of AppSPEAR is supported by a software developer framework. As discussed, trusted execution mechanisms provide a promising basis for the trusted enforcement of application security policies. Specifically, we include support for Intel SGX due to its popularity and availability in current Intel CPUs. Since the official SDK is only available in C++, we implemented the framework in C++ as well. Our AppSPEAR developer framework comprises the following features: (1.) transparent and flexibly configurable isolation according to the AppSPEAR instantiation variants and the selection of isolation mechanisms discussed,[2] (2.) transparent and flexibly configurable communication via proxy objects, (3.) transparent reduction of communication effort via in-proxy caching.

The key idea for implementing these features lies in an RPC/RMI-alike communication model: Any communication between the components of a policy-controlled application is handled via pairs of proxy objects, similar to stubs known from RPC/RMI implementations. Those proxies act as intermediaries representing the callee on the caller side and the caller on the callee side. Consequently, any communication handled by these proxies is be performed *transparently*, so that the consideration of isolation-mechanism-specific needs is not required to be considered by developers, such as the serialization of application-specific data structures and accordingly their deserialization. In addition, techniques for securing communication, such as encryption, hashing, or integrity certificates, and reducing communication effort, such as caching of access requests and corresponding decisions, are transparently integrated into proxies.

The developer framework provides a set of requester/responder proxy pair implementations, one for each AppSPEAR component taking the specific characteristics of each isolation and corresponding communication mechanism into account:

- language/compiler-based isolation and communication via local procedure/function/method calls,
- process-based isolation and IPC-based communication via local domain sockets[3], and
- TEE-based isolation via SGX enclaves and boundary-crossing communication (ecalls/ocalls).

For implementing the AppSPEAR components, base classes provide basic functionality (according to § 2.2 and abstract member objects for proxy-based communication. Concrete implementations are derived from those base classes where their instantiation also configures the particular isolation and communication mechanisms to be used, which is usually done at compile time. Alternatively, to be able to compare different implementation variants (e. g. for testing purposes), the configuration can be done flexibly at runtime.

In order to reduce communication effort and multiple processing of the same inputs, caching mechanisms can be used. In principle, caching is possible for (1.) application-logic-to-TOM communication, and (2.) TOM-to-TPS communication.[4] Since the application logic is assumed to be untrusted, caching of TOM calls and results on the application logic side counteracts our goals for rigorous and trustworthy policy enforcement. Nevertheless as argued in § 3.1, many applications use databases to store application objects such that DBMS caches can achieve an overhead reduction when retrieving application object data.

Caching of TPS requests and decisions on the TOM side avoids multiple identical requests (if the policy has not changed compared to the initial request) and reduces the runtime overhead to the level of local function/method calls – especially for implementation variants with stricter TPS/TEP isolation (see Fig. 2b and 2d). To also keep access times to cache entries as low as possible, the cache is implemented using a container organized based on a hash table. If, in the case of a stateful policy, the execution of an operation results in the modification of the policy state (Schlegel and

[2]Note that the framework implementation is conceptually not limited to this selection and can be easily extended.

[3]Sockets were chosen based on the out-of-the box compromise of flexibility (in terms of message content/size), performance, simplicity, and availability in commodity OSs.

[4]Communication between the TPS and the TEP is asynchronous based on callbacks or periodic updates (see § 2.2). Since received context values are temporarily stored anyway, a separate cache is not useful here.

Amthor, 2020; Schlegel and Amthor, 2021), it may be necessary to invalidate corresponding cache entries.

3.2 Case Study: OpenMRS

This section describes a case study in which we apply the AppSPEAR framework to an existing database-backed application. The studied subject is OpenMRS, an electronic medical record (EMR) and medical information management system (OpenMRS Inc., 2021a). OpenMRS is an optimal representative for a policy-controlled application since the AC policy is part of its architecture and directly visible in the source code.

Software and Security Architecture. OpenMRS aims at being adaptable to resource-constrained environments such as healthcare facilities of low-income countries (Wolfe et al., 2006). This motivation is reflected in the web application architecture consisting of three logical layers: (1.) the *UI layer*, providing the user interface with input and query forms, (2.) the *service layer*, implementing the basic functionality, data model interaction, and a corresponding API, and (3.) the *database layer*, realizing the data model.

A tightly integrated role-based AC (RBAC) policy is responsible for controlling accesses to application objects such as EMRs or medication plans. The policy semantics are similar to the RBAC model family (Ferraiolo et al., 2007): a logged-in user's accesses to objects are regulated according to her assigned and activated roles (e. g. physician or nurse), to which certain permissions for actions are assigned (e. g. *read/modify EMR* or *create/delete patient*).

RBAC policy and enforcement are implemented through an AOP mechanism (`AuthorizationAdvice`) that wraps each service layer method call with a policy call, and custom Java annotations ("`@Authorized`") which initiate checking the privileges of the currently authenticated user. Since required permissions are directly attached to each service layer method, the policy is hard-coded and distributed over 635 points (OpenMRS core version 2.3.1). This renders the policy as well as its underlying model static, contradicting the goal of simple adaptability and flexible configurability. Beyond that, policy decision and enforcement functionality are isolated from application logic only via language-based mechanisms and, thus, misses the opportunity of using stronger isolation.

Applying the AppSPEAR Framework. While OpenMRS is implemented in Java, the official SGX SDK and the AppSPEAR developer framework are implemented in C++. Therefore, for the scope of this study, we decided to prototype OpenMRS in C++.

While the basic software architecture is the same, few functional differences exist. On the UI layer, we have implemented a minimal command line interface for requesting service layer functionality. On the service layer, we have implemented a selection of core services for data model interaction: the (system) user service (*login*, *activate/deactivate role*, *logout*), the person service (*create/delete person object*, *get/set address*), and the patient (EMR) service (*create/delete patient object*, *get/set patient diagnosis*), each derived in an OOP manner from an abstract service class. On the database layer, SQLite (Hipp et al., 2020) is used as relational DBMS because of its ability to fully store a database in-memory enabling its trusted execution within an SGX enclave. We modified SQLite version 3.32.3 for in-enclave execution by wrapping system calls through trampoline functions that temporarily exit an enclave (*ocalls*) or, where possible, by an SGX-compatible variant provided by the SDK.

Each service forms a TOM and manages its own objects (users, patients, EMRs, etc.) stored in the database and provides corresponding operations on them (generally such as *create/destroy object* and *read/modify object attribute*). When using the communication proxies provided by the developer framework, operations on TOM-managed objects called within the UI layer are transparently forwarded to the proxy counterpart of the respective TOM; depending on the isolation mechanism used, either via function/method calls, local domain socket send/receive (IPC) or SGX enclave calls. An analogous pattern applied for the TOM-to-TPS communication initiates requests to the RBAC policy regarding access permission or denial for each TOM operation to be executed. The TPS realizes the security policy by means of either a database or, leading to a smaller TCB, model-tailored data structures.

4 EVALUATION

The evaluation addresses the practical feasibility of the application-level policy enforcement approach. We compare the different architecture instantiation alternatives in terms of their runtime performance. Each alternative is implemented by configuring our developer framework to use each of the following isolation mechanisms: (1.) language/compiler-based isolation serving as a baseline, (2.) process-based isolation as a basic OS-level mechanism, and (3.) SGX/enclave-based isolation as a widely available trusted execution mechanism. Subsequently, § 4.1 describes the evaluation method and § 4.2 discusses the results.

4.1 Evaluation Methodology

Test Cases. We study two test cases: A basic application with an *always-allow* AC policy highlights the baseline for the runtime to be considered for isolation and communication in each architecture implementation variant (baseline microbenchmark). The application only comprises a single synthetic operation passing an operation identifier (required to determine the minimum costs). The prototypical reimplementation of OpenMRS comprising an RBAC policy (see § 3.2 for details) serves as a real-world use case. In particular, the layered software architecture and the usage of a database are representative for a multitude of applications and yields an impression of potential costs also relevant for other scenarios. The database provided by the OpenMRS community comprises an anonymized data set of 5,000 patients and 500,000 observations (OpenMRS Inc., 2021b).

We run two types of benchmarks: four microbenchmarks show efforts for typical create, read, update, and destroy (CRUD) operations, whereas a mixed macrobenchmark based on an OpenMRS workload extracted from logs (Chen et al., 2019) (called "Action" there) puts the execution of individual operations into a bigger context. Our adapted workload assumes the following occurrences of patient service CRUD operations: 25 % of create patient, 38 % of read patient diagnosis, 12 % of update patient diagnosis, and 25 % of delete patient. The individual operations appear mixed over the entire execution. For comparability reasons, the measured runtimes are divided by 100 (number of executed operations within the macrobenchmark).

Since especially enclave-based isolation involves high communication costs, we have also implemented and evaluated two techniques for decreasing runtime overhead: (1.) caching of policy decisions in TPS proxies located in the TOMs (see § 3.1) and (2.) asynchronous enclave calls (Weisse et al., 2017) which are provided by the SGX SDK as *switchless calls* because they do not involve costly enclave switches.

Metrics and Measurements. The runtimes are measured in CPU clock cycles by using the RDTSCP instruction. To avoid the typical behavior of "cold" CPU caches, each measurement is preceded by a warm-up phase of 1,000,000 iterations. To filter out potential outliers, we perform 1,000,000 iterations for each measurement and calculate medians. All measurements were performed on desktop hardware with an Intel Core i7-7700K CPU at 4.2 GHz and 32 GiB DDR4 RAM at 2,400 MHz. The machine runs Ubuntu 18.04.4 LTS with Linux kernel 5.3.0 including mitigations for critical CPU vulnerabilities. We use the Intel SGX driver, SDK, and platform software in version 2.7.1 (Intel Corporation, 2019). We compile using GCC 9.2, SGX hardware mode, and SGX SDK Prerelease configuration known to have production enclave performance (Johnson et al., 2016). The Enclave Page Cache (EPC) size is set to the maximum of 128 MiB of which ca. 93 MiB are usable. During all experiments, we disable dynamic CPU frequency scaling, Turbo Boost, and Hyper-Threading to avoid erratic runtime behavior and reduce potential outliers, by adjusting scheduling priorities and interrupt affinities.

4.2 Evaluation Results

Figs. 3a – 3d show the measurement results. The implementation variants (x-axis) are labeled according to the type of isolation/communication between (1.) application logic and TOMs, and (2.) TOMs and TPS/TEP. In each subfigure, absolute runtimes are illustrated in the upper part (unit 10^3 clock cycles on the left y-axis, unit microseconds calculated for the Intel i7 CPU with 4.2 GHz on the right y-axis) and in the lower part the relative runtime overhead compared to the fully integrated, intra-application implementation of AppSPEAR. The error bars show 95 % confidence intervals.

First of all, Fig. 3a illustrates the results of the baseline microbenchmarks showing the runtime of each of the considered unoptimized AppSPEAR implementations. Each variant requires at least a relative runtime overhead of more than 2 orders of magnitude compared to the intra-application implementation of AppSPEAR. Due to isolation and communication effort occurring twice in the IPC/IPC and IPC/SGX variants, the costs are also about twice as high compared to the single process- and SGX-isolated variants (ca. 30k cycles vs. ca. 55k/60k cycles).

Considering the CRUD microbenchmarks, in the LPC/LPC implementation, the read operation requires only about half of the runtime of create, update, and destroy operations (ca. 10,5k cycles vs. 18,2k/21,5k/25,8k cycles) because no changes to application objects stored in the database are made. Using process-based isolation in the LPC/IPC, IPC/LPC, and IPC/IPC implementation variants, the results are comparable considering the costs for isolation and communication. The LPC/IPC and IPC/LPC variants yield a similar level of runtime overhead, while the IPC/IPC variant has an increased amount of additional overhead due to isolation using two separate processes which nevertheless does not double the costs due to a compact small-volume TOM-to-TPS communication.

While for the same reason, the SGX-based isolation of the TPS/TEP (LPC/SGX variant) has a runtime overhead comparable with the LPC/IPC variant, the

SGX/LPC variant highlights that the joint isolation of TOMs and TPS/TEP, and the resulting in-enclave code execution can lead to a considerable additional overhead beyond pure isolation/communication costs: The approximately twice as high runtime effort results from the execution of SQLite within the isolated enclave, since temporary enclave exits (ocalls) may occur several times (see also § 3.2).

The hybrid IPC/SGX variant shows a middle way solution in terms of rigor and costs compared to the previously discussed variants: TOMs are separated from the application logic by process-based isolation and only the TPS/TEP is completely trusted through isolation via SGX. Although the baseline overhead is the highest in the field of comparison, the CRUD microbenchmark results rank between the IPC/IPC and SGX/LPC variants because SQLite is executed in a regular process. Moreover, the macrobenchmark results fit into the picture of results drawn so far. The runtimes for the CUD operations in favor of the R operation are relativized in all AppSPEAR implementation variants according to a real-world workload.

In order to reduce the runtime overhead (especially caused by communication), we have implemented and evaluated two techniques for runtime reduction, whose impact we will now discuss: caching of policy requests and decisions within TOMs (see Fig. 3b) as well as SGX switchless calls (see Fig. 3c). The runtimes of cache-optimized AppSPEAR implementations assume caches already filled with corresponding entries (done in measurement warm-up).

It can be seen in Fig. 3b that especially those im-

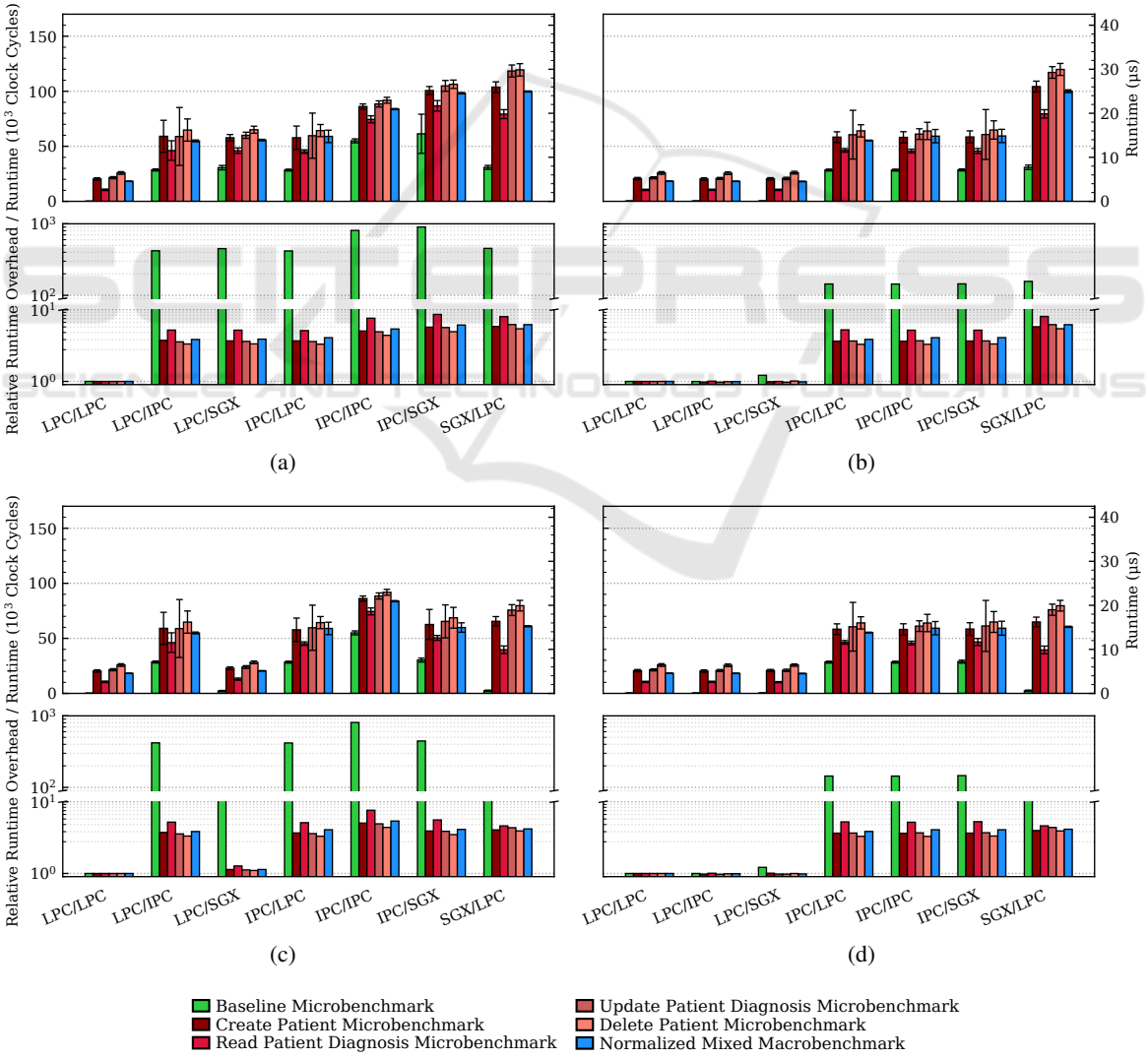

Figure 3: Benchmarking results: (a) standard/unoptimized implementations, (b) implementations optimized by caching, (c) implementations optimized by switchless calls, (d) implementations optimized by caching and switchless calls.

plementation variants that separate the TPS/TEP from TOMs via process- or SGX-based isolation (LPC/IPC, LPC/SGX, IPC/IPC, and IPC/SGX implementation variants) benefit from this measure: A cache hit effectively results in a significant reduction of the runtime overhead by eliminating IPC or enclave-boundary crossing. For the LPC/IPC and LPC/SGX variants, the runtime is reduced to the level of the intra-application implementation (LPC/LPC variant) and for the IPC/SGX and IPC/IPC variants to the cost level of IPC/LPC. For all other implementation variants (IPC/LPC, IPC/IPC, and SGX/LPC) querying the cache requires a little more runtime.

Compared to the AppSPEAR standard implementations, switchless calls improve runtimes in all considered implementation variants which use enclave-based isolation, i. e. LPC/SGX, IPC/SGX, and SGX/LPC (see Fig. 3c). For the LPC/SGX and SGX/LPC variants, this approach leads to a reduction of the basic runtime overhead by more than one order of magnitude, while for the IPC/SGX variant it is more than half of the runtime. The results of the CRUD micro- and macrobenchmarks also show a runtime reduction on average by more than half for the LPC/SGX variant and by less than two-thirds for the other two IPC/SGX and SGX/LPC variants. The results of all other variants remain unchanged compared to Fig. 3a.

Finally, Fig. 3d shows the results of combining the two previously discussed optimizations. Taking into account both caching (assuming cache hits) and switchless calls, the runtimes for TPS/TEP isolating implementation variants via processes or SGX (LPC/IPC and LPC/SGX) are reduced to the cost level of an intra-application implementation (LPC/LPC variant). Although the other implementation variants have 3 to 4 times higher runtimes, due to a much higher effort for communication and isolation, they also allow for much stronger isolation guarantees (see also § 3.1). In particular, the trusted execution of all AppSPEAR components within an enclave (SGX/LPC variant) is now possible almost at the run-time level of process-based isolation, putting the expected costs of using SGX trusted execution technology for our application-level policy enforcement approach into perspective.

5 RELATED WORK

This section summarized works related to our application-level policy enforcement approach considering a precise TCB perimeter.

A step towards more precisely identifiable application TCB perimeters is observable in SELinux with user-space object managers (Loscocco and Smalley, 2001). The approach is based on auxiliary application-level constructs for managing application objects as SELinux policy objects and for supplementing the system-wide policy with application policy rules based on OS-specific policy semantics (Carter, 2007; Walsh, 2007). Although collecting and locating all policy rules in the kernel's policy runtime environment is effective in terms of policy protection and analyzability, this approach causes an increase of each application TCB by the OS policy as well as its runtime functionality and gives up any policy individuality.

An approach beyond is the SELinux Policy Server Architecture (Tresys Technology, 2014; MacMillan et al., 2006). With the goal of a clear separation between OS-level and application-level policies, the user-space security server (USSS) is placed at the application level running all application-specific policies. Although the USSS fulfills a comparable role to the AppSPEAR TPS, the USSS exists only once at the application level and is not instantiated individually per application in contrast to AppSPEAR TPSs. This approach leads to considerably larger application TCBs and, additionally, has to deal with problems known from multi- and meta-policy systems (Bonatti et al., 2002). Due to unknown reasons, the project was abandoned.

6 CONCLUSION

This paper tackles the problem of large and complex TCBs of current approaches for application-level security policy enforcement. We propose the security architecture framework AppSPEAR: By isolating AppSPEAR components from untrusted application logic and by applying isolation between AppSPEAR components using mechanisms on different hardware/software levels, implementation variants enable a fine-grained balancing of rigor regarding the reference monitor principles as well as isolation/communication costs, and thus, adjusting an AppSPEAR implementation to specific application requirements. The practical evaluation shows that the expected runtime overhead of using TEE/enclave-based isolation can be significantly reduced by using caching and asynchronous enclave calls. While considerably reducing TCB implementation size and complexity compared to conventional mechanisms such as process-based isolation, SGX enables trusted enforcement of application-level policies in AppSPEAR implementations.

Ongoing work focuses on two main areas: (1.) We are investigating approaches for increasing memory safety of AppSPEAR implementations. The first step is being taken by implementing our framework in Rust.

(2.) We are extending developer support: By enabling compiler-supported code generation of policy and policy runtime environments (TPSs) from policy representations in domain-specific languages (Amthor and Schlegel, 2020), we aim to embed our approach into model-based security policy engineering workflows.

REFERENCES

Alam, M., Zhang, X., Khan, K., and Ali, G. (2011). xDAuth: A Scalable and Lightweight Framework for Cross Domain Access Control and Delegation. In *SACMAT '11*, pages 31–40.

Amthor, P. and Schlegel, M. (2020). Towards Language Support for Model-based Security Policy Engineering. In *SECRYPT '20*, pages 513–521.

Anderson, J. P. (1972). Computer Security Technology Planning Study. Tech. Rep. ESD-TR-73-51, Vol. II.

Bezawada, B., Haefner, K., and Ray, I. (2018). Securing Home IoT Environments with Attribute-Based Access Control. In *ABAC '18*, pages 43–53.

Bhatti, R., Ghafoor, A., Bertino, E., and Joshi, J. B. D. (2005). X-GTRBAC: An XML-based Policy Specification Framework and Architecture for Enterprise-wide Access Control. *TISSEC*, 8(2):187–227.

Biswas, P., Sandhu, R., and Krishnan, R. (2016). Label-Based Access Control: An ABAC Model with Enumerated Authorization Policy. In *ABAC '16*, pages 1–12.

Bonatti, P. A., De Capitani di Vimercati, S., and Pierangela, S. (2002). An Algebra for Composing Access Control Policies. *TISSEC*, 5(1):1–35.

Carter, J. (2007). Using GConf as an Example of How to Create an Userspace Object Manager. In *SEinux Symp. '07*.

Chen, J., Shang, W., Hassan, A. E., Wang, Y., and Lin, J. (2019). An Experience Report of Generating Load Tests Using Log-recovered Workloads at Varying Granularities of User Behaviour. In *ASE '19*.

Crampton, J., Gutin, G., and Watrigant, R. (2016). Resiliency Policies in Access Control Revisited. In *SACMAT '16*, pages 101–111.

Ferraiolo, D., Kuhn, D. R., and Chandramouli, R. (2007). *Role-Based Access Control*. Artech House. Sec. Ed.

Ferraiolo, D. F., Chandramouli, R., Kuhn, R., and Hu, V. C. (2016). Extensible Access Control Markup Language (XACML) and Next Generation Access Control (NGAC). In *ABAC '16*, pages 13–24.

Ferraiolo, D. F., Gavrila, S. I., and Jansen, W. (2015). Policy Machine: Features, Architecture, and Specification. Tech. Rep. NISTIR 7987 Rev 1.

Fong, P. W. L. (2011). Relationship-Based Access Control: Protection Model and Policy Language. In *CODASPY '11*, pages 191–202.

Gupta, M., Patwa, F., and Sandhu, R. (2018). An Attribute-Based Access Control Model for Secure Big Data Processing in Hadoop Ecosystem. In *ABAC '18*, pages 13–24.

Hipp, D. R., Kennedy, D., and Mistachkin, J. (2020). SQLite Version 3.32.3. https://www.sqlite.org/src/info/7ebdfa80be8e8e73.

Hsu, A. C. and Ray, I. (2016). Specification and Enforcement of Location-Aware Attribute-Based Access Control for Online Social Networks. In *ABAC '16*, pages 25–34.

IBM (2016). Db2 11.1 – Label-based Access Control Overview. https://www.ibm.com/support/knowledgecenter/en/SSEPGG_11.1.0/com.ibm.db2.luw.admin.sec.doc/doc/c0021114.html.

Intel Corporation (2019). Intel® SGX SDK for Linux* OS – Developer Reference. https://download.01.org/intel-sgx/sgx-linux/2.7.1/docs/Intel_SGX_Developer_Reference_Linux_2.7.1_Open_Source.pdf.

Intel Corporation (2021). Intel® Software Guard Extensions. https://software.intel.com/en-us/sgx.

Jin, X., Krishnan, R., and Sandhu, R. S. (2012). A Unified Attribute-Based Access Control Model Covering DAC, MAC and RBAC. In *DBSec '12*, pages 41–55.

Johnson, S., Zimmerman, D., and B., D. (2016). Intel® SGX: Debug, Production, Pre-release. https://software.intel.com/en-us/blogs/2016/01/07/intel-sgx-debug-production-prelease-whats-the-difference.

Loscocco, P. A. and Smalley, S. D. (2001). Integrating Flexible Support for Security Policies into the Linux Operating System. In *ATC '01*, pages 29–42.

MacMillan, K., Brindle, J., Mayer, F., Caplan, D., and Tang, J. (2006). Design and Implementation of the SELinux Policy Management Server. In *SELinux Symp. '06*.

Ni, Q., Bertino, E., and Lobo, J. (2010). Risk-Based Access Control Systems Built on Fuzzy Inferences. In *AsiaCCS '10*, pages 250–260.

OpenMRS Inc. (2021a). OpenMRS. https://openmrs.org.

OpenMRS Inc. (2021b). OpenMRS Demo Data. https://wiki.openmrs.org/display/RES/Demo+Data.

Oracle (2018). Oracle Label Security Administrator's Guide, 18c. https://docs.oracle.com/en/database/oracle/oracle-database/18/olsag/index.html.

Rizvi, S. Z. R., Fong, P. W., Crampton, J., and Sellwood, J. (2015). Relationship-Based Access Control for an Open-Source Medical Records System. In *SACMAT '15*, pages 113–124.

Schlegel, M. and Amthor, P. (2020). Beyond Administration: A Modeling Scheme Supporting the Dynamic Analysis of Role-based Access Control Policies. In *SECRYPT '20*, pages 431–442.

Schlegel, M. and Amthor, P. (2021). The Missing Piece of the ABAC Puzzle: A Modeling Scheme for Dynamic Analysis. In *SECRYPT '21*.

Shebaro, B., Oluwatimi, O., and Bertino, E. (2014). Context-based Access Control Systems for Mobile Devices. *TDSC*, 12(2):150–163.

Shu, R., Wang, P., Gorski III, S. A., Andow, B., Nadkarni, A., Deshotels, L., Gionta, J., Enck, W., and Gu, X. (2016). A Study of Security Isolation Techniques. *Comp. Surv.*, 49(3):50:1–50:37.

Smalley, S. and Craig, R. (2013). Security Enhanced (SE) Android: Bringing Flexible MAC to Android. In *NDSS '13*.

Smalley, S. D., Vance, C., and Salamon, W. (2001). Implementing SELinux as a Linux Security Module. NAI Labs Rep. 01-043.

Spencer, R., Smalley, S. D., Loscocco, P. A., Hibler, M., Andersen, D., and Lepreau, J. (1999). The Flask Security Architecture: System Support for Diverse Security Policies. In *Secur. '99*, pages 123–139.

Stoller, S. D., Yang, P., Gofman, M. I., and R., R. C. (2011). Symbolic Reachability Analysis for Parameterized Role-Based Access Control. *Comp. & Secur.*, 30(2–3):148–164.

Sze, W. K., Mital, B., and Sekar, R. (2014). Towards More Usable Information Flow Policies for Contemporary Operating Systems. In *SACMAT '14*, pages 75–84.

Tresys Technology (2014). SELinux Policy Server. http://oss.tresys.com/archive/policy-server.php.

Tripunitara, M. V. and Li, N. (2013). The Foundational Work of Harrison-Ruzzo-Ullman Revisited. *TDSC*, 10(1):28–39.

Walsh, E. F. (2007). Application of the Flask Architecture to the X Window System Server. In *SELinux Symp. '07*.

Wang, Q. and Li, N. (2010). Satisfiability and Resiliency in Workflow Authorization Systems. *TISSEC*, 13(4):40:1–40:35.

Watson, R. N. M. (2013). A Decade of OS Access-control Extensibility. *Queue*, 11(1):20:20–20:41.

Weisse, O., Bertacco, V., and Austin, T. (2017). Regaining Lost Cycles with HotCalls: A Fast Interface for SGX Secure Enclaves. In *ISCA '17*, pages 81–93.

Wolfe, B. A., Mamlin, B. W., Biondich, P. G., Fraser, H. S. F., Jazayeri, D., Allen, C., Miranda, J., and Tierney, W. M. (2006). The OpenMRS System: Collaborating Toward an Open Source EMR for Developing Countries. In *AMIA Ann. Symp. '06*, page 1146.