UiT The Arctic University of Norway

Faculty of Science and Technology
Department of Computer Science

Diggi

A Distributed Serverless Runtime for Developing Trusted Cloud Services

Anders Tungeland Gjerdrum

A dissertation for the degree of Philosophiae Doctor – July 2020

Til Jan-Henry (1956 - 2018)

Abstract

Cloud computing offers the convenience of outsourcing storage and processing power to a public shared environment. Physical infrastructure is managed by the cloud provider, allowing hosted services to be deployed without any upfront investment. Cloud infrastructure may additionally manage deployment, migration, scalability, and fault tolerance, transparently from the hosted service. Serverless computing, and more specifically FAAS, is a natural continuation of this trend, narrowing the computational scope down to individually deployable *cloud functions*, which are scalable and billable on demand.

Contemporary cloud services require that sensitive data such as user identifiable information be protected from unauthorized access. However, a conventional cloud security threat models assumes that the underlying public cloud infrastructure can be trusted. Physical attacks on server hardware conducted by an unfaithful employee may compromise the entire software stack. Moreover, a compromised operating system or hypervisor may directly inspect information in less privileged execution contexts.

Dedicated hardware such as TEEs mitigate such attacks by enabling privileged application containers, protected from inspection by the untrusted underlying system. Intel SGX introduces one such hardware system implementing support for hosting secure segments of code and data (enclaves) on commodity x86-64 platforms. Enclaves may be attested remotely, however the attestation evidence is limited to the enclave's initial state. SGX is considered feature rich compared to similar TEEs, however, the threat model of SGX leads to some architectural intrinsics which may impact the runtime performance.

This thesis present the design and implementation of Diggi; an efficient trusted cloud function runtime implemented in SGX. Diggi enables the development of secure applications, composed of multiple persistent and accountable cloud functions which may be jointly authenticated through co-attestation. We demonstrate that the design of Diggi is practical, and additionally, that it reduces the overhead of SGX compared with standard runtime execution techniques. We further demonstrate the applicability of Diggi by implementing two pseudo-real application workloads demonstrating a database management system and a machine learning inference pipeline on top of the Diggi runtime.

Acknowledgements

I would like to express my sincere gratitude to those who made this dissertation possible.

First of all, i would like to thank UiT: The Arctic University of Norway, for hosting me and funding my research for the last 4 years, it has been an honor.

I would also like to extend my gratitude towards the technical staff and administration, both at the department and faculty level. Particularly i would like to thank Svein Tore, Jan, Maria, Kai-Even, Jon Ivar and Ken Arne for being able to answer just about any inquiry, be it acquisition of hardware, sick-leave or other administrative work; simplifying my life as a Phd-student.

My co-advisors, Håvard, Dag and Robbert have contributed with endless discussions on core computer science problems, and have thought me the fundamentals of distributed systems research, and for that i am forever grateful. I would also like to thank the other Phd-students at our lab: Magnus, Enrico, Tor-Arne, and Aakash, for valuable insights and interesting discussions. Additionally, i would like to thank Robert, Lars, Helge and Eleanor for their direct contributions to Diggi, both engineering work and in-depth discussion on trusted distributed systems. Moreover, Fred, for much needed input on scientific writing.

Research is hard on the spirit, and i would especially like to thank all my friends and family for supporting me and cheering me on throughout this process.

Lastly, a huge thank you to my life partner, Sigrun, without whom, none of this would have been possible. I love you.

Contents

List of Figures

Acronyms

AEX Asynchronous Enclave Exit
AP Application Processor
API Application Programming Interface
ARM Advanced RISC Machine
ASIC Application-Specific Integrated Circuit
ASLR Address-Space Layout Randomization
ATM Automatic Teller Machine
AWS Amazon Web Services
AXI Advanced eXtnesible Interface
CDN Content Delivery Network
CISC Complex Instruction-Set Computer
CLR Common Language Runtime
CPU Central Processing Unit
CU Computation Unit
DBMS Database Management System
DMA Direct Memory Access
DNS Domain Name System
DRAM Dynamic Random-Access Memory
EPC Enclave Page Cache
EPCM Enclave Page Cache Map
EPID Enhanced Privacy ID
FaaS Function-as-a-Service
GDPR General Data Protection Regulation
HIPAA Health Insurance Portability and Accountability Act
HMAC Hash-based Message Authentication Code
HTTP Hypertext Transfer Protocol
IaaS Infrastructure-as-a-Service
IAS Intel Attestation Service
IIC Integrated Circuit Card
IOT Internet of Things
IPC Inter-Process Communication
IPI Inter-Processor Interrupt
ISA Instruction Set Architecture
ISV Independent Software Vendor
JIT Just-In-Time

JVM Java Virtual Machine
LE Launch Enclave
MAC Message Authentication Code
MEE Memory Encryption Engine
MMU Memory Management Unit
MSE Mean Square Error
NIC Network Interface Card
NUMA Non-Uniform Memory Access
OLTP OnLine Transaction Processing
OS Operating System
PE Provisioning Enclave
PGP Pretty Good Privacy
PRM Processor Reserved Memory
QE Quoting Enclave
QPI Quick Path Interconnect
RDMA Remote Direct Memory Access
RISC Reduced Instruction-Set Computer
RNG Random Number Generator
ROM Read-Only Memory
RPC Remote Procedure Call
RTSP Real-Time Streaming Protocol
RTT Round Trip Time
SDK Software Development Kit
SECS SGX Enclave Control Structure
SEP Secure Enclave Processor
SGX Software Guard eXtensions
SHTTP Secure Hypertext Transfer Protocol
SIM Subscriber Identity Module
SLOC Source Lines Of Code
SMC Secure Monitor Call
SMM System Management Mode
SMT Simultaneous Multi-Threading
SoC System-on-a-Chip
SSA Save State Area
STL Standard Template Library
TCB Trusted Computing Base
TCP Transmission Control Protocol
TCS Thread Control Structure
TEE Trusted Execution Environment
TLB Translation Lookaside Buffer
TLS Transport Layer Security
TPM Trusted Platform Modules
TSX Transactional Synchronization Extentions
VA Version Array

VM Virtual Machine
VMM Virtual Machine Monitor
WAL Write Ahead Log
WAN Wide Area Network

1

Introduction

Connectivity is considered an essential part of modern life. The Internet offers new ways to manage personal memory, interaction, and consumption through online services such as cloud storage, social media, instant messaging, and online shopping. Connected devices perceive and record large quantities of personal information enabling online services to infer and provide *intelligent* capabilities to end-users. Monetization occurs through the promise of access to rich, convenient and delightful services in exchange for targeted advertisement.

Physical infrastructure for hosting online services is often managed by a public cloud; an Infrastructure-as-a-Service (IaaS) provider. Delegating the management of infrastructure reduces operational expenses, increases service reliability, and provide more predictable cost estimates [59]. The risk of investment is reduced, as cloud infrastructure is rented rather than owned.

Hosting privacy sensitive data and computations in a public cloud requires that services trust the underlying infrastructure. Infrastructure may be compromised by exploits or tampering, rendering the upper layers of the software and hardware stack visible to an attacker. An unfaithful employee of the cloud provider could potentially mount a privileged physical attack against a hosted service. Additionally, software may contain misconfigurations or bugs, which cause involuntary information leakage. System software is particularly susceptible to bugs and misconfigurations due to the complex nature of low-level engineering. Supply-chains may also be compromised by tampered infrastructure, where logging devices may be placed on hardware system buses [150]. Given these concerns, privacy-compliant online services should implement techniques to shield application software and data from the untrusted infras-

tructure.

For modern cloud services, composability and separation of concerns simplify scaling and fault tolerance. *Microservices* provide a composable abstraction for developing complex and highly scalable cloud services [130]. Unlike traditional monolithic services, microservices are developed as multiple single-purpose distributed units of application logic. These loosely coupled components present a composite service through well defined networking protocols such as Hypertext Transfer Protocol (HTTP), Apache Thrift [11] or Google Remote Procedure Call (RPC) [73]. Microservices are commonly deployed at scale using container technology, implementing an isolated and virtually dedicated Operating System (OS) despite a shared infrastructure. Deployment, fault tolerance and scaling is simplified using technologies such as Docker Swarm [54] or Kubernetes [34].

Serverless computing [37] iterates on these technological achievements to offer automatically managed cloud infrastructure. *Cloud Functions*, or FaaS, are the primary manifestation of this paradigm, reducing the unit of scalable computation to individual event-driven functions, deployed on demand. Events may trigger function invocation through client requests, message queues, database changes, or timer-based operations. Complex online services may be implemented as collections of cloud functions interacting in synchrony.

Most major cloud providers support cloud functions, including AWS Lambda Functions [16], Microsoft Azure Functions [17], and Google Cloud Functions [42]. This single-purpose service-oriented abstraction automates many properties of cloud software, including fault tolerance, scalability, availability, and placement. Application code is decoupled from explicit knowledge of the underlying computing resources, allowing services to automatically scale on demand, hence the moniker *serverless computing* [82]. Compared to traditional cloud hosting in which servers are rented per *time unit*, serverless functions are rented per *invocation* [99].

This thesis investigates techniques for shielding contemporary cloud services from an untrusted infrastructure, including the replacement of current shielding techniques with more efficient ones. We include a preliminary analysis of available technological foundations for securing software in an untrusted environment, and additionally a selective empirical study of eligible technology.

1.1 Trusted Execution Environments

Modern commodity processors implement hardware support for shielding applications from an underlying system [153] [124]. TEEs enable hosted code to execute in secure and trusted *compartments* without requiring explicit trust in the underlying platform. Distinct CPU modes separate secure and non-secure

execution; providing the ability for secure modules to remain encrypted in memory during execution. The system bus will prohibit all access requests to *secure memory* from the non-secure execution mode. Code executing inside a secure module is able to prove the correctness of both software and hardware to a third party. This is achieved through *software attestation*, a process in which the trusted hardware produces a *quote*, containing a signed hash of the secure modules' code and data. The key used to sign the hash, or *measurement*, is derived from material unique to the hardware platform. By serving this quote upon request, the secure module is able to prove the following two properties: the identity of the initial state (code and data), and the authenticity of the hardware platform, firmware and trusted platform services.

The most mature and available TEE is Intel SGX. Although proprietary, SGX has since release received significant scrutiny from the research community [14, 22, 31, 180, 116, 194, 158, 164, 75, 138, 190, 195, 177]. Most Intel CPUs developed after 2015 support SGX, both in server-grade hardware and client desktops/laptops[1].

SGX allow multiple mutually distrusting *enclaves* to run concurrently on a single physical host. This property is unique to SGX, whereas other TEEs only provide a single *secure world* per physical host [153]. All parts of the secure platform are implemented in signed firmware or hardware [46] and no operation conducted by the trusted hardware is visible to the untrusted host operating system. We observe that the compartmentalization of software, emblematic of serverless computing, lends itself elegantly to the *enclave* programming abstraction.

SGX enclaves are compiled and deployed as regular shared library objects, however, limited by memory consumption. During boot-up, SGX-cabable firmware sets aside a range of dedicated physical memory exclusively for SGX. This is currently restricted to 128MB. Over-subscribing memory will cause physical pages to be multiplexed among multiple enclaves; similar to conventional virtual memory. There is, however, an additional performance penalty for multiplexing secure memory [65]. The initial version of SGX only permits enclaves to statically allocate memory at creation time, an issue addressed in a later revision [125]

Enclaves execute in a higher privilege-level than the surrounding system, yet relies on untrusted system software to handle interrupt processing and resource management. Interrupts generated by system software and hardware must explicitly exit this privileged mode before being serviced. This indirection adds a significant performance penalty.

Enclaves may be authenticated by remote attestation, however submitted evidence is solely based on the initial measurement identifying its predicate state. As execution progresses, state is mutated, diverging from the initial identity. For long running enclaves, authenticating the initial state is a weaker

1. github.com/ayeks/SGX-hardware

identity for the mutated state.

The threat model for enclaves further provides no guarantee for verifiability of execution. An enclave may be interrupted or subverted by the host, and determining whether an event has verifiably occurred requires additional security measures.

Code executing inside an SGX enclave considers the environment outside to be untrustworthy, including the operator of the hardware. The untrusted software and hardware may actively attempt to subvert execution in order to gain access to data. SGX protects the integrity and confidentiality of code and data inside an enclave. The authenticity of the hardware platform, firmware and code executing inside an enclave may be verified remotely through attestation. Secrets may, following a successful attestation process, be securely provisioned to the enclave through a confidential and integrity preserving communication channel. All interaction with the outside world is visible to an attacker and may be stored, modified and replayed back to the enclave at any point. SGX does not protect against exploitable side-channel attacks such as cache analysis [75], however modifications to software and/or hardware have proven to harden systems against such attacks [133]

SGX, and TEEs in general, do not protect from denial-of-service attacks which prevent application code from progressing. The underlying system may actively withhold resources, such as network packets, memory pages, I/O-resources or thread time-slices.

1.2 Thesis Statement

Serverless computing is a contemporary cloud service abstraction which simplify deployment, scale, management and billing of distributed applications in a public cloud [99].

Physical attacks, bugs and misconfigurations may compromise cloud infrastructure, rendering less privileged services exposed. A protected runtime should shield privacy-sensitive data from an untrusted cloud and protect the authenticity and verifiability of execution. TEEs enable cloud hosted, and authentic software, shielded from an untrusted underlying infrastructure.

Developing an efficient serverless application runtime capable of TEEs requires significant work in analysis, design and engineering.

The main hypothesis of this thesis is therefore:

TEEs can be leveraged to build a secure and efficient serverless application runtime for trusted computing in a public cloud.

This thesis will have particular focus on the design and implementation of *Diggi*, a distributed runtime for secure native cloud functions.

To evaluate the applicability of TEE in context of serverless computing, we conduct a precursory survey of serverless computing systems, detailing the opportunities and challenges in implementing a secure serverless runtime. The set of capabilities supported by SGX suggests it to be the most applicable TEE for cloud computing. To validate this claim, we compare it to alternative candidate TEEs and capable trusted hardware systems.

A comprehensive baseline performance analysis of the programming primitives comprising the SGX platform will allow us to deduce a set of general advisory principles for implementing efficient and secure systems using Intel SGX. Insights gathered from this analysis will then be integrated, along with the threat model, into a complete system outlining the design requirements of the Diggi runtime.

To demonstrate that the Diggi runtime is practical, we will implement a prototype satisfying these requirements. Our evaluation will confirm this practicality, by micro and macro benchmarks. This includes a layered analysis of the compounding effects of different security measures implemented in Diggi. Additionally, to demonstrate applicability of our prototype system, we implement a rudimentary application simulating a privacy-sensitive workload.

1.3 Scope and Limitations

We design and implement a prototype serverless runtime for trusted execution in a public cloud. Completeness is not a first-order concern and throughout, this thesis assumes a set of properties at the boundary of our limited prototype design:

- Cloud providers are able to host Diggi on top of bare-metal physical hosts or equivalent virtual machines which make SGX capabilities available to the guest OS.
- Legislative requirements may altogether prohibit hosting certain privacy-sensitive data in a public cloud. However, we conjecture that the techniques described here may partially alleviate articles in regulatory frameworks such as the General Data Protection Regulation (GDPR) [60] or the Health Insurance Portability and Accountability Act (HIPAA) [80].
- Although automatic management and scaling of cloud functions is a core feature of serverless computing, we refrain from discussing it in significant detail in this dissertation. We consider this an engineering problem, orthogonal to the focus of this thesis. We primarily focus on the development of a secure *runtime* for hosting cloud functions, and where applicable, we demonstrate scaling potential to support this claim.
- Diggi assumes a crash-stop [32] model for all distributed processes, and assume benign execution of authenticated functions in TEEs. Runtime

behaviour violating these preconditions, will cause the system to intentionally crash.
- Although Diggi cloud functions demonstrate persistence, we consider state in functions to be ephemeral and state is not replicated. However, persistence is not a fundamental limitation to serverless computing; future system may include the ability to maintain distributed persisted state, with application-tailored consistency models.
- Serverless platforms isolate the host operating system and cohosted tenants using virtual machines. Diggi supports cross-tenant isolation but does not implement host protection. We consider this problem complementary and previous works demonstrate that a solution is practical [14] [85].

1.4 Methodology

Modern science may be defined as the endeavor of repeated systematic study of phenomenon, both ethereal and physical. It encompasses the organization of knowledge into verifiable or reproducible claims, of which further study is built upon. Following controversy, scientific progress is achieved when reaching consensus on claimed truths. Natural, or formal sciences, follow the hypo-deductive method [68]. Observations of processes or phenomena lead to the formation of a generalization in form of a hypothesis, tested by logical deduction or experiments. This process is iterated until the test results matches the expected outcome, i.e there are no errors in the experiment.

The final report of the ACM Task Force on the Core of Computer Science [43] presents a new taxonomy for classifying computing as a science. Computing research is rooted in three paradigms, *theory*, *abstraction* and *design*. *Theory* is the foundation for logical reasoning and mathematical sciences. *Abstraction* is the applied method of natural sciences, where the formation of hypothesis and models are validated through experiments. *Design* is defined by the iterative process of solving problems based on specification and implementation.

Theory is rooted in mathematics and consists of the following procedure:

1. Describe or characterize the phenomenon.
2. Pose a hypothesis based on the characterization.
3. Prove the hypothesis by way of logical deduction to determine the truthfulness of the characteristics.
4. Analyze the results, testing the proof to the object or phenomenon in observation.

These steps are iterated for as long as errors or inconsistencies are present.

Abstraction is founded in the experimental scientific method with 4 distinct stages:

1. Form a hypothesis based on the reasoned relationship of phenomenons or expected logical outcomes.
2. Construct a model representing this hypothesis, and predict observed behaviour.
3. Design experimental parameters to increase the certainty of the phenomenons.
4. Collect and analyze empirical evidence from the resulting experiments in congruence with the initial hypothesis.

The stages may be iterated upon until the hypothesis successfully predicts observed behaviour.

Design is founded in the practical engineering discipline, and involve the following steps:

1. Based on a series of observation, state a set of requirements for which a system must fulfill.
2. Design and implement a prototype system according to these specifications.
3. Construct a set of tests to evaluate the system in conformance with the initial requirements.

These steps may be repeated until the requirements are fulfilled.

The paradigms are archetypes of the scientific method, and computer science borrows from all three. In algorithm research, the construction of algorithms to model the characteristics of a phenomenon and the formal proof thereof, are funded in theory.

In machine learning, or computational intelligence, hypotheses are formed based on a expected correlation of separate phenomenon. A learned model is then constructed to predict this correlation, and experiments validate the conformance through classification.

Software engineering describes an initial problem statement through requirements and specifications of a system, which is implemented and its validity tested in conformance with the requirements. Although all computer science is permeated by theory, computer science is not only a science of the artificial [51]. Information processes and computing preexist the earliest descriptions of physical computing devices.

This thesis is funded in the subfield of computing referred to as *systems research*, mixing abstraction, theory and design. We use abstraction to describe

the proprietary Intel SGX platform, and test our hypothesis by experimental measurements. Additionally theory, to reason about the security properties of our hypothetical system, proposing algorithmic solutions to preserve confidentiality, integrity and authenticity. Based on the proposed hypotheses, we design a system model through a set of requirements. We then implement a prototype system, *Diggi*, and evaluate its conformance with stated requirements through experimentation.

1.5 Research Context

The research presented in this dissertation was conducted in the context of the *Corpore Sano* research group, exploring the intersection between computer science and life sciences. Notable research targets elite sports performance development and injury prevention, preventive healthcare, large scale population screening, and epidemiological health studies. By applying systems research to these fields, Corpore Sano aims to disrupt state-of-the-art monitoring and analysis within the field. In support of these goals, systems research conducted by the group covers all abstraction of the software stack. This includes data collection and processing, big data and machine learning applications, fault tolerance, and software mechanism for privacy and policy enforcement.

The Corpore Sano center is a natural continuation of research conducted as part of the iAD (information Access Disrupted) research group, funded by the Norwegian Research council as a center for research driven innovation (SFI 2007-2015). To place this dissertation in context, we summarize relevant previous research contributions by the group.

With the advent of commodity support for hardware based virtual machines, the public cloud enable consolidation of computing resources to provide IaaS. The Vortex omni-kernel [111] implements a cloud centric operating system from the ground up to support resource isolation between tenants. Vortex introduced a novel approach to resource scheduling and attribution using message aggregates which delegate messaging resources based on policy.

The Internet revolution lead to vast improvements in indexed knowledge, and introduce information retrieval tools to a global audience. To ensure the relevance of retrieved content, sophisticated processing tools and computation pipelines are necessary. Cogset [179] implements a distributed big-data processing engine based on the map-reduce pattern [48] exploring a novel approach to scheduling compute tasks based on data-locality, leading to an increase in performance relative to competing systems [191].

In publish subscribe systems, active queries are able to process streams of information, ensuring responsive and practical inference on data. Streaming queries are generally stateful constructs which are difficult to distribute across multiple physical hosts. Brenna et al. [29] extend the Cayuga [28] stream

processing engine to enable scalable distribution of queries represented through non-deterministic finite state automata.

Peer-to-peer systems promises a highly scalable way of organizing computing resources into structured or unstructured groups. Systems which implement decentralized computing are more fault tolerant and elastic than their centralized counterparts. The blockchain [170] is the most prominent example of a decentralized architecture which share a common ancestry to these systems. Fireflies [98] implemented a practical scalable overlay network which supports Byzantine Fault-Tolerant (BFT) membership agreement. By organizing members into a probabilistically verifiable pseudo-random network, attackers are unable to modify membership views of correct participants.

Cloud providers promise large scale computation for enterprises at low cost and low risk of investment. However, enterprises may have restrictions in migrating workloads onto the cloud. Policies such as lack of vendor lock-in, hybrid clouds, data locality, and retention, require precisely tagged and described data. Balava [131] investigates data migration between heterogeneous cloud providers using *metacode*, including combinations of private and public (hybrid) clouds. Balava reduces vendor lock-in, and enables timely migration of on-premise software stacks onto the cloud.

Content Delivery Networks (CDNs) provide a simple way of caching read-only data close to consumers, such as video on-demand services. This read-only infrastructure is not well suited to handle write workloads and dynamically adaptable content. Jovaku [140] investigates reusing existing cloud infrastructure to create a middle-tier caching layer using the Domain Name System (DNS).

Ensuring authentic access to sensitive data is adamant to develop secure distribution. Renesse et al. [148] implement a mobile smart access control abstraction through meta-code embedded in x509 certificates. LoNet [97] implements policy enforcement through automated transparent information-flow de-identification enforced by an inter-positioned reference monitor between data and requestor.

Proof of concept implementations of these systems address many initial questions emerging from the cloud revolution. As cloud technology has matured, more diverse frameworks which simplify development of cloud services push more of the management responsibility onto the infrastructure provider. The container abstraction lets the developer focus on software development, rather than operating system management. Serverless computing reduces the scope of computing further, automating management and scaling completely. As more aspects of cloud software is managed by cloud providers, consumers still require strict privacy and security guarantees for software executing in the public cloud. This thesis focuses mainly on the mechanism necessary for secure distributed analytics and data processing in the cloud.

1.6 Impact

Outside of publications, the work presented in this dissertation have afforded several distinct and noteworthy contributions and collaborations which are listed here.

As part of special curriculum requirements for master students at *UiT: The Arctic University of Norway*, two student project assignments were written based on the foundations of Diggi. These explored the development of an encrypted file system in SGX supporting durable storage of secrets, and an investigative survey on software replacements for TEEs using homomorphic encryption schemes.

In his master thesis Hoff [83] developed a distributed in-memory key-value store for caching sensitive data built on top of the Diggi runtime. The prototype achieved practical performance, demonstrating the applicability of Diggi.

Part of the dissertation period included a research internship at Microsoft Research Systems Group in Redmond, Washington. The internship lead to explicit acknowledgement for contributions made to published work [161]. Additionally, it awarded co-authorship on two US patent [35][36], and a submitted journal article detailing Multi-Version Concurrency Control, pending review [162].

As part of the development of Diggi, posters introducing core aspects of the design were presented at ACM SIGOPS Symposium on Operating Systems Principles 2017 Shanghai, China, and the ACM SIGOPS 1st Summer School on Advanced Topics in Systems 2018. Several talks on the foundations of Diggi have been made as part of seminars related to the masters level course "*INF-3203: Advanced Distributed Systems*" at UiT. Diggi is moreover part of the official curriculum for the spring semester of 2020.

1.7 Summary of Contributions

This thesis is based on the work presented in [64, 65, 62, 25, 63]. We map the individual contributions in each publication to the work presented in this thesis, and list the novel concepts.

1.7.1 Publication I

Anders T Gjerdrum et al. “Implementing informed consent as information-flow policies for secure analytics on ehealth data: Principles and practices.” In: *2016 IEEE First International Conference on Connected Health: Applications, Systems and Engineering Technologies (CHASE)*. IEEE, 2016, pp. 107–112

This paper introduces the *CSano* architecture, which outlines a privacy-compliant distributed system architecture for curation and processing of privacy-sensitive data gathered from cyber-physical systems. CSano introduces the concept of *vaults* for storing gathered data and Computation Units (CUs) for processing data. Vaults and CUs are composed into a distributed processing pipeline where *information-flow* policies, enforced through security labels, ensure that taint does not cross isolation boundaries in a multi-tenant pipeline. CSano describes a distributed tamperproof log structure in which all events in the processing pipeline are logged. Accountability is achieved through deterrence in view of a possible audit. An audit process may replay the logging events to determine the correctness of processed data. The audit process may additionally prove revocation of rights in the event where consent is withdrawn from the system. Computational units and vaults are implemented as individual containers, isolated from one another. Vaults are implemented as personal mysql databases hosted in separate containers. The cohosting potential for multi-tenant environments is demonstrated empirically, achieving practical cohosting of 70 separate vaults.

This work introduces the foundation for Diggi, presenting an initial prototype distributed application runtime for protecting privacy-sensitive data. It introduces the concept of accountable reproduction of application execution and tenant/user isolation of compute and storage resources. However, this work does not protect from a malicious host, and requires a high degree of trust in the cloud infrastructure, an issue addressed in later work.

1.7.2 Publication II and III

Anders T Gjerdrum et al. “Performance of Trusted Computing in Cloud Infrastructures with Intel SGX.” In: *CLOSER*. SCITEPRESS, 2017, pp. 668–675

This paper explores the performance implications of hosting software in SGX enclaves. We evaluate the technology by constructing a set of micro-architectural benchmarking experiments targeting key traits of the SGX platform, including context-switching, memory management and deployment. We analyze the empirical evidence from the experiments in conformance with detailed descriptions of the internals, and devise a set of programming principles for developing efficient software with Intel SGX.

Anders T Gjerdrum et al. “Cloud Computing and Service Science: 7th International Conference, CLOSER 2017, Porto, Portugal, April 24–26, 2017, Revised Selected Papers.” In: vol. 864. Springer, 2018, pp. 1–18

A selection of papers from the conference where subsequently invited to contribute extended versions for the Springer Cloud Computing and Services

Science Journal. The extended journal version included additional benchmarks on multithreading performance, and principles recommending asynchrony to maximize system utilization.

The Diggi runtime is built from the ground-up to satisfy these principles.

1.7.3 Publication IV

Eleanor Birrell et al. "SGX Enforcement of Use-Based Privacy." In: *Proceedings of the 2018 Workshop on Privacy in the Electronic Society*. WPES 18. Toronto, Canada: ACM, 2018, pp. 155–167

This work evaluates several architectural layouts for for interposing reference monitoring to enforce use-based privacy. By delegating enforcement to authenticated SGX enclaves external to the trust domain, distributed systems may reduce data transfers by enforcing data access remotely, local to the host consuming the data. Program attestation may enforce fine-grained policies as the attestation evidence is inherently tied to evidence of usage.

The evaluation was implemented in an early incarnation of the Diggi runtime prototype, and the concepts introduced here further influence the design of Diggi.

1.7.4 Publication V

Anders T Gjerdrum et al. "Diggi: A Secure Framework for Hosting Native Cloud Functions with Minimal Trust." In: *The 1st IEEE International Conference on Trust, Privacy and Security in Intelligent Systems, and Applications (TPS)*. IEEE, 2019

This work introduces Diggi, a distributed cloud function runtime for hosting native privacy-sensitive applications in an untrusted FaaS cloud. Functions in Diggi are developed through an asynchronous flow-based programming abstraction. This abstraction enables efficient usage of limited SGX resources, maximizing the cohosting potential per server instance of Diggi. Diggi implements co-attestation of cloud function to implement distributed trusted architectures, consisting of hundreds of cloud functions. We evaluate the efficiency of Diggi and demonstrate that our solution is practical, reducing the TCB compared to previous work. To the best of our knowledge, Diggi is the first asynchronous native FaaS runtime for hosting trusted cloud functions using TEEs.

This work presents the core runtime and security properties of the Diggi prototype, including secure ephemeral storage, co-attestation, asynchronous

execution, and communication via tasks and flows.

1.7.5 Novel Concepts

Founded in the publications listed above and additional research, a we summarize the core novel concepts introduced in this thesis:

Persistent cloud functions: To broaden the spectrum of applications which may be implemented via *cloud functions*, we introduce persistence. Each function may securely manage ephemeral session state, available throughout its lifetime. A function may additionally be *sticky*, and implement multi-request session interaction, beneficial for streaming workflows.

Accountable function execution: We introduce a message-based dynamic attestation process for identifying enclave software beyond initial attestation. By storing message exchanges in a tamperproof logging structure, functions in Diggi are accountable. This property further extends to non-deterministic functions. Execution may be audited by replaying messages and comparing the expected result to the dynamic attestation proof.

Shielded native cloud function execution runtime: We design and implement an efficient, native and fully asynchronous shielded runtime for cloud function execution. The runtime adopts best practices for shielded software in SGX, including low memory footprint, exit-less communication and careful partitioning of program logic.

Protocol for multiparty co-attestation: We extend the Intel-provided stock attestation protocol and introduce a multiparty secure attestation and key distribution protocol for distributed applications. A cloud function may identify and authenticate several other cloud functions in a non-trusted cloud.

1.8 Outline

This thesis is structured as follows:

Chapter 2 studies serverless systems, including challenges and opportunities for serverless computing in an untrusted cloud.

Chapter 3 describe several Trusted Execution Environments and other hardware-based trusted computing systems, with particular emphasis on SGX.

Chapter 4 conducts a baseline experimental analysis on the micro-architectural features of the SGX TEE. Based on the outcome, we derive 7 *performance principles* for designing efficient SGX-capable software. We then derive a set of functional and non-functional requirements for which a secure and efficient serverless cloud application runtime must satisfy. Based on these requirements, we introduce the design of Diggi.

Chapter 5 and 6 details the Diggi prototype implementation, securing serverless applications from an untrusted cloud. The prototype implementation includes a trusted asynchronous micro runtime, ephemeral state protection, record-and-replay accountability, system call translation for legacy libraries, cloud function deployment and co-attestation.

Chapter 7 evaluates the prototype implementation, and demonstrates the practicality of the system design through analysis and discussion of the empirical results. To demonstrate applicability, we introduce a sample application implemented as as a collection of Diggi cloud functions.

Chapter 8 discusses Diggi in the context of relevant and topically similar related work.

Chapter 9 presents concluding remarks, discussing contributions, along with opportunities for future research.

2

Serverless Computing

Cloud computing enables the development of scalable software, through rented hardware resources on-demand [59]. Serverless computing [82] [99] extends this by empowering developers to create complex and demanding online services without any up-front investment. Serverless applications are able to automatically scale on platform engagement, and costs are directly proportional to revenue streams such as click-based advertisement. This linear relationship reduces commercial risk for businesses interested in developing online services.

A distinction is made between renting computing capabilities (*Serverfull*), versus paying for computations (*Serverless*). Services dynamically scale the allocated physical infrastructure based on incoming request load, and service owners only pay the cost of resources actually consumed. Contrary to Infrastructure-as-a-Service (IaaS), FaaS provides a simple and fine-grained interface through stateless, reactive singular units of computation called *cloud functions*.

Serverless is incorrectly used for an assortment of concepts boasting automatic scalability. Serverless also defines a finegrained and self-contained compute unit separate from infrastructure [21]. The scope of this treatment is therefore limited to FaaS, or *Cloud Functions*.

2.1 Advantages of Serverless Computing

Serverless is, despite what the name suggests, dependent on servers. However, developing serverless applications arguably involves less management than the conventional counterparts. Operational concerns such as scalability, load-balancing, deployment, availability, geo-location are automatically manageable by cloud infrastructure software. The full software stack, ranging from Virtual Machine (VM)s to application libraries, is transparently managed by the cloud provider. This leads to a potentially lower bar for onboarding developers, as much of the complexity of the cloud is hidden.

From the cloud provider's point of view, decoupling execution and management simplifies the upgrade procedure. Less heterogeneous runtime environments also simplify the development and testing of cloud infrastructure software. A cloud provider may also exercise flexibility in choosing the physical hardware required hosting cloud functions. Renting IaaS implies specifying the tier of resources to claim, including storage capacity, network bandwidth and compute capabilities [59]. FaaS however, allows computing instances from multiple generations of hardware, increasing longevity of infrastructure investments.

Pricing per function-invocation reduces the risk of investment for both the cloud consumer and provider. Consumers may develop services where costs are directly proportional to engagement. For providers, datacenter resource consumption is tied to revenue, mirroring the cost of power. Additionally, packing fine-grained units of compute efficiently onto physical hosts permits higher infrastructure utilization.

2.2 The Cloud Function Abstraction

Cloud functions are stateless, event-driven, and independently scalable unit of application logic. Statelessness simplifies scalability and load balancing as functions are not required to keep multi-session data. Subsequent requests for a service may be directed to any available instance, or a new, triggering a scale-out. Functions requiring state are traditionally coupled with secondary storage components such as databases or blob storage. Persisted state services are often long running, and cost is therefore harder to precisely estimate.

Serverless computing is considered more beneficial for compute-intensive tasks or applications with bursty access patterns. For IO bound services, partitioning the application architecture into separate functions may be cost-efficient, however, increase development complexity.

Major providers support cloud function development through a wide variety of programming languages, either natively or through workarounds allowing arbitrary code execution. The most popular natively supported choices are

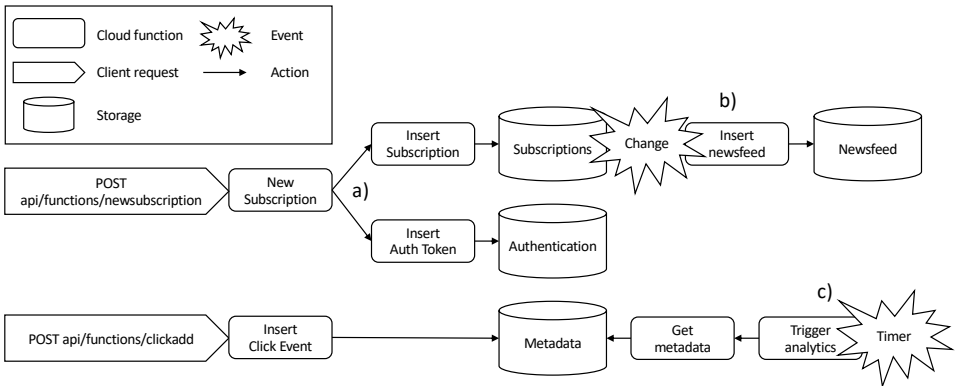


Figure 2.1: Conceptualization of a sample serverless architecture: a) Simplify complex APIs by aggregation. b) Allows change-based triggers to implement propagation of information. c) Allows batch oriented tasks for triggering analytics workflows. A reactive version may trigger analytics similarly to b).

Golang, Python, C# and Node.js (JavaScript). The natively supported among each, offer more tightly integrated library and infrastructure support as opposed to workarounds.

Cloud functions are reactive, listening for particular events which trigger the execution of the functions. Typical event-triggers include database changes, or more complex operations such as newsfeed subscriptions or social media account creation. Events may be triggered on timed intervals, similar to *cron*-jobs, performing maintenance tasks, and batch processing. Similar to microservices, cloud functions are single purpose units of computations, and may be composed into more complex services, as illustrated in Figure 2.1. Complex cloud APIs may be aggregated by cloud functions into simpler client-facing APIs. Events may lead to other function invocations, which again produces new events to create a chain of computing behaviour.

2.3 Pricing Model

Attribution of cost in serverless architectures is fine-grained; cloud functions are invoked on-demand and priced per invocation. This allows developers to precisely monitor the cost of operation of a service, and automatically scale out in the event of a request surge.

A FaaS-runtime attempts to optimize execution of cloud functions to maximize placement. Assuming hardware has a fixed cost of acquisition per unit C_h, including power, space rental, developer costs etc. Given a set of invocations per host I_h, an optimal execution plan for hosting cloud functions on a given

host will maximize the utilizations ratio (UR_h):

$$UR_h = \max_{C_h} \frac{I_h}{C_h}$$

Maximizing utilization may be solved by either minimizing the cost of hardware or increasing invocations per hardware unit.

Provisioning cloud functions must also be efficient, and execution latency is directly proportional to utilization. Considering a cloud function which takes 10s to complete and an additional 0.5s to provision. Given that the resource is occupied for the duration, this would imply a theoretical 10 percent underutilization.

Although billed per invocation, most FaaSs additionally charge for longevity of invocation, and particularly long running functions are restricted by design [16]. Precision in attribution therefore comes at the expense of functionality.

2.4 Architecture

Multi-tenancy is a predicate for efficient consolidation of hardware resources in serverless computing. Cloud functions executing in the context of different tenants must therefore be isolated; both by fair resource scheduling, and confidentially, ensuring no information leakage across tenants. Serverless infrastructure implements several techniques to isolate function execution. Most commonly in VMs, encapsulating execution environments into separate instances. To reduce the overhead of provisioning and execution, other mechanisms for isolation may be used in combination, such as containers [2] [78]. Redundant layers provide host protection and efficient tenant runtime provisioning. Unikernels [118] hosted on top of VMs, library operating systems inside containers [144] and high-level language runtimes [44] [74] [95] are also possible isolation mechanism. A common technique to offset some of the provisioning cost of VMs, containers, and runtimes, is to create pools of dormant compute capability which are preallocated in anticipation of request load. Additionally, runtimes belonging to the same tenant may be reused to save provisioning costs. Some support ephemeral (temporary) storage for cloud functions, erased once the runtime is decommissioned.

To implement scheduling for cloud function invocations, a fault-tolerant distributed queuing system must be in place. The queue must accept request and schedule them in a timely manner onto free execution slots as advertised by the physical hosts. If cloud functions are required to be idempotent, as is the case for most, the failure model may be simplified. Stricter *exactly-once*[24] execution semantics may be substituted for *at-least-once*. Functions with side-effects, such as a financial transaction, must internally ensure that "double

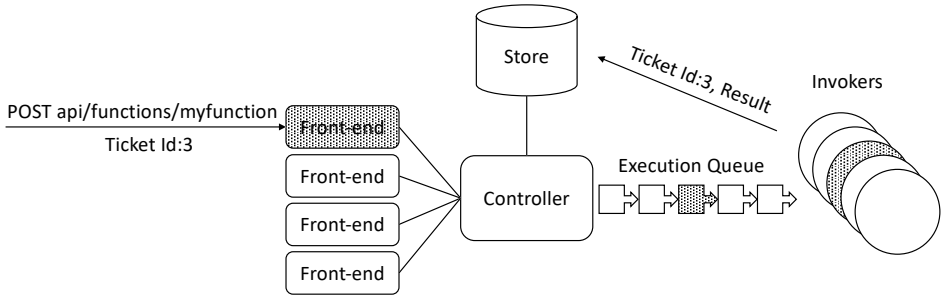


Figure 2.2: Invoking a cloud function in a serverless application framework. The client first requests execution through a REST-full API, the front-end forwards the request to the controller, which authenticates and schedules it for execution. The result is stored, and may be retrieved through a subsequent request.

spending" is detected and corrected.

Figure 2.2 depicts how a serverless application framework invokes a cloud function. The frontend receives the request through a stateless interface, and forwards it to the controller. The controller authenticates the requester and authorizes the access, fetches the corresponding function and places it on the execution queue. Once placed on the queue, it receives a ticket id. Depending on how functions are scheduled, the queued request is eventually assigned to an invoker, a VM or container. The output, or result, is then persisted in a database under the corresponding id. The invoker issues an additional request to obtain the result of the function execution.

2.5 Challenges

The latency for provisioning resources in a serverless runtime significantly impacts the total revenue for a cloud provider by how many functions are charged per time-unit for a given physical host. The overhead, or *cold start* latency, should therefore be minimal. Isolation capabilities such as containers and virtual machines serve multiple purposes; security isolation, preventing information leakage, and service management. In current serverless infrastructure, these present overlapping functionality by redundant security measures [178]. A JavaScript engine shares isolation capabilities provided by the underlying container, VMs or both.

A cold start penalty may partially be caused by virtual machine or container deployment. For optimized serverless runtimes, pre-provisioned virtual machines are specialized for a particular tenant during the cold start process [178]. Bootstrapping the language runtime environment and function

binary may additionally contribute to latency. Dormant functions may exist in a pre-provisioned state, referred to as *warm start*. Mismanaged capacity planning (pre-provisioning) may additionally cause the execution queue to grow boundless. Chained functions which fan-out execution, may propagate latency or increase demand. This may have cascading effects for composite serverless applications where an increase in load from a function upstream may unpredictably increase load downward in the chain [126].

In addition to temporal restrictions, cloud functions have a fixed limit on temporary storage and memory usage. Cost is attributed per invocation and determining the execution time for individual cloud functions is difficult. A common problem is scheduling request queues without the ability to precisely determine execution time. A pessimistic scheduling approach might under-provision functions to target hosts, however, over-provisioning may violate QoS constraints.

Conventional cloud functions are only suitable for developing a particular class of applications. Software library support is often restrictive and application developers must adhere to the suite of APIs available in a given runtime environment. Scaling stateful application services while maintaining consistency is considered a non-trivial problem. Services such as Spanner [45] and DynamoDB [49] attempt to relax consistency requirements, but essentially require complex agreement protocols [112] to manage distribution of state. Cloud functions are therefore ordinarily limited to stateless applications.

2.6 Comparable Concepts

The concept of serverless may arguably not be constrained to the cloud, however, may be interpreted as a symbiotic construct to edge computing. In Fog/Edge computing, IoT devices with serverless application constructs may offset some of the bandwidth cost by performing upstream evaluation on sensor data prior to ingestion by cloud services [163].

Platform-as-a-Service (PaaS) predates FaaS, however serve a similar purpose. PaaS provides a full-fledged application framework and storage service for developing automatically scalable web applications [72]. Cloud functions serve as a more lightweight generalization of the concepts initially made available through PaaS [99]. Figure 2.3 illustrates where FaaS fit into the cloud computing continuum compared to other common abstractions.

The concept of a software construct executing single purpose operations based on requests, bears similarity to mobile software agents [96] [105]. A software agent is a computer program which acts on behalf of a program or end user to achieve a prescribed task. Software agents do not require user interaction and are self activated. Agents may furthermore be persistent, exist in a waiting state while perceiving context, and at any point autonomously

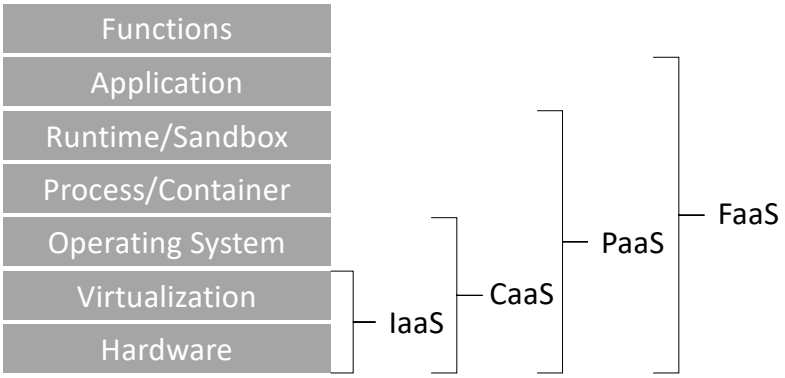


Figure 2.3: The cloud computing continuum of abstractions.

decide to activate itself. There are no architectural limitations in serverless computing which prohibit this comparison. Serverless functions may be triggered by temporal parameters, essentially implementing this behaviour. Although traditionally serverless computing does not demonstrate persistence, we argue that it is not an innate property of the paradigm.

	Google	**Azure**	**AWS**
Edge	No	No	Lambda@Edge
Max longevity	540s	600s[1]	900s
Arbitrary code	?	No	Yes
Storage Options	Object, Cloud, File, Block, Firebase, and Cloud SQL	Azure Blob, Queue, Files, CosmosDB, and Table	AWS S3, DynamoDB, Aurora, RDS, SQS, and Kinesis Streams
Max Memory	2GB	1.5GB[1]	3GB
Isolation[2]	GVisor	Windows Nanoserver (Docker)	EC2 container service (Amazon Linux)
Native Languages	Node.js, Go, Python	C#, Java, JavaScript, PowerShell, Python, TypeScript	Java, Go, PowerShell, Node,js, C#, Python, Ruby
Request Size	10MB	No Limit	6MB[3]
Composition	Cloud Composer (Apache Airflow)	Azure Logic Apps, Durable Functions	Step Functions

Table 2.1: A comparison of the major cloud providers' FaaS offerings.

1. Consumption plan
2. Not including language runtime
3. For synchronous functions

2.7 Proprietary Implementations

The largest public cloud providers all offer FaaS as part of their suite of cloud services [16][17][42]. We detail the similarities and differences between these platforms where reliable documentation is available, a summary of which is shown in table 2.1. All provide automatic scalability, fault tolerance, infrastructure management and broad support of cloud service connectors. Functions may implement sophisticated conditions for activation, including modifications to storage and temporal triggers for maintenance tasks. All offer pricing models driven by cost attribution per invocation. All support development through JavaScript and Python, but only AWS support API extensions for arbitrary function binaries.

Cloud functions are hosted in containers to simplify deployment, additionally hosted in virtual machines. While containers offer resource and dependency isolation, VMs are traditionally considered more appropriate tools for security isolation. However, colocated tenants on a single VM are vulnerable to side-channel attacks [149]. Wang et al. [184] demonstrate colocation of cloud functions from multiple tenants for Azure cloud functions. No successful colocation was observed for AWS or Azure. Investigations [184] confirm that cloud functions for all major providers are hosted on heterogeneous hardware transparent to the developer, optimizing infrastructure utilization.

Wang et al. [184] studies the characteristics of the three public cloud serverless platforms described, by way of large scale deployment and runtime experiments. All serverless platforms studied exhibit relatively high cold-start latencies. And the non-native runtimes studied show a significant overhead on compute heavy workloads. No multisession function support is mentioned in any of the publicly available documentation. Multiple requests will not necessarily hit the same runtime container, and to support sessions, functions instead require external storage with strong consistency. *Stateful* services are implemented via function composition frameworks, coupling stateless cloud functions with storage services.

2.7.1 AWS Lambda

AWS *Lambda* is the most mature public FaaS platform, launched in 2014. Each Lambda is hosted inside a container which again are hosted in per-tenant dedicated VMs. Tenant separation only applies to developer identities and the implementer must ensure that different application tenants, belonging to the same developer identity, do not share containers [184]. AWS supports Lambda development through multiple language runtimes; an example implemented in JavaScript is listed in 2.1. This function may be configured to respond to storage events, client events or other function requests.

```
// EventName: HelloFunction
exports.handler = (event, context, callback) => {
        callback(null, "Hello," + event.who + "!");
};

// Event Configuration
{
    "who": "AWS Step Functions"
}

// State Machine Definition
{
  "Comment": "A Hello World example",
  "StartAt": "HelloWorld",
  "States": {
    "HelloWorld": {
      "Type": "Task",
      "Resource": "aws:lambda:*...*:HelloFunction",
      "End": true
    }
  }
}
```

Listing 2.1: Example Lambda Step Function, consisting of a event function, configuration and state machine definition.

AWS *Step Functions* allow developers to create workflows which combine Lambdas, storage services and other autoscaling services to create application workflows. Steps are executed as a state machine consisting of individual events with inputs and outputs, crafting long-running workloads for applications. Figure 2.1 illustrates the state machine definition for a simple step function. Examples of use include machine-learning inference, big data ingestion and batch processing, revenue stream report generation, and subscription services for email newsletters.

Lambdas scale automatically based on load and provide fault tolerant access across availability zones within each geolocated region. Execution parameters may specify pre-provisioned resources for applications requiring lower latency. Pre-provisioned functions incur an additional cost per unit of memory (GB per seconds) allocated.

Lambdas are essentially restricted containers allowing execution of arbitrary code, simplified through developer APIs for code submission and preparation. Once Lambdas complete, the execution context is suspended along with all processes executing within that container. The architecture reuses execution contexts for subsequent function execution and memory objects may persist across invocations, however, container reuse is not guaranteed.

Execution contexts additionally provide temporary filesystem storage which remains throughout its lifetime. Lambdas reuse execution contexts between invocations to reduce cold-start latency. This is not a deterministic property of

the system, and cannot be used to reliably preserve function state.

AWS Lambda manages all system software and library packages, limiting developers to the provided software modules. To ease developer transition in the event of changes to software library support, Amazon lists the planned deprecation time for various software runtimes in the Lambda developer guide.

2.7.2 Azure Functions

Azure functions, similarly to AWS, support a variety of different programming languages. Package managers for C# and JavaScript, nuGet and NPM are supported, providing more flexibility in application composition. Like Lambdas, Azure also supports pre-warmed functions, with an additional cost as a fixed price per time unit.

Azure supports *Durable Functions*, which is a framework for designing stateful applications similar to Step Functions. These implement a dedicated function composition pattern; with orchestrator, entity, activity and client functions. Orchestrator functions are used to organize the execution of other durable functions in a function app. Activity functions are general purpose and act as the most basic unit of work in durable function composition. They may be used for any task requiring I/O or compute, and are considered stateless. Activity functions may be orchestrated in any way, in parallel, serially or as a chain. Reliability is achieved by entity functions, declared to store state in an append only record store, providing eventual consistency guarantees for transactional data. This property also enables audit through stored historical records of a functions state, called the Durable Task Framework. All persisted state in durable functions is stored in *Azure Storage* transparently, and high availability is achieved by deploying passive function state in parallel to separate failover regions. As is, functions are required to be idempotent and the framework guarantees *at-least once* semantics during execution of functions.

Client functions act as an initial request point triggering an orchestration of durable functions to create application behaviour. Like AWS, Azure supports trigger APIs connected to external cloud services, such as *Azure Blob Storage*, *Data Lake* and hybrid cloud connected software. Azure functions run in a custom runtime on top of a dedicated per-instance container. Although not specified by public documentation, we speculate that the container service is additionally hosted in a virtual machine.

2.7.3 Google Cloud Functions

Google cloud functions offer very similar set of features compared to the previously detailed proprietary FaaS platforms. However, through a comparatively

reduced selection of programming languages; Go, JavaScript and Python. Functions are separated into multiple types:

- *HTTP functions* - trigger on explicit request events, either by clients or other functions.
- *Background functions* - trigger on service events, such as changes in cloud storage, analytics, and timed events.

Like Azure and AWS, developers are expected to implement idempotent functions, however semantics differ among function types. *Background functions* are invoked *at-least once* while *HTTP functions* are invoked *at-most once*.

Cloud functions are hosted in GVisor [78] container sandboxes. Gvisor implements a hardened container runtime, with dedicated per-container kernel to limit the interaction with the host OS. Conventional containers are vulnerable to privilege escalation attacks, and previous efforts have demonstrated the ability to escape containment [66]. This hardening would allow multi-tenancy in containers, which could improve function packing (Utilization). This is, however, not confirmed by any public documentation.

2.8 Open Source Implementations

Baldini et al. [20] introduce the first open source serverless computing framework for developing cloud native event-based mobile applications. OpenWhisk implements FaaS through what it refers to as *actions* and *triggers* which are bound together by via *rules*. This *triple* is packaged and deployed in docker containers to provide automatic scaling capabilities. Figure 2.2 illustrates a prototype architectural composition inspired by the OpenWhisk architecture.

Actions may be implemented in multiple high-level languages, including binary code hosted in Docker containers. Functions may be configured to trigger on a multitude of event types, including everything from Internet of Things (IOT)-based sensor readings to database changes. OpenWhisk uses a Nginx frontend to handle incoming trigger requests, and Apache Kafka to manage reliable request queues for function execution. Access control lists, authentication metadata, function code, parameters, session state and results are all stored in CouchDB. A custom load-balancer handles scheduling of requests for execution onto correct request queues for execution in containers. Containers may be located in virtual machines if necessary. Once requests are stored on the queues, an execution id is returned to the trigger source for asynchronous execution. A synchronous mode also exists in which the trigger waits for completion.

IBM Cloud Functions [86] are based on Apache OpenWhisk, sporting similar capabilities to that of AWS, Azure and Google Cloud. OpenLambda [82], Open-

FaaS [160], the Fn Project [58] and Iron Functions [135] are other noteworthy examples of open source FaaS frameworks.

2.9 FaaS in Research Literature

Akkus et al. [4] implement a high performance serverless runtime using application-level sandboxing and a hierarchical message bus. Sand claims to provide lower latency, better resource efficiency, and more elasticity than existing platforms, and achieves a 43 percent speedup compared to OpenWhisk. Sprocket [10] introduces a video processing framework which uses intra-video parallelism to create a low latency scalable and cost efficient serverless pipeline built on AWS Lambda. Kaffes et al. [100] argue the case for a cluster-level centralized scheduler for serverless computing. They claim to reduce load skew and queuing in function deployments, and reduce the interference between cohosted functions.

Jangda et al. [94] examine the unique properties of serverless computing, and present formal descriptions of these, outlining some of the inherent problems with this computing abstraction. They examine the problems of maintaining state in serverless functions, privacy implications of runtime reuse to avoid cold start, and idempotent execution. The authors argue that these problems may be further increased for function composition. By defining a naive function as the ideal, without state persistence, concurrency and idempotency, the authors are able to define precisely when these low-level runtime internals can be ignored by developers.

Alpernas et al. [7] introduce dynamic information flow control to manage taint in cloud functions. This work recognizes the problem of large Trusted Computing Base (TCB)s in monolithic cloud applications and leverages serverless application composition to reduce taint propagation for unmodified applications.

2.10 Summary

This chapter has discussed the architectural background for serverless computing, detailing several deployed cloud services available for use, demonstrating the advantages of a simplified cloud computing paradigm.

Despite the benefits, all contemporary cloud services require that applications trust the underlying infrastructure. In the next chapter we will detail hardware systems which protect applications and services from an untrusted underlying system.

3

Trusted Execution Environments

Services hosted in a public cloud trade convenience for security, implicitly trusting the surrounding infrastructure owned and operated by the cloud provider. TEEs implement general computing capabilities in shielded application containers, accommodating secure software despite an untrusted software/hardware stack. This section details the attributes of serverless computing and several widely available trusted hardware systems precluding the design of a trusted cloud-computing runtime.

Modern Reduced Instruction-Set Computer (RISC)/Complex Instruction-Set Computer (CISC) processors partition software into separate privilege groups. The highest privileged mode, or level of access, is granted rights which are a superset of lower privileged modes. For example, an operating system has the ability to access memory pages and persisted data belonging to a process executing in the less privileged user-level execution mode. The process abstraction allows for portable and modular software decoupled from the underlying system architecture, including physical media, concurrency, and memory management. Processes moreover fail independently without compromising the integrity of system software.

This computing model assumes that system software is more trustworthy than less privileged software. Modern operating systems are considered highly complex constructs, which have repeatedly been shown to be vulnerable to exploits compromising its integrity [33]. Because of the privilege layer abstraction present in modern processor architectures, this implicitly compromises

the integrity of all lower privileged software.

In cloud-native services, where the public cloud provider hosts infrastructure and system software, it is reasonable to require protection from a potentially malicious underlying infrastructure. Unfaithful servants with physical access to the hardware may further attempt to compromise the security of hosted software.

TEEs are isolated execution constructs which grant the ability to develop secure application modules despite a potentially malicious environment. Depending on implementation, a secure module is shielded from inspection by most non-secure software and hardware, including the hypervisor, operating system and process runtime. Code executing inside a secure module is able to prove the correctness of both software and hardware to a third party through remote software attestation. The Global Platform Consortium claims the term TEE through specific standards detailing system architecture and secure application APIs [67]. We apply a broader description of the term, as described above.

The following section describes different approaches to implementing TEEs, comparing their strengths and weaknesses. Based on reasoning which will become apparent, particular emphasis is held on describing Intel SGX. The different approaches to TEEs guard against different types of attacks, and where practical we will make the distinction.

3.1 Intel Software Guard Extensions

Intel SGX extends the x86 instruction-set first introduced for the Skylake generation of Intel's *Core* microarchitecture [124]. SGX *enclaves* contain application code and data segments which are shielded from the underlying operating system and other non-trusted physical hardware. By shielded, we imply that the content is confidential, and the integrity of the enclave is preserved. Enclave code may use most of the instructions available in the x86-64 Instruction Set Architecture (ISA), which enables existing software libraries to be ported into secure modules at near-native performance. Distinct privilege modes separate secure and non-secure execution, and architectural modifications to the processor cores and Memory Management Unit (MMU) implement shielded enclave memory. The system bus will protect against unsolicited access requests to enclave memory by non-privileged execution modes. SGX holds a smaller inherent TCB than other mechanisms for authenticating software, comprising the enclave, CPU package and the secure firmware implementation.

Enclaves may be authenticated remotely via attestation, which enables external services such as clients to verify the identity of an enclave. To attest an enclave, trusted hardware constructs a *quote*, containing a signed hash of the secure modules' code and data pages. The signature is generated by a

key uniquely identifying a known benign TEE hardware platform instance. By serving this quote, the enclave is able to prove to a requestor exactly what code is executing, and the identity of the underlying hardware platform instance.

SGX is mostly implemented in the CPU microcode-architecture, not accessible from the operating system, executing underneath the ISA abstraction. The microcode facility in modern x86 processors enable complex instructions to be implemented as a composition of multiple simpler instructions. This essentially bridges the gap between the CISC architecture which the instruction set exposes to system software, and a RISC architecture executing on the processor cores. By implementing more complex operations in microcode, development time for new processor functionality is significantly reduced, and microcode firmware may be updated to account for bugs and vulnerabilities in processors after manufacturing.

The internals of SGX are sparsely described in literature released by the manufacturer [87] [88]. However, Costan and Devadas [46] provide a comprehensive introduction to the internals of SGX based on publicly released developer guides, the Intel x86-64 Software Developer Manual and patent applications, bridging the knowledge gaps with qualified conjecture. Serving as a de facto source for SGX internals within the research community, we base most of our description on the details in this comprehensive treatment. This section only describes the details which are necessary for reasoning about the design and implementation of Diggi. We refer to this treatment for a more complete description of the SGX implementation.

McKeen et al. [125] introduce improvements to SGX by adding dynamic paging to the SGX specifications. Hardware supporting this improvement was not available to us during the work on this thesis. These modifications are considered incremental and does not significantly impact the security model nor the design choices made in this thesis.

3.1.1 Security Model

Traditional operating system or hypervisor software consider application software containers to be untrusted, and access to privileged resources are mediated though software interrupts or system call operations. Policies set by the operating system specifies resource consumption and access to physical data for processes.

Privileged system software have unrestricted access to the execution context of less privileged containers. SGX is built around the assumption that system software may itself be compromised. Non-trusted software, firmware and hardware may actively attempt to subvert the execution of enclaves in order to involuntarily leak secrets.

Adversaries may attempt to mount physical attacks which tap the system

bus in order to record memory accesses between the CPU package and Dynamic Random-Access Memory (DRAM). Additionally, malicious peripheral devices attached to the system bus may attempt to construct Direct Memory Access (DMA) requests targeting memory regions holding enclave secrets. SGX outsources memory management of enclave memory to system software, and although pages are encrypted, adversaries may still attempt to replay or swap evicted pages to manipulate the program flow in an enclave. During a page-fault, eviction policy alters the mapping between the linear virtual addresses seen by a process/enclave and the corresponding physical address. These stale entries in the Translation Lookaside Buffer (TLB) must be invalidated to account for changes in physical memory mapping. Adversaries may exploit these stale translations to alter program flow inside an enclave. A more sophisticated approach may attempt to swap the evicted pages on persisted media, or untrusted DRAM, preserving the virtual mapping but altering the contents of the pages.

Generally, the untrusted environment may attempt to record, modify, and replay any information or operations performed by enclaves in order to subvert the security of an enclave.

We assume that the correctness of a TEE's hardware and software platform may be attested remotely. Secrets should only be provisioned to the platform following a successful authentication across a secure communication channel. All code and data inside an authenticated TEE is considered confidential and integrity protected.

3.1.2 Known Vulnerabilities

The SGX security model does not protect against side-channel attacks, a class of vulnerabilities which TEEs generally are not well equipped for. SGX exposes several vectors for mounting side-channel attacks aimed at extracting secrets from enclave memory.

Cache timing attacks are proven to be practical [27] and among the most powerful, as they are mountable from non-privileged execution modes. A malicious enclave may utilize high resolution timers to infer information about a cohosted victim enclave. By exploiting architectural details about how Intel x86-64 processors synchronize on-core caches, an attacker may artificially construct memory operations targeting the same cache lines as the victim enclave. By interrogating these lines, the attacker is able to populate them, and infer which lines are accessed in the victim enclave by identifying cache misses through timed measurements. Data-dependent memory operations will then reveal the contents of the surveyed memory addresses.

Since SGX memory is managed by system software, page-fault operations also leak information to the untrusted system [185]. An attacker may target data dependent computations via memory management in system software.

While cache timing attacks are able to infer access at the resolution of a cache line (64 bytes), this attack vector may only observe page-level operations at 4KB intervals, reducing its potency.

SGXs system-bus memory reads and writes are encrypted before exiting the CPU-package, however, address-references are not. These may be observed by an attacker tapping the Quick Path Interconnect (QPI). Although data-dependent memory-fetches may leak information, policies such as prefetching and multicore cache-coherency protocols may generate noise, which reduces the practicality of such attacks.

An attacker may infer information on executing processes by performing power analysis external to the host. Measuring the power draw of hardware may allow an attacker to infer complexity of cryptographic operations on the host computer. Additionally, thermal inspection of hardware may leak information on internal processes from a distance.

Approaches described above may be deployed in combination with other attacks to increase precision. Some of these side-channels may be closed by clever engineering techniques such as oblivious memory [187, 156]. However, due to the high manufacturing and development cost, it is impractical to implement protection mechanisms against all side-channels. Hunt et al. [84] discuss the orthogonal design criteria for modern operating system, trading performance and interference for security. Caching, multiplexing and resource sharing will leak data between hosted environments. A complete redesign of modern system software is necessary to mitigate these vulnerabilities.

Physical attacks against SGX capable CPUs may reveal secrets directly from the core-die, however, have a considerable up-front cost in equipment and analysis. Additionally, dismantling the CPU core in order to observe the cpu logic internals through electron microscopy is a destructive, difficult and error-prone process.

Software is inherently dependent on physical hardware to execute computer instructions, making *denial-of-service* attacks by the underlying system very hard to protect against. The system may withhold resources, such as memory pages, network packets, kernel scheduler time-slices or power.

Developers of SGX applications must ensure the integrity of the development process prior to deployment. The development environment must have a trusted and verified software stack including the operating system and compiler, and be hosted on trusted physical hardware, with vetted developers and operators.

SGX is vulnerable to IAGO attacks [38] by the untrusted host operating system. Call-gate operations, ingress or egress to the enclave, may be hijacked to deliver false information. System software should not be trusted with procedures which alter the program flow. Hardware based Random Number Generators (RNGs) or high-resolution timers served by operating system device drivers may be modifiable by an attacker. Examples include uniform distribution scheduling, randomized sample selection, or encryption algorithms. The

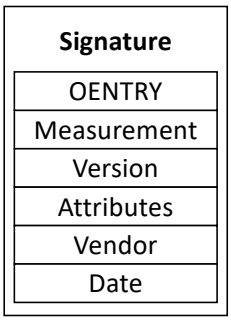


Figure 3.1: The SIGSTRUCT certificate structure identifies a deployable enclave and corresponding author. Additionally, it contains valid entry points (OENTRY), version and product line identifiers, and feature (attribute) masks to specify enabled CPU-modes.

attacker may attempt to alter program flow to escalate privileges or disclose secrets. SGX implements dedicated services for obtaining secure randomness and secure time.

Design choices in hardware may additionally impact the security of SGX enclaves. Simultaneous Multi-Threading (SMT) features in modern CPUs, implement a shared execution pipeline for hardware threads. Instruction scheduling into pipeline segments by two distinct threads may leak timing information. Rowhammer [103] exploits hardware bugs in DRAM cell storage to trigger bit-flips in neighboring memory cells. Seaborn and Dullien [159] demonstrate how this bug may be exploited to gain unsolicited kernel privileges. Both vulnerabilities may be modified to target a SGX enclave, as illustrated by Jang et al. [93].

Weichbrodt et al. [188] detail a tool for exploiting synchronization bugs in SGX software by precise control over thread scheduling and artificial Asynchronous Enclave Exit (AEX) interrupts.

3.1.3 Enclave Lifecycle

Describing the runtime execution model of SGX is contingent on first understanding how enclaves are developed and deployed onto an initially untrusted system.

Compilation. Enclaves are distributed to the host computer as shared library objects (.so). Developers compile the enclave binary along with support libraries, and packages it together with the SIGSTRUCT, illustrated in Figure 3.1.

The SIGSTRUCT is a certificate data structure holding the enclave and author

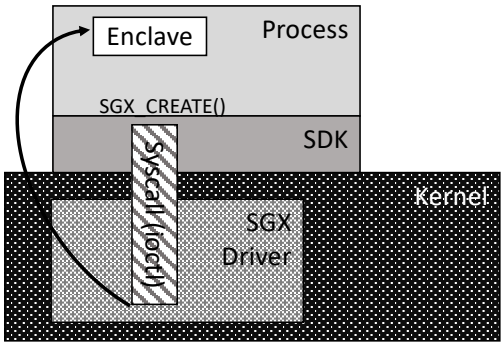


Figure 3.2: SDK interacting with the SGX kernel driver to create an enclave. Implemented via pseudo-character device, controllable through the *ioctl* system call.

identity data, the measurement, and the versioning and capability mask detailing which architectural extensions the enclave supports. The measurement is a 256-bit SHA-2 sequential hash of all compiled code and data segments. This preserves a proof of the relative page ordering in memory, as illustrated during the initialization procedure in Figure 3.4. The shared library object represents the recipe which the target SGX platform must use to identically recreate the memory layout as captured by the measurement.

The SIGSTRUCT-certificate is signed by the developers private key, P^D_{priv}. The corresponding public key (P^D_{pub}), certificate and binary is then distributed to the target physical host.

Deployment. During deployment, privileged system software is tasked with initializing the enclave. The SGX device driver manages enclave memory indirectly through dedicated instructions. Figure 3.2 illustrates the device interaction between the Linux kernel driver and the Intel SGX SDK embedded in a non-privileged process hosting the enclave.

The host process delivers the SIGSTRUCT, enclave binary and public key P^D_{pub} to the kernel through the SGX-SDK. To begin deployment, the SGX driver issues a special instruction, ECREATE, which creates a SECS allocated in the Enclave Page Cache (EPC). Each enclave has a unique SECS, responsible for storing and maintaining metadata about the enclave, including its base linear address in process memory and its size(ELRANGE), illustrated in Figure 3.3.

Additionally, the SECS contains data from the SIGSTRUCT; the author certificate, enclave version id, architectural extension masks, and the measurement created when loading the enclave.

The SECS is stored in enclave memory, however unlike other enclave support data structures, it is only accessible through SGX instructions. Mapping

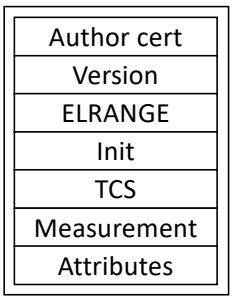


Figure 3.3: The SECS stores metadata for each unique enclave.

the SECS page into an enclave's address-space would allow it to modify its own measurement, compromising the security model. All SGX instructions are parametrized on virtual addresses and the SECS address is used to uniquely identify an enclave. ECREATE sets the *Init*-field in the SECS to false, signifying that the enclave is not yet initialized. All SGX instructions expecting an initialized enclave will check the SECS, and fails the operation if not.

SGX supports multiple, mutually distrusting, enclaves per physical host, either inside the same process or in multiple.

Thread Control Structure. ECREATE additionally allocates Thread Control Structures (TCSs) for each logical processor expected to execute inside a given enclave. During development, authors must specify how many TCS to provision and the legal entry points to the enclave. Processors may only enter *enclave-mode* through predefined call gates, and the the TCS stores valid entry points (OENTRY) defined by the SIGSTRUCT during compilation. Predefined entry points avoid uncontrolled jumps into enclave-memory, skipping crucial protection mechanisms developed to validate and sanitize input on entry.

The TCS additionally stores segment registers used for thread-local storage inside enclaves. Similarly to SECS, TCS may only be accessed by the SGX microcode architecture.

Each TCS references multiple Save State Areas (SSAs) organized contiguously in EPC memory following the TCS. The content of an SSA is populated with a thread's execution context prior to egress out of enclave mode. This includes the general registries, stack information, instruction pointer, and architectural dependent extensions such as the floating point context, SSE and AVX.

Loading enclave pages. The SECS is mapped to the physical pages owned by the enclave through the EPCM, also allocated in the Processor Reserved Memory (PRM). These include TCS and SSA datastructures, in addition to regular code

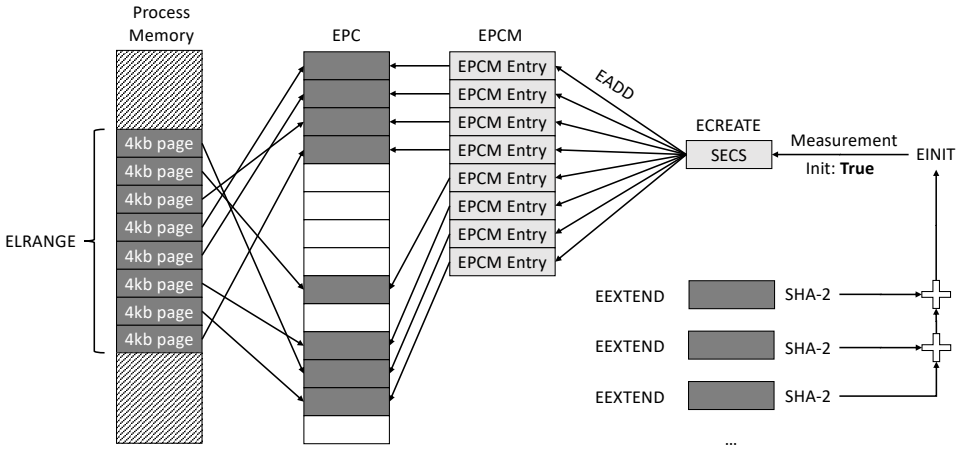


Figure 3.4: Enclave memory organization and initialization procedure. Each enclave is mapped to physical memory pages through the EPCM. The initialization procedure sequentially measures each page for comparison with the SIGSTRUCT.

and data pages. Pages may only be added to the enclaves linear address-space while the SECS is in the uninitialized state via the EADD instruction, additionally used for adding TCS pages.

The EADD instruction accepts as input each page's linear address, the access permission bits, the target memory page, and SECS address. The target memory page is then copied into a free EPC-page tagged via the EPCM, as belonging to the target enclave.

The SECS base address (ELRANGE) is used to place the position independent code into the correct virtual enclave address mapping preserving memory ordering and measurement integrity, as illustrated in Figure 3.4.

EADD is issued repeatedly by system software for each code and data page belonging to the enclave binary. Each is followed by one or more EEXTEND instructions which update the enclave measurement using the SHA-2 secure hashing algorithm. This measurement is used to verify the integrity of the loaded enclave conforming to the developer specifications via the measurement stored in the SIGSTRUCT, and additionally by software attestation.

The Launch Enclave. Once loaded into EPC-memory, an Intel-provided Launch Enclave (LE) establishes the authenticity and integrity of the enclave. The LE compares the measurement against the contents of the SIGSTRUCT and validates the SIGSTRUCT-certificate using the developers public key, P^D_{pub}. A token is issued to system software upon the successful completion of this process. This token is input to the EINIT instruction which finalizes the enclave and the measurement. EINIT moreover sets the initialized field in the SECS, after which no more pages may be added to the enclave. Once the initialization

process reveals that the signature and measurement matches that of the loaded segments, the enclave is considered successfully loaded and may be executed. The EINIT token may be stored and reused to avoid redundant steps during subsequent enclave invocations. Costan and Devadas [46] conjecture that pre-provisioned architectural enclaves such as the LE, hold hardcoded signatures enabling them to execute without a token.

Enclave Teardown. An authentic enclave executed by a correctly behaving SGX implementation will, within the limits of the security model, guarantee erasure when enclave operations halt. The teardown procedure is initiated by the host process in synchrony with the enclave. Enclaves expect that a benign process defers teardown until the enclave finishes execution.

System software is responsible for de-allocating enclave memory by issuing the EREMOVE instruction targeting each page individually. The mapping between the enclave identified by the SECS and the page is invalidated by setting the valid bit in the corresponding EPCM to zero. Additionally, EREMOVE verifies that no thread is executing inside the enclave of the target page, failing if the check is affirmative. This check serves to stop address translation attacks where pages are reused while a thread is still active. Contents of removed EPC pages are purged prior to becoming available for allocation. Lastly, the SECS and associated TCSs/SSAs are invalidated, at which point the enclave is considered successfully removed.

Evicted pages situated in regular DRAM are deallocated by benign system software. However, should the system be compromised, any recorded pages are still encrypted.

3.1.4 Memory Model

During the boot-up procedure of an SGX-enabled CPU, firmware prepares a contiguous region of memory exclusively for use by the SGX implementation. This region, referred to as PRM, sets aside a maximum of 128MB of physical DRAM only accessible from SGX microcode and enclave-mode. PRM is again divided into 4KB pages, collectively referred to as the EPC.

SGX shares a similar virtual memory layout to regular process memory. This simplifies development as existing applications may be ported with relative ease into SGX. Attempts to read or write to memory addresses within the PRM from a non-privileged mode is prohibited. This includes both kernel (ring 0) and user (ring 3) privilege-modes. Given a prohibited memory reference, write operations will be ignored completely while reads will invariably return a specific value (-1) [87].

Each enclave maintains exclusive access and ownership of allocated EPC pages, preventing secure shared memory. However, enclaves may share infor-

mation via the untrusted process memory address space, as all enclaves may read and write unrestricted the host process' heap. Inversely, SGX does not architecturally restrict enclaves from being mapped into multiple processes' address space. However, as a simplification, we treat each enclave as belonging to a single process.

Virtual enclave memory organizes enclave pages into a continuous range (ELRANGE), specified in the SECS. Addresses outside this range are mapped to the same memory layout as the host process, illustrated in Figure 3.4. Only memory within the enclave range is protected by the SGX security model, and any access to the host process memory reveals information to the untrusted system. Accessing untrusted memory is the practical vessel for which parameters to/from enclave entry/exit operations are delivered.

DMA transactions targeting memory addresses within the PRM region are rejected by the on-core memory controller. This prevents PCI-E connected devices, Network Interface Cards (NICs) or other peripherals from accessing protected memory regions. EPC pages are encrypted on write-back from the last-level cache into DRAM by the Intel Memory Encryption Engine (MEE). Attackers are not able to infer the content of memory pages stored in DRAM, however, cached memory regions in the on-core caches are kept in plaintext.

Host support. The first revision of SGX released along the *Ice Lake* generation of the Intel x86-64 core architecture, supports dynamically loading pages after initialization [125]. This enables large enclaves to be initialized with partially evicted pages, lazily loaded upon request. The revised version additionally permits more PRM to be set aside. We refrain from discussing these alterations further, since the work presented in this thesis predates general availability for hardware supporting these revisions.

Linux systems support virtual enclave paging through kernel driver extentions [89]. Developers may specify arbitrarily large enclaves, at the cost of multiplexing available physical memory. Windows does not augment the memory management system to support page eviction from PRM, and therefore has a fixed upper limit of enclave memory usage. It is unclear if this is an artifact of Windows kernel design limiting extendibility or security concerns as a result of side-channels exposed by EPC page eviction [185].

EPC Memory Multiplexing. To efficiently utilize available PRM, unused enclave pages are encrypted and evicted to DRAM. This enables practically unrestricted enclave memory usage, however with a performance penalty.

EPC-pages are assigned to enclaves via the operating system's virtual memory manager. The EPCM keeps track of EPC-pages by maintaining a map between the SECS and EPC-pages assigned to a particular enclave. Each EPCM entry stores bit fields specifying the read, write and execute permissions along with allocation and eviction state for a given page. The operating system should not be

able to access the EPC or EPCM, and consequently, page-evictions/-assignments are handled indirectly by the kernel through SGX instructions.

When an enclave attempts to access a page not present in the EPCM, a page-fault interrupt triggers an AEX. Along with all other registry context, the lowermost bits of the CR2 registry specifying the intra-page fault address are flushed prior to exit. The kernel observes the page-level faulting address, but is unable to infer intra-page access patterns. Prior to eviction, any additional logical processor cores executing within the affected enclave also perform an AEX.

Stale address translations from another processor core may modify program behavior during page eviction. Each page targeted for eviction is tagged as *blocked* in the corresponding EPCM entry to ensure no new address translations are cached in the TLB. Any reference to a blocked page either via address translation or SGX-instructions will trigger a page-fault interrupt.

The operating system is expected to expel threads from enclaves with blocked pages through an Inter-Processor Interrupt (IPI), triggering an AEX for each thread. To ensure compliance, the SGX implementation keeps track of all threads in enclaves with blocked pages.

Eviction. Page evictions are implemented through the EWB instruction, which encrypts subject pages using an ephemeral symmetric key only known to the SGX implementation. EWB first ensures the target page is blocked, and the ETRACKED instruction confirms that the relevant TLB entries are flushed. Encryption keys are purged during power transitions, rendering evicted pages unintelligible. Each page is integrity protected by a Hash-based Message Authentication Code (HMAC) stored in dedicated EPC pages (Version Arrays (VAs)). These pages may themselves be evicted from the EPC creating a tree-like structure where leaf-nodes are enclave pages.

These precautions guard against page inspection, modification, address translation manipulation, and replay attacks from a malicious system. Encrypted pages are stored in preallocated regions in DRAM managed by system software. Since page-fault interrupts triggers an AEX for each affected logical processor, the page eviction process is considered quite costly. To amortize this cost SGX supports batching eviction operations together. Not being able to inspect intra-page fault address patterns reduces the ability for memory management to predict access patterns and implement smart prefetching policies.

Assignment. Once the eviction process is complete, the resulting free EPC pages are populated by the faulting enclave. The virtual page number is extracted from the upper bits of the faulting address in the CR2 registry. The operating system memory manager retrieves the encrypted page from a map, indexed by the virtual page number. The ELDB/ELDU instruction then decrypts and copies the page into the free EPC page, and verifies the HMAC-tag saved

in the corresponding VA entry. The difference between ELDB and ELDU is that the former sets the blocked state of the page to 1, whereas the latter does not. ELDB is intended for Virtual Machine Monitor (VMM)s which must restore the correct state of a page, and uses ELDB or ELDU depending on the state when it was evicted. Regardless, ELDB/ELDU marks the VA entry as empty, preventing an attacker from reintroducing the page in a subsequent page-fault. Once the operating system is finished processing the page-fault, control is passed back to the enclave thread. It subsequently restores the execution context saved in the SSA, and retries the faulting instruction.

3.1.5 Attestation

SGX enclaves may be authenticated remotely via software attestation. The initial state is represented by the measurement constructed during enclave initialization. Platform and enclave authenticity is guaranteed by requesting that the Quoting Enclave (QE) sign this measurement using a key derived from the *Attestation Key* A_k; creating a *Quote*. The QE is a pre-provisioned enclave endowed with special privileges, allowing it to access A_k directly. A_k is provisioned to the platform instance via the Provisioning Enclave (PE), another privileged pre-provisioned enclave, as illustrated in Figure 6.6.

Local Attestation. To facilitate secure communication between the PE and QE, a mechanism for generating shared secrets is required. Local attestation allows enclaves to prove its identity to another using the EREPORT instruction. The instruction creates a cryptographic report bound by a Message Authentication Code (MAC) covering a custom provided message and the enclave certificate and measurement. The MAC is computed by a symmetric key only known to the SGX implementation and the target enclave. The report may be shared with the target enclave through untrusted memory. The recipient may then verify the authenticity by issuing the EGETEKEY instruction, returning a report key used to verify the MAC. This report key is derived by a combination of an embedded secret in the processor and the target enclave's measurement. The target enclave may similarly again issue its own report to the initial enclave. By implementing the *Diffie-Hellman* Key-Exchange protocol [52] through the custom message field in the report, a stable secure channel may be established between the two entities. This symmetric key may be used for sharing authenticated secrets to the target enclave. The report scheme does not include any guarantees for freshness, and any secure encryption protocol channel must provide such guarantees.

Enclave identities and measurements are uniquely tied to a versioning scheme, allowing multiple versions of enclaves without changing encryption keys. The SGX implementation allows local attestation to migrate secrets be-

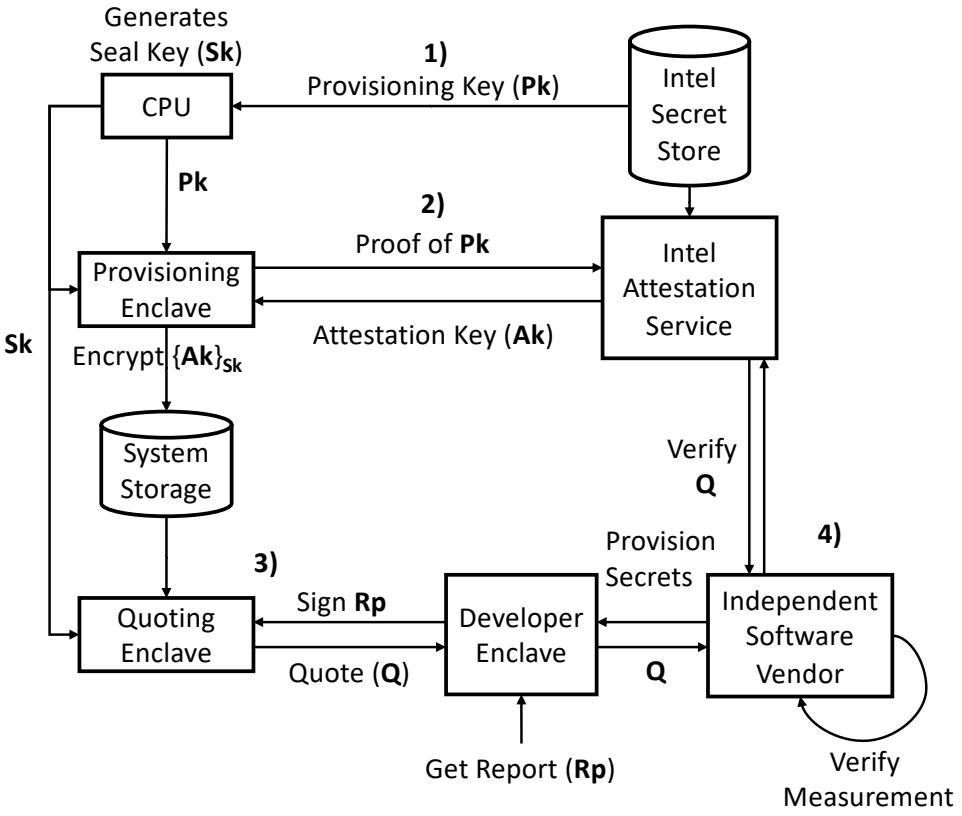


Figure 3.5: Conceptual presentation of the Intel SGX remote attestation process. 1) Intel provisions P_k to the physical chip during the manufacturing process. 2) The provisioning enclave submits a signature to IAS, proving an authentic P_K, and in response receives A_k. 3) Quoting enclave decrypts the stored A_k from storage, signs the proposed report producing the Quote. 4) The ISV receives the quote, checks the measurement and requests IAS to verify the signature.

tween different versions of the same enclave. This is however only one-way; migrating secrets from older (deprecated) enclaves to newer versions of the same enclave.

During deployment, the SIGSTRUCT loads the SECS with a PRODVN and a SVN identifier, indicating the product and enclave software version, respectively. Included in all measurements, these distinguish different versions of the same enclave. Pre-provisioned enclaves (QE, LE) also include version numbers. The attestation procedure may then prove that the SGX implementation is up to date and reject requests from deprecated implementations.

Provisioning Enclave. An SGX-capable CPU stores two secrets, the *seal secret* and the *provisioning secret*, burnt into the chips E-fuses. The provisioning secret

is generated by Intel and stored in a secret database. During the production of an SGX-enabled CPU, Intel's key generation process will provision this secret to the CPU. The seal secret is generated inside the processor after manufacturing and not known to Intel. Both are input to a derivative process resulting in the symmetric keys retrieved via the EGETKEY instruction. EGETKEY enables enclaves to retrieve two keys, the report key described above, and the seal key. By this scheme, the key distribution achieves *perfect forward secrecy*. An attacker who compromises the key-generation process, will not reveal the key generated by EGETKEY and universally compromise the security of SGX.

The *Provisioning enclave* is able to retrieve the attestation key A_k from the Intel Attestation Service (IAS)s by proving ownership of the provisioning secret P_k. A_K is then encrypted using a specifically derived seal key S_k targeting the QE, and persisted to system storage.

These steps happen during the process of bootstrapping the SGX implementation, and possibly in the event of software/firmware upgrades. Before they can use the service, developers are enrolled to the attestation service manually. The provisioning enclave must verify the developer signed measurement (SIGSTRUCT) in order to retrieve the attestation key.

Remote Attestation. The process of attestation begins by a requester, or Independent Software Vendor (ISV), issuing a request for the enclave to authenticate itself. The enclave is provisioned according to the steps outlined in Section 3.1.3 on an untrusted platform and able to receive information from the ISV through an insecure communication channel. Secrets may only be provisioned to the enclave after this process completes.

After receiving an attestation-request, the enclave asks the QE to sign a report generated by EGETREPORT, similar to local attestation. The quoting enclave decrypts A_k and signs the report. For remote attestation, the MAC is replaced with a signature produced by A_k. The attestation key uses Intel's Enhanced Privacy ID (EPID) group signature scheme, which provides anonymity for signing participants. The quote is then forwarded via ISV to the IAS, verifying that A_k signed the quote. The ISV is expected to hold the measurement identifying a valid enclave and compare it to the retrieved measurement from the quote.

The SGX SDK contains a message preparation API for remote attestation which implements a modified SIGMA protocol [107, 182]. This includes a Diffie-Hellman key exchange on top of the report exchange to establish an authentic symmetric key encrypted channel for communication between the ISV and enclave. We defer a detailed description of the protocol to Section 6.6, in the context of Diggi.

3.1.6 Context Switches

Enclaves execute in what is essentially a higher privileged cpu-mode than all other software, including operating system kernel, hypervisiors, and System Management Mode (SMM). When not in enclave-mode, any references to PRM is rejected by the memory controller. However, enclave execution is still mapped inside the same address-space as the host process. This abstraction protects system software from buggy or malicious code executing inside enclaves.

Enclaves support multithreading, however, all threads are created in non-privileged user-mode and enter enclave-mode explicitly. Code executing in enlave-mode is prohibited from issuing system calls or interrupts not explicitly handled by the SGX implementation. This implies that all interaction with host system software (OS) or the host process, must be invoked through enclave exit operations. The enclave developer defines call-gates or entry-points along with entry or exit parameters during development. These are loaded as part of the SECS when initializing the enclave, in the OENTRY-field.

Synchronous Transition. The EENTER instruction causes the executing thread to switch privilege level to *enclave-mode*, and invoke a controlled jump by setting the instruction pointer to a predefined point in EPC memory, as specified by the OENTRY field in the TCS.

The EENTER instruction accepts the virtual address of a single TCS as input. Additionally, SGX verifies that the TCS contains at least one SSA for capturing enclave context information. Prior to entry, the execution context for the thread is saved, for use once the thread eventually exists enclave-mode. For *production enclaves*, EENTER moreover checks and disables debugging and instrumentation features implemented in the processor core, such as the ability to set hardware breakpoints and hardware event sampling.

This operation bares similarity to other call-gate mechanisms, such as virtual machine entry/exits. However, EENTER may only be invoked from privilege level 3 (user level).

EENTER is invoked through the SGX SDK, which additionally sets aside a memory region within the ELRANGE to stacks. To protect against IAGO-attacks [38], parameters entering an enclave must be diligently validated and integrity checked. The trusted system marshals and boundary checks parameters before copying any potential arguments into the enclave from untrusted memory.

An enclave may exit synchronously, either to complete execution gracefully or to fetch information from the untrusted system. A thread may only invoke the EEXIT instruction while executing in enclave-mode. Prior to exit, enclave software saves the execution context in EPC memory, before restoring the stack pointer and instruction pointer to its contents prior to entry. If an enclave performs an outbound call to an untrusted resource, the enclave creates the

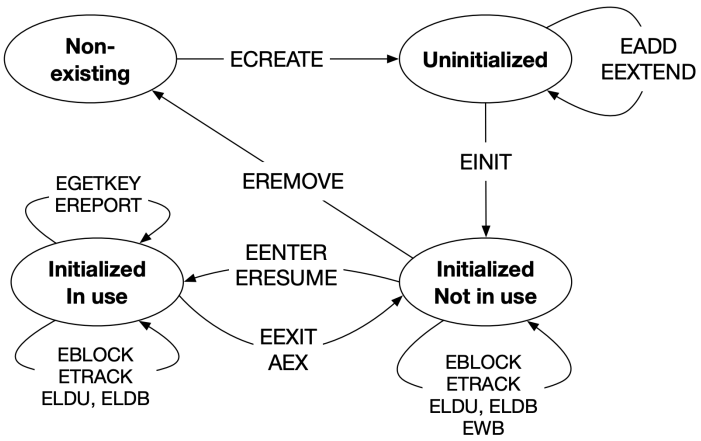


Figure 3.6: State transition diagram describing the lifecycle of an enclave [46].

stack and instruction registries to the expected function in untrusted memory. The return of an outbound call is managed through an EENTER instruction. If more SSA structures are available, the enclave may perform nested calls via EENTER targeting an OENTRY inside the outbound call.

A state transition diagram representing the lifecycle of an enclave is depicted in Figure 3.6.

Asynchronous Transition. In the event of an interrupt, system software with direct access to memory mapped device interfaces must be able to service that interrupt. For enclaves, this requires any logical core affected by the interrupt to exit enclave mode first. The processor triggers an AEX which first restores the execution state prior to enclave entry, and then prepares stack and instruction registries pointing to an asynchronous exit handler in process memory. The location of this exit handler is specified by the EENTER instruction when entering the enclave.

The enclave saves the execution state into a free SSA entry and subsequently scrubs all registries by setting them to predetermined values, before exiting. After exit, the operating system's interrupt handler assumes control and traps execution to a dedicated handler in privileged mode (ring 0). In the special event of a page-fault, the uppermost bits of the CR2 registry is left unchanged to ensure the page-level address is readable by the kernel handler.

Once the interrupt is serviced, control returns to the asynchronous exit handler in process memory. The handler invokes the ERESUME instruction which accepts a TCS virtual address to the faulting thread as input. ERESUME uses the last occupied SSA to restore execution context to the faulting address in enclave-memory.

In the event of a page-fault for a given enclave, system software is expected to issue an IPI triggering an AEX for all logical processors. A TLB shoot-down

then invalidates the affected memory translations, to ensure that stale address translations for EPC memory pages are not served by the MMU. The ETRACK instruction ensures that pages are *blocked* from being targeted by other SGX-instructions while waiting for a TLB flush by system software.

3.1.7 Side-Channel Attacks and Mitigation

Side-channels are a class of vulnerabilities where information systems causes unwanted side-effects that may be monitored by an attacker to illicit information. Examples relevant to SGX include cache, page-fault, timing and power analysis. By using analysis, an attacker may learn a statistical correlation between the information emitted by an operation, and particular programmatic behaviour. An attack which targets an encryption scheme may reduce the cryptographic key space into a problem which may be solved *brute force* within a reasonable amount of time[193].

Kocher et al. [104] demonstrate practical cache-based side-channel attacks for modern intel x86 based CPUs. The only mitigation of which is to disable branch prediction logic in the CPU. Additionally, hyper-threading (SMT) has also been explained as a source of side-channel. However, the consequence of disabling these is a significant reduction in performance and any system should carefully assess the tradeoff.

Controlled channel attacks are a class of side-channel attacks which target the shared caching infrastructure common in commodity operating systems. Xu et al. [192] detail a practical attack against the page-fault mechanism of shielded systems such as Overshadow [40] and Haven [22] despite only revealing page-granular addresses.

Brasser et al. [27] introduce the first practical shared cache side-channel attack against SGX. This work assumes a malicious privileged OS, which grants access to noiseless high resolution monitoring of cache evictions, reducing the sample requirements in a prime-and-probe attack. Lee et al. [114] introduce branch shadowing. This is a technique which exploits the branch prediction algorithm for modern processors to infer fine grained execution control of enclaves. SGX does not flush the branch history when transitioning out of enclave mode, which is exploited to predict the content of enclave memory.

Wang et al. [186] present a systematic exploration of the side-channel threats in the SGX virtual memory subsystem. The paper proposes circumvention techniques for reducing the side-channel size by decreasing the AEX count.

Van Bulck et al. [180] introduce a powerful practical side-channel attack enabling the extraction of keys from an SGX enclave. By exploiting Spectre [104], *Foreshadow* is able to extract launch keys from the pre-provisioned LE, allowing them to forge and launch unsolicited production enclaves.

Ahmad et al. [3] counter system call snooping, pagefault-based and cache-based timing attacks for SGX-enabled filesystems. By using path-ORAM [168],

confidential file systems used in SGX are shielded from such attacks. Zheng et al. [195] uses similar oblivious memory techniques to mask memory access patterns inside enclave memory. T-SGX [164] introduces a technique for eradicating caching side-channel attacks by using the Transactional Synchronization Extentions (TSX) for transactional memory. T-SGX disallows prime and probe cache attacks by using transactional memory in which cache operations by an attacker on a region partaking in a transaction, will trigger an abort on the operation. This abort interrupt is relayed to the enclave application, leaving the operating system oblivious to the operation. T-SGX claims it able to isolate the effect of cache snooping, and expunge this distinct attack vector completely. Oleksenko et al. [133] improve upon this work for concurrently accessed pages and caches. Additional research by Orenbach et al. [136], proposes a set of modifications to the ISA which hide page level access from the host and give the enclave full control over its own page-faults.

Schwarz et al. [158] introduce SGX-capable malware, which uses prime and probe cache side-channel attacks to infer information about cohosted mutually distrusting enclaves. Additionally, this work was the first to describe a technique which uses SGX to conceal malicious code from anti-malware software.

3.2 ARM TrustZone

Similarly to Intel, ARM processors implement modular security extensions enabling secure containers which may execute on an untrusted software stack. ARM Trustzone [13] specifies a set of intellectual property modules, which when combined, shield application containers from an untrusted system. A licensed manufacturer may selectively implement the modules required by the target threat-model.

Multiple System-on-a-Chips (SOCs) specifications, ranging from microcontrollers (Coretex-M) to mobile device chips (Coretex-A) implement TrustZone components. Manufactured SOCs with TrustZone capabilities include Qualcomms QSEE, Huawei's TrustedCore, Kinbi from Trustonic and AMDs Advanced Processing Units (APU)-coprocessors.

Fundamentally, TrustZone separates all hardware and software into two isolated worlds; a *secure world* and a *normal world*. Each physical core is virtualized into a secure and non-secure core. Hardware then multiplexes between these two modes for concurrent execution.

Main memory is partitioned into ranges with dedicated cache lines for the secure and non-secure world respectively. They are separated by a bit set on all Advanced eXtnesible Interface (AXI)-bus requests to protect against unauthorized access from non-secure hardware. This does, however, require TrustZone-capable hardware modules to enforce the separation. TrustZone-aware caches enforce the memory policy of the secure world by appropriately

setting the bit for memory requests. Each capable core implements separate address translation units for secure and nonsecure world, simplifying virtual memory management. Dedicated cache-lines and separate MMUs reduce the potential for side-channels, contrary to the shared architecture of SGX.

The TrustZone specifications describe support for secure peripherals, exclusively mapped to the secure world across the AXI-bus. Moreover, in order to provide secure memory able to withstand physical attacks, a TrustZone-aware memory controller, encryption unit, and a DMA-controller are necessary. However, since TrustZone is a modular design these are not mandatory.

In the boot-up procedure, all TrustZone capable cores are initially placed in secure mode, which configures the TrustZone platform by executing firmware stored in an on-chip ROM.

Context Switches between these worlds happen by invoking a Secure Monitor Call (SMC); an instruction which allows pre-specified messaging between the worlds. The invocation causes execution to trap into a dedicated *monitor mode* which forwards control to a predefined handler specified in the Monitor Vector Base Address Register (MVBAR) only accessible from the secure world. This monitor must also handle hardware exceptions and correctly route them to the target world.

The secure world has unrestricted access to non-secure memory, and system software may share information between the worlds using shared memory. Some ARMv8 designs do not implement the SMC and rather handle transitions in shared memory.

Contrary to SGX, TrustZone only hosts a single secure world per SOC and if one application misbehaves, the integrity of all other cohosted applications are compromised.

TrustZone implements few restrictions on the capabilities of software hosted in a secure world. Secure exceptions and explicit interrupt handling enable extensive control for system software, contrary to SGX. However, to support rich secure APIs, such as program attestation, complex system software is required. OP-TEE [134] implements support for the *Global Platform* consortium's TEE client API-specifications [172] for hosting *trusted applications* in an ARM-based TEE. OP-TEE incurs a significantly larger TCB than that of a SGX-based enclave. This comparison, however, excludes the insufficiently described microcode architecture of the SGX platform.

Since Advanced RISC Machine (ARM) only provide reference designs for modules (*TrustZone Blocks*), reasoning about the security properties of combinations of modules is challenging. Moreover, hardening TrustZone has proved a daunting task [141], as a wide array of vulnerabilities have surfaced in different implementations thereof.

3.3 Additional Trusted Hardware Systems

Unlike TEEs, which provide integrated selective shielding of application modules, *secure co-processors* implement hardware modules external to the main processing unit. Secure co-processors vary in complexity, but may include a processing unit, I/O controller, main memory and firmware/operating system hosted in a physically separated and *protected* environment. Mostly implemented as special purpose processors for storing secrets and authentication-assistance, we list some of the most commonly deployed here, and point out core aspects of the security model for each.

The Trusted Platform Modules (TPM) [91] is an international standard for implementing a secure cryptographic processor. The co-processor implements software attestation by measuring the initial state of the entire software stack on the physical host, leading to weaker security properties compared to SGX. This measurement includes the OS and device drivers, and it follows that combinations of composite system software makes it impossible to distinguish a correct software stack using the measurement value. The TPM may additionally be used to store cryptographic keys, an example of which is the BitLocker disk encryption scheme[1].

Integrated Circuit Card (IIC) [92] or smart cards embed an integrated circuit chip onto a "card". Early incarnations of smart cards provide tamperproof and immutable storage on chip, for purposes such as handling account balance, and preventing double spending situations. Modern incarnations contain a secure microcontroller with embedded memory, implementing a RISC processor, often based on the ARM architecture. These store the internal operating system in Read-Only Memory (ROM), with additional non-volatile memory on-chip for storing applications and data. More complex IICs may implement MMUs which provide isolation mechanisms for applications executing on chip. Additionally, some offer Java Virtual Machine (JVM) support [41] for developing Java applications, which may be deployed to multiple different architectures.

IIC is implemented in a variety of technologies including GSM Subscriber Identity Module (SIM) for mobile telephony authentication, 2-factor authentication schemes, and Automatic Teller Machine (ATM) networks. The smart card may interface with external components through either physical contact, or close proximity radio frequency. This interface also provides electrical power to the chip, through physical connectors or electromagnetic induction. The interface exposed by the smart card is not inter-positioned as a mandatory operation in software executing on a general purpose CPU, and therefore, may not protect against a malicious software stack.

Modern Apple mobile devices have a dedicated security co-processor for storing sensitive information, called the Secure Enclave Processor (SEP)[77]. The SEP implements biometric access control (Touch ID) to avoid key management at the server-side, while providing authenticated offline access. The SEP chip implements hardware-based random number generation and encrypted

memory, protecting stored data in the event of a compromised kernel. Communication between the traditional kernel, and the coprocessor occurs on an isolated interrupt-driven shared memory Inter-Process Communication (IPC) channel. Pages allocated to the secure enclave are guarded by a special bit mask, and we speculate that the SEP shares a MMU with the common cores, enforcing access to shared memory accordingly. This is, however, unconfirmed by public documentation. The secure enclave has a dedicated ROM storing secure-boot firmware, which establishes trust before booting a dedicated operating system. The operating system, SEPOS, is based on the L4 microkernel architecture [55], and signed by Apple. The secure-boot ROM creates an ephemeral encryption key used to encrypt the device memory used by the secure coprocessor. Following, the OS kernel signature is verified by the boot procedure before control is passed to the kernel initialization routine.

Depending on chip-model, integrity is guarded by either a memory protection key (> A7) or an integrity-tree in later chips (A11). The integrity-tree is authenticated by the memory protection key and itself guarded by nonces stored on on-chip SRAM. According to reverse engineering efforts [119], SEPOS does not implement stack hardening with cookies or Address-Space Layout Randomization (ASLR), reducing the security of the SEP. The secure enclave considers the Application Processor (AP) untrusted, however, the interface exposed through shared memory IPC, and the complexity of applications running on the SEP poses a significant attack surface. Moreover, implementing an entire secure microkernel increases the TCB significantly. If protected memory share MMU or caches with the regular APs, a side-channel attack may leak information to the untrusted system.

3.4 Summary

This chapter has discussed the architectural background for serverless computing, detailing several deployed cloud services available for use, demonstrating the advantages of a simplified cloud computing paradigm.

Despite the benefits, all contemporary cloud services require that applications trust the underlying infrastructure. Several widely available commodity hardware solutions for establishing trust exist. However, most are either severely limited in the security guarantees provided for software, have severe performance limitations or include large parts of the software stack as part of the TCB. Intel SGX provides the desirable trait of having a relative small attestable software stack with sufficient support for hosting software systems at near-native performance. Moreover, SGX is available in most Intel produced CPUs after 2016 (Skylake), increasing the potential for widespread adoption of a trusted serverless system.

4

Design

Based on our preliminary study on serverless computing (Chapter 2) and TEEs (Chapter 3), we present the following observations:

- Fine grained units of computation allow increased utilization of server hardware via smart scheduling, however, resource sharing of physical infrastructure requires secure isolation between tenants.
- Current runtimes isolate cohosted functions in separate VMs and/or containers, detrimental to performance [123].
- Cloud infrastructure is implemented through complex systems software. Exploits, misconfigurations, bugs or unfaithful operators may compromise hosted applications.
- Privacy-sensitive data may implement use-based policies, and accountability mechanisms are necessary to establish policy compliance [25] [64] [97].
- Stateless serverless abstractions simplify scalability, however, reduce adoption for applications requiring persistence of session or state.

We conjecture that existing serverless runtimes are unable to sufficiently shield applications from an untrusted public cloud. A conjecture shared by similar efforts, surveying the potential and challenges of serverless systems[99][81].

We determine that Intel SGX is the most suited TEE for designing an efficient and trusted distributed system. However, SGX is a proprietary technology, and previous research lack a comprehensive performance analysis.

This chapter is structured as follows: First we present a precursory performance analysis of SGX, followed by a set of principles and requirements for designing a trusted serverless system. Lastly, we present the design of Diggi;

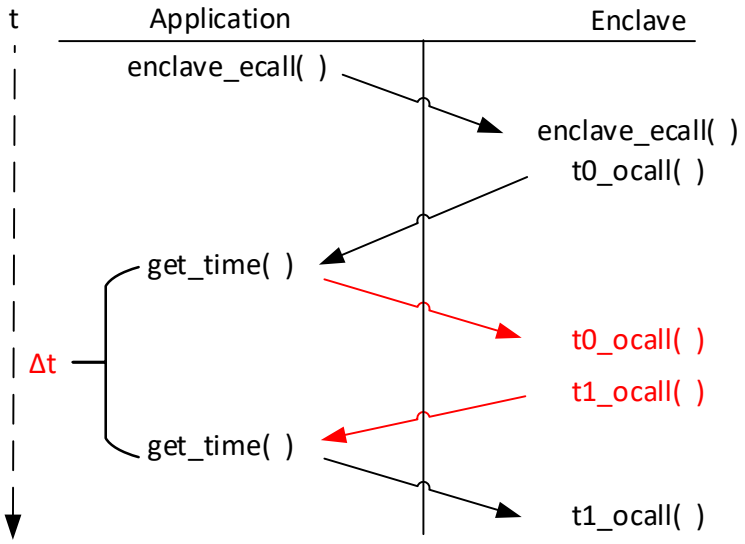


Figure 4.1: Sequence of events involved in measuring time spent inside enclaves [65]. To obtain the measurement between $t0$ and $t1$, each point must exit the enclave to reach the timing facility (get_time). The timing delta captures the entry and exit labeled in red [65].

an efficient and accountable trusted serverless runtime.

4.1 SGX Benchmark

To capture the intrinsic behaviour of SGX, we derive a set of experiments tailored to isolate the enclave runtime cost of memory consumption, thread management, context switching, and provisioning. Our experimental setup uses an Intel Core i5-6500 CPU @ 3.20 GHz with 4 logical cores and 2×8 GB of DDR3 DIMM DRAM. Dynamic frequency scaling and low energy hibernation are disabled throughout our experiments to avoid interference. We set the PRM size to its maximum alotted 128 MB. The experiments are hosted on the Ubuntu 14.04 Linux distribution, loaded with Intel's open source kernel SGX driver.[1]

The current setup of SGX hardware does not support RDTSC or any other native timing facilities in enclave-mode. Measurements must exit enclave-mode for each point in time and consequently, all intervals therefore include the time taken to enter and exit the enclave, described as the sequence of events detailed

1. https://github.com/01org/linux-sgx-driver

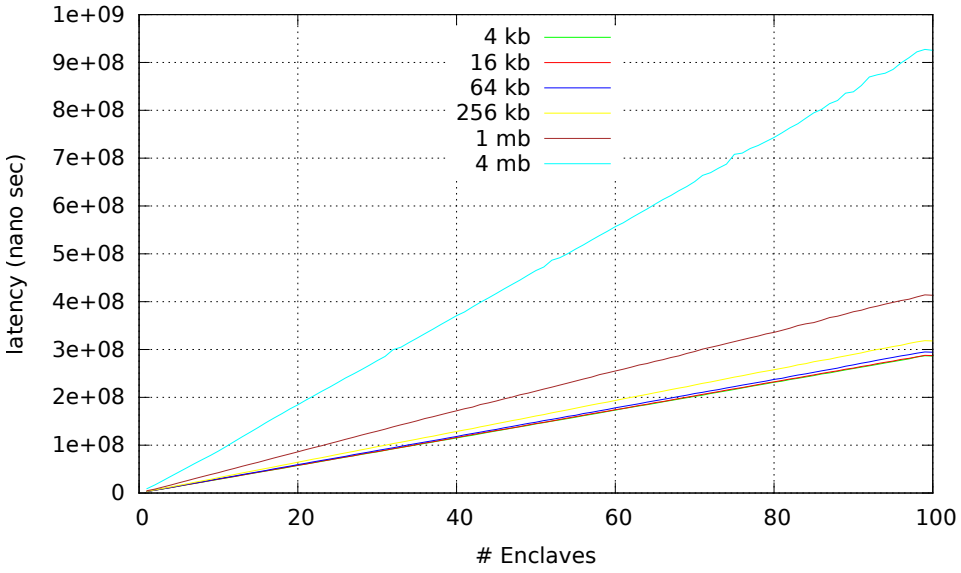


Figure 4.2: Latency as a function of number of enclaves created simultaneously, for differing sizes of enclaves [65].

in Figure 4.1.

4.1.1 Enclave Creation

Each SGX-capable CPU may host multiple enclaves, and tenants in a public cloud may share physical resources while remaining isolated from each-other and the underlying system.

The cost of enclave deployment directly impacts responsiveness of hosted cloud services. Low-latency enclave creation is therefore a primary concern for mission critical systems. To examine this cost, we measure the latency of concurrently deployed enclaves per host as a function of enclave size, illustrated in Figure 4.2. We observe that creation time increases linearly proportional to the size of the enclave. This linearity conforms with our expectations, as each page of a deployed enclave is loaded from binary and measured sequentially, described in Section 3.1.3; once the enclave grows in size, so too does pages measured. Additionally, we observe that the experiment produces a surge of page-fault events once the total memory consumption increases beyond the physically available PRM.

We observe that creation costs are similar for enclaves less than or equal to 64KB in size. For small enclaves, we conjecture that the dominant factor is metadata initialization; the SECS, TCS and SSA.

It is reasonable to assume that capable software consumes more than 4MB of memory, and we may therefore conclude that the cost of enclave creation

is significant. For systems requiring real-time responsiveness, pre-provisioning may prove advantageous, at the cost of increasing PRM consumption.

4.1.2 Memory Management

Hosting complex and demanding software in enclave-mode increases pressure on memory management, as detailed in Section 3.1.4. Depleting the PRM will cause enclave pages to be evicted with increasing intensity proportional to the oversubscription. To precisely quantify this overhead, we construct an experiment which forces oversubscription, and measure the overhead at both kernel and user-level.

The subject enclave is configured to consume twice the amount of physically available PRM, and the experiment progresses by writing bytes selectively to each 4KB page in a contiguous region of enclave memory. The enclave will not physically fit all pages in PRM, triggering evictions to regular DRAM.

Figure 4.3 illustrates this overhead as observed by both the kernel page-fault handler and the enclave. The y-axis denotes the cost and the x-axis the elapsed time, in nanoseconds. The red dots mark page eviction events and latency as observed by the kernel, and the black solid line represents total time spent in the kernel page-fault handler. The green line visualizes time spent writing to a particular region of memory as observed by the enclave. *Enclave-mode* measurements additionally include the cost of entry/exit, as illustrated in Figure 4.1.

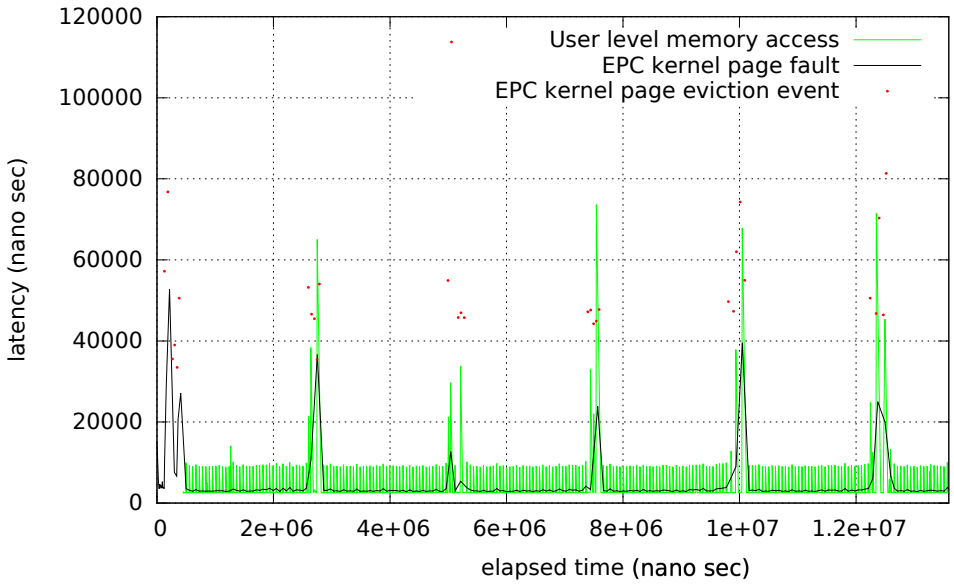


Figure 4.3: Paging overhead in nanoseconds as a function of time elapsed while writing sequentially to enclave memory [65].

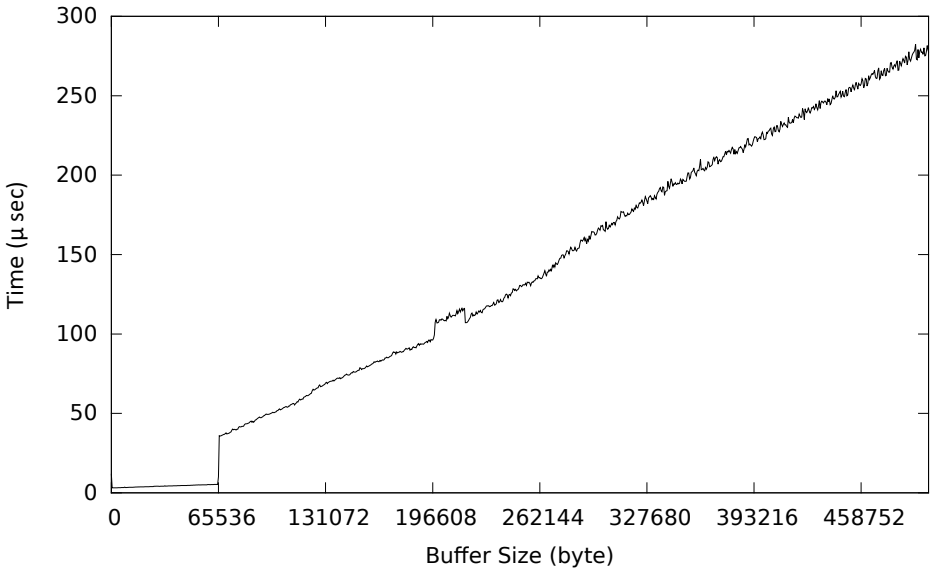


Figure 4.4: Enclave transition cost as a function of buffer size [65].

The experiment demonstrates the impact of page-fault operations for large enclaves, and additionally the initial provisioning of the enclave. Each cluster of evictions made by the kernel is correlated with increased memory write latency as observed by the enclave. We further observe that the kernel assumes a sequential access pattern, prefetching contiguous pages when evicting cluster of enclave pages. We base this on the observation that following each page-fault, the enclave is repeatedly allowed to touch an identical number of pages without interruption. Additionally, intra-page information is scrubbed from CR2 prior to enclave exit, and the kernel page-fault mechanism is therefore unable to infer access patterns internal to each page.

We observe that oversubscription of enclave memory is costly, and conclude that enclave memory should be used conservatively by applications. The experiment additionally demonstrates that PRM resources are not exhausted despite observable oversubscription. We conjecture the reason is a lack of predictable access-patterns observed by the kernel. For enclaves on a dedicated host, there is an opportunity for developers to optimize EPC usage based on predetermined access patterns.

Our experiments demonstrate that paging has a profound impact on performance, and a natural follow-up would be to measure the performance characteristics of dynamic paging support proposed in the SGX version 2 specifications.

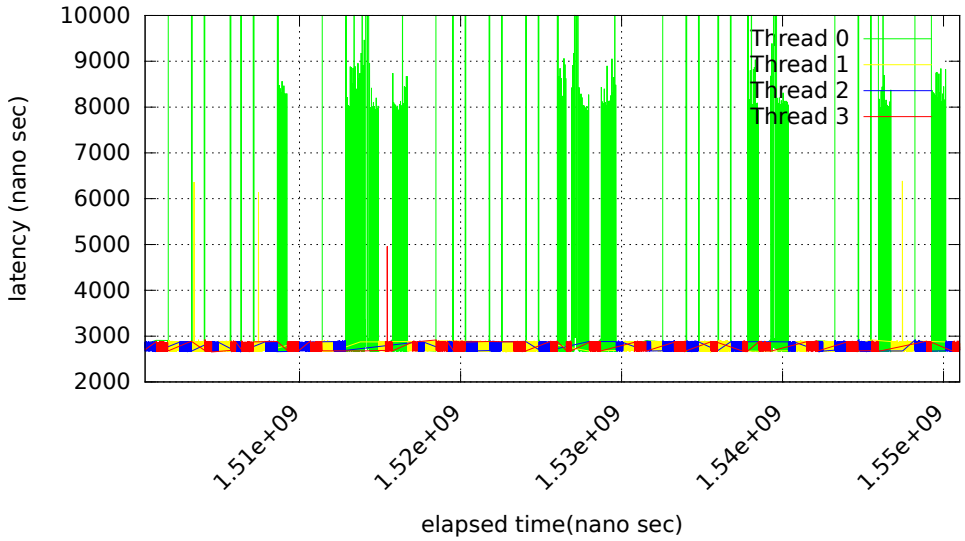


Figure 4.5: Execution overhead for multiple threads pinned to a single core, with page-fault events occurring [62].

4.1.3 Context Switches

Enclaves must be able to communicate with the external system in order to be practical. The rate at which information and control may be transferred into and out of enclave-mode is critical in determining the potential performance of shielded software.

To evaluate the predicted overhead, we measure the cost of switching between non-privileged and enclave-mode. Additionally, as part of the switch, we measure the cost of transferring information into enclave memory for processing.

Figure 4.4 illustrates this latency as a function of buffer size, simulating information transfer into the enclave. Latency increases linearly with transfer size, as the buffer grows beyond 64KB. As detailed in section 3.1.4, enclave-mode reads information directly from the host address space into enclave memory. The MEE then encrypts target cache-lines written back to PRM from the L3 on-core cache.

Based on these operations we conclude that enclave transitions (context switches) and data-transfers are costly if invoked frequently as part of the *hot-path* in application execution. Services employing SGX to shield select application logic should partition code to minimize data transfer between non-privileged and enclave-mode. We additionally observe that the cost may be compounded by page-fault events for large enclaves.

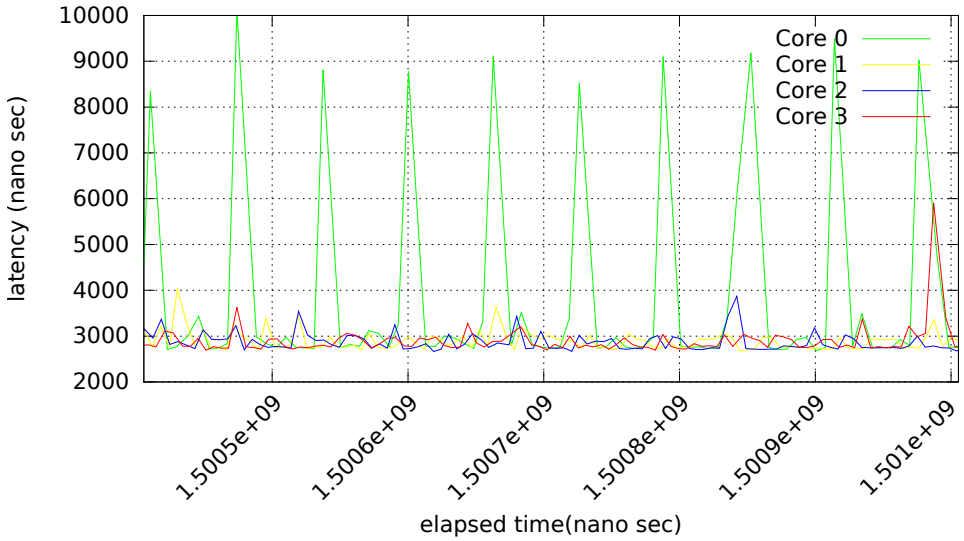


Figure 4.6: Execution overhead for multiple threads running on separate logical cores, with page-fault events occurring [62].

4.1.4 Multithreading

Web-scale data processing requires concurrent algorithms to operate efficiently on modern computers. SGX supports concurrent enclave execution, however, documentation suggests that interrupt processing may incur a significant overhead for multithreaded code. Before servicing an interrupt, all threads on the affected logical core must exit enclave-mode. Page-fault and timer interrupts are both examples which frequently invoke AEX.

We elicit this overhead by repeating the memory management experiment from Section 4.1.2, for multiple concurrent threads. We measure the access time of writing to contiguous regions of memory inside an enclave as a function of elapsed time. The experiment uses two different thread affinity techniques to demonstrate the impact on performance. We use the available OS-scheduler affinity capabilities to force threads onto distinct logical processor cores, referred to as *pinning*.

For each experiment, *thread* 0 (green) triggers page-fault events by sequentially writing to memory, while the others measure scheduler frequency. Scheduler frequency is measured as a timed interval with no operations in between, executed in a loop. Software threads then detect if the scheduler dequeues execution to multiplex across other logical cores.

The first experiment, illustrated in 4.5, pins multiple software threads onto the same logical core (hardware thread). We observe that in addition to page-faults, software threads pinned to the same core are subject to timer interrupts. Timer interrupts will cause affected threads to exit, in order to transfer control

between software threads, as described in 3.1.6.

Inversely, for the second configuration we observe that threads affinitized to separate logical cores, are not impacted by page-fault, as illustrated in 4.6. As expected, individual logical cores are independently interrupted without interference.

4.2 Performance principles

Founded in the experimental observations made in the previous section, we deduce a set of performance recommendations in form of guiding principles for designing efficient SGX-capable software.

In 4.1.3 we observe that buffer size is the dominant factor for enclave entry cost, particularly visible for buffers above 64 KB in size. We conjecture that for smaller buffers, the enclave memory write operations only hit the on-core cache, avoiding the cost incurred when encrypting last-level cache evictions to PRM. We suspect that increasing the buffer-sizes will cause more write-back operations to PRM. Our principle therefore states:

The Cohesion Principle. *Applications should partition its logical components such that data-copy across the enclave boundary is minimized.*

Following this, a possible design would be to place all application logic into a single enclave. Haven [22] is a prominent example of this approach, implementing a library OS inside enclave memory.

However, this contradicts the observations in 4.1.2, which demonstrate the significant cost incurred by excessive memory consumption. Although a limited resource, the available pool of EPC memory is not exhausted by the experiment, even in the presence of high memory contention. As detailed in 3.1.4, the intra-page faulting address is not revealed to the untrusted kernel memory manager, making it difficult to predict memory access patterns. We therefore state that:

The Access Pattern Principle. *Prior knowledge about an application's memory consumption and access pattern may be used to optimize kernel enclave memory management.*

Section 4.1.1 demonstrates the latency of enclave creation as a function of size. By pre-provisioning whenever service load patterns can be predicted, the application is able to hide some of this cost. However, once an enclave is used, it might be tainted with secret data. Recycling enclaves to a common pool can therefore potentially leak secrets from one domain to the next; invalidating the isolation guarantees. We therefore state that:

The Pre-provisioning Principle. *Application authors able to accurately predict enclave usage, should pre-provision enclaves in a disposable pool of resources that*

guarantees no reuse between isolation domains.

The enclave creation procedure includes provisioning a SECS, one TCS for each logical core executed inside an enclave, and at least one SSA for each thread. Costan and Devadas [46] suggest that to simplify the hardware implementation, some of these structures are allocated at the beginning of an EPC page dedicated to that instance. If we assume this is true for all data structures managed by SGX, enclaves executing on 4 logical cores may have at least 9 pages (34 KB) of metadata in total allocated to it, excluding code and data segments.

To offset the cost of having multiple enclaves, application authors should consider security separation as a continuous scale. Role-, or tenant-based isolation might be sufficient for some services, rather than individual isolation of all users. Application authors should precisely determine the required granularity of isolation, as a finer granularity includes the added cost of enclave creation.

Executing multiple software threads from the same core inside a single enclave degrades concurrent performance. Although non-enclave execution behaves similarly, the overhead associated with enclave page-faults becomes significant when memory footprint increases. In a multi-enclave system, assigning additional software thread to the same logical core will cause them to be multiplexed between enclaves and the host system. Each software thread must exit the enclave when switching context, incurring an additional overhead. As a consequence, the number of software threads in enclave-mode should ideally not exceed the logical core count for a given system. We therefore establish the following principle:

The Affinity Principle. *Enclave applications should not affinitize multiple software threads to the same logical core.*

Section 4.1.3 demonstrates the baseline cost of enclave entry/exit operations. It follows that to reduce the overhead incurred by transitions, threads should remain in enclave-mode once provisioned. We therefore state:

The Pinning Principle. *Application authors should pin threads to enclaves to avoid costly transitions.*

Threads may alternatively transport data egress/ingress through efficient message queuing/polling. To handle both simultaneously, each operation must either execute on a dedicated thread, or implement non-blocking queuing/polling, serviceable from a single thread. The latter conserves logical core use, and is preferable given the Affinity Principle defined above. These considerations produce the following principle:

The Asynchrony Principle. *All execution inside enclaves should be asynchronous.*

Threads are pinned inside enclaves to amortize transition cost, however,

thread count should not exceed logical core count. Core logic executing in enclave-mode should remain responsive at all time, servicing both incoming requests and processing data. Rather than allocating multiple threads to the same enclave, execution should be fully asynchronous, increasing resource utilization and improving overall application performance.

4.3 Trusted Serverless Runtime

Serverless cloud infrastructure reduces the need for explicit management of resources, however, privacy-compliant services requires implicit trust in the underlying infrastructure. Trusted Execution Environments (TEEs) and more specifically, Intel SGX, implements the necessary security model to decouple trust between a hosted cloud services and its underlying infrastructure.

Based on the challenges for serverless computing stated in Section 2.5, the security model presented in Section 3.1.1, and the performance principles derived in the previous section, we introduce a set of functional and non-functional requirements for designing a *trusted serverless runtime* using Intel SGX:

Functional: Shielded Privacy-compliant application services require a runtime capable of maintaining confidentiality and integrity of execution, its memory, persisted state and communication. The untrusted system may attempt to read and modify the control-flow and state of an application. Moreover, persisted memory and communication may be subject to replay attacks where stale state is presented to the runtime by the untrusted system. Runtime services should, when possible, securely request services from the untrusted system and augment operations to safeguard the security of the runtime.

Functional: Authentication Application services executing on an untrusted platform must be able to remotely authenticate itself. Before any secrets are provisioned, clients must be able to know: *who* is executing the request, and *what* that function does. Without the ability to authenticate applications remotely, a malicious system may modify application behavior undetected by the client. Once the client provisions secret data, despite shielded execution, the modified application may be coerced into disclosing secrets to the untrusted system. Distributed applications should similarly be able to jointly authenticate individual components, establishing composite trust.

Functional: Revocation Secrets provisioned to cloud services should only be persisted while the service is active. Additionally, continuous consent by the end user is required to store personal identifiable data. GDPR specifies

strong rights for revocation on behalf of end-users through the right to be forgotten, and services must provably purge consumer state on request. A consumer should be able to *revoke* data from a cloud service, and securely purge remanence from memory or persisted media. Revocation should moreover be accountable and applications may, during an audit, submit evidence to prove the revocation.

Functional: Accountability Actions performed by trusted applications, including revocation should be verifiable. A untrusted platform may withhold infrastructure resources such that operations are delayed indefinitely. A secure application runtime should produce irrefutable evidence proving the authenticity of committed operations. For privacy-sensitive data subject to access control policies, verifiable evidence on how data is processed and stored, provides proof of compliance.

SGX measurements serve as proof of the initial state, however, long running applications alter internal state. Any operation which mutates program state should therefore be logged. Authenticating programs beyond the initial state becomes intractable as the state diverges. Applications may additionally implement non-deterministic behavior through temporal or random information, which complicate reproducibility. To ensure non-reputability, any side-effects on function state should be stored securely. In the event of an audit, deterministically reproducing interactions with the environment and observing the alterations to application state will uniquely define application execution.

Functional: Persistence Although existing serverless systems predominantly treat cloud functions as a stateless construct, we argue that the definition should not be restricted by a higher order feature. Systems which require strict serializability for persisted state should implement support on top of the serverless abstraction. Protocols such as Real-Time Streaming Protocol (RTSP) require identifiers to track concurrent sessions and synchronize delivery streams. Persistence include the ability for preserving connection mapping between serverless execution and consuming clients for multi-step requests. Such state is persisted, but not permanently stored, the loss of which does not have fatal consequences for the service. We define such persistent state as *ephemeral*.

Non-Functional: Efficiency Applications hosted in rented cloud infrastructure are billed as time-per-rented-unit. For organizations, lower infrastructure costs imply more capacity for growth, through scale or investment. User-engagement will have a measurable cost to infrastructure spendings, and minimizing this equation while providing the same service is an important goal. Additionally, increasing the cohosting potential drives the cost down for consumers. A contemporary cloud application runtime must therefore be *efficient*.

This requirement may have multiple improvement dimensions, including memory footprint, compute, storage, provisioning time, scalability, throughput and latency.

Non-Functional: Reducing Trust Bugs, misconfigurations and exploits may violate privacy-compliance and leak personal data to an untrusted party. It follows that reducing the probabilistic vulnerability of a trusted system implies reducing the TCB. The TCB of an application is defined through the complexity of surrounding software, implicitly defined as trustworthy. We measure this empirically through Source Lines Of Code (SLOC) [5]. Correctly partitioning runtime components into what must be trusted and what may be left untrusted requires careful systems engineering. The bisection should be such that security is preserved while reducing the assumptions on trust in the system, and with minimal impact on application performance.

Non-Functional: Practicality For a technology to garner widespread adoption, it must be simple to use. The economy of scale states that an increase in production will trigger a proportional saving per unit sold. For cloud, this implies that an increase in adoption will trigger synergy by consolidation and homogenization of infrastructure investments. Trends in cloud computing skew towards simplicity, rather than breath of choice. A contemporary cloud application platform should provide a simple and familiar programming interface enabling development of powerful cloud services. This computing model should abstract away security concerns and provide support for integrating legacy software. Serverless computing [99] is one such abstraction; complex and rich application services may be implemented through *cloud functions*, with minimal infrastructure management.

Non-Functional: Granularity Aside from the underlying system, individual cloud functions should be able to specify colocation trust. Multiple cloud functions executed in the context of a single consumer, may share security context. Colocated functions not trusting one another should execute in separate security contexts. Separate functions hosted in the same context may trust one another if jointly authenticated.

4.4 Design

The following section introduces the design of *Diggi*; an efficient and secure serverless runtime. Diggi is designed from the ground up as an asynchronous runtime exclusively based on message-passing, conforming to the principles outlined in Section 4.2. Diggi cloud functions are shielded from the underly-

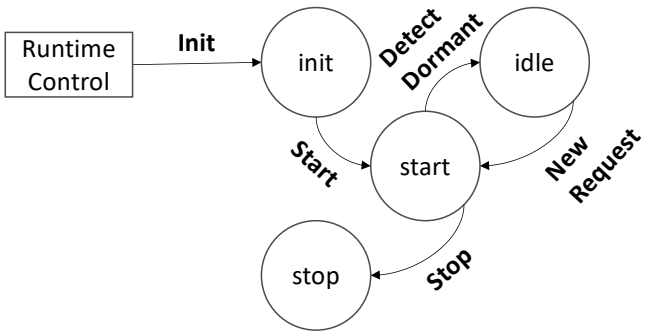


Figure 4.7: State diagram representing the lifecycle and transitional events of a Diggi cloud function. Idle is an internal state, hidden from the cloud function developer.

ing system, and additionally, verifiable via message record-and-replay. Diggi supports deep revocation of data via ephemeral encryption of session state. Collections of cloud functions may be deployed and jointly authenticated to create complex trusted serverless applications.

4.4.1 Diggi Persistent and Accountable Cloud Functions

Diggi implements persistent cloud functions which allow continuous sessions which may last for the longevity of the host, or until a cloud function explicitly shuts down. Invokers may additionally create multiple sessions per cloud function instance, and each cloud function may host multiple session *types* simultaneously. Figure 4.7 illustrates the lifecycle states and transition events which each cloud function must implement.

Idle cloud function instances may temporarily enter a dormant state, relinquishing resources to other cloud functions. Functions are subsequently revived by the trusted runtime once incoming requests are detected. This state transition does not require intervention from the function programmer. This is similar to conventional FaaS systems implementing a pre-provisioned (*hot-state*) function. Following the Pre-Provisioning Principle, however, separately isolated persistent cloud functions cannot be reused or repurposed by another security domain, as it is tainted by information belonging to the former.

Diggi does not concern discoverability, as it is outside of the scope of this thesis, but expects a repository to store identifiable running functions. Managing cloud functions as persistent entities requires us to relax idempotency. Functions are long lived, implying that all invocations will have side-effects that alter subsequent requests in a session. Additionally, if we assume a faulting function may prove crash-stop[32] fault tolerance to an invoker via remote software attestation, Byzantine faults become crash-faults.

To maximize the potential of cohosted functions per physical host, the runtime performance is important and functions in Diggi are developed as native C/C++ components. Just-In-Time (JIT) compiled language runtimes such as the Common Language Runtime (CLR) implement highly efficient programming languages, and previous work have demonstrated the ability to host non-native languages in SGX [30]. However, native compiled code continues to be the most performant, and arguably consume less memory for loaded executables and library extensions. Compiled languages additionally reduce the TCB, as they do not require a dedciated virtual machine or interpreter runtime. Such runtimes may comprise millions of lines of code, not including the standard libraries and external dependancies.

Diggi permits applications to invoke legacy code with minimal changes by exposing selected C/C++ standard library and POSIX-like system services to cloud functions.

A deployed enclave is identified through the author certificate, attestation key, and measurement. In Diggi, the measurement represents the initial state of a cloud function hosted in an enclave. Long-running persistent invocations will diverge significantly from this initial state, reducing congruity with its initial identity. Under the security model for SGX, enclaves do not protect against denial-of-service attacks, preventing function invocation. A claim presented to an invoker regarding the correct execution of a function should be tested.

Accountability is achieved through the capture and replay of state mutations for individual cloud functions. All interactions are realized through message-passing, and we represent these state mutations as the collection of messages sent and received. Although a cloud function may read state directly from the untrusted system, we expect an initially identifiable cloud function and runtime to only use a monitored media for communication. Given a single cloud function $f(msg_i)$ where msg is the input message to the cloud function. The initial state of the function is $state_{init}$ and for each message msg_i processed by the function, the next state becomes:

$$state_{i+1} = msg_i + state_i$$

Mutated state for message i uniquely identifies a cloud function beyond the initial measurement performed during enclave initialization, as described in section 3.1.3. State transitions for a given function is stored in an encrypted, non-reputable message-log for verification [8]. By storing messages, Diggi supports individual replay of cloud functions in the event of an audit. Cloud functions also consume randomness and temporal services through messages, and mutated state may be identified despite non-determinism by logging these parameters.

Cloud functions may implement stateful applications through *ephemeral state*. State is only local to individual functions, and its lifetime is limited to the host systems lifetime. Since regular memory is abundant and cheap, state is persisted in DRAM, however, through the same security model as provided

by SGX. Preservation is best effort, and should the host become unavailable, all state is purged. Applications which require durability and fault tolerance must implement it explicitly through state replication, or alternatively, log replay.

In-memory state may in theory be stored in enclave memory, however, contrary to the recommendations detailed in the Size- and Pre-Provisioning Principles (4.2). Should EPC be multiplexed, cloud functions may interfere with other colocated functions, increasing the variance in request latency. Explicit state management further allow dormant functions to remain deployed without consuming EPC memory.

State allocated to a given cloud function may only be accessed by that unique cloud function, or a derived version of that function. Despite storing state in the untrusted system, Diggi ensures that revocation of state is supported by ephemeral encryption keys. State encryption is uniquely tied to the cloud function identity, rendering encrypted data useless without an authentic cloud function.

4.4.2 An Asynchronous Trusted Runtime

The Affinity, Pinning and Asynchrony Principles stated in 4.2, leads to an execution model where application threads are permanently pinned in enclave-mode. Over-provisioning of software threads onto hardware threads should be avoided, and ideally, dedicated to separate cloud function runtimes. To maximize the utilization of each, the trusted enclave runtime is implemented as a non-blocking system.

Communication occur through asynchronous message-passing, where all internal APIs, including storage, networking and threading, are serviced through messages.

Multi-session communication with clients or other cloud functions is implemented by an abstraction allowing callbacks (*tasks*) to be registered as continuations of invoked operations. Application logic is implemented as chains of such task invocations, or *flows*, as illustrated in 4.8. Flows are stored in a task list, which saves intermediate state, forwarded to the next invoker.

Cloud functions are reactive and sessions are triggered on the receipt of a message sent by an invoker. A persistent cloud function is able to host multiple sessions simultaneously where communication processing and business logic are interleaved onto the same physical thread. Figure 4.9 illustrates multiple concurrent flows, interleaved onto the same physical thread. Blocking operations, interacting with slower APIs such as I/O and networking, are wait-able through runtime hooks, allowing separate flows to execute in the interim.

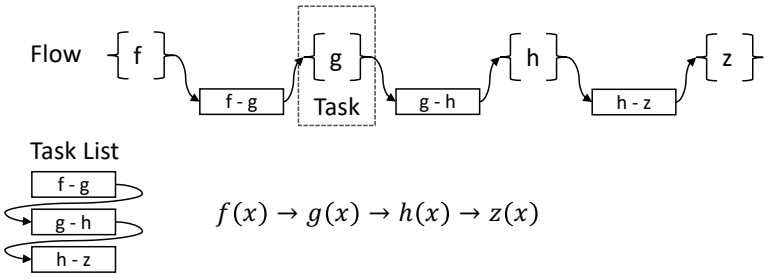


Figure 4.8: A chain of callbacks (tasks) implementing a flow. Each task executes independently, however serialized. The flow progresses by invoking the next following the completion of a precursor task.

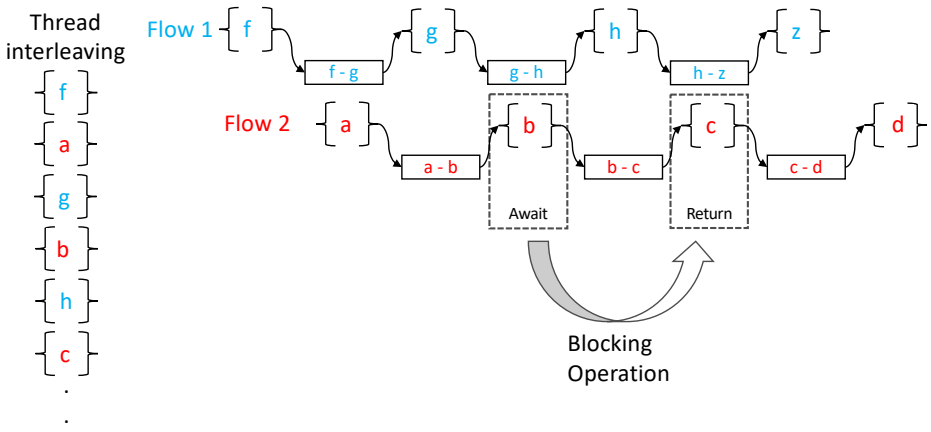


Figure 4.9: A cloud function interleaving multiple flows onto a single thread. Blocking operations are writable tasks, where the continuation is scheduled once the results are produced.

4.4.3 Deployment and authentication

Diggi cloud functions are composable, where functions may be chained together to form more complex functionality, creating secure and verifiable distributed applications. A manifest stores all configuration parameters and placement information for an application expressed as a collection of cloud functions. Diggi then uses this information to distribute and deploy cloud functions to physical hosts. The manifest contains the following information:

- Physical location; Which functions should be deployed to a given physical host.
- Communication permissions; Which functions are allowed to communicate, and which may allow external communication through a function proxy.
- Resource usage; The allocations of memory, compute, and storage, for a given cloud function.
- Static resources; Configuration parameters unique to a given cloud function.

During the process of loading a new application, the manifest is bundled together with function executables and shipped to target hosts, which deploy the necessary resources. Before deployment, the host ensures that the binary is compiled by a known authority and that the content is authentic through the enclave initialization process outlined in Section 3.1.3. The manifest is injected into the binary before measurement, ensuring it becomes part of the identity. The manifest indicates which functions have joint communication abilities and each must individually authenticate themselves, before establishing a joint confidential channel for communication.

We assume that an open architecture permits the attestation service and keys be controlled explicitly. Ideally, under the security model presented, a cloud function application should not have a single root of trust. This authentication scheme would require each cloud function to embed information which identify other correct cloud functions and attestation keys. However, embedding this information alters the measurement, invalidating others embedded measurement. Measurement information may alternatively be delivered through a signed measurement at runtime requiring a trusted principal to sign and deliver measurements. Additionally, this would permit stale measurements unless enclaves store an identifier, which again leads to the problem of circular joint measurement. Figure 4.10 illustrates the circular measurement problem.

It is arguably impossible to not prescribe trust external to the enclave, yet in its minimal form we can assume a single root of trust. Diggi implements a trusted principal, ensuring the authenticity of reported quotes, and on behalf of each, certifies the joint authenticity of the deployed collection of functions.

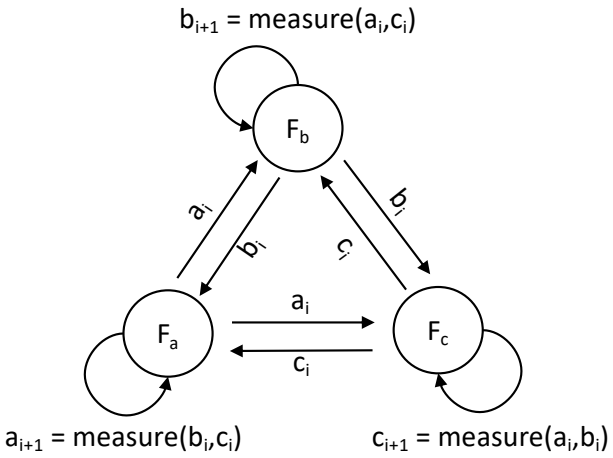


Figure 4.10: The circular measurement problem: Each cloud function, *F*, includes the two others in its own measurement, which alters the others measurements.

Additionally, as mentioned in Section 3.1.1, SGX assumes a trusted development environment for compiling, measuring and signing cloud functions prior to deployment. For simplicity we can assume these are the same principal.

4.5 Summary

To efficiently support shielded execution of application software in an untrusted cloud, we evaluate Intel Software Guard eXtensions (SGX). We demonstrate that although SGX is an attractive Trusted Execution Environment (TEE), there are intrinsic properties of the platform which require careful system design. Our analysis shows that thread management, call-gate transitions, memory management and application partitioning must be first-class concerns when designing an efficient application cloud runtime with SGX.

Based on this analysis and threat model for privacy compliant services, we state a set of functional and non-functional requirements for designing a trusted serverless runtime. These requirements lead to the design of Diggi; a persistent, accountable and shielded serverless runtime. Diggi is designed to support simple development of secure applications hosted in a public cloud. The next chapters will detail the implementation of a prototype trusted serverless runtime satisfying the Diggi design.

5

Cloud Function API

The Diggi serverless runtime is designed to shield applications from an untrusted public cloud. Diggi enables the development of complex distributed cloud services through the *persistent accountable cloud function* abstraction. The next two chapters will detail our prototype implementation of the Diggi system, introduced in the previous chapter, specifically conforming to the requirements stated in Section 4.3.

This chapter details the services endpoints available for developing secure, persistent and accountable cloud functions on top of the Diggi prototype runtime supporting Intel SGX.

```
void func_echo_cb(msg_async_response_t *resp)
{
    auto api = (DiggiAPI*)resp->context;
    auto mm = api->GetMessageManager();
    auto new_msg = mm->allocateMessage(resp->msg, resp->msg->size);
    new_msg->dest = resp->msg->src;
    new_msg->src = resp->msg->dest;
    memcpy(new_msg->data, resp->msg->data, resp->msg->size);
    mm->Send(new_msg);
}
void func_init(DiggiAPI *api)
{
    auto mm = api->GetMessageManager();
    mm->registerTypeCallback(function_echo_cb, ECHO_MESSAGE, api);
}
void func_start(DiggiAPI *api)
{
    //noop
}
void func_stop(DiggiAPI *api)
{
    auto log = api->GetLogObject();
    log->Log(LRELEASE, "Stopping the echo-function\n");
}
```

Listing 5.1: An example using the Diggi runtime API to implement an echo function. *func_init* registers a callback task for handling incoming messages of the type *ECHO_MESSAGE*. The callback receives a *msg_async_response_t*, encapsulating the inbound message and a discretionary context pointer.

The Diggi cloud function runtime exposes a set API-endpoints for developers of cloud functions, comprising functionality for concurrency, configuration, debugging, storage, and communication. All runtime operations and APIs are asynchronous (non-blocking) to maximize the potential utilization per thread. Functions are persistent, and sessions enable an invoker to request the same instance repeatedly, implemented as *flows*. These behave similarity to reactive programmable systems [18], where dynamic queries *observe* incoming data streams and apply processing rules on-demand. Reactive programming is beneficial for several application architectures including stream processing analytics, publish subscribe systems, and video on-demand services; all of which are difficult to implement in conventional stateless FaaS systems. An example function implementing an *echo server* which retransmits incoming data back to the sender, is shown in Listing 5.1.

```
class DiggiAPI : public IDiggiAPI {
        IThreadPool * tpool;
        IStorageManager * stomanager;
        IMessageManager * msgmanager;
        INetworkManager * netmanager;
    ISignalHandler * shandler;
        ILog * logr;
        aid_t aid;
    sgx_enclave_id_t enclaveid;
        void * dl_handle;
        json_node configuration;
public:
        DiggiAPI(
        IThreadPool * pool,
        IStorageManager * smngr,
        IMessageManager * mmngr,
        INetworkManager * nmngr,
        ISignalHandler * sighandler,
        ILog * log,
        aid_t id,
        void * handle);
        IThreadPool * GetThreadPool();
        IStorageManager * GetStorageManager();
        IMessageManager * GetMessageManager();
        INetworkManager * GetNetworkManager();
        sgx_enclave_id_t GetEnclaveId();
        ISignalHandler *  GetSignalHandler();
        void * GetDlHandle();
        ILog * GetLogObject();
        aid_t GetId();
        json_node & GetFuncConfig();
};
```

Listing 5.2: The Diggi cloud function API class definition which acts as the aggregate API, gathering a collection of interfaces comprising the joint functionality available to Diggi cloud functions. These include individual APIs for storage, networking, messaging, concurrency, signaling and logging.

5.1 Lifecycle management

Figure 4.7 illustrates the lifecycle of a Diggi cloud function. To handle lifecycle events, each cloud function must implement three event callback handlers: *init*, *start*, and *stop*, as demonstrated in Listing 5.1 These handlers are invoked by the Diggi runtime to notify the cloud function of significant events altering execution state.

- **Init** – Before application-defined task callbacks are allowed to execute, the runtime prepares communication primitives, stacks, storage interface and other internal data structures. The init event callback, defined by the cloud function, is ran once these initialization procedures are completed. This callback is executed serially by the runtime and may block to guarantee that all preconditions are met before the cloud function begins servicing incoming requests. The init function is a utility for developers which should prepare function-specific internal state, initialize libraries, etc. The callback receives the Diggi API object as input, exposing all runtime services available to the cloud function, detailed in Listing 5.2. Once the callback is completed, the Diggi runtime expects that the cloud function is ready for execution, and will begin forwarding inbound messages.
- **Start** – Once ready, the start event callback is triggered by the runtime to notify the function that it is ready to process requests. Both the init and start callback are triggered on the primary thread of the cloud function. However, unlike init this operation must be non-blocking. If the cloud function has multiple threads, the start event will be scheduled on the thread with the lowermost identity.
- **Stop** – Cloud functions persist for as long as the cloud function or runtime permits. When the execution of a cloud function ends, a stop event callback is triggered, either by the function itself or the runtime. Prior to invocation, the runtime attempts to relinquish all dedicated threading resources. Any nested task callback operations remaining in the cloud function are aborted, and the physical thread resources returned to the runtime. Cloud functions are permitted to gracefully terminate, reducing the potential for races in multithreaded cloud functions. The callback is invoked by the runtime on a separate runtime thread to ensure any blocking operations internal to the cloud function do not prevent termination. This is a mandatory operation, implying that the runtime expects the function to cooperate in the cleanup process. Diggi does not guarantee that all shutdown events will be preceded by a notification via this callback. In the event of a system crash, this will be ignored.
- **Idle** The system allows dormant functions which are not participating in active request processing to relinquish resources and enter an idle state. The runtime periodically polls for incoming requests and upon

the receipt, returns resources to the cloud function. The cloud function does not require idle-aware programming; the runtime transparently suspends and awakes cloud functions.

5.2 Asynchronous Programming

Asynchronous programming allows computational events to execute separately without a global synchronized clock. In an asynchronous system all possible ordering of computational events are valid and must be handled correctly. Cloud functions in Diggi are implemented as *continuation-style* asynchronous task callbacks, allowing efficient scheduling of runtime operations. These chains of operations (*tasks*) are in Diggi referred to as *flows*, which are the primary building block for application logic in persistent cloud function. This abstraction renders the following properties for asynchronous programming of cloud functions. The Diggi programming model is more restrictive than *pure* asynchrony, where any ordering, causal or otherwise is permitted. However, we present the Diggi API as asynchronous given the following assumptions. For a given thread:

- *Non-Blocking*: All causally dependent operations which await a response must asynchronously schedule that continuation to allow other operations to execute in the interim.
- *Starvation-free*: All operations which are compute-intensive must gracefully share resources by yielding control to other operations periodically.
- *Intra-flow order*: All operations internal to a particular asynchronous flow will maintain causal ordering when executed asynchronously.
- *Inter-flow interleaving*: Operations on separate flows may be interleaved and reordered.

Through flows, cloud functions are able to handle message delivery and business logic for a given function concurrently on a single thread without context-switches and enclave interrupts, which increases the co-hosting potential.

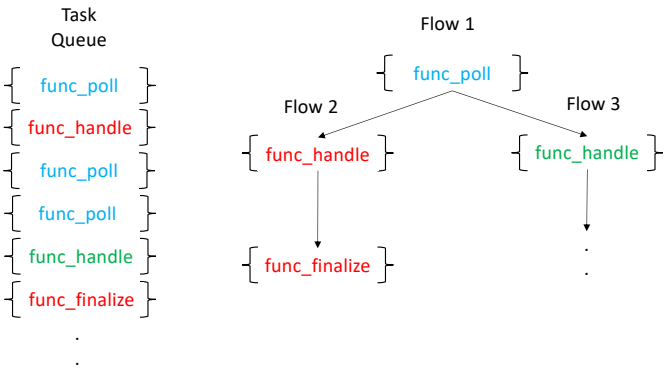


Figure 5.1: Interleaving of three flows on a queue of tasks; blue is the polling flow, while red and green are separate flows processing items retrieved. Cloud functions may interleave communication and processing on a single thread.

```
void func_poll(DiggiAPI * api)
{
    if(item_present(pollable))
    {
        api->GetThreadPool()->Schedule(func_handle, api);
    }
    //Reschedule to periodically poll resource
    api->GetThreadPool()->Schedule(func_poll, api);
}
void func_handle(DiggiAPI * api)
{
    auto item = get(pollable);
    api->GetThreadPool()->Schedule(func_finalize, api);

}
void func_finalize(DiggiAPI * api)
{
    free(item)
    .
    .
}
```

Listing 5.3: Example code illustrating the asynchronous polling pattern. These flows may alternatively be serialized into one flow by rescheduling *func_pool* in *func_finalize*. Both patterns permit interleaving separate operations.

Listing 5.3 presents this phenomenon for a single cloud function through a simplified representation of the continuation-style pattern. Figure 5.1 illustrates an example execution plan demonstrating one (of many) correct orderings of interleaved asynchronous flows onto a task queue consumed by a single thread. One implements a periodic item polling, while the other handles available items. In the illustration, two separate items become available, resulting in

three flows (red, blue, green).

The Diggi runtime is itself internally implemented through tasks and flows, and all Diggi APIs exposes a consistent pattern for passing state between tasks, illustrated in Listing 5.4.

```
task(arg)
{
   internal_state = input.operation();
   schedule(task_next, internal_state);
}
task_next(void *internal_state)
{
   internal_state.next_operation();
   ...
}
```

Listing 5.4: An example pseudocode pattern illustrating how Diggi preserves asynchronous state and delivers it to the next task, once ready.

This approach solves a common problem in asynchronous systems; preserving state between asynchronous operations adds complexity to the system. By allowing Diggi to manage state internally, we minimize the boilerplate code necessary for asynchronous communication, and additionally incur a potential reduction in TCB.

5.3 Programming Language

Cloud functions are commonly developed in high-level programming languages, such as JavaScript or Python. High-level languages are beneficial for rapid development, as most of the low-level details such as explicit memory management and library loading are handled automatically by the runtime. Because of this simplicity, high-level languages also have a bigger developer base than their low-level counterparts. High-level languages offer features such as automated memory management by garbage collection and byte code interpretation for portability across different hardware platforms. Some high-level languages provide JIT compilation which ship code in an intermediate language representation, which is then compiled on-demand to native platform dependent code. This ensures higher performance without sacrificing portability.

As a general rule, simplification is a tradeoff which inherently sacrifices performance. Code requiring precise performance tuning, such as cache alignment, memory management, lock-free concurrent programming, and instruction set extensions such as x86 AVX, SSE or TSX will benefit from explicit control over the underlying architecture.

Additionally, high-level languages contribute negatively to the TCB of a trusted enclave, requiring more runtime code and system libraries than its

compiled counterpart. JIT-ed languages require dynamic linking and memory mapping features not available in the first revision of SGX. Supporting interpreted languages in SGX is straightforward, however, loading generic virtual machines into an enclave violates the ability to perform program attestation. Arbitrary programs may execute inside the enclave, and careful engineering is required to ensure the identity of interpreted code is included in the attestation evidence. Arbitrary code execution additionally increases the attack-surface of the enclave. Runtime and library support for high-level languages incur a larger memory footprint, contrary to the Size Principle stated in Section 4.2.

Enclave-mode permits direct access to host process memory, similar to other TEEs. Interpreted languages may provide memory safety, reducing the potential for memory reference bugs which accidentally writes unshielded data to a non-trusted memory location.

Rust is a high-performant, lean, compiled language which mediates memory safety problems inherent in C/C++ development during compile-time, disallowing null pointer references and enforcing explicit pointer-object attribution and ownership [122]. Diggi currently supports cloud functions through C and C++, with on-going work to support the Rust programming language in the future.

The SGX SDK supports enclave development through C/C++ out of the box, including library support for most of Standard Template Library (STL) and gLibc. However, this excludes all functionality which require system services, such as network, filesystem, time, crypto and peripheral devices.

5.4 Legacy

Ideally, Diggi would be able to support a broad software library, enabling all applications to be developed as native asynchronous and flow-based cloud functions, reducing the need for external dependencies. This is, however, impractical due to the the increased TCB and memory consumption required [22]. Additionally, existing software libraries are often tied directly to the host system call api exposed by the OS. To support existing (legacy) software libraries in a shielded serverless systems, Diggi exposes runtime constructs to allow cloud functions to implement select storage, concurrency, and networking system calls; securely and asynchronously.

Traditional *blocking* system calls in procedural programming expect execution to transfer control to the system for processing halting execution of the caller. Diggi cloud functions are dependent on *wait-free* task execution to maximize thread utilization. Existing software depending on blocking procedures to synchronize application progress cannot easily be refactored. To avoid costly context-switches, operations must be fulfilled through exit-less asynchronous message-passing.

Diggi saves the execution context of calling code and allow a dedicated *translator* to prepare an asynchronous message representing the operation which is then delivered to the untrusted system. The translator must, depending on operation, protect the integrity and confidentiality of the message. The untrusted system then processes the message via a *server*, interacting directly with the host operating system. The response is then delivered back to the translator and the task scheduler then restores the execution context of calling code before returning to the caller with the translated results. During this procedure the caller yields control, permitting concurrent operations to be scheduled on the task queue, as depicted in Figure 4.8.

Each translated system call will increase the exposure for Iago-based attacks [38]. Some may be emulated in enclave-mode, avoiding the need for interacting with the untrusted system. Networking operations do not change the security model based on an untrusted system, as Wide Area Network (WAN) communication is expected to be protected by Transport Layer Security (TLS) or application layer specific measures such as Pretty Good Privacy (PGP) and Secure Hypertext Transfer Protocol (SHTTP). Other system calls such as *time* and *rand* are difficult to emulate in enclave-mode, and impossible to shield by translation. These may be fetched or pre-provisoned (cached) from a remote trusted resource. SGX supports trusted time and randomness through dedicated platform services (Intel Management Engine). However, these are implemented in firmware and only retrievable through pre-provisioned platform enclaves which incur a significant invocation overhead.

To support legacy software which require system interaction, a cloud function implements for each external service, a translators and server pair, as illustrated in Listing 5.5. The server implements the untrusted host system interaction, while the translator mediates system call access and protects the confidentiality and integrity of translated system call operation.

Applications must themselves ensure that the translator shields the system call arguments correctly, and validates the return parameters, to ensure no tampering. For data system calls, such as network and storage, this implies ensuring confidentiality and integrity protection. Translator-server pairs are satisfied as messages and may be distributed across multiple machines, transparent to the calling software. For example, secure timestamps may be pushed from a trusted machine external to the public cloud. Depending on the synchronization requirements of the calling software, timestamps may be cached if only monotonicity is required. Cloud function-specific translators allow each to tune the semantic behaviour and security of system calls individually. The caveat, however, is an increase in complexity for developing cloud functions, where each must implement support for bespoke system call operations. However, exposing a simplified translator-server API to cloud function developers reduces this complexity.

```
\\ Server
void open_server_task(void *msg, int status)
{
    auto ctx = (msg_async_response_t *)msg;
    auto ptr = ctx->msg->data;
    mode_t md = Pack::unpack<mode_t>(&ptr);
    int oflags = Pack::unpack<int>(&ptr);
    int encrypted = Pack::unpack<int>(&ptr);
    /* assumes null terminated character */
    const char *path = (const char *)ptr;

    /* Direct system call */
    int fd =  __real_open(path, oflags, md);

    auto msg_n = api->GetMessageManager()->allocateMessage(
                                                ctx->msg,
                                                sizeof(int));
    msg_n->src = ctx->msg->dest;
    msg_n->dest = ctx->msg->src;
    auto ptrt = msg_n->data;
    Pack::pack<int>(&ptrt, fd);
    api->GetMessageManager()->Send(msg_n, nullptr, nullptr);
}

\\ Translator
int open_translator_task(const char *path, int oflags, mode_t mode)
{
    char *path_n = normalizePath((char *)path);
    auto path_length = strlen(path_n);
    size_t request_size = sizeof(int)
                        + sizeof(mode_t)
                        + sizeof(int)
                        + path_length
                        + 1;
    auto msg = api->GetMessageManager()->allocateMessage(
                                            "Server-Destination",
                                            request_size,
                                            CALLBACK,
                                            CLEARTEXT);
                                            msg->type = FILEIO_OPEN;
    /* Marshall */
    auto ptr = msg->data;
    Pack::pack<mode_t>(&ptr, mode);
    Pack::pack<int>(&ptr, oflags);
    Pack::pack<int>(&ptr, (int)encrypted);
    memcpy(ptr, path_n, path_length + 1);
    mngr->Send(msg, set_response, ctx);
    msg_t *single_p_response;
    msg_t **double_p_response = &single_p_response;
    double_p_response = nullptr;
    /* Wait and yield to task queue */
    auto return_msg = wait_for_response(double_p_response);
    \*Unmarshal results*\
    int fd = Pack::unpack<int>(&return_msg->data);
        return fd;
}
```

Listing 5.5: A server/translator pair implementing the open system call. For brevity, this example does not include code for encryption or integrity protection.

```
"Diggi-node-1": {
 "network":"192.168.1.137:6000",
 "functions": {},
 "enclave": {
  "functions": {
   "echo-function@1": {
    "skip-attestation": "1",
    "threads": "1"}}}},
"Diggi-node-2": {
 "network": "192.168.1.138:6000",
 "functions": {
  "load-function@1": {
   "skip-attestation": "1",
   "master": "1",
   "package-size": "87040",
   "connected-to": "echo-function@1",
   "threads": "1",
   "duration": "10" }}}
```

Figure 5.2: An example Diggi application configuration, consisting of two functions; an echo-function and a load-function.

5.5 Deployment

Serverless applications may be composed of multiple persistent cloud functions, specified through an application configuration. The application configuration specifies co-hosting properties, placement and resource usage. Each cloud function has a sub-configuration for specifying custom input parameters. Rich configuration options may simplify development, however, generalize the cloud functions. General code will reduce the identity properties of enclave measurement. To avoid generalizability, the sub-configuration is itself compiled and linked into each function binary, becoming part of the enclave identity by measurement. An example application configuration is listed in 5.2, which implements a server (*echo-function*) based on Listing 5.1 interacting with a client (*load-function*).

During compilation, the configuration is used as a recipe for creating the units of deployable binaries, later distributed to the target hosts. The build environment must be hosted on a fully trusted software and hardware stack, and able to communicate securely with the trusted principal authority for program authentication.

For each individual cloud function, the build environment creates a deployment key pair, $\{P_d, S_d\}$. The binary B_i and the sub-function configuration C_i is measured and signed by the private key S_d, forming the SIGSTRUCT certificate:

$$Cert_i = Sign(S_{di}, Measurement_i(B_i, C_i))$$

Each F_i includes the binary B_i, per function deployment configuration C_i and

the certificate $Cert_i$:

$$F_i = \{Cert_i, \{B_i, C_i, \}\}$$

Before cloud functions are deployed to physical hosts, a daemon host process and untrusted runtime must be initialized. The Diggi trusted root distributes binaries according to the placement specifications in the application configuration. For the deployable set of functions D_N, the individual function bundles and the per-node config C_N are distributed to a given node N:

$$D_N = \{F_1, F_2, ...F_i, C_N\}$$

The daemon process deploys applications according to the configuration by creating an enclave for each cloud function binary as described in section 3.1.3. The LE compares the actual measurement of the cloud function binary against the expected measurement and signature:

$$\forall i, Verify(F_i, P_{di}) \wedge (M_{hw} = M_i)$$

Once completed, the cloud function may be certain that it is an authentic cloud function signed by the trusted root, identical to the binary submitted by the developer.

5.6 Summary

This chapter details the implementation of the Diggi API for developing secure serverless applications in an untrusted cloud, through a collection of persistent, shielded and accountable cloud functions. Diggi is implemented from the ground up to support a performant trusted runtime in SGX, through asynchronous programming. The Diggi API facilitates cloud function development through an asynchronous task and flow based abstraction built on top of efficient message-passing interfaces. Cloud functions may implement rich features through existing libraries by translating system dependancies into translator/server pairs, where procedural system call operations are translated into asynchronous messages. The next chapter will detail the Diggi daemon process, the trusted and untrusted asynchronous runtime, responsible for efficiently hosting collections of shielded cloud functions.

6

Runtime

A Diggi serverless application may be composed of multiple distributed cloud functions communicating to deliver a composite service. Diggi is designed to be hosted on physical hardware or alternatively, a virtual machine interacting with the SGX driver directly, capable of mapping enclave memory into a guest operating system process. This section details the implementation of the Diggi runtime prototype for shielded, persistent and accountable cloud function execution. Our prototype comprises some 31K lines of C/C++ code, 17K of which implements the trusted runtime.

For each host, Diggi is executed by a single daemon process, referred to as a *Diggi node*. Inside the process' address-space, cohosted functions share a single *untrusted runtime*, handling deployment and lifecycle management, configuration, messaging, networking, and filesystem interaction.

Each distinct cloud function implements a dedicated *trusted runtime* within their respective enclave's address-space, as illustrated in Figure 6.1. The trusted runtime is responsible for handling message delivery, state preservation, threading/concurrency, and attestation/key exchange.

To shield applications from an untrusted system, the trusted runtime must ensure the integrity, authenticity and confidentiality of cloud functions contained within an enclave. All communication with the untrusted runtime and other cloud functions must additionally be shielded. In the absence of a denial of service attack, we expect a benign untrusted runtime to correctly delegate resources and deliver messages to the trusted runtime. The trusted runtime is responsible for, under the performance principles outlined in section 4.2, managing these resources optimally.

A first-order concern for the implementation of a trusted runtime is reducing

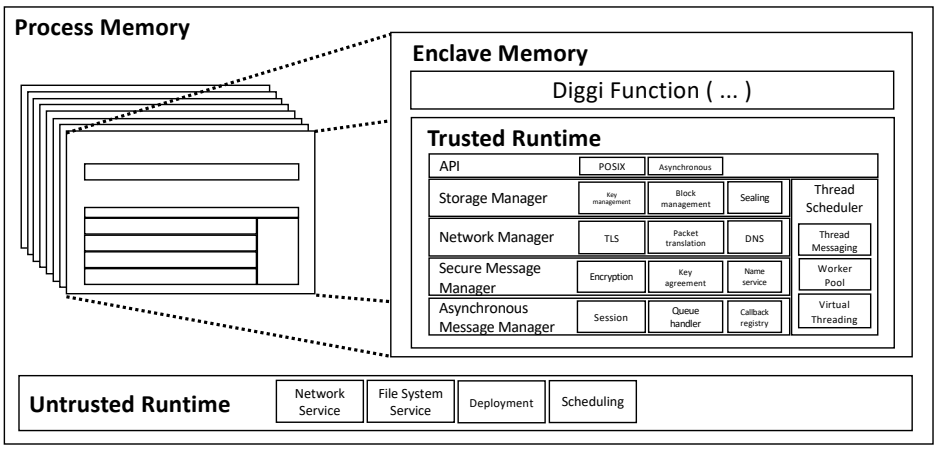


Figure 6.1: The Diggi daemon process layout. Each function receives a dedicated enclave and trusted runtime, but shares the untrusted runtime with all cohosted functions in regular process memory.

memory consumption and TCB for a cloud function; a trade-off between performance and security. The Cohesion Principle recommends a bisection which reduces communication between the trusted and untrusted system, yet that reduction may imply more complexity in the trusted runtime.

6.1 Task Scheduler

The trusted runtime delegates units of scheduled computations through *tasks*. Tasks are placed onto task queues, consumed by physical threads through a lock-free scheduler; the API of which is listed in 6.1.

Physical threads may schedule tasks for execution on other threads through inter-thread messaging. Each logical core allocated to a cloud function manages a dedicated task queue, addressable via this API. Tasks are submitted by placing a callback and argument pair onto a task queue. The scheduler consumes tasks from each queue in FIFO order, preserving *intra-flow order* of execution. FIFO execution guarantees the following: given well behaved co-operative flows where tasks complete within a given timed interval, if a flow is allowed to execute a task, it will be able to progress by scheduling another. That task will eventually execute, i.e *starvation-free*.

```
class ThreadPool : public IThreadPool
{
  public:
        ThreadPool(size_t threads, threading_mode_t mode);
        ~ThreadPool();
        // Create a new thread identifier
        unsigned get_thrd_id();
        // Explicitly yield execution and saves stack
        // and prepares another virtual thread for execution.
        void Yield();
        // Schedule a task to be executed
        // on current physical thread
        void Schedule(async_cb_t cb, void *args);
        // Schedule a task to be executed
        // on target physical thread
        void ScheduleOn(size_t id, async_cb_t cb, void *args);
        // Get current physical thread id,
        // relative to the current trusted runtime
        int currentThreadId();
        // Get physical thread count
        // assigned to the current trusted runtime
        size_t physicalThreadCount();
        // Get current virtual thread count.
        size_t currentVThreadId();
        // Align stack for new virtual thread
        static void  alignstack(void * ptr);
        // Loop for processing tasks off the task queue.
        // One for each physical thread.
        static void  SchedulerLoop(void *ptr, int status);
        // Entry function for provisioning
        // physical threads to trusted runtime.
        void InitializeThread();
        // Stop scheduler loop and purge all task queues
        void Stop();
        // Check if task scheduler is alive.
        bool Alive();
};
```

Listing 6.1: API for the lock-free trusted runtime task scheduler. Physical and virtual threads are provisioned during cloud function initialization, and each physical function is addressable through inter-thread messaging.

6.1.1 Physical Threads

Cloud functions support two threading-modes, *shared* or *dedicated*. During the initialization procedure, the untrusted runtime delegates a physical core to a cloud function by feeding it to the enclave via a special entry point. A single cloud function may have an unbounded amount of physical cores delegated to it, determined by the application configuration. Once inside the enclave, threads are captured by the task scheduler, and begin executing tasks on behalf

of the cloud function.

Given the availability of physical resources, each cloud function receives dedicated cores for execution, following the Affinity Principle defined in Section 4.2. However, oversubscription will cause threads to be shared among multiple cloud functions. Shared threads are multiplexed between different enclaves incurring additional overhead through AEX, as described in Section 3.1.6.

6.1.2 Virtual Threads

To support legacy software, code which expects blocking behaviour is simulated as a multithreaded system in Diggi. The trusted runtime implements virtual non-preemptive threading through a cooperative FIFO scheduler. Each pinned physical thread in a particular cloud function is mapped to a set of virtual enclave-mode threads. Each virtual thread receives a dedicated stack and execution context, which the assigned physical thread switches between as illustrated in Figure 6.2. Non-preemption is a design property derived from the Affinity Principle in Section 4.2. Preemptive thread scheduling would induce an enclave exit for interrupt-driven context switches.

Virtual threading permits a blocking operation to yield control to the task scheduler, allowing concurrent tasks to execute while waiting for the response. The execution context and stack is preserved in the interim, enabling a perceived synchronous application to asynchronously process multiple requests on a single physical thread. Each physical thread may have an unrestricted amount of virtual threads allocated to it, only limited by memory usage. Figure 6.2 illustrates a blocking read task which permits another virtual thread to execute while waiting for fulfillment.

The trusted runtime implements partial *pthread* support for concurrent programming in cloud functions, fully realized through virtual threads. Threads may be created, started, joined and destroyed from within the cloud function. Additionally, pthread mutexes synchronize perceived concurrent operations by yielding control to the task scheduler on contention.

6.1.3 Oversubscription of Physical Threads

Ideally, Diggi should not oversubscribe physical thread resources. Section 4.1.4 demonstrates the performance impact of sharing physical threading resources between multiple enclaves. However, if the trusted runtime detects that a cloud function is idle, physical threads are relinquished pending new requests. Threading resources allocated to idle functions may be distributed to active functions in the interim. This retains performance in the presence of oversubscription, as long as a subset of cloud functions are idle so that shared physical threads may act as though they are dedicated.

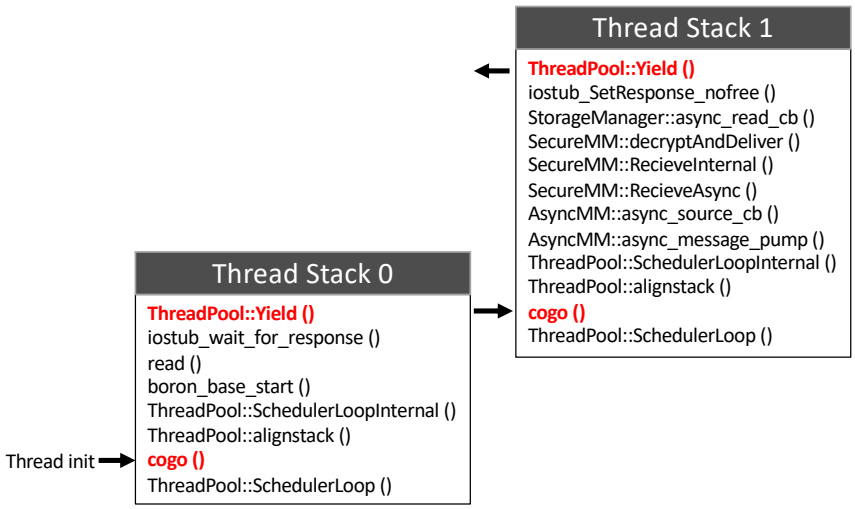


Figure 6.2: Virtual threading in the trusted runtime; The physical thread performs a context switch between two virtual threads. Thread 1 may receive messages pending the return of a read operation on Thread 0 [63].

The trusted runtime detects an idle function by tracking the incoming request frequency. If a function receives no incoming requests for a given configurable observational period, the runtime revokes scheduling of its thread for linearly increasing segments of time. These threading resources are relinquished to the untrusted runtime, which distributes them to other functions. The trusted runtime receives resources to periodically poll for incoming messages, and upon the arrival of a message, the idle function is re-assigned its threading resources. Packet polling intervals must be carefully tuned so not to impact other cloud functions significantly, yet reduce the start latency when a request arrives for a dormant function.

6.2 Messaging

The trusted runtime implements a single message-passing abstraction used by all services. Both internal concurrency, storage and system calls (Translator-Server-pairs), as well as external communication between cloud functions are implemented through this single abstraction. Messaging is implemented as *tasks* and *flows*, complementing the task based abstraction described in the previous section. Conforming to the Pinning Principle stated in Section 4.2, all messages are relayed through an exit-less and asynchronous communication channel.

Diggi messages provide a minimal, efficient, secure and reliable interface

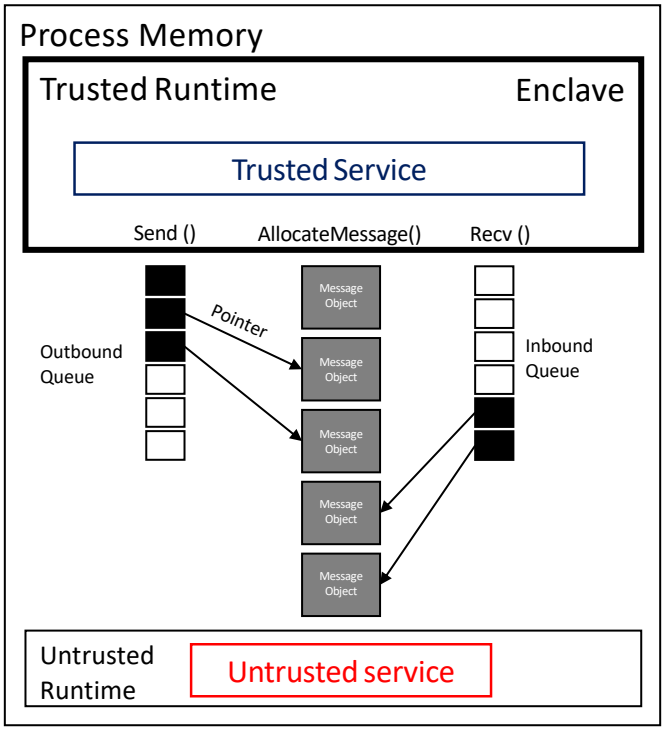


Figure 6.3: Shared memory queue and memory objects used for exit-less communication [63].

between cloud functions. However, more complex feature such as service discovery, publish-subscribe, automatic endpoint code generation and interoperability should be implemented on top. We expect application developers to implement necessary service protocol schemes through native Diggi messages, such as Thrift, protobuf or AMQP.

6.2.1 Concurrent queuing

Exit-less communication is achieved through concurrent queues. Threads pinned inside the trusted runtime send messages by placing them onto an outbound queue, *readable* by the untrusted system. Inversely, the trusted runtime polls another queue for inbound messages, *writable* by the untrusted system. The enclave memory model detailed in Section 3.1.4 explains that SGX allows enclaves to read and write to memory in the host process' address space. Queues used for communicating are stored in untrusted memory to allow read and write access from both parties, as illustrated in Figure 6.3. A concurrent thread in the untrusted system produces messages onto the inbound queue and consumes messages of the outbound queue for a given trusted runtime.

Serializable access to concurrent data structures requires synchronization, however, our design requires all computations to be *wait-free*. We solve the bounded buffer problem by implementing a multi-producer multi-consumer lock-free queuing algorithm modified from work presented by Krizhanovsky [108].

Given that a queue has free slots available for placement, this algorithm implements wait-free consumption and production onto the queue. In the event of an empty or full queue, the operation aborts immediately and control will return to the caller, allowing other tasks to execute concurrently. Synchronization among multiple threads is realized through atomic memory operations. Additionally, each thread holds a unique view of the concurrent queue-state during read/write operations. The algorithm permits the following operations given contention:

- Read-ahead: A thread may consume available messages beyond the lowermost tail as long it does not exceed the lowermost head.
- Write-ahead: A thread may produce messages onto the queue beyond the lowermost head, as long as it does not exceed the lowermost tail.

Before attempting an operation on the queue, each concurrent thread records the latest global head and tail into its local state. These may lag behind the actual head and tail, and on concurrent accesses, an atomic operation acts as the arbiter. The non-committed must retry for an operation on an item beyond the contended. Although lock-free, on highly concurrent access, the algorithm is wait-free for consumption but not for production. The Diggi runtime implementation of the algorithm is shown in Listing 6.2.

```
void produce(void *message, size_t prod_thrd)
{
   thrd[prod_thrd].head = global_head;
   thrd[prod_thrd].head = atomic_inc(global_head);

   while (thrd[prod_thrd].head >= last_tail + qsize){
      auto min_tail = global_tail;

      for (size_t i = 0; i < cons_thrds; ++i){
         auto tmp_tail = thrds[i].tail;
         memory_barrier();
         if (tmp_tail < min_tail)
            min_tail = tmp_tail;
      }

      last_tail = min_tail;

      if (thrds[prod_thrd].head < last_tail + qsize)
         break;
      memory_barrier();
   }

   q[thrds[prod_thrd].head & qmask] = message;
   thrds[prod_thrd].head = max_value;
}

void *consume(size_t cons_thrd)
{
   if(!thrds[cons_thrd].in_progress){
      thrds[cons_thrd].tail = global_tail;
      thrds[cons_thrd].tail = atomic_inc(global_tail);
      thrds[cons_thrd].in_progress = 1;
   }

   if (thrds[cons_thrd].tail >= last_head){
      auto min_heas = global_head;
      for (size_t i = 0; i < prod_thrds; ++i) {
         auto tmp_head = thrds[i].head;
         memory_barrier();
         if (tmp_head < min_head)
            min_head = tmp_head;
      }

      last_head = min_head;

      if (thrds[cons_thrd].tail < last_head)
         break;
      return NULL;
   }

   void *message = q[thrds[cons_thrd].tail & qmask];
   thrds[cons_thrd].tail = max_value;
   thrds[cons_thrd].in_progress = 0;

   return message;
}
```

Listing 6.2: Send and receive queue implementation; Wait-free for all operations except send on a full queue. Both update the local view before attempting enqueue/dequeue.

6.2.2 Message structure

To protect the confidentiality and integrity of messages, the trusted runtime encrypts each outbound message using an AES-128-GCM symmetric-key and a unique session nonce for replay protection. Messages are encrypted in enclave memory and copied into free messages buffers in untrusted memory.

Applications may implement redundant methods for protecting confidentiality and integrity, such as TLS or PGP. Similarly, runtime services such as ephemeral storage implement bespoke protection using authenticated block encryption. The trusted runtime therefore supports a non-encrypted mode, where messages are sent as plaintext.

Plaintext messages are populated directly onto the outbound message buffers, avoiding additional memory operations. Inversely on receive, plaintext messages are directly consumable by the cloud function. All messages inbound to the enclave are explicitly validated by the trusted runtime to avoid IAGO attacks [38].

A message datastructure contains the following members:

$$msg = (t, d, s, i, b, c, p)$$

The message type t is customizable by cloud functions and distinguish between multiple message purposes. The source s and destination d uniquely identifies the message source and message destination. For each session between two communicating parties, i identifies the next task in a flow, designated as the recipient of this message. b is payload size and c holds the intra-session identifier, used for sorting out of order delivery of messages. Lastly, p contains the payload.

Individual Cloud functions are addressable through a 64 bit internal identifier:

$$f_{id} = \{t, n, t, m, a\}$$

where t signifies the function type, n and t address the physical node and thread, m identifies the mode and a the attestation group identifier.

6.2.3 Message Flows

Messaging in Diggi is implemented using the asynchronous programming model described in 5.2, where cloud functions are implemented through *tasks* composed into *flows*. For messages, two fundamental communication modes exist, type- or flow-based. All initial messages begin as a typed message, and cloud functions subscribe to the receipt of messages of a given message type. Type subscriptions are defined through a triple consisting of the type, task callback and context object, stored in the trusted runtime. A cloud function

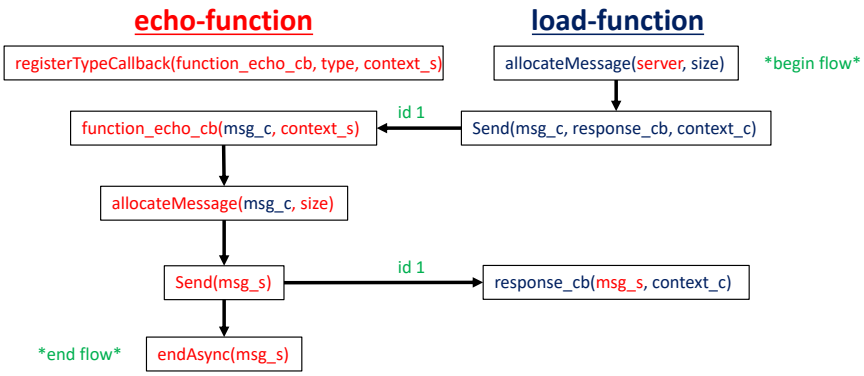


Figure 6.4: An example use of the Diggi messaging API; asynchronous continuation-style flow between two functions, an echo function and a load function [63].

may produce an unbound number of different simultaneous type subscriptions, distinguishing multiple different message end-points. As an example, a cloud function may accept client requests defined by one distinct type, and configuration- or control-requests via another type subscription. Whenever a typed message is received by the runtime, it is handled by the task callback stored when the cloud function subscribe to the type.

Following a typed request, cloud functions may implement a multi-step communication session through a message *flow*. Messages sent as responses are allocated by the messaging API using the originating request message, as illustrated in Listing 5.1. The response generates a *flow-id*, or *i*, set in the message header described above. This identifier is used by all subsequent tasks executed by either side of the communication channel for the longevity of the flow. When sending a response, a single-use task callback and context object tuple is stored by the trusted runtime. The next response for the given flow-id triggers the stored task, receiving the inbound message as input in addition to a discretionary context object for managing state. A given cloud function may implement an unbound amount of additional tasks, defining complex protocols and business logic.

The trusted runtime supports multiple concurrent flows, tasks and registered types for a single cloud function. However, for a unique flow id, the response will always invoke the associated task callback. An example flow based message exchange implementing both sides of the interaction detailed in Listing 5.1, is depicted in Figure 6.4. Typed callbacks are not single-use and may be invoked multiple times, contrary to flow based callbacks, which must be defined for each subsequent step of the flow. Once a flow ends execution, the last response will be sent without a callback task, causing the runtime to clear the flow id from the message and tear down support session state stored to keep track of task execution. On the receiving end, the lack of flow-id will also cause a

teardown of session state. One-way messages, not expecting a response, are similarly sent without a task callback. These do not generate a flow id, as they only target the initial typed subscription task callback set up by the opposing communicating endpoint. The Diggi messaging API is defined in Code Listing 6.3.

```
class SecureMessageManager: public IMessageManager
{
public:
   std::map<uint64_t, key_exchange_context_t> callback_map;
   std::map<std::string, aid_t> name_servicemap;
   TamperProofLog *tamperproofLog_inbound;
   TamperProofLog *tamperproofLog_outbound;
   // Constructor
   SecureMessageManager(
      IDiggiAPI *dapi,
      IIASAPI *api,
      IAsyncMessageManager *mngr,
      std::map<std::string, aid_t> nameservice_updates,
      int expected_thread,
      IDynamicEnclaveMeasurement *dynMR,
      ICryptoImplementation *crypto,
      bool record_func,
      bool trusted_root_func_role);
   // Destructor
   ~SecureMessageManager();
   // Allocate message for destination based on HRN,
   msg_t *allocateMessage(std::string destination,
                          size_t payload_size,
                          msg_convention_t async,
                          msg_delivery_t delivery);
   // Allocate message for destination based on 64bit address.
   msg_t *allocateMessage(aid_t destination,
                          size_t payload_size,
                          msg_convention_t async,
                          msg_delivery_t delivery);
   // Allocate response message based on incomming.
   msg_t *allocateMessage(msg_t *msg, size_t payload_size);
   // End Message flow
   void endAsync(msg_t *msg);
   // Send message.
   void Send(msg_t *msg, async_cb_t task, void *contextobject);
   // Get map of reachable functions (HRN, unique identifier)
   std::map<std::string, aid_t> getfuncNames();
   // Stop message recording (used for shutdown)
   void StopRecording();
};
```

Listing 6.3: The messaging API for developing flow-based asynchronous messaging between cloud functions.

The trusted runtime delivers messages by periodically polling the inbound message queue. Messages off the queue are delivered to the correct task

callback by looking up the type and flow id in an internal index structure. The task scheduler ensures that messages are delivered to the correct task and on the correct thread. All message task responses in Diggi are guaranteed to surface on the same thread as the original request. Multithreaded cloud functions are individually addressable per thread, specified in the recipient field of a message.

The trusted runtime may reach a condition where it is able to deliver requests faster than the cloud function can process them. Over-consumption of incoming requests may cause the host enclave to reach its memory limit and subsequently crash. The task scheduler handles this by throttling incoming message requests, limited to a finite number of parallel tasks in flight at any given moment.

6.3 Ephemeral Storage

Cloud-scale services implementing stateful computing require the ability to store large amounts of information. The trusted runtime could store information in enclave memory, but experiments conducted in Section 4.1.2 suggest that oversubscription of the EPC incurs a significant overhead. To avoid provisioning large enclaves, state which is not part of the most recently used pool of application objects is stored in non-enclave memory.

The trusted runtime implements shielded ephemeral storage of cloud function state through an asynchronous storage API listed in 6.4. Each unit of storage persisted by a given cloud function is referred to as a *state object*. State objects may persist across cloud function lifetimes, for the longevity of the Diggi node. In the event of node failure, state is permanently lost, a property we define as *ephemeral*. Applications depending on fault-tolerant storage should implement state replication explicitly.

```
class StorageManager : public IStorageManager
{
    IDiggiAPI *func_context;
    ISealingAlgorithm *sealer;
    std::map<int, int> pending_write_map;
    std::map<int, off_t> lseekstatemap;
    crc_vector_t block_2_crc;
    std::map<int, off_t> size_of_file;
    std::map<std::string, int> object_name_to_id;
    std::map<int, std::string> object_id_to_name;
public:
        // Constructor
    StorageManager(IDiggiAPI *context, ISealingAlgorithm *seal);
        // Retrieve Current state of replay vector table
    void GetCRCReplayVector(crc_vector_t **vectors);
        // Set replay vector table
        void SetCRCReplayVector(crc_vector_t *vectors);
        // Close the interaction on a state object
        void close(int object_id, bool omit_from_log);
        // Seek to an internal offset within the state object.
        int seek(int object_id, int offset, int whence);
        // Open a state object given a object name.
    void async_open(    const char *object_name,
                            async_cb_t task,
                            void *context,
                            bool encrypted,
                            bool omit_from_log);
    // Read from a state object
    void async_read(    int object_id,
                            size_t nbyte,
                                async_cb_t task,
                            void *context,
                            bool encrypted,
                            bool omit_from_log);
    // Write to a state object
    void async_write(int object_id,
                                    const void *buf,
                                    size_t nbyte,
                                    async_cb_t task,
                                    void *context,
                                    bool encrypted,
                                    bool omit_from_log);
    //Destructor
    ~StorageManager();
};
```

Listing 6.4: Interface implementing persistent ephemeral storage for cloud functions. Access is initiated by retrieving an object identifier from a human readable name. This identifier is used for subsequent operations on the state object.

Exit-less state preservation is implemented asynchronously through message flows, as discussed in the previous section. Storage is organized as individ-

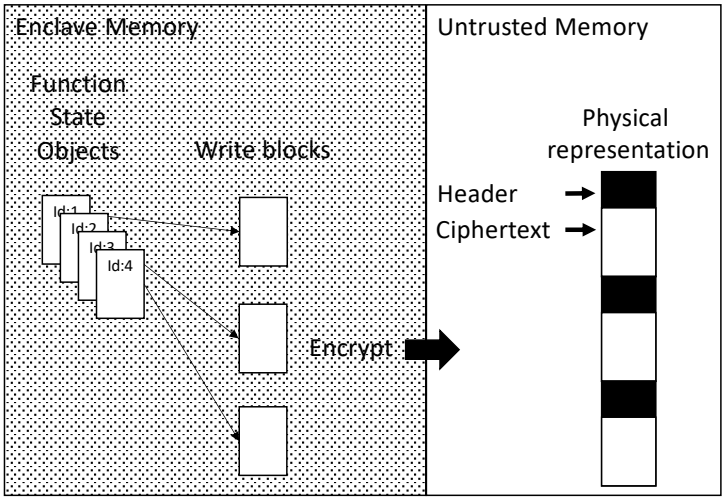


Figure 6.5: Function state preservation using encrypted ephemeral storage in Diggi [63].

ual objects, addressable via human readable names. Objects may be partially read/written to and expand in size. An internal position indicator directs the current point within an object. Read and write operations are translated to encrypted blocks in the trusted runtime and relayed to the untrusted system through messages. Blocks are stored as a virtually contiguous buffer in untrusted memory and each distinct cloud functions encrypts state using different keys. Blocks stored in untrusted memory are additionally protected from unsolicited modification and data-replay.

The cipher-text representation of an encrypted block expands in size compared to its plaintext counterpart. The block header includes size s and MAC-tags t_{mac}, while the encrypted payload contains a CRC-32 hash h_{crc} of the plaintext for a given block:

$$block = \{s, t_{mac}, enc\{h_{crc}, payload\}\}$$

Blocks are encrypted using a 128bit AES-GCM cipher, derived from a combination of the SGX platform seal key, embedded in hardware, and the SIGSTRUCT certificate signature. Blocks can only be decrypted by the same physical hardware and a cloud function signed by the same developer key. This property allows multiple cloud functions signed by the same developer to share persisted state. Multiple versions of a cloud function may moreover share the encrypted state, simplifying the upgrade procedure. A stricter mode which additionally requires an identical cloud function measurement is also configurable. The absence of an authenticated cloud function renders session state stored in untrusted memory useless, guaranteeing revocation.

Replay protection. While blocks written to untrusted ephemeral storage are encrypted and protected from modifications, a malicious attacker may subvert the execution of an instance by serving stale data back to the application. Replay attacks for communication primitives and replay attacks for storage differ in one significant aspect, data entries *should* be remembered and replayed. However, only the last entry stored at a given location should be returned upon an explicit request.

The trusted runtime implements a replay protection protocol where each block of encrypted data is associated with a 32 bit value defining the block version. Each stored object is associated with a block vector, and all read/write operations targeting blocks within an object must correctly check/update each version number. Block versions are derived from a *CRC32*[1] hash of the plaintext block data, stored in an encrypted field of the persisted cipher-text.

To ensure performant lookup, this vector table is stored in enclave memory. Given an encrypted block size of 4 KB, the corresponding vector table will account for 1MB of enclave memory per GB of data written. Large objects may increase the memory footprint, and as an optimization, block vector arrays may themselves be stored in separate state objects to reduce enclave memory consumption.

Given a write operation to a particular block, the associated vector is updated, encrypted and stored along with the block in untrusted memory. Any subsequent reads of the block will compare the contents of the block header with the expected entry in the vector block table. Algorithm 1 outlines the protocol steps.

The vector block scheme is partially inspired by the enclave memory integrity implementation detailed in section 3.1.4. Given a pre-existing persisted state object, the associated vector block array may be delivered securely to the runtime from a trusted source during cloud function deployment. An empty state object is represented initially by an empty block vector. Block vectors expand dynamically as the stored state object increases in size.

Replay protection requires serializability and the trusted runtime therefore does not support concurrent operations on the same block. Multithreaded cloud functions are instead expected to implement synchronization explicitly.

6.4 Accountability

SGX cannot protect against denial of service, where the underlying infrastructure reduces or revokes computing resources for an ongoing trusted enclave computation. Determining whether a cloud function has verifiably executed is important for non-idempotent operations with side-effects, such as transferring

1. https://create.stephan-brumme.com/crc32/

Algorithm 1 Block vector operations

```
1: procedure CHECK BLOCK RANGE
2:     bi ← Index of block
3:     cnt ← Block read count
4:     for i ← bi to cnt + bi do
5:         if Hash(i) ≠ Stored(j) then
6:             goto error
       return ok
1: procedure UPDATE BLOCK RANGE
2:     bi ← Index of block
3:     cnt ← Block write count
4:     for j ← bi to cnt + bi do
5:         if Hash(j) ≠ Stored(j) then
6:             goto error
7:         else
8:             Stored(j) ← Hash(j) + 1
       return ok
```

funds between two accounts. An untrusted system may claim the operation concluded, despite the contrary. For such operations to be verifiable, a cloud function must be able to prove that it executed correctly.

Cloud functions are identifiable through the enclave measurement representing its initial state. However, identifying long-running functions is difficult when state diverges from the initial measurement.

The trusted runtime implements accountability for Diggi cloud functions through the ability to *record* and *replay* execution thereof. As described in 4.4.1, we define the identity of a cloud function as the state predicate $i(0)$ initially derived from the enclave measurement. Following the first input message, the next identity of a cloud function becomes:

$$i(1) = i(0) + i(message_1)$$

For any following state x the identity is defined by:

$$i(x) = i(x-1) + i(message_x)$$

To identify any given state, the trusted runtime stores the derived state. The premise of trusting recorded state verification evidence requires a trusted runtime with a known predicate state. Auditing the execution of a cloud function may be realized through two different modes, each with different tradeoffs.

The first mode implements a dynamic attestation module creating a new state identifier for each incoming message. The state identifier is derived through the same process as initial enclave measurement, a SHA-256 hash of

the inbound message. The function preserves sequential ordering by hashing each state and its predecessor state as follows:

$$state_i = hash(state_{i-1} + message_i), \forall i$$

The identity of a cloud function after i messages is represented by $state_i$. A hypothetical cloud function may after 1000 received messages deliver its identity $state_{1000}$ as proof of execution. For deterministic cloud functions, this proof will guarantee that a cloud function executed without interference by the untrusted system. This mode only requires that the final state is preserved, and a successful audit will arrive at the same concluding state.

For non-deterministic cloud functions, using temporal or random input, identity will differ between executions, making verification difficult. To support verification of non-deterministic cloud functions Diggi implements a tamperproof log for storing message states. The message-states are stored as an append only data structure encrypted in untrusted memory using the storage interface defined in Listing 6.4. Two logs are stored, one for input state and one for output state. As described in section 6.3 encryption keys for storage are uniquely tied to cloud function identity. This guarantees that only a cloud function from the same issuer will be able to read the tamperproof log. Non-deterministic operations such as *gettime()* and *rand()* are realized through translators into asynchronous messages, as described in Section 5.4. Just as all other interactions, these operations are recorded on the tamperproof log. To verify the cloud function execution through an *audit*, a reference cloud function, signed by the developer, is loaded into the trusted runtime. The runtime replays the tamperproof log input to the cloud function, comparing the actual output messages with the expected. If the output differs or the execution is incomplete, the reference implementation will have evidence of tampering.

Both modes require one additional execution to audit the cloud function. The prior only stores a current state hash value and is more memory conserving. A fully replay-able audit in the presence of non-deterministic execution requires a full record of all internal state. Audit procedures are only executed if necessary, and we expect that accountability achieves deterrence through the perceived probability of an audit.

6.5 Untrusted Runtime

Cloud functions deployed on the same physical host, share an *untrusted runtime* responsible for lifecycle management and intra-node message delivery. To maximize co-hosting potential, satisfying the Pinning Principle defined in section 4.2, the untrusted runtime consumes as few dedicated threads as possible.

We implement the untrusted runtime as an asynchronous system with the task and flow computing abstraction detailed in section 5.2. A polling task repeatedly schedules itself onto the task scheduler, as defined by the pattern in Listing 5.3. The task polls inbound messages from network and outbound messages from the individual cloud function queues. An outbound message object is moved from the respective queue to the correct outbound socket defined by the destination cloud function identifier. Incoming messages are likewise consumed from the network interface and delivered to the correct input queue. Diggi messaging is protocol agnostic, but requires that the underlying communication media provides error correction and reliable packet delivery. Inter-node communication is in its current form implemented across non-blocking TCP sockets.

During deployment, the untrusted runtime reserves physical threads for message processing and delivery. The default allocates a single physical thread to the untrusted runtime, however many-core systems may allocate several.

The untrusted runtime bootstraps execution by loading the trusted runtime and cloud function into SGX enclaves according to the deployment configuration; one for each defined cloud function. Once the trusted runtime is initialized, physical threads enter the enclave and are captured by the trusted runtime task scheduler.

Message queues are allocated by the untrusted runtime and delivered to each trusted runtime during initialization. Each receive a dedicated input queue which may be addressed by the untrusted runtime and other cloud functions directly. Depending on the recipient, the trusted runtime chooses a target queue for relaying messages. Dedicated queues exist for ephemeral storage, operating system "servers" and outbound inter-node messages. Queues only store pointers to message objects, and the untrusted runtime maintains a global pool of free message object buffers concurrently consumable by all cloud functions. The process is illustrated in Figure 6.3.

6.6 The Diggi Trusted Root

The SGXs security model requires a trusted principal for secure application deployment and joint software attestation of cloud functions. Three principals exist for an SGX application; the build environment compiling and distributing binaries, IAS responsible for identifying a correct hardware platform, and the developer service, which deploys and authenticates the enclave. All require an implicitly trusted underlying infrastructure, and distributing these roles may increase scalability, but also increase the TCB.

Section 4.4.3 describes how a serverless application may be composed of multiple distinct enclaves. These are compiled and signed according to the application configuration, and delivered to the target host for deployment.

Although conceptually different, Diggi implements compilation, deployment, key distribution and authentication into a single principal; the *trusted root*. The trusted root is hosted and owned exclusively by the service developer, external to the cloud, and interacts with Diggi node processes hosted in a public cloud.

During deployment, the trusted root measures and signs each cloud function binary, creating the SIGSTRUCT certificate, used by SGX to authenticate a binary. This certificate is stored by the trusted root in an index, for use during the attestation process.

The platform attestation key, described in 3.1.5 is made available to the trusted root, used to verify quotes. SGX requires this key material be integrated into a fuse array embedded in the hardware during the manufacturing process. This design is a departure from the actual architecture of SGX which does not disclose the attestation key to developers. We simulate the attestation key available through the Diggi trusted root. Diggi is not inherently tied to the SGX platform and we conjecture that an open TEE architecture replaces the IAS by enabling application developers to provision hardware secrets to platforms directly.

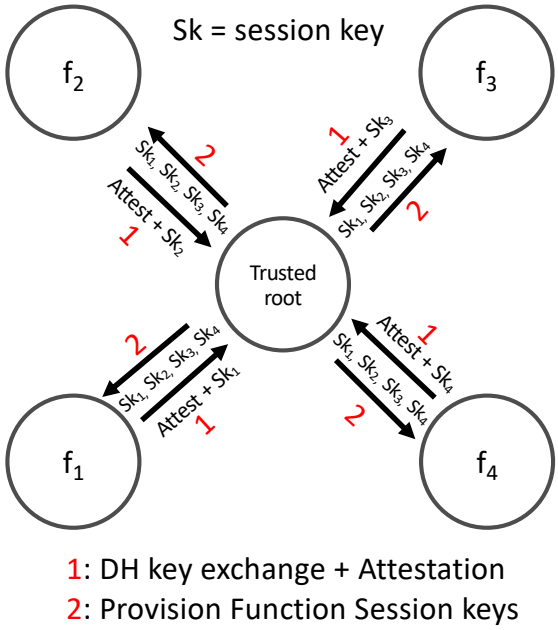


Figure 6.6: The cloud function attestation process. Each function individually authenticates themselves to the trusted root. Once all are authenticated, the trusted distributes session keys to each. By the transitive property, each attested cloud function may now trust one another [63].

Once the trusted root successfully deployed all binaries in an application configuration to the respective Diggi host nodes, it awaits requests for attes-

tation. Cloud functions are divided into attestation groups, as defined in the application configuration. The application configuration specifies the granularity of attestation groups. An application may have multiple attestation groups per service, depending on the tradeoffs between resource consumption and individual access control and authentication. A cloud function cannot belong to multiple attestation groups.

Each cloud function requests an attestation for its allocated attestation group from the trusted root. This protocol is derived from a modified Sigma protocol [9] which piggybacks the delivery and verification of attestation evidence on-top of the Diffie-Hellman symmetric key exchange. The Intel SGX SDK provides an API for message preparation in each step for attesting a single client. We modify the protocol to implement multi-party co-attestation.

Each cloud function is compiled with a static public key tr_p, for which the trusted root stores the corresponding private key tr_s. The initial request contains the SGX quote signed by the QE, a nonce and the Diffie-Hellman [52] initial modulo p, base g, and exponent g^a sent to the trusted root, encrypted using tr_p:

$$enc_{tr_p}(p, g, Q_c, n, g^a) \xrightarrow{\text{send}} TR$$

The trusted root decrypts the message using the private key tr_s, and sends a request to the Intel attestation authority, with the received quote. The attestation authority may then, based on the signature of the received quote, verify that the attestation key used by the QE to sign the quote belongs to a valid hardware platform. The trusted root then verifies that the measurement in the quote Q_c is identical to the SIGSTRUCT certificate. If the process is successful, the trusted root then responds to the client, with a response to the initial nonce n_r and the exponent g^b, signed by the private key tr_s:

$$CF \xleftarrow{\text{send}} \text{sig}_{M_p}(g^b, n_r)$$

The host then verifies the signature using the public key tr_p.

The process is repeated for all participants in the attestation group. Attestation verifies to all functions jointly, the initial state of each function and the authenticity of the hardware platform where each function is executing. Once concluded, the chain of trust within an attestation group follows from the transitive property, if

$$CF_i \xrightarrow{trust} TR$$

and

$$TR \xrightarrow{trust} CF_{i+1}$$

where CF are cloud functions and TR is the trusted root, then

$$CF_i \xrightarrow{trust} CF_j \forall i, j$$

The trusted root completes the attestation by distributing session keys to all participants, encrypted by the trusted roots own session key. All cloud functions within a group may now communicate across a secure and authenticated channel. The high-level steps of this protocol is illustrated in Figure 6.6.

Multiple mutually distrusting attestation groups may be cohosted simultaneously, either through the same or different trusted roots. For a single attestation group, only a single trusted root may complete the co-attestation protocol. We expect attestation groups to be bounded in size. A single attestation group should not require scalability per trusted root.

6.7 Summary

This section has detailed the Diggi prototype runtime for shielding cloud function execution from an untrusted public cloud using the Intel Software Guard eXtensions (SGX) Trusted Execution Environment (TEE). Through an asynchronous task scheduler and flow messaging API, Diggi efficiently utilizes threading resources to maximize function hosting. Accountable cloud functions may prove a correct execution through recorded state transitions in a tamper-proof log which may be replayed in the event of an audit. Cloud functions may additionally store shielded ephemeral data in the untrusted system, without impacting responsiveness or cohosted functions. Serverless applications consisting of multiple cloud functions are compiled, deployed and authenticated jointly using the trusted root principal, demonstrating how a distributed serverless application may be hosted securely on untrusted infrastructure.

7

Evaluation

Chapter 4 describes the design of Diggi, a secure serverless runtime for hosting secure online services in an untrusted public cloud. Based on this design, we developed a prototype system to support efficient use of the SGX TEE, detailed in the previous chapter. To investigate the thesis conjecture, we must evaluate whether Diggi is efficient enough to demonstrate a practical potential. This chapter will detail the evaluation of our prototype implementation, and moreover seek to answer the following *research questions*:

- **Research Question 1:** To demonstrate a practical design, our prototype runtime should be applicable and add value. *Can Diggi be applied to create contemporary secure cloud services for managing privacy sensitive data?*
- **Research Question 2:** Securing software systems from an untrusted public cloud will inherently add overhead in each measure taken, but an efficient system should minimize this overhead. *What is the penalty in performance of executing cloud functions in the Diggi trusted runtime, and can this overhead be characterized as reasonable?*
- **Research Question 3:** Ensuring high utilization of hardware is an important goal in the design of serverless systems, and runtimes should aspire to pack as many cloud functions onto the same host as efficiently possible. *How does cloud functions scale per host, and are resources distributed optimally among cohosted instances?*
- **Research Question 4**: Accountable cloud functions require explicit protection and storage to ensure that a verifiable proof is preserved during function execution. *What is the overhead of storing a full transcript of cloud function execution vs. storing the final measurement, and additionally, what*

is the cost of an audit operation?

7.1 Experimental Setup

To answer the research questions stated above, we evaluate our prototype on an experimental setup which synthesizes the expected infrastructure environment of an untrusted public cloud. Cloud infrastructure consists of racks of connected server-grade computing resources, with high-speed interconnect between racks, and aggregate backbone connections between multiple racks in a datacenter. We synthesize the hardware configuration of a single rack in a public cloud by connecting multiple physical machines together on a dedicated physical network. Each host runs an Ubuntu 16.04 Linux-based OS along with version 2.5 of the Intel SGX SDK and kernel driver. A single Diggi daemon process is provisioned per physical machine, capable of hosting the necessary cloud functions.

All experiments are carried out on Dell R330 servers with Xeon E3-1270 v6 3.8 GHz processors with 4 cores (8 Hyper Threads), configured with 64 GB UDIMM DDR4 RAM and 10GBaseT Intel X540 NICs connected together by a Dell Networking N1524 10 Gbit Ethernet switch. Each is configured with 8 Dell 1.2 TB 10K RPM 400-AHNO hard drives connected through a Symbios Logic MegaRAID SAS-3 3108 raid controller with a 1 GB write cache, configured in RAID 5 (striped with parity). We disable CPU frequency throttling and other power management features across all machines, and set the PRM to its maximum allowed 128MB.

The interconnect between nodes is dedicated to Diggi traffic, and all external connections to WAN occur through a separate network on a 1Gbit NIC, as illustrated in Figure 7.1. All experiments are configured such that each Diggi node dedicates one logical core for intra-node message processing. The Diggi runtime may be hosted outside SGX without any major modifications to the codebase. To illustrate the inherent overhead, we selectively compare against baseline experiments hosted in regular memory.

Except where evident from context, all measurements are performed on pre-deployed Diggi cloud functions with co-attestation and audit logging disabled. The Diggi runtime is hosted in a dedicated environment which minimizes interference. Unless explicitly stated, experimental results are displayed as averages across 10 individual runs. For experiments where the *relative standard deviation* is negligible, we omit error bars for clarity. Only experiments which explicitly test resource sharing/contention across cloud functions demonstrates a *relative standard deviation* above 2.4 percent.

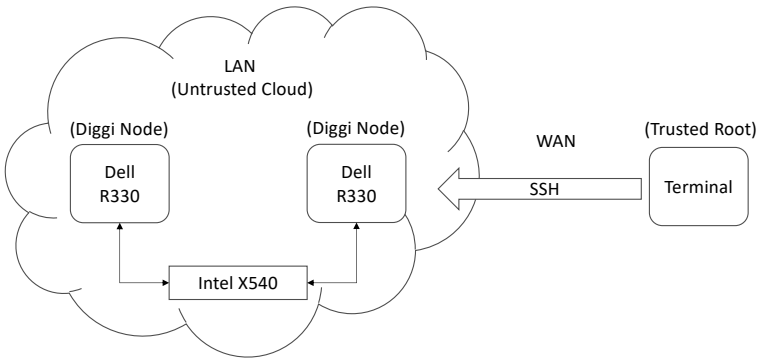


Figure 7.1: Experimental setup synthesizing an untrusted cloud. Each physical host represents a Diggi Node running the deamon process for deploying cloud functions. The terminal client serves as the Trusted Root external to the untrusted cloud.

7.2 Cohosting Cloud Functions

Section 2.3 describes the importance of maximizing utilization; packing more cloud functions onto each physical host increases the revenue. Diggi cloud functions are cohosted without virtualization since SGX isolates each trusted runtime from the underlying system and other cloud functions. Additional security isolation is redundant, and may instead increase the runtime overhead of executing cloud functions.

As detailed in Section 3.1.4, once total memory consumption exceeds the architectural limits, SGX will encrypt unused enclave pages and evict them to untrusted DRAM, increasing memory access latency. In order to efficiently host multiple isolated cloud functions per physical host, the Diggi trusted runtime is built from the ground to conserve memory usage.

We evaluate the packing potential of cohosted Diggi cloud functions by measuring the overhead of hosting a collection of isolated simple functions on the same physical host. The function of choice is the echo-function listed in 5.1, which returns whatever messages it receives immediately to the sender. Because of its simplicity, this provides us with a lower bound on memory usage per secure runtime, and an indication of optimal per-node scaling for Diggi cloud functions.

We measure the throughput as the number of messages processed by each echo function, and additionally, the Round Trip Time (RTT) between the request and response as observed by the load-generating function. Each echo function consumes a total of 2.2MB including the trusted runtime, and we expect to be able to host 40 parallel functions per host before exhausting the pool of physical trusted memory(PRM). Once the physical threshold is reached, latency and throughput performance decreases due to memory multiplexing.

Each cloud function is initially allocated a single physical thread and two

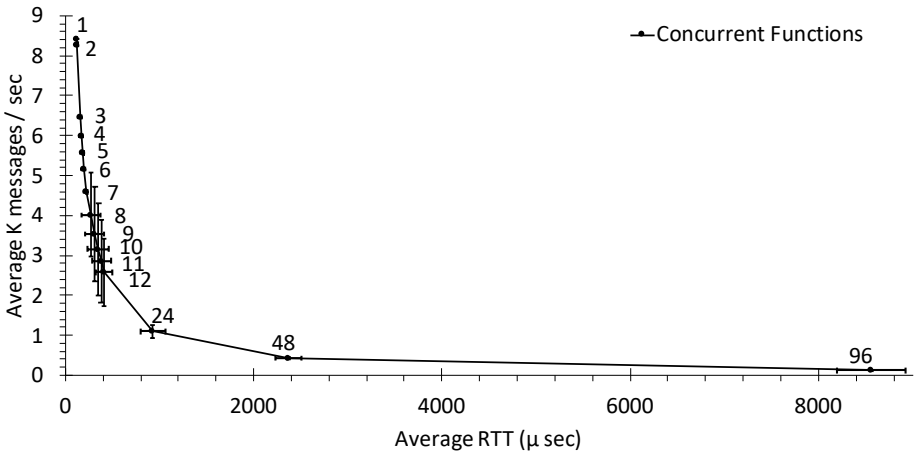


Figure 7.2: Average throughput for cohosted instances vs. average round-trip time.

virtual threads, each with 64KB of memory set aside for the runtime stack. By allocating just two virtual threads to each, the Diggi runtime is able to handle blocking system call operations while preserving memory consumption from stack allocation. However, once the number of deployed cloud functions per host exceeds the logical core count, each physical thread is shared across multiple functions, which may impact performance. High resolution timers are not available in enclave mode, and so to accurately measure RTT, the load generating functions are executing in regular process memory. Load generating functions and echo-functions are hosted on separate physical hosts, connected by a high-speed interconnect, as illustrated in figure 7.1, where each load-generating function targets exactly one echo-function.

To avoid warmup latency for TCP handshake operations, each connected Diggi node uses a single connection to send and receive data to another node. To minimize the probability of a network link becoming the bottleneck, we disable the TCP send buffer (Nagle) algorithm for all connections, and use a small Diggi message size of 1KB. Our intuition is that a high packet rate will increasingly stress the non-blocking runtime and packet-scheduling in each node, rather than saturate the network. Figure 7.2 illustrates the average perceived per-function end-to-end latency and throughput of the echo experiment as the amount of cohosted echo-functions increases.

Figure 7.3 displays the total throughput measured for all cohosted functions over the average per-function latency measured. We observe that the total throughput peaks at around 7 cohosted functions, the point at which Diggi must share the 7 available logical cores (1 set aside for message scheduling) among multiple enclaves. While this suggests that the experiment is CPU bound, the runtime is still able to delegate resources fairly across 12 functions without decreasing the total throughput or latency significantly and even more across

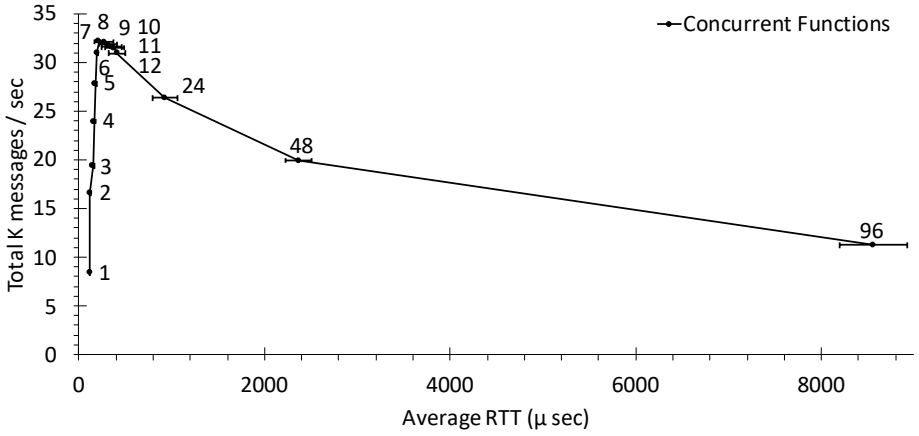


Figure 7.3: Total throughput for cohosted instances vs. average round trip time.

24 functions without significant increases in latency. This suggest that the cap on throughput is dominated by network bandwidth.

Although the interconnect is expected to carry roughly 1.1 GB per second, disabling packet buffering and transmitting small messages reduce this potential significantly. Additionally, to simplify the experiment, each load generating function only schedules a single message exchange at-a-time, awaiting the response of the prior, between each exchange.

The variance in both experiments increases above 7 cores, illustrating resource interference between cloud functions. All threading resources are shared preemptively through virtual threads, and as the cohosted functions grow in number, interrupts will trigger AEX-operations to share physical cores among multiple enclaves, contributing to the variance in results. Diggi additionally maps cloud functions to available cores in round-robin, and as more enclaves are forced to share cores, the throughput variance drops.

We allocate a single core for message scheduling, and the total throughput experiment reveals that this is sufficient for serving 7 cloud functions, without becoming the bottleneck. For processors with more parallelism, increasing the cores allocated to message scheduling is necessary to ensure further vertical scaling. However, logical cores share some caching, bus bandwidth and pipeline execution steps which will limit scalability for each added core in the event of a pipeline stall. For increased vertical scalability, Diggi should implement Non-Uniform Memory Access (NUMA)-awareness to handle non-uniform access patterns by separate NUMA-cores accessing main memory.

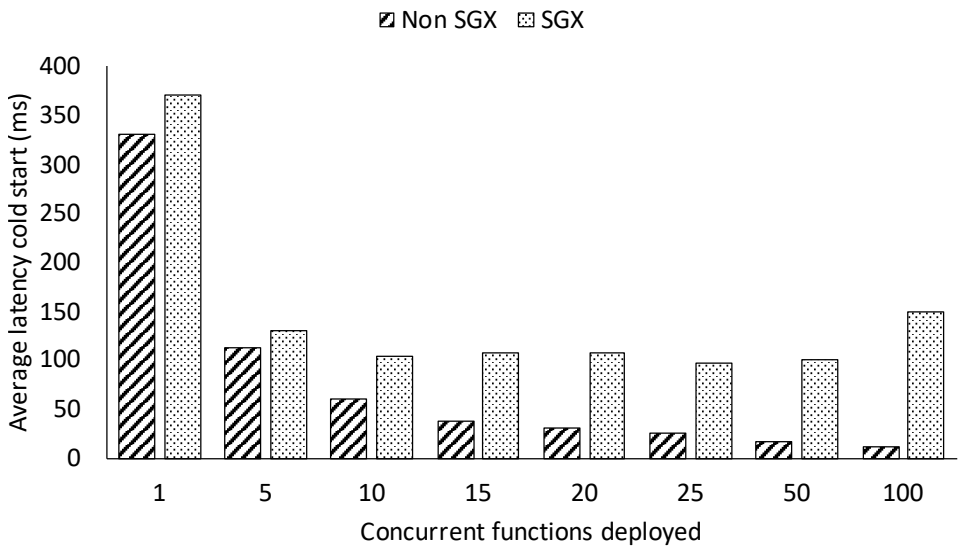


Figure 7.4: Average cold start deployment latency for Diggi cloud functions executing in SGX and outside.

7.2.1 Cold-start

Section 2.5 describes the challenge of *cold-start* provisioning serverless systems. Reducing cold-start latency for cloud functions is essential for application responsiveness and maximizing revenue potential. We measure the cold-start performance of Diggi as the persieved latency of concurrently deployed cloud functions. We additionally compare the responsiveness to a baseline consisting of a non-SGX cloud function hosted in regular process memory. Each cloud function implements the echo-function detailed above, using roughly 2.2MB of PRM.

As expected, function deployment inside SGX is on average 4x slower than the baseline. For this experiment, cloud functions have not been preprocessed by SGX in any form, and launching an enclave therefore includes the binary measurement and signature verification for each function. Deploying 100 functions concurrently on a single physical host in Diggi renders an average cold-start latency of 149 ms.

Precisely comparing the Diggi prototype implementation against existing proprietary systems is not possible due to the lack of documentation and peer-reviewed literature. As an approximation, Wang et al. [184] conduct a series of *black-box*-experiments revealing the cold and warm-start latency of cloud functions hosted by AWS, Google, and Azure, listed in Table 7.1. These experiments measure the network (end-to-end) latency as experienced by an invoker targeting a datacenter within the same region. This includes the cost of allocation, scheduling and provisioning physical resources, as detailed

Service	Cold Start	Warm Start	Runtime
Azure Functions	3640ms	320ms	3320ms
Google Cloud Functions	493ms	79ms	414ms
AWS Lambda	265ms	25ms	240ms
Diggi	n/a	n/a	149ms

Table 7.1: The cold and warm-start performance of several serverless cloud providers compared to the Diggi runtime. Runtime overhead is calculated by subtracting warm-start from cold-start.

in Section 2.4. The Diggi prototype consists of a secure serverless runtime for deploying cloud functions, excluding some of the necessary architectural components required to implement a full FaaS framework. Consequentially, our experiments only measure the runtime overhead of provisioning cloud functions. The majority of the warm-start overhead reported by Wang et al. [184] may be attributed to authentication, load-balancing and network latency. We arrive at a probable number for comparing the runtime provisioning costs by by subtracting this figure from the cold start measurement, as displayed by the third column in Table 7.1.

Diggi does not use virtual machines or containers for isolation, and despite the overhead of SGX, the runtime cost of provisioning isolated function instances is comparable to these competing systems. However, based on the heterogeneous hosting environment of our synthetic cloud and the proprietary public clouds listed, the results may vary.

Ephemeral Storage The previous experiments have all demonstrated a lowermost boundary for provisioning Diggi cloud functions, with a peak memory consumption of 2.2MB of PRM. Additional memory consumption caused by application state will increase this figure and impact the cold start performance as SGX enclaves must measure and account for all memory consumption during deployment. The experiment conducted in Section 4.1.1 demonstrates that the cost of enclave provisioning increases as a function of its size. The Diggi runtime avoids this overhead by allowing applications to manage state through ephemeral storage, persisted across function lifetimes without impacting the provisioning cost.

7.3 Communication Overhead

Diggi communication primitives may be implemented on top of a reliable communication protocol of choice. The current prototype implements internode communication using TCP. To evaluate the overhead of hosting functions inside of enclaves, we evaluate the end-to-end latency and throughput of

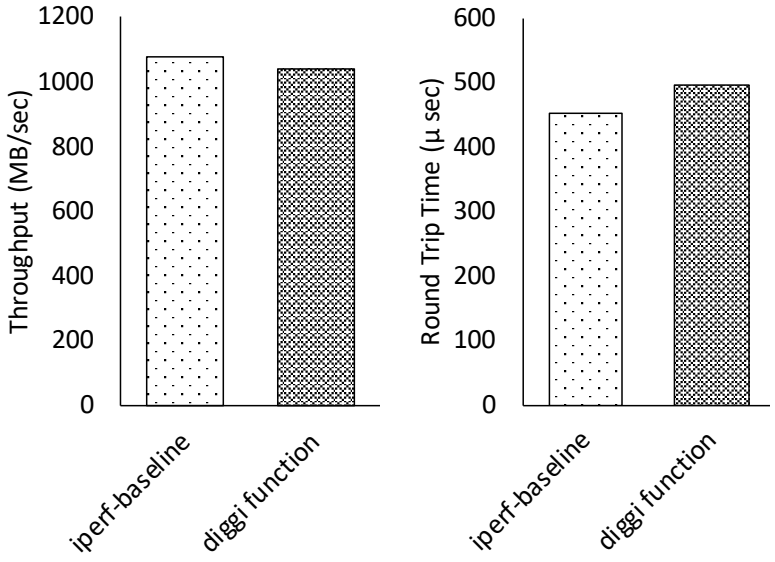


Figure 7.5: (*1*) Throughput measurements for the baseline and Diggi cloud functions. (*2*) Round-trip time for the baseline measurements and the Diggi runtime.

message delivery in Diggi compared to a baseline.

Similar to the previous experiment we use two functions, a load generating and an echo function hosted on two distinct physical machines. The experiment measures the throughput and RTT of a message to be delivered back to the sender. To be able to use high resolution timers, we host the load-generating function in regular process memory.

As a baseline, we evaluates the peak Transmission Control Protocol (TCP) bandwidth and latency between two physical machines in our synthetic cloud infrastructure rack using *iperf*[1]. To achieve a fair mix between latency and throughput performance we set the TCP write buffer size to 128KB for both iperf and Diggi, and the load-generating function to 128KB per Diggi message. *iperf* uses all available cores to generate load onto the network, and is able to achieve a maximum TCP-layer throughput of 1.08GB per second, compared to a theoretical peak load of 1.1GB per second across the data link layer of the 10Gbit interconnect. We expect no packet-loss between the two physically connected machines, and attribute this difference to additional transport and IP-layer message header information. Diggi is able to achieve 1.04GB per second using only 4 load-generating functions each using a single logical core, demonstrating that the experiment is network bound. The difference in maximum throughput is here attributed to the additional message header information, sent as part of the Diggi message. Messaging in Diggi adds 42 µsec to the latency (9 percent), which we consider negligible. This overhead is caused by additional message

1. https://iperf.fr/

scheduling and encryption in the Diggi runtime.

7.4 Trusted Runtime System Call Translation

In order for the Diggi runtime to be practical, cloud functions may import legacy libraries by implementing secure system dependent access explicitly through translator/server pairs, as detailed in Section 5.4. We demonstrate the overhead of translated system calls in Diggi by implementing simple file system operations and measuring the performance overhead compared to a baseline.

Figure 7.6 and 7.7 demonstrate the latency of reads and write operations as a function of block sizes. We compare the native execution speed of the gLibc library against encrypted IO operations which explicitly exit the enclave (ocalls), and Diggi *translated* exit-less (async) IO; both with and without encryption.

The baseline gLibc benchmark is executed in regular memory with no encryption. Block encryption is dependent on the write size and we observe that the latency increases exponentially for write operations, both for ocalls and exit-less translations. Read operations are less impacted by encryption since they are not re-entrant. However, ocalls degrade significantly in performance for larger reads, and we theorize that this is due to memory multiplexing for large reads copied across the enclave boundary, similar to the results presented in Section 4.1.3.

Compared to the gLibc baseline, there is a significant added cost of executing IO operations in SGX. For IO originating from within SGX, there is at least 5-10x slowdown, not including encryption. However, there is a noticeable performance benefit for both reads and writes by using asynchronous (exit-less) IO in Diggi as opposed to explicit context switches (ocalls). Unencrypted asynchronous write operations are less impacted by larger write sizes since they do not exhibit the re-entrant behaviour required by encrypted writes.

Experiments are subject to spikes in latency for IO operations interfacing with the operating system. We attribute this to file system and/or RAID controller caching, which intermittently trigger expensive *cache-miss* behaviour. Additionally, ocalls are increasingly impacted by intermittent expensive enclave page-faults due to the additional memory operations required to complete each IO request.

In its current form, the Diggi trusted runtime does not cache any intermediate data in trusted memory, but rather pushes all writes out of the enclave immediately. Caching blocks inside the trusted runtime may increase performance but also increase enclave memory consumption.

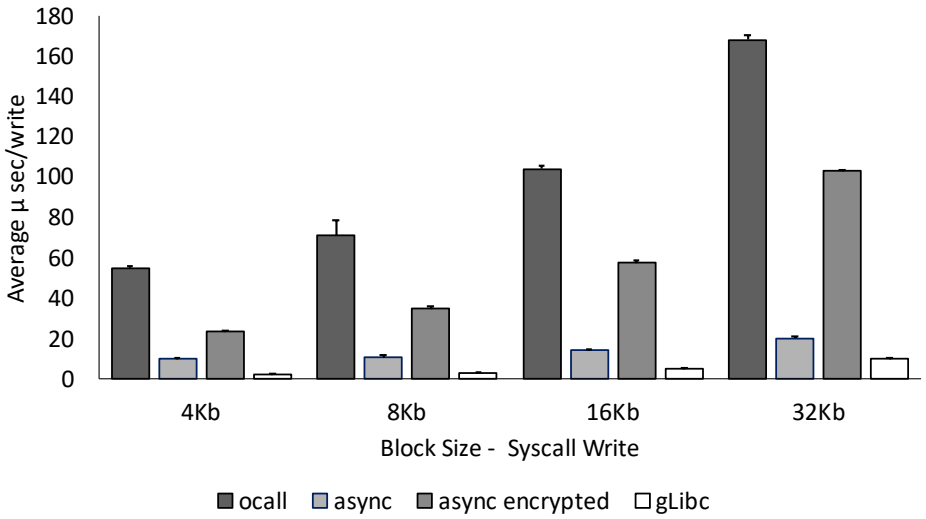


Figure 7.6: A comparison of asynchronous (exit-less) write latency in Diggi versus synchronous (ocalls) and gLibc as a baseline.

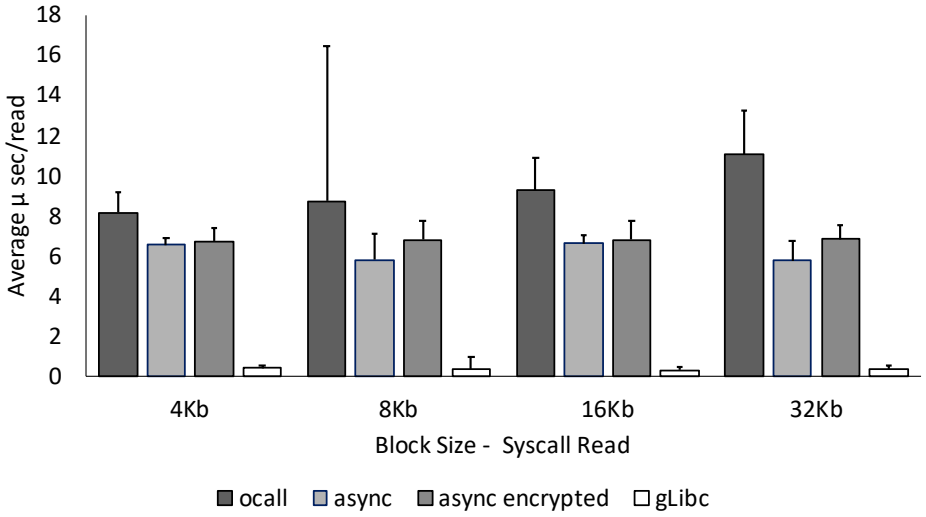


Figure 7.7: A comparison of asynchronous (exit-less) read latency in Diggi versus synchronous (ocalls) and gLibc as a baseline.

7.4.1 Supporting Legacy Libraries in Diggi Cloud functions

To demonstrate the portability of existing software, we include the SQLite3[2] Database Management System (DBMS) library in a cloud function and implement the necessary system call translators and servers. The SQLite3 library comprises some 128K lines of code, and aside from compiling, no modifications were made to support it in the Diggi trusted runtime. UNIX Large File System and Write-Ahead-Logging mode, requiring shared memory, was disabled by compiler flags. Extending library support, providing associated data structures, and translating system call operations securely, added some 5K lines of code[3] to the trusted runtime. Although this increases the complexity of the trusted runtime, the bulk of this code implements system call translators, encryption and pthread support, potentially reusable by subsequent library translations.

System calls are implemented by translating requests to secure messages which are relayed to *servers* in the untrusted runtime interfacing directly with the host OS. Any such interaction with the untrusted system may leak information, given data dependent access. Each translation is prepared independently to ensure minimal leakage of information by integrity protection and encryption.

More work could reduce the number of translations further by emulating more behaviour in enclave-mode. However, this would increase the complexity and TCB of the trusted runtime.

Table 7.2 and 7.3 list the system calls and library support implemented to support this particular scenario. For system calls which are not translated, and library operations outside of pthread support, we emulate the behaviour expected by SQLite3 in the trusted runtime to avoid security issues and reduce complexity. Other legacy libraries may require more complex (true) behaviour for these operations. To evaluate the performance of the ported library, we implement a cloud function able to serve as a OnLine Transaction Processing (OLTP) database, by using the SQLite3 library, and evaluate the resulting performance against a TPC-C transaction mix load.

The TPC-C benchmark models a series of warehouses with stock changing hands, and orders and payments processed[4]. We execute TPC-C in full through a simple database protocol implemented on top of Diggi messages. Prior to each experiment, TPC-C loads content simulating a set of warehouses into the database. For all experiments, unless stated otherwise, we configure the benchmark to load the content simulating a single warehouse, and each experimental iteration generate transactional load for 10 seconds against the target database. The loading phase serves as a warmup for issuing transactional load to the system, and as the experiments will illustrate, we do not see signifi-

2. https://www.sqlite.org
3. Measured using the Source Lines of Code (SLOC) tool.
4. http://www.tpc.org/tpcc/

System Call	Implementation Technique
lstat	Translation
stat	Translation
fstat	Translation
close	Translation
access	Translation
fsync	Translation
lseek	Translation
open	Translation
read	Translation
write	Translation
unlink	Translation
umask	permissions are allways 0777 because file is encrypted
readlink	Identical path
getpid	Static process identity
time	Monotonically incremented
getcwd	Single Directory
fcntl	(Multi-Process file locking) No-Op
geteuid	static user identity
nanosleep	Yield to runtime task scheduler

Table 7.2: System calls required by the SQLite3 library, and implementation techniques to securely serve them from within the Diggi trusted runtime.

Standard Library Support	Implementation Technique
pthread_mutex_init	atomic compare and swap
pthread_mutex_lock	atomic + yield to scheduler
pthread_mutex_unlock	atomic compare and swap
pthread_mutex_trylock	atomic compare and swap
pthread_create	atomic + yield to scheduler
pthread_join	atomic + yield to scheduler
errno	set according to error
sysconf	static value, SQLite3 only asks for pagesize

Table 7.3: Standard library additions and implementation techniques to support SQLite3 in a Diggi trusted runtime.

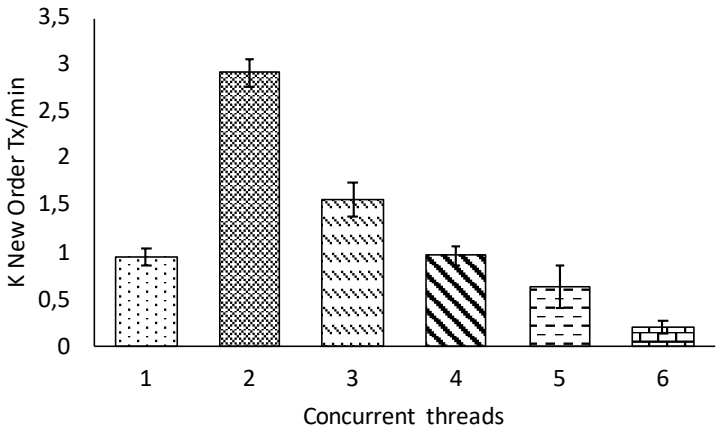


Figure 7.8: Tx/m vs. concurrent dedicated threads to Diggi server instance.

cant variance in results across multiple runs. We report the performance as new-order transactions committed per minute, as is common.

The cloud function implements SQL-query requests from clients and IO operations exit-less and concurrently on a single physical thread. Because TPC-C is dependent on runtime randomness and high-resolution timers, we do not execute the TCP-C benchmark in trusted memory. However, we do not expect this to significantly impact performance measurements; rather slightly decrease the potential load put onto the OLTP cloud function.

Concurrency. SQLite3 implements concurrency via the pthread library, and we demonstrate the multiprocessing performance by varying the physical core count dedicated to the OLTP cloud function. The Diggi runtime pthread implementation uses the trusted runtime task scheduler to distribute concurrent operations across physical threads. If the pthread library expects more than the cloud function is allotted, virtual threads are used instead.

Figure 7.8 illustrates new order transactions per minute as given by TPC-C benchmark versus different core count allocated to the OLTP cloud function. Our findings are in tune with the expected concurrency scaling, as write-transactions in SQLite3 require exclusive database access. The best per-thread performance is achieved using two dedicated hardware threads for each instance. As the dedicated core count grows, Diggi delegates an increasing amount of cores to the trusted runtime, leaving less resources for external messaging and system call servers. The limited concurrency of the SQLite3 engine decreases the utility of adding cores beyond 2. However, prior work [146] has demonstrated that enabling Write Ahead Log (WAL) may improve performance in SQLite3 by a factor of 28x.

The experiment additionally illuminates a curious result where the single

core experiment is underperforming. This issue did not surface for in-memory workloads, and we expect this hyper-linear vertical scalability to be attributed to an increasingly hot filesystem or RAID controller cache. Additionally, the raid controller may optimize reads/writes based on drive head position for multiple simultaneous requests, beneficial for concurrent IO.

Runtime Overhead. Based on the prior experiment on concurrency, two dedicated cores achieved the best performance and we evaluate this configuration of the OLTP cloud function against 7 other non-standard configurations, listed here:

- **Baseline, In-Memory:** SQLite3 configured as in-memory executing on two threads in regular process memory.
- **Baseline, 1 thread, In-Memory:** SQLite3 configured as in-memory executing on a single thread in regular process memory.
- **In-Memory, 1 thread (SGX):** SQLite3 configured as in-memory executing on a single thread in enclave mode.
- **Baseline, Durable:** SQLite3 configured as using the file system executing on two threads in regular process memory.
- **In-Memory (SGX):** SQLite3 configured as in-memory executing on two threads in enclave mode.
- **Durable, No encryption (SGX):** SQLite3 configured as using the file system executing on two threads in enclave mode, without encryption or integrity protection.
- **Durable (SGX):** SQLite3 configured as using the file system executing on two threads in enclave mode.
- **Ephemeral Storage:** SQLite3 configured to store data in diggi ephemeral object storage, on a single thread.

Based on initial experimentation, we observed that our setup was network bound and favors Diggi by closing the performance gap, only resulting in a 2x decrease in performance compared to the In-Memory Baseline. To reduce potential slowdown from networking and more precisely determine the runtime overhead, we ran the experiment against TCP-C benchmark clients on the same Diggi node. The overhead compared to the baseline is now measured as a 5x slowdown in performance, as seen in Figure 7.9. This is consistent with observations made by comparable experiments in SCONE, with SQLite3 hosted in Secure Containers [14].

We observe that the in-memory configurations are increasingly impacted by lock contention, compared to the durable counterparts. The best in-memory performance is achieved by using a single thread for both enclave and regular process memory experiments. This is expected as SQLite3 implements strict serializability for write transactions via mutex locks. For durable configurations,

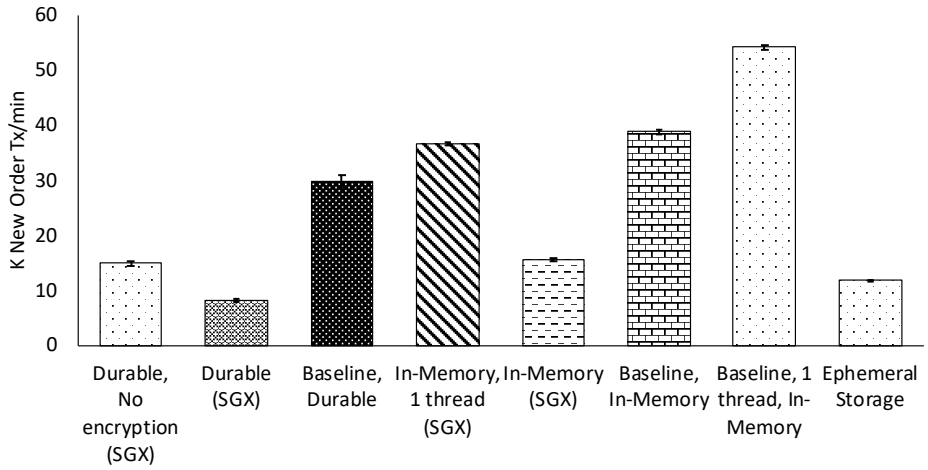


Figure 7.9: Tx/m for different configurations, load generated on the same host.

Diggi achieves higher performance with two threads both for regular process memory and SGX, as the system is able to increasingly saturate blocking writes via multiple virtual and physical threads.

We note the cost of Diggi is significant compared to configurations running in untrusted memory. Performance of TPC-C is reduced by 40 percent with IO encryption turned on, and running Diggi in memory within SGX without any IO operations yields a significant improvement in performance over that of durable. The peak memory consumption for all configurations except the in-memory, never exceed 13Mb. Although the in-memory configuration exhibits the best performance in isolation, it will impact the cold start performance, and additionally, the memory access latency of cohosted cloud functions.

Lastly, we implement a dedicated runtime configuration which instead of translators, use Diggi ephemeral object storage to persist SQLite3 data. The experiment illustrates the performance of ephemeral storage compared to the in-memory SGX configuration, where ephemeral storage results in a 67 percent drop in new order transactions per minute. We chose a single warehouse as the TPC-C configuration across all experiments to illustrate the best case situation for in-memory configurations of an enclave database. Our preliminary experiments on SGX memory performance suggests that the performance gap between ephemeral storage and enclave in-memory execution will likely narrow if executed against a larger database consisting of more warehouses.

We implemented Diggi to safely and efficiently enable co-location of multiple mutually distrusting cloud functions on the same host, despite the performance limitations introduced by Intel SGX. The in-memory SGX configuration will increase deployment time, as demonstrated in Section 4.1.1. Additionally, it exhausts all available PRM for a given host, which decrease the expected

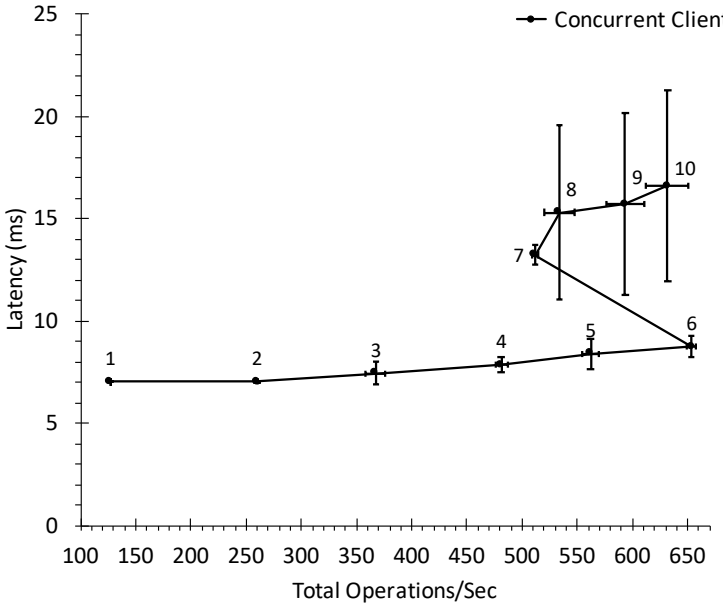


Figure 7.10: Average latency for cohosted instances vs. total throughput.

per-instance performance for cohosted OLTP cloud functions.

Cohosting OLTP cloud functions. Section 7.2 illustrate the optimal cohosting potential for a simple cloud function, allowing us to scale to 24 cohosted cloud functions without significant drop in latency. We repeat this experiment with a more complex example; the previously detailed OLTP cloud function, hosted via system call translations.

For all iterations of this experiment we dedicate two logical cores to the untrusted runtime, one for managing system call *servers* and the other for message scheduling. Each operation consists of a *select* query retrieving 100 rows from a single database table.

Figure 7.10 shows the throughput (total operations per second) over average latency of requests as experienced by each cloud function. We observe that scaling concurrent Diggi instances beyond 6 cores reduces the average throughput by 30 percent while increasing latency by 50 percent. Each host CPU has 8 logical cores and Diggi is therefore no longer able to dedicate threads to separate instances and must schedule multiple on each core. We observe that allocating two cores to the runtime is sufficient for serving 6 simultaneous OLTP cloud functions.

Similar to the previous experiments, over-provisioning concurrent instances beyond the available cores has a negative effect on performance. System call *servers* initially receive dedicated cores, however are assigned resources from the same pool as cloud functions. Once the round robin process completes

an assignment round, it will assign leftover requestors to already occupied cores. For 7 concurrent OLTP cloud functions, one of them becomes colocated with the system call server, causing a dip in total throughput, not observed in previous cohosting experiments.

Without over-provisioning, we observe that cohosted Diggi instances are not particularly impacted by one another. Performance is expected to deteriorate further when the total memory consumption grows beyond the physically allotted 128MB.

7.5 Accountable Cloud Functions

Section 4.4.1 details the design of accountable cloud functions which may capture state mutations consistently, allowing an auditor to verify function execution. Section 6.4 details the implementation of this concept into the Diggi trusted runtime, consisting of two distinct modes to establish accountability; a single dynamic attestation proof or a recorded log.

The former minimizes storage requirements by only storing a 32 byte value representing the current mutated state. Given a deterministic cloud function, this value will not change for repeated executions, and may be used in a subsequent audit process to prove that the cloud function executed as expected. However, to support non-determinstic cloud functions, only a full record of the execution state may prove that the cloud function executed correctly. We evaluate these two modes of accountability against a baseline to detect the overhead of execution.

Additionally, a full record log requires us to replay execution on a reference implementation. To determine the cost of replaying messages, we measure the time taken to execute the same experiment by simply replaying the previously recorded input operations onto the OLTP cloud function.

Any cloud function implemented in the Diggi runtime may be *accountable* as all state mutations happen through message-passing, including system call translators.

To demonstrate that a complex cloud function may be accountable, we use the OLTP cloud function from the previous experiment and test against a TPC-C transaction mix load. Our tamperproof log design stores the content asynchronously into Diggi ephemeral storage, and the performance of the log structure hinges on this runtime service being performant.

We test the total runtime for a preconfigured TPC-C test, configured to generate 5 seconds of transactional load targeting the OLTP cloud function before completing execution. In Figure 7.11 we observe that recording a full log of state mutations decreases performance by 42 percent compared to the baseline (No-Record). Recording only the dynamic attestation proof for each message-exchange results in no performance degradation for this inherently

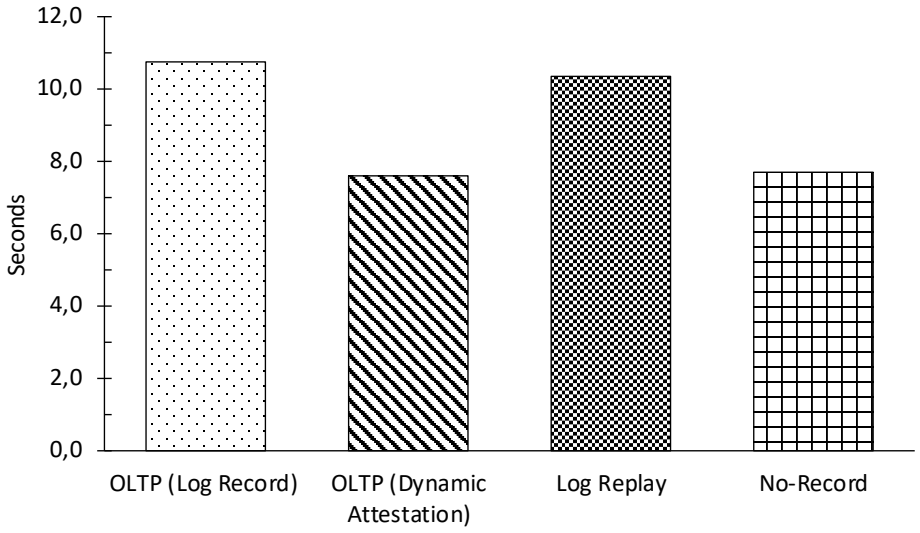


Figure 7.11: Execution time for 5 seconds of TPC-C transaction mix load, including bootstrapping initial tables.

IO bound experiment. We suspect that a CPU bound experiment may yet illicit a degradation in performance. Replaying the inbound messages for our OLTP cloud function exhibits similar performance to that of the recording experiment. We attribute most of the overhead to encrypting/decrypting ephemeral storage objects. The replay experiment runs the OLTP cloud function in isolation, committing inbound messages from ephemeral storage to the cloud function in the exact order they were recorded. These messages replay both interaction with the TCP-C loading function and the system call translations. Both replay and record functionality in the Diggi trusted runtime is implemented asynchronously to reduce the impact on cloud function execution. However, the translator servers and ephemeral storage facility in untrusted memory share logical cores, which may impact performance slightly. An experiment stressing the CPU may achieve a different performance profile.

7.6 Use Case: A neural network image classification pipeline

The cloud enables processing and inference at scale, not generally cost-effective in on-premise systems. We demonstrate that Diggi is practical by developing an end-to-end application mirroring a real-world use case. We implement a

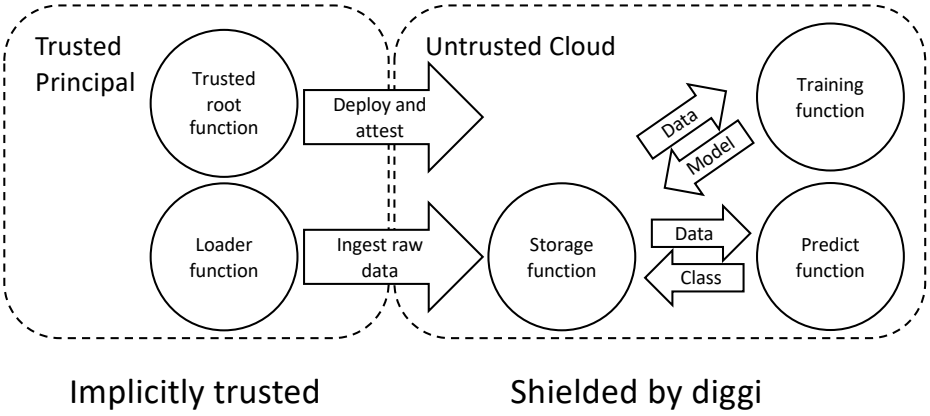


Figure 7.12: A machine learning pipeline implemented as Diggi cloud functions, shielded by the Diggi trusted runtime, deployed and authenticated by the trusted root.

cloud inference pipeline for neural network classification of images where data and computations are shielded from the untrusted cloud by the Diggi runtime. The pipeline consists of 5 distinct stages implemented as Diggi cloud functions, organized into a single attestation group as illustrated in Figure 7.12:

- *Loader function*: Responsible for preparing and ingesting imagery data, storing it in cloud storage. The loader is analogous to a sensory service, delivering raw sensor data to the pipeline.
- *Trusted root function*: The trusted root is, as detailed in Section 6.6, responsible for initializing a secure and authenticated communication channel between all parties in the group through co-attestation.
- *Training function*: The trainer initializes the neural network, and streams data from the storage function in batches, applying them to the network for training.
- *Predict function*: Once complete the neural network is serialized and persisted to structured storage. The predict function is then able to retrieve the trained model, and classify new incoming samples.
- *Storage function*: A secure data repository for storing structured training/test data and the trained models.

Training, predict and storage functions are shielded by the Diggi trusted runtime individually, enabling them to be hosted in a public cloud. Listing 7.13 depicts the Diggi configuration used for this application pipeline.

```
"Diggi-node-1": {
 "network": "127.0.0.1:6000",
 "funcs": {
  "trusted_root_func": {
   "trusted-root": "1",
   "threads": "1",
   "messageencryption": "1"
  },
  "mnist_loader_func": {
   "skip-attestation": "0",
   "train-sample-count": "60000",
   "test-sample-count": "10000",
   "threads": "1",
   "messageencryption": "0",
   "attestation-group": "1",
   "train-image-path": "train-images-idx3-ubyte",
   "train-label-path": "train-labels-idx1-ubyte",
   "test-image-path": "t10k-images-idx3-ubyte",
   "test-label-path": "t10k-labels-idx1-ubyte",
   "load-target-db-func": "sql_server_func"
 }},
 "enclave": {
  "funcs": {
   "structured_storage_func": {
    "record-func": "0",
    "skip-attestation": "0",
    "attestation-group": "1",
    "threads": "1",
    "messageencryption": "1",
    "syscall-interposition": "1",
    "fileencryption": "1",
    "in-memory": "0"
   },
   "neural_network_train_func": {
    "skip-attestation": "0",
    "attestation-group": "1",
    "threads": "1",
    "messageencryption": "1",
    "syscall-interposition": "0",
    "output-layers": "10",
    "hidden-layers": "300",
    "epochs": "30",
    "learning-rate": "0.1",
    "data-source": "sql_server_func"
   },
   "neural_network_predict_func": {
    "skip-attestation": "0",
    "attestation-group": "1",
    "threads": "1",
    "messageencryption": "1",
    "syscall-interposition": "0",
    "data-source": "sql_server_func"
}}}}
```

Figure 7.13: The Diggi configuration for a neural network training pipeline, consisting of 5 components, implemented as Diggi persistent functions.

Figure 7.14: Sample hand written digits from the MNIST dataset.

We test our learning pipeline on a well documented pattern recognition benchmark, the MNIST handwritten digit dataset[113]. This dataset consists of grayscale 28 by 28 pixel images depicting handwritten digits between 0 and 10, each pixel occupying a byte, with associated labels. This benchmark is commonly used as a reference point for developing image classifiers, and a sample collection of the image data is depicted in Figure 7.14. We chose this set over a more contemporary dataset such as *imagenet*[50], mainly to reduce scope and complexity.

We implement a 2-layer fully connected feed-forward neural network in C++ where each neuron models the perceptron with a sigmoid activation function[151]. The MNIST database partitions images and labels into a training set of 60,000 and a test set of 10,000 images, and we adjust the networks parameters based on the outcome of the training set. We set the learn rate to 0.1, and settle on 29 epochs for training. Each pixel is mapped to an input neuron, totaling 784, and a single hidden layer is set to 300 neurons. The output layer is set to 10 neurons, one for each class. Aside from normalization, no pre-prosessing, dimensional reduction or component analysis were performed on the raw input dataset. Imagery data is retrieved from structured storage in batches of 40 to reduce memory consumption. Initial observations show that regardless of data transfer costs, neural network training dominates pipeline execution time.

The choice of hidden layer neurons are based on the results reported in [113], and we observe a similar loss rate to that reported in this publication on testing; *5.5 percent* loss rate on the test dataset at *29 epochs*. However, we observe the Mean Square Error (MSE) and training loss-ratio to be low, suggesting potential overfitting.

To evaluate the overhead of executing compute heavy workloads in SGX, we compare the overhead of executing our training function in SGX versus the

equivalent in regular process memory. Figure 7.15 compares the training time of each batch of imagery data in enclave-mode compared to an identical run in regular process memory. Enclave-mode exhibits 2.4x the overhead of regular process memory. A batch size of 40 images of 784 bytes ensures that all can fit inside a single memory page, most likely cached. Results isolate compute cost from memory achieving a higher comparative performance than other workloads with more memory intensive operations.

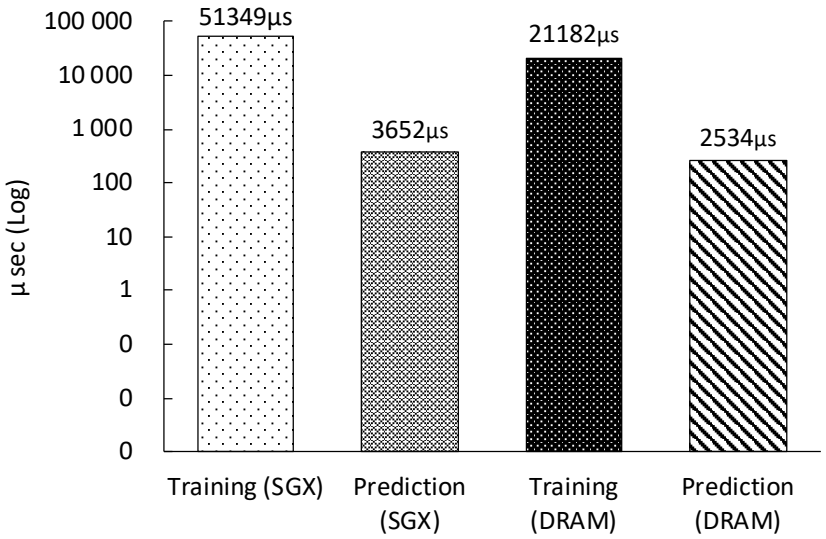


Figure 7.15: The training and prediction overhead for a 2-layer fully connected feed-forward perceptron neural network, in SGX and regular DRAM respectively. Training is measured on a 784 x 40 batch matrix, with 40 samples. Prediction is measured by classifying the digit of a single image.

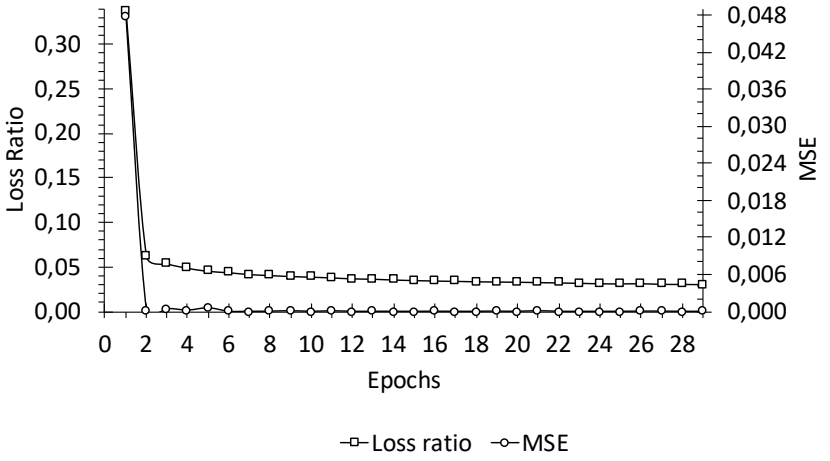


Figure 7.16: Loss ratio on training data and the Mean Square Error(MSE) as epochs progress.

This proof-of-concept implementation only uses a single hardware thread for training. Work to distribute load across multiple threads and multiple nodes are left to future work. Additionally, our neural network implementation uses no specialized library for matrix operations, SIMD or vector instructions (although AVX and SSE are available in SGX). We expect performance to improve given the application of these optimizations. Neural networks, and more contemporary convolutional neural networks (deep learning) are better trained on massively data-parallel hardware architectures, such as GPUs, TPUs or FPGA layouts. However, classification is cheap and may be served in SGX without significant impact to performance. In our example, the Loader function is placed on an implicitly trusted system. Trusted edge-computing may complement this pipeline by implementing secure sensory collection in an untrusted environment.

7.7 Summary

We demonstrate that our prototype implementation of Diggi is efficient, practical, and despite the overhead associated with encrypted block storage and secure memory, support responsive applications. Although we only evaluate Diggi on the Intel SGX TEE, comparable facilities for AMD and ARM exhibit similar operational characteristics; we expect that our concepts and ideas may also apply there.

The next chapter will discuss Diggi in context of concurrent and related work, with particular focus on trusted distributed system. Lastly we will conclude the thesis by discussing the observations made here and map our findings in search

of answers to the *research questions* initially stated, and furthermore present the next steps for future scientific work on the material presented.

8

Discussion

Distributed systems are favorable in situations which require scale, availability and fault tolerance. However, these systems require bespoke design decisions for security where distributing state implies distributing secrets to multiple physical locations. Distribution increases the attack surface (TCB) of a software system, and complex inter-process relationships may moreover lead to bugs which are traditionally hard to discover.

Diggi is designed and implemented to enable the development of secure distributed services hosted in a untrusted public cloud. The work presented in this thesis is in part influenced by, and topically situated, among a tapestry of related research. This chapter presents a selection of the most relevant concepts and systems which protect hosted applications from an untrusted underlying system.

8.1 Mitigating and improving SGX-based systems

Native code executing in enclave memory is exposed to bugs or exploits which trigger data-leaks to the writable host process' memory address space. Systems which instrument memory references [137] or implement memory safety as part of a natively compiled language [183] are able to protect against ROP-style attacks or stray pointers triggering unsolicited writes to untrusted memory. Similarly, Kuvaiskii et al. [110] implement memory safety in SGX using tagged memory pointers, which hide allocation metadata in the unused upper 32 bits of each enclave memory reference.

Diggi supports legacy systems, by exposing a translator server abstraction to simplify partitioning and porting existing systems into SGX. Similar efforts simplify development by integrating enclave components as a programming language extensions which minimize the porting work necessary to partition applications into trusted and untrusted components [61].

Orenbach et al. [136] conceal the controlled side-channel in the SGX memory model by proposing a set of modifications to the SGX ISA which gives an enclave exclusive control over its own page faults.

Rollback attacks may return an application to a known good prior authentic state in non-ideempotent systems triggering behaviour such as *double spending*. Forking attacks are implemented by selectively rolling back state to different clients for a given application. Diggi solves this problem by implementing accountable cloud functions, where the burden of a provable interaction is on the cloud function to submit evidence back to the client. Brandenburger et al. [26] introduce *Lightweight Collective Memory*, a distributed protocol to detect integrity and consistency violations among multiple participating system clients, for operations with side-effects.

Diggi is built from the ground up to be efficient, adhering to the performance principles outlined in Section 4.2. A large part of this work, revolves around managing enclave memory usage and context switch frequency. The memory integrity protection features for SGX incurs a significant overhead and Taassori et al. [171] addresses the overhead by simulating an alternative organization of the EPC into a variable arity unified tree (VAULT). Similar to our findings, Weisse et al. [190] realize that context switches are expensive and implement HotCalls. Through a spin-lock based mechanism, dedicated threads within and outside are able to conduct system calls in synchrony, exhibiting close to native performance.

Cloud services require elasticity to be efficient, and migrating runtimes is therefore a requirement for capacity management. As discussed in Chapter 3, SGX enclaves use ephemeral keys to encrypt both memory and sealed state, uniquely tied to the hardware platform instance. Additionally, migration of trusted enclaves is susceptible to fork and rollback attacks by the untrusted cloud. Gu et al. [76] propose a secure protocol for live migration of SGX enclaves. Alder et al. [6] extend this, by also supporting architectural services such as monotonic counters and sealed data.

Intel SGX is an inherently proprietary system, several aspects of which are confidential. Systems which interact with IAS to attest production enclave signatures are required to be explicitly onboarded onto the SGX ecosystem. Although Intel supports developer provisioned attestation keys, the process still involves approval, and the reliance on, proprietary undocumented technology. Chen et al. [39] propose OPERA, an open SGX attestation service decoupled from Intels IAS. OPERA is a standalone, open source attestation service providing the same privacy-preserving and verifiable properties as IAS, without relying on a single point of failure.

Costan and Devadas [46] conduct a comprehensive security analysis of SGX, falsifying some of its guarantees by explaining in detail exploitable vulnerabilities within the architecture. These lead the authors to implement Sanctum [47], an alternative hardware architectural extension providing many of the same properties as SGX, but targeted towards the Rocket RISC-V chip architecture.

8.2 Formal Methods, Verifiable Execution and Policy Enforcement

As described above, despite the benefit of trusted execution, bugs, misconfigurations or system level attacks such as rollback or denial of service, require additional measures to protect from an untrusted environment.

Moat [166] is a tool which utilizes formal verification to ensure the confidentiality of code executing within enclaves. Based on precise adversarial models and formal specification of SGX system properties, Moat uses information flow analysis and automatic theorem evaluation to prove the confidentiality of trusted execution.

Subramanyan et al. [169] introduce the concept of a trusted abstract platform (TAP); a formal description of an ideal enclave system with a parametrizable adversarial model. TAP presents machine checkable proofs showing that the platform is confidential, integrity preserving and provides secure remote attestation. Moreover, the authors additionally demonstrate how existing enclave systems such as SGX and Sanctum implement secure enclaves under the respective adversarial models.

Ferraiuolo et al. [57] acknowledge the extreme complexity of the SGX microarchitecture and its limited potential to mitigate design flaws and bugs. Komodo decouples trusted hardware and securely hosted software through a privileged software *monitor*. The correctness and security properties are similarly machine-checkable through formalized proofs.

Delegated enforcement through trustable software constructs, mediates both access latency for policy verification, and increases the potential richness of remotely enforceable policies. Birrell et al. [25] evaluate multiple architectural designs for implementing use-based policy enforcement using Intel SGX. Software attestation enables reference monitor designs outsourcing enforcement traditionally only executed at source. By moving logic closer to data via attested enclaves, the overhead of policy enforcement is reduced. This work is implemented on an early version of Diggi. Krahn et al. [106] recognize a similar opportunity, and enable rich policy enforcement on storage separate from the I/O stack by using SGX. Policy Enhanced Secure Object Store (PESOS) may host policy compliant data on untrusted systems. Matetic et al. [121] introduce

brokered delegation, which allow flexible delegation of credential and access rights to third party systems, protected by TEEs. Djoko et al. [53] use SGX to implement credential based access controll of storage resources protected by SGX. Nexus avoids expensive re-encryption of storage as all access to physical volumes occur through enclaves.

PeerReview [79] implements accountability in distributed systems, where Byzantine failures are detectable and linked to faults. Moreover, a correct node may prove itself against false accusations. PeerReview implements support for these properties through a secure log of all messages transmitted by each node in the system. PeerReview requires a correct node to be deterministic, sign messages and periodically audit the system. SGX-Log [101] implements tamper-proof and encrypted storage of system logs for use in an audit procedure, such as forensic analysis. Similarly, CUSTOS [139] is a framework implementing tamper evident audit procedures of Operating Systems using TEEs. CUSTOS additionally implements a decentralized audit protocol, capable of real-time detection of integrity violations. This approach bares similarity to Lightweight Collective Memory [26], discussed in the previous section. LibSeal [15] implements a non-repudable service log which is able to arbit claims for SLA violations between a service provider and clients, through TEEs. LibSeal interposes a TLS library mediating all network access, logging all interaction to sealed storage.

8.3 Secure Analytics and Storage systems

Securing data processing pipelines and storage from an untrusted cloud is made possible through progress in hardware design, encryption techniques and algorithm research. For SGX specifically, hosting large volumes of data in enclave memory is infeasible due to the expected overhead. Data replay attacks may attempt to repeat stale information to hosted enclaves. Additionally, data-dependent operations on sensitive data will inherently leak information to the untrusted system.

EnclaveDB [145] partitions the Microsoft Hekaton in-memory SQL Server by moving parts of the query processing into trusted memory. EnclaveDB mitigates replay attacks by reusing transaction identifiers as nonces. The in-memory Hekaton engine stores all memory slabs in PRM and is, due to memory restrictions in current SGX hardware, only evaluated at scale on a simulated test bench by injecting speculative performance penalty timings into the executable binary. EnclaveDB briefly mentions hosting mutually distrusting database instances within the Hekaton engine, however does not describe the concept further. TrustedDB [19] and Cipherbase [12] support queries on encrypted data using trusted hardware, however does not provide complete confidentiality or integrity protection. Furthermore, the two are limited to custom hardware

co-processors, not generally available on commodity platforms. Sartakov et al. [154] implement a secure rack-scale in-memory database using SGX and SQLite. Similarly to Diggi ephemeral storage, STANLite uses user-level paging to efficiently use regular DRAM for data storage rather than the restrictive pool of EPC. Additionally, STANLite uses Remote Direct Memory Access (RDMA) to distribute storage across a rack of SGX capable systems. Weiser and Werner [189] proposes a generic I/O architecture for creating trusted paths connecting generic peripheral devices securely to enclave software, without inspection from the hypervisor or host OS.

More complex processing and lookup techniques require additional care to not expose data-dependent operations to the host system. ObliDB [56] implements generic analytical query processing for SGX-capable DBMSs, protecting execution from data-dependent processing. Similarly, Oblix [127] implements an oblivious search index structure for protecting server index operations and responses from clients by doubly-oblivious data-structures. These data-structures protect both client interrogation and server interrogation from inferring data-dependent access patterns. X-Search [129] implements a private search engine proxy, hosted in an software attested enclave. Queries are obfuscated via a novel algorithm which packs k random past queriers, ORs them together, before submitting them to the search engine endpoint. Pires et al. [142] similarly solves private web-search through a SGX-based client browser extension.

Ahmad et al. [3] implement a data oblivious filesystem by filtering read and write operations to the host system through an ORAM protocol. A similar approach may be implemented to shield against data-dependent access for select system call translations and ephemeral storage in Diggi.

Similarly to storage, analysis of data may also expose information through data-dependent processing. VC3 Schuster et al. [157] introduce a distributed map-reduce system for trusted analytics in the cloud using SGX. VC3 runs on unmodified Hadoop, however ensures that it, the operating system and hypervisor is held outside of the TCB. Zheng et al. [195] implement a distributed oblivious analytics platform based on Apache Spark SQL. Opaque uses SGX as an optimized memory cache to speed up oblivious memory operations. A trade-of between generic implementation and performance leads Opaque to implement oblivious relational queries through modifications to the Catalyst query optimizer. Several distributed oblivious relational operations are proposed, solving a broad category of filtering and data aggregation tasks.

Ohrimenko et al. [132] implement secure multi-party machine learning. By storing datasets in enclaves, two untrusted parties may share data for a joint machine learning task without sharing source data. Trained models may then be downloaded for use into SGX enclaves. To avoid side-channels they introduce *data obliviousness* which constructs data independent machine learning algorithms for SVM, matrix factorization, neural networks, decision trees and k-means clustering. We consider this work complementary to Diggi, and other prior work have demonstrated the practical application of effective

general countermeasures against memory dependent side-channel attacks[164]. This work does not detail the differential privacy concerns of the joint trained model nor the revocation procedure for guaranteed erasure once learning is complete. Küçük et al. [109] apply this concept similarly by utilizing SGX to solve privacy preserving multi-party energy metering.

Statistical inference on sensitive datasets will inherently contain traces of the original data. Differential privacy is therefore very hard to ensure in trained models. Each query towards a private model will leak some small delta of information. Song et al. [167] evaluate the concept of "memorizing" a trained model, through blackbox access, and demonstrate that the resulting model can again demonstrate high predictive power. Mo et al. [128] similarly recognize this problem and evaluate a framework for protecting the privacy of trained Deep Neural Network (DNN)-models on TrustZone-capable edge devices.

8.4 Trusted runtimes in TEEs

The Diggi trusted runtime is designed from the ground up to support efficient execution of cloud functions in SGX with a small TCB. Additionally, Diggi implements support for legacy features by allowing selective implementation of external system translators. Table 8.1 compares the TCB of Diggi as measured by SLOC[1] to that of noteworthy similar trusted runtimes in SGX.

Baumann et al. [22] were the first to explore legacy applications executing inside SGX. By modifying a library OS implementation of Windows 8 [144], Haven is able to host unmodified applications entirely inside an SGX enclave. The Haven TCB comprises some 5 million SLOC executing inside the enclave, exposing a considerable attack surface. Haven implements block encryption of IO, and stores nonces for replay protection separately, using Merkle-trees for integrity-protection of data. This work predates the general availability of SGX and all experimental evaluations are therefore conducted on a proprietary emulator provided by Intel.

Ryoan [85] implements a distributed sandbox facilitating untrusted computing on secret data residing on third-party cloud services. Ryoan proposes a new request oriented data-model where processing modules are activated once without persisting data input to them. Ryoan creates a shielding construct supporting mutual distrust between the application and the infrastructure by combining sandboxing techniques with SGX. Through remote attestation, Ryoan is able to verify the integrity of sandbox instances and protect execution. As Haven, Ryoan predates availability of SGX, and large parts of its evaluation is conducted in an SGX emulator based on QEMU. A similar effort by Goltzsche et al. [69] implements this concept in a 2-way sandbox to solve

1. https://dwheeler.com/sloccount/

resource accounting between two mutually distrusting parties.

Arnautov et al. [14] design a SGX-capable container runtime by investigating the tradeoff between multiple designs supporting the execution of unmodified legacy applications inside of SGX. Three different design configurations for the partitioning of the application stack between the trusted and untrusted environment are evaluated; a library OS, a standard application library, and a minimal stub interface. Each choice holds several performance and security tradeoffs. More importantly, this work demonstrates the performance gain of implementing exit-less user-level threading and system calls, an approach which has inspired the design of Diggi. Vaucher et al. [181] further expand on SGX-capable containers and describe the design and implementation of support inside the Kubernetes container orchestrator.

Tsai et al. [177] implement support for unmodified applications executing in SGX by modifying the Graphene library OS [176]. This work includes support for multiprocessing applications using system features such as process fork and IPC. Eleos [138] improve upon this work to implement exit-less system call operations and enclave controlled virtual memory management. This does however require some modifications to existing applications as Eleos uses software address translation via a C++ template pointer class for memory references.

Tian et al. [173] implement a custom library OS kernel similar to Haven and Graphene. SGXKernel uses techniques for asynchronous system calls adopted from the work in SCONE. The authors argue that enclave managed system calls replicate functionality present in commodity operating systems and advocate only presenting a minimal stub interface to enclave-hosted legacy applications. SGXKernel reports that its secure runtime comprises only 7K lines of code excluding the 80K MUSL standard library. However, this work does not describe how and if system-call integrity is preserved via encryption or replay prevention. Due to these shortcomings, we do not list them as a viable candidate for comparison in Table 8.1, as the threat model is potentially different.

Lind et al. [116] implement automatic application partitioning using static data-flow analysis based on developer annotated source code. Sensitive components are automatically placed inside the enclave, and transitions are signed and encrypted. Analysis may move components into enclaves given evidence of a potential performance improvement, however does not implement any optimizations i.e. exit-less operations. Glamdring has a much lower TCB than comparative solutions.

Panoply [165] focuses on reducing of TCB rather than performance, and much like SCONE, Haven, and Graphene, enables execution of unmodified application binaries. Panoply uses a minimal shim for trusted mediation of system services implementing explicit enclave exit operations. Panoply does not implement IO encryption nor detail how replay attacks may be prevented.

Goltzsche et al. [71] implement trusted execution of JavaScript through TrustJS. TrustJS may be integrated into a commodity browser, providing servers

Secure runtime	Haven	Panoply	Graphene	SCONE	Google V8	Ductape	Diggi
SLOC	5 000K	20K	53K	97K	17 000K	185K	18K

Table 8.1: The SLOC of Diggi compared with similar work implementing secure runtimes in SGX. SLOC is a commonly used metric to measure the TCB of a software system, indicating the implementation complexity and the circumference of assumed trust. Measurements for SCONE, Graphene, Panoply, Ducktape and Diggi exclude the Intel provided SGX SDK code as well as stock implementations of standard libraries (glibc/musl). Depending on use, we expect this to add roughly 100K SLOC. As an exception, Haven depends on the Drawbridge library OS, and is likely not using gLibc.

with an attested client-side component for storing/processing.

8.5 Distributed Systems and Coordination

Remote attestation of processes simplify fault tolerance and distributed consensus, by assuming that coordinated entities are benign once attested.

Behl et al. [23] present a hybrid state-machine replication protocol which uses a trusted subsystem to assume a crash-stop fault model. This reduces the overhead of solving Byzantine Fault Tolerance (BFT) in scalable distributed systems. Li et al. [115] enable the creation of decoupled BFT clients through trusted proxies situated on the server. Clients may transparently access a BFT system through the proxy, where traditional client code is executed on the server side. The benefit of this is easy upgradable consensus protocols, where clients are able to access replicas regardless of protocol version. Bloxy [152] expands on this concept and applies them to blockchain protocols.

Teechain [117] implements efficient off-blockchain transactions using TEEs as a secure *treasury* for handling payment and settlement off-blockchain. Treasuries are replicated in a committee and vote on settlements onto the blockchain, asynchronously from the side-chain.

Lightweight client verification of blockchain payments may leak information about the client transactions. Matetic et al. [120] leverage TEEs to protect full node verification on behalf of a lightweight client, while also addressing inherent side-channels exposed by the protocol.

Kim et al. [102] describe how the security and privacy of Tor anonymous routing network can be enhanced by using TEEs. SGX-Tor reduces the possibility for an adversary to inspect or modify the software state of a Tor node. Additionally, the reduced trust model increases the potential to scale the Tor network without implicitly trusting nodes.

Zhang et al. [194] introduce an authenticated data feed, Town Crier, to consume trusted data sources in execution of autonomous smart contracts on blockchains. TC acts as a mediation layer and backend store, between a smart

contract provider blockchain, and existing HTTPS enabled websites.

Brenner et al. [31] implement SecureKeeper, a secure Zookeeper variant in SGX, which preserve the confidentiality and integrity of potentially sensitive cloud configuration data.

In-network computing is proliferating in cloud infrastructures due to the influx of capable routing hardware. Software Defined Networking (SDN) and Network Function Virtualization (NFV) enable flexible backbone architectures, but inspecting real-time traffic to make smart routing decisions comes at the risk of privacy. Diggi shares some architectural traits with other persistent distributed secure constructs such as network functions (NF). Poddar et al. [143] use trusted hardware(SGX) to implement shielded NFs which only expose encrypted traffic to the cloud provider and preserves the integrity of NFs. Trach et al. [175] implement a similar concept through *Secure Middleboxes* and Goltzsche et al. [70] propose outsourcing Middleboxes to network clients through trusted computing.

Google Asylo[2] and the Open Enclave Project[3] implement trusted open source frameworks for developing distributed TEE applications, decoupled from the hardware mechanisms implementing shielded execution.

Brenner and Kapitza [30] implement a secure runtime for JavaScript FaaS hosting, based on the minimal footprint Ducktape JavaScript engine, comprising an order of magnitude more code than Diggi. They also evaluate the Google V8 engine, which comprises some 17 million lines of code. Both use webpack to allow dependency bundling of legacy applications by downloading all dependent JavaScript code from the web. Code is read from untrusted storage into the secure context, leaving the system susceptible to malicious code execution. The benefit of measuring the initial enclave state using this approach is also greatly reduced as enclaves are generic interpreters/runtimes which allow arbitrary code to be executed without evidence in the initial attestation process. Moreover, all cohosted functions share a single secure runtime environment, and although interpreted, are subject to bugs which may cause information leakage. For compatibility reasons, the Google V8 engine further requires all enclave pages marked as writable, which reduces robustness in the presence of bugs. Qiang et al. [147] implement a secure serverless runtime built on top of the OpenLambda project, however, functions in OpenLambda are similarly implemented in JavaScript and the runtime requires a significant amount of code executing within the secure TEE context.

Trach et al. [174] mediate cold-start latencies in an SGX-based FaaS by using the second generation of SGX capable hardware, introduced in desktop CPUs along with the 2019 Ice Lake generation of Intels Core architecture. By using the new dynamic memory allocation feature [125], enclaves are able to load and unload EPC pages on demand and in batches. Pages are loaded upon

2. https://github.com/google/asylo
3. https://openenclave.io/

access, reducing the startup-time for each function significantly. Similarly to Diggi, Clemmy implements a novel messaging format for ensuring confidential and integrity preserving communication across multiple functions. Clemmy is implemented as a modification to the ApacheWhisk open source FAAS framework, using the SCONE POSIX runtime described above, where SGX based functions are implemented on top of a Python interpreter. Additionally, Clemmy is able to verify execution order among multiple functions. Diggi is similarly able to verify function execution order, and unlike Clemmy, function internal execution ordering by message logging.

EActors [155] implements a related persistent distributed trusted computing paradigm. Similar to Diggi, the EActors framework creates a simplified concurrent abstraction for composing applications consisting of multiple enclaves together. EActors are similarly focused on asynchrony (non-blocking) execution, a small memory footprint, and a low TCB.

8.6 Summary

Our thesis is situated among a massive corpus of similar research into trusted computing in an untrusted cloud. Concurrent efforts have inspired and validated our work, and additionally pinpoints the unique concepts and contributions which separate Diggi from the rest.

Several works implement trusted efficient runtimes, and some even in the context of serverless systems. However, to the best of the authors knowledge, Diggi is the only trusted serverless runtime supporting asynchronous flow-based persistent cloud functions.

Techniques for optimization such as exit-less communication and memory offloading via ephemeral storage are concepts which several related works use. Systems for logging state mutations for verifiability have been implemented in several domains, both for accountability and debugging. Trusted systems use hardware support to create tamperproof logging of system metadata for use in forensics. Diggi is, to the best of the authors knowledge, the only generic POSIX-enabled trusted runtime capable of accountable execution using log record and replay.

The next chapter will conclude this thesis answering our initial conjecture, and additionally, highlight opportunities for further research in the context of Diggi.

9

Concluding Remarks

This thesis presents Diggi; a trusted runtime system for implementing secure accountable serverless applications on top of an inherently untrusted public cloud. Our initial thesis aims to test the following conjecture:

> ***TEEs can be leveraged to build a secure and efficient serverless application runtime for trusted computing in a public cloud.***

The security model presented in Section 3.1.1, the opportunities and challenges for serverless systems presented in 2.5, and the performance principles derived in Section 4.2 resulted in a set of requirements for the design of a trusted efficient and accountable serverless runtime. Based on these requirements, we subsequently design, implement and evaluate a prototype of the Diggi runtime.

We conclude this thesis by discussing the experimental results from the previous section and seek to answer the research questions stated conforming to the design requirements and our initial conjecture. Additionally, we address some of the limitations of Diggi and opportunities for further research.

9.1 Conclusion

In the previous chapter we initially stated four research questions which our experimental evaluation seeks to answer. Moreover, the design and implemen-

tation of Diggi is based on a set of functional and non-functional requirements stated in section 4.3.

We first list each *research question* individually and argue that the evaluation has provided sufficient evidence to corroborate an affirmative answer.

Research Question 1: Section 7.4.1 illustrates the ability for the Diggi runtime implementation to host legacy libraries without modification in the trusted runtime. Applications must implement translators of system interaction befitting their application, and we demonstrate that doing so for an example library illustrating that a high degree of system interaction is plausible and practical. Additionally, Section 7.6 illustrates an end-to-end use case demonstrating that rich applications may be securely developed on top of the Diggi runtime. The MNIST handwritten digit dataset illustrates a real-world use case implementing secure and trusted distributed inference pipeline on an untrusted underlying infrastructure.

Research Question 2: All experiments compare the result of Diggi performance against a baseline. Communication overhead adds 9 percent latency overhead compared to the iperf baseline. System call translations add between 5-10x of overhead compared to direct interaction with the system call interface, and message logging adds 42 percent to benchmark runtime to achieve accountable cloud functions. The relative overhead of Diggi is in tune with similar work [14], and the MINIST dataset and TPC-C benchmark demonstrate the application of pseudo-real workloads. Regardless of these results, any applied use of Diggi should carefully analyze the tradeoff between security and performance.

Research Question 3: Section 7.2 demonstrates the potential for cohosting mutually distrusting could functions in distinct enclaves. Despite multiplexing physical hardware resources, 24 echo server cloud functions are able to communicate across the network before the load function experiences any significant latency overhead. Section 7.2.1 demonstrates that the responsiveness of deploying cloud function as the number of concurrent functions increase, achieves an acceptable average latency of 150 ms for 100 cohosted cloud functions. The echo function demonstrates the simplest logic for a Diggi cloud function, consuming roughly 2.2MB of memory. Ephemeral storage ensures that this figure does not grow significantly nor impact the cohosting potential. Each cloud function may explicitly manage application state securely in untrusted memory.

Research Question 4: Section 7.5 demonstrates the overhead of recording all state mutations for a cloud function into a tamperproof encrypted message log. Compared to the baseline, the impact on performance is significant. Al-

ternatively, the cloud function may store a dynamic proof of execution, which does not impact performance significantly. However, the persisted evidence may then only be used to verify deterministic cloud functions.

Non-Functional Requirements Additionally, the experiments demonstrate that the following non-functional requirements are met:

- **Practicality**: We demonstrate that existing systems and full-fledged distributed serverless applications are hostable in Diggi.
- **Granularity**: Composite applications may define the appropriate distributed security context through attestation groups.
- **Efficiency**: Exit-less system call translations demonstrate an increase in performance compared to conventional ocalls. Moreover, reduced memory consumption through Diggi ephemeral storage increases the cohosting potential.
- **Reducing Trust**: Bespoke system interaction through explicitly implemented translator/server pairs reduces the need for broadly implemented system compatibility which would increase the attack surface and TCB.

Functional Requirements Argumentatively, the design and implementation of Diggi satisfies the following functional requirements:

- **Shielded**: Intel SGX is used to develop a confidential and integrity protected runtime capable of hosting trusted computations and data in an untrusted environment. The Diggi runtime and ephemeral storage extends the SGX security model to implement shielded data storage without impacting cohosting potential.
- **Authentication**: Collections of Diggi cloud functions may be jointly authenticated and identified through software attestation. We extend the Intel provided software attestation protocol for multiparty attestation, to support *attestation groups*.
- **Revocation**: All information in the Diggi runtime, including storage, is encrypted using ephemeral keys derived from the SGX platform. In the presence of hardware failure, these keys are discarded by the SGX implementation, rendering persisted state useless. Additionally, given a known attested trusted runtime, we may remotely attest the ability for the system to use these ephemeral keys correctly and guarantee revocation.
- **Accountability**: By recording all inbound and outbound messages we capture the set of state mutations which define a given cloud function execution. All interactions with the external system, including temporal or random events are recorded through the message interface. A reference implementation is able to decrypt and replay the interactions, and verifiably decide if the cloud function executed correctly.

- **Persistence**: Streaming services and reactive real-time analytics processing services are examples of a class of systems which require a weaker form of persist-able state; defined as *ephemeral state*. Diggi ephemeral storage, together with persistent message endpoints which enable repeated interaction between unique cloud function instances, implement persistent cloud functions.

We conclude that the initial thesis conjecture may be confirmed, based on the outcome of the experimental evaluation and satisfaction of the functional and non-functional requirements.

TEEs may be leveraged to build a secure and efficient serverless application runtime for trusted computing in a public cloud.

Through the Diggi runtime, applications are protected from a potentially malicious service environment, all while maintaining acceptable performance. Additionally, distributed applications may identify themselves remotely through software co-attestation, and be held accountable through verifiable execution. The code implementing the Diggi prototype runtime is available as open source on GitHub[1].

9.2 Future Work

Serverless computing is at its core defined by precision in attribution, holding a linear relationship between consumption and cost. Attribution of usage in a cloud-driven revenue model, requires precisely measuring resource units. Diggi implements system services, such as memory, storage and compute through message-passing, meaning attribution of cost may be bound to delivery of messages to the cloud function rather than a binary metric.

Host infrastructure may differentiate usage based on the resource consumed, scaling out hardware components individually similar to Kvalnes et al. [111]. For persistent cloud functions, attributing cost per invocation will underestimate computational expenses, because of the potential longevity. Multiple models for logging attribution require joint trust in the surrounding system. Intuitively, it is the cloud providers responsibility to correctly attribute cost. The consumer must then trust that resource usage is not over-stipulated. Inversely, if the consumer is responsible for measurement, more granularity may be possible by subdividing recorded consumption, shielded from the untrusted cloud provider. An *attribution arbiter*, which both cloud and consumer agree is trustworthy by attestation, may act on behalf of both to record usage correctly. This concept is similar to the work presented by Goltzsche et al. [69].

Diggi implements a runtime prototype component of a traditional serverless

1. github.com/andersgjerdrum/diggi

system. A fully realized persistent cloud function framework requires managing non-idempotency in execution. Conventional systems use reliable message queues to implement at-least-once semantics, however Diggi persistent cloud functions may have side-effects which require exactly-once execution.

Since the work on this thesis began, Intel has released a modified remote attestation service, enabling developer provided attestation evidence. Additionally, *Open Enclaves* attempt to create an open standard for enclave development creating a hardware agnostic platform for developing trusted applications. Moreover, cloud providers are beginning to support several types of trusted computing services [90]. Future work for Diggi involves supporting multiple types of trusted hardware, decoupling the dependency on a particular TEE and associated attestation services.

Diggi ephemeral storage is built on the realization that memory management in SGX is costly. With the release of dynamic memory management for SGXV2, enclaves may scale memory usage beyond the initial reserved. We still expect an overhead associated with enclave memory operations, however the impact brought on by this new memory model is unknown. Additionally, enclaves which are able to add EPC pages at runtime must be accounted for in a revisited state mutation recorder.

In the event of an audit, accountable cloud functions provide evidence to the auditor, which replays the account against a reference implementation. However, this does not solve non-repudiation in the event of an unavailable system. Future work should explore how non-repudiation can be achieved by logging results to a BFT ledger.

Operating systems, firmware, and all multi-user software such as databases, web-servers and message queues are subject to interference by the host system. Both software and hardware construct share caches, registries, pipelines, ALU and peripheral storage bus interfaces, to name a few. Hunt et al. [84] discuss the challenges of a shared system architecture in pursuit of performance versus non-interference. Future work should investigate a reference design for solving trusted shared-nothing runtimes in an untrusted cloud.

For resource isolation, modern operating systems are capable of isolating resources with high precision using features such as c groups or process groups, additionally used in container technology. In its current form, Diggi is unable to implement fairness among multiple cohosted cloud functions once threading resources are oversubscribed. An architecture which allow enclave-managed interrupt processing may enable fair scheduling of resources among mutually distrusting participants.

Application-Specific Integrated Circuits (ASICs) such as GPU, TPU and FPGAs demonstrate a considerable advantages for highly data-parallel workloads such as convolutional neural networks, compared to conventional CPU architectures (SISD). A high-performance data-parallel trusted processing system should investigate how to protect analysis of data by multiple mutually distrusting applications on shared ASIC-hardware.

The diggi serverless runtime is realized through a limited prototype implementation which demonstrate selective applicability for hosting sensitive data in an untrusted cloud. Future research should evaluate the broader applicability of Diggi for developing trusted distributed systems.

Bibliography

[1] *AES-CBC + Elephant diffuser A Disk Encryption Algorithm for Windows Vista*. `https://download.microsoft.com/download/0/2/3/0238acaf-d3bf-4a6d-b3d6-0a0be4bbb36e/BitLockerCipher200608.pdf`. Accessed: 2020-04-29.

[2] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. "Firecracker: Lightweight Virtualization for Serverless Applications." In: *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. 2020, pp. 419–434.

[3] Adil Ahmad, Kyungtae Kim, Muhammad Ihsanulhaq Sarfaraz, and Byoungyoung Lee. "OBLIVIATE: A Data Oblivious Filesystem for Intel SGX." In: *Proceedings of the Symposium on Network and Distributed System Security (NDSS)*. NDSS, 2018.

[4] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. "SAND: Towards High-Performance Serverless Computing." In: *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX, 2018, pp. 923–935.

[5] Allan J. Albrecht and John E Gaffney. "Software function, source lines of code, and development effort prediction: a software science validation." In: 6. IEEE, 1983, pp. 639–648.

[6] Fritz Alder, Arseny Kurnikov, Andrew Paverd, and N Asokan. "Migrating SGX enclaves with persistent state." In: *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2018, pp. 195–206.

[7] Kalev Alpernas, Cormac Flanagan, Sadjad Fouladi, Leonid Ryzhyk, Mooly Sagiv, Thomas Schmitz, and Keith Winstein. "Secure serverless computing using dynamic information flow control." In: 2018.

[8] Lorenzo Alvisi and Keith Marzullo. "Message logging: Pessimistic, optimistic, causal, and optimal." In: vol. 24. 2. IEEE, 1998, pp. 149–159.

[9] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. "Innovative technology for CPU based attestation and sealing." In: *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*. Vol. 13. ACM, 2013.

[10] Lixiang Ao, Liz Izhikevich, Geoffrey M Voelker, and George Porter. "Sprocket: A serverless video processing framework." In: *Proceedings of the ACM Symposium on Cloud Computing*. ACM, 2018, pp. 263–274.

[11] *Apache Thrift*. `https://thrift.apache.org`. Accessed: 2020-06-17.

[12] Arvind Arasu, Spyros Blanas, Ken Eguro, Raghav Kaushik, Donald Kossmann, Ravishankar Ramamurthy, and Ramarathnam Venkatesan. "Orthogonal Security with Cipherbase." In: *6th Biennial Conference on Innovative Data Systems Research (CIDR 13)*. Citeseer. Microsoft Research, Jan. 2013.

[13] *Arm TrustZone Technology*. `https://developer.arm.com/ip-products/security-ip/trustzone`. Accessed: 2020-03-17.

[14] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, et al. "SCONE: Secure Linux Containers with Intel SGX." In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX, 2016, pp. 689–703.

[15] Pierre-Louis Aublin, Florian Kelbert, Dan O'Keeffe, Divya Muthukumaran, Christian Priebe, Joshua Lind, Robert Krahn, Christof Fetzer, David Eyers, and Peter Pietzuch. "LibSEAL: revealing service integrity violations using trusted execution." In: *Proceedings of the 13th European Conference on Computer Systems*. ACM, 2018, pp. 1–15.

[16] *AWS Lambda*. `https://aws.amazon.com/lambda/`. Accessed: 2020-03-09.

[17] *Azure Functions documentation*. `https://docs.microsoft.com/en-us/azure/azure-functions/`. Accessed: 2020-03-12.

[18] Engineer Bainomugisha, Andoni Lombide Carreton, Tom van Cutsem, Stijn Mostinckx, and Wolfgang de Meuter. "A survey on reactive programming." In: vol. 45. 4. ACM, 2013, pp. 1–34.

[19] Sumeet Bajaj and Radu Sion. "TrustedDB: A trusted hardware-based database with privacy and data confidentiality." In: vol. 26. 3. IEEE, 2014, pp. 752–765.

[20] Ioana Baldini, Paul Castro, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, and Philippe Suter. "Cloud-native, event-based programming for mobile applications." In: *Proceedings of the International Conference on Mobile Software Engineering and Systems*. ACM, 2016, pp. 287–288.

[21] Ioana Baldini, Paul Castro, Kerry Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Aleksander Slominski, et al. "Serverless computing: Current trends and open problems." In: *Research Advances in Cloud Computing*. Springer, 2017, pp. 1–20.

[22] Andrew Baumann, Marcus Peinado, and Galen Hunt. "Shielding applications from an untrusted cloud with Haven." In: vol. 33. 3. ACM, Aug. 2015, 8:1–8:26.

[23] Johannes Behl, Tobias Distler, and Rüdiger Kapitza. "Hybrids on steroids: SGX-based high performance BFT." In: *Proceedings of the Twelfth European Conference on Computer Systems*. ACM, 2017, pp. 222–237.

[24] Sumeer Bhola, Robert Strom, Saurabh Bagchi, Yuanyuan Zhao, and Joshua Auerbach. "Exactly-once delivery in a content-based publish-subscribe system." In: *Proceedings International Conference on Dependable Systems and Networks*. IEEE, 2002, pp. 7–16.

[25] Eleanor Birrell, Anders Gjerdrum, Robbert van Renesse, Håvard Johansen, Dag Johansen, and Fred B. Schneider. "SGX Enforcement of Use-Based Privacy." In: *Proceedings of the 2018 Workshop on Privacy in the Electronic Society*. WPES 18. Toronto, Canada: ACM, 2018, pp. 155–167.

[26] Marcus Brandenburger, Christian Cachin, Matthias Lorenz, and Rüdiger Kapitza. "Rollback and forking detection for trusted execution environments using lightweight collective memory." In: *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2017, pp. 157–168.

[27] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. "Software grand exposure: SGX cache attacks are practical." In: *11th USENIX Workshop on Offensive Technologies (WOOT 17)*. USENIX, 2017.

[28] Lars Brenna, Alan Demers, Johannes Gehrke, Mingsheng Hong, Joel Ossher, Biswanath Panda, Mirek Riedewald, Mohit Thatte, and Walker White. "Cayuga: a high-performance event processing engine." In: *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*. ACM, 2007, pp. 1100–1102.

[29] Lars Brenna, Johannes Gehrke, Mingsheng Hong, and Dag Johansen. "Distributed event stream processing with non-deterministic finite automata." In: *Proceedings of the 3rd ACM International Conference on Distributed Event-Based Systems*. ACM, 2009, pp. 1–12.

[30] Stefan Brenner and Rüdiger Kapitza. "Trust More, Serverless." In: *Proceedings of the 12th ACM International Conference on Systems and Storage*. SYSTOR 19. Haifa, Israel: ACM, 2019, pp. 33–43.

[31] Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Christof Fetzer, Peter Pietzuch, and Rüdiger Kapitza. "Securekeeper: confidential zookeeper using intel SGX." In: *Proceedings of the 17th International Middleware Conference*. ACM, 2016, p. 14.

[32] Navin Budhiraja, Keith Marzullo, Fred B Schneider, and Sam Toueg. "The primary-backup approach." In: vol. 2. 1993, pp. 199–216.

[33] *Bugzilla, Kernel.org*. `https://bugzilla.kernel.org/`. Accessed: 2020-03-15.

[34] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. "Borg, omega, and kubernetes." In: vol. 14. 1. ACM, 2016, pp. 70–93.

[35] Miguel Castro, Dushyanth Narayanan, Aleksandar Dragojevic, Matthew Renzelmann, Alexander Shamis, Richendra Khanna, Stanko Novakovic, Anders Gjerdrum, and Georgios Chatzopoulos. *Clock synchronization*. US Patent App. 15/933,214. Sept. 2019.

[36] Miguel Castro, Dushyanth Narayanan, Aleksandar Dragojevic, Matthew James Renzelmann, Alexander Shamis, Richendra Khanna, Stanko Novakovic, Anders Gjerdrum, and Georgios Chatzopoulos. *Performing transactions in distributed transactional memory systems*. US Patent App. 15/933,230. Sept. 2019.

[37] Paul Castro, Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. "The Rise of Serverless Computing." In: vol. 62. 12. ACM, Nov. 2019, 44–54.

[38] Stephen Checkoway and Hovav Shacham. "Iago attacks: why the system call API is a bad untrusted RPC interface." In: vol. 41. 1. ACM, 2013, pp. 253–264.

[39] Guoxing Chen, Yinqian Zhang, and Ten-Hwang Lai. "OPERA: Open Remote Attestation for Intel's Secure Enclaves." In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2019, pp. 2317–2331.

[40] Xiaoxin Chen, Tal Garfinkel, E Christopher Lewis, Pratap Subrahmanyam, Carl A Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan RK Ports. "Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems." In: vol. 42. 2. ACM, 2008, pp. 2–13.

[41] Zhiqun Chen. *Java card technology for smart cards: architecture and programmer's guide*. Addison-Wesley Professional, 2000.

[42] *Cloud Functions*. `https://cloud.google.com/functions`. Accessed: 2020-03-12.

[43] D. E. Comer, David Gries, Michael C. Mulder, Allen Tucker, A. Joe Turner, and Paul R. Young. "Computing As a Discipline." In: ed. by Peter J. Denning. Vol. 32. 1. ACM, Jan. 1989, pp. 9–23.

[44] *Common Language Runtime (CLR) overview*. `https://docs.microsoft.com/en-us/dotnet/standard/clr`. Accessed: 2020-03-10.

[45] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. "Spanner: Google's globally distributed database." In: vol. 31. 3. ACM, 2013, p. 8.

[46] Victor Costan and Srinivas Devadas. "Intel SGX Explained." In: vol. 2016. 086. 2016, pp. 1–118.

[47] Victor Costan, Ilia Lebedev, and Srinivas Devadas. "Sanctum: Minimal hardware extensions for strong software isolation." In: *25th USENIX Security Symposium (USENIX Security 16)*. USENIX, 2016, pp. 857–874.

[48] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters." In: vol. 51. 1. ACM, 2008, pp. 107–113.

[49] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. "Dynamo: amazon's highly available key-value store." In: *ACM SIGOPS Operating Systems Review*. Vol. 41. 6. ACM, 2007, pp. 205–220.

[50] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. "Imagenet: A large-scale hierarchical image database." In: *2009 IEEE Conference on Computer Vision and Pattern Recognition*. Ieee. IEEE, 2009, pp. 248–255.

[51] Peter J. Denning. "Computing is a Natural Science." In: vol. 50. 7. ACM, July 2007, pp. 13–18.

[52] Whitfield Diffie and Martin Hellman. "New directions in cryptography." In: vol. 22. 6. IEEE, 1976, pp. 644–654.

[53] Judicael B Djoko, Jack Lange, and Adam J Lee. "NEXUS: Practical and Secure Access Control on Untrusted Storage Platforms using Client-side SGX." In: *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2019, pp. 401–413.

[54] *Docker Swarm*. `https://docs.docker.com/engine/swarm`. Accessed: 2020-06-17.

[55] Kevin Elphinstone and Gernot Heiser. "From L3 to seL4 what have we learnt in 20 years of L4 microkernels?" In: *Proceedings of the 24th ACM Symposium on Operating Systems Principles*. ACM, 2013, pp. 133–150.

[56] Saba Eskandarian and Matei Zaharia. "ObliDB: oblivious query processing for secure databases." In: vol. 13. 2. VLDB Endowment, 2019, pp. 169–183.

[57] Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. "Komodo: Using verification to disentangle secure-enclave hardware from software." In: *Proceedings of the 26th ACM Symposium on Operating Systems Principles*. ACM, 2017, pp. 287–305.

[58] *Fn Project*. `https://fnproject.io/`. Accessed: 2020-03-14.

[59] Armando Fox, Rean Griffith, Anthony Joseph, Randy Katz, Andrew Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, and Ion Stoica. "Above the clouds: A berkeley view of cloud computing." In: vol. 28. 13. 2009, p. 2009.

[60] *General Data Protection Regulation*. `https://gdpr-info.eu/`. Accessed: 2020-03-17.

[61] Adrien Ghosn, James R Larus, and Edouard Bugnion. "Secured routines: language-based construction of trusted execution environments." In: *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX, 2019, pp. 571–586.

[62] Anders T Gjerdrum, Robert Pettersen, Håvard D Johansen, and Dag Johansen. "Cloud Computing and Service Science: 7th International Conference, CLOSER 2017, Porto, Portugal, April 24–26, 2017, Revised Selected Papers." In: vol. 864. Springer, 2018, pp. 1–18.

[63] Anders T Gjerdrum, Håvard D Johansen, Lars Brenna, and Dag Johansen. "Diggi: A Secure Framework for Hosting Native Cloud Functions with Minimal Trust." In: *The 1st IEEE International Conference on Trust, Privacy and Security in Intelligent Systems, and Applications (TPS)*. IEEE, 2019.

[64] Anders T Gjerdrum, Håvard D Johansen, and Dag Johansen. "Implementing informed consent as information-flow policies for secure analytics on ehealth data: Principles and practices." In: *2016 IEEE First International Conference on Connected Health: Applications, Systems and Engineering Technologies (CHASE)*. IEEE, 2016, pp. 107–112.

[65] Anders T Gjerdrum, Robert Pettersen, Håvard D Johansen, and Dag Johansen. "Performance of Trusted Computing in Cloud Infrastructures with Intel SGX." In: *CLOSER*. SCITEPRESS, 2017, pp. 668–675.

[66] *GKE Sandbox: Bring defense in depth to your pods*. `https://cloud.google.com/blog/products/containers-kubernetes/gke-sandbox-bring-defense-in-depth-to-your-pods`. Accessed: 2020-030-14.

[67] *Global Platform Specification Library*. `https://globalplatform.org/specs-library/?filter-committee=tee`. Accessed: 2020-03-17.

[68] Peter Godfrey-Smith. *Theory and reality: An introduction to the philosophy of science*. University of Chicago Press, 2009.

[69] David Goltzsche, Manuel Nieke, Thomas Knauth, and Rüdiger Kapitza. "AccTEE: A WebAssembly-based Two-way Sandbox for Trusted Resource Accounting." In: *Proceedings of the 20th International Middleware Conference*. ACM, 2019, pp. 123–135.

[70] David Goltzsche, Signe Rüsch, Manuel Nieke, Sébastien Vaucher, Nico Weichbrodt, Valerio Schiavoni, Pierre-Louis Aublin, Paolo Cosa, Christof Fetzer, Pascal Felber, et al. "Endbox: Scalable middlebox functions using client-side trusted execution." In: *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2018, pp. 386–397.

[71] David Goltzsche, Colin Wulf, Divya Muthukumaran, Konrad Rieck, Peter Pietzuch, and Rüdiger Kapitza. "Trustjs: Trusted client-side execution of javascript." In: *Proceedings of the 10th European Workshop on Systems Security*. ACM, 2017, pp. 1–6.

[72] *Google App Engine*. `https://cloud.google.com/appengine`. Accessed: 2020-03-11.

[73] *Google RPC*. `https://grpc.io`. Accessed: 2020-06-17.

[74] *Google V8 Engine*. `https://v8.dev/`. Accessed: 2020-03-10.

[75] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. "Cache attacks on Intel SGX." In: *Proceedings of the 10th European Workshop on Systems Security*. ACM, 2017, p. 2.

[76] Jinyu Gu, Zhichao Hua, Yubin Xia, Haibo Chen, Binyu Zang, Haibing Guan, and Jinming Li. "Secure live migration of SGX enclaves on un-

trusted cloud." In: *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2017, pp. 225–236.

[77] Manu Gulati, Michael J Smith, and Shu-Yi Yu. *Security enclave processor for a system on a chip*. US Patent 8,832,465. Sept. 2014.

[78] *gVisor: Container Runtime Sandbox*. `https://github.com/google/gvisor`. Accessed: 2020-03-10.

[79] Andreas Haeberlen, Petr Kouznetsov, and Peter Druschel. "PeerReview: Practical accountability for distributed systems." In: vol. 41. ACM, 2007.

[80] *Health Insurance Portability and Accountability Act of 1996*. `https://aspe.hhs.gov/report/health-insurance-portability-and-accountability-act-1996`. Accessed: 2020-03-17.

[81] Joseph M Hellerstein, Jose Faleiro, Joseph E Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. "Serverless computing: One step forward, two steps back." In: 2018.

[82] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. "Serverless computation with openlambda." In: *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*. USENIX, 2016.

[83] Helge Hoff. "SecureCached. Secure caching with the Diggi framework." MA thesis. UiT Norges arktiske universitet, 2018.

[84] Tyler Hunt, Zhipeng Jia, Vance Miller, Christopher J Rossbach, and Emmett Witchel. "Isolation and Beyond: Challenges for System Security." In: *Proceedings of the Workshop on Hot Topics in Operating Systems*. ACM, 2019, pp. 96–104.

[85] Tyler Hunt, Zhiting Zhu, Yuanzhong Xu, Simon Peter, and Emmett Witchel. "Ryoan: A distributed sandbox for untrusted computation on secret data." In: vol. 35. 4. ACM, 2018, p. 13.

[86] *IBM Cloud Functions*. `https://www.ibm.com/cloud/functions`. Accessed: 2020-03-14.

[87] *Intel Software Developer's Manual*. `https://www.intel.com/content/dam/www/public/emea/xe/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3d-part-4-manual.pdf`. Accessed: 2020-03-17.

[88] *Intel Software Guard Extensions Programming Reference*. `https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf`. Accessed: 2020-04-29.

[89] *Intel(R) Software Guard Extensions for Linux* OS, linux-sgx-driver*. `https://github.com/intel/linux-sgx-driver`. Accessed: 2020-03-17.

[90] *Introducing Azure confidential computing*. `https://azure.microsoft.com/en-us/blog/introducing-azure-confidential-computing/`. Accessed: 2020-06-17.

[91] *ISO/IEC 11889-1:2009 Information technology — Trusted Platform Module*. `https://www.iso.org/standard/50970.html`. Accessed: 2020-06-23.

[92] *ISO/IEC 7816-2:2007 [ISO/IEC 7816-2:2007]*. https://www.iso.org/standard/45989.html. Accessed: 2020-03-17.

[93] Yeongjin Jang, Jaehyuk Lee, Sangho Lee, and Taesoo Kim. "SGX-bomb: Locking down the processor via rowhammer attack." In: *Proceedings of the 2nd Workshop on System Software for Trusted Execution*. ACM, 2017, pp. 1–6.

[94] Abhinav Jangda, Donald Pinckney, Samuel Baxter, Breanna Devore-McDonald, Joseph Spitzer, Yuriy Brun, and Arjun Guha. "Formal Foundations of Serverless Computing." In: 2019.

[95] *Java Virtual Machine Technology Overview*. https://docs.oracle.com/javase/10/vm/java-virtual-machine-technology-overview.htm. Accessed: 2020-03-10.

[96] Dag Johansen, Robbert van Renesse, and Fred B Schneider. "An introduction to the TACOMA distributed system. Version 1.0." In: Universitetet i Tromsø, 1995.

[97] 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*. ACM, 2015, pp. 1–7.

[98] Håvard D Johansen, Robbert Van Renesse, Ymir Vigfusson, and Dag Johansen. "Fireflies: A secure and scalable membership and gossip service." In: vol. 33. 2. ACM, 2015, pp. 1–32.

[99] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, et al. "Cloud programming simplified: A berkeley view on serverless computing." In: 2019.

[100] Kostis Kaffes, Neeraja J Yadwadkar, and Christos Kozyrakis. "Centralized Core-granular Scheduling for Serverless Functions." In: *Proceedings of the ACM Symposium on Cloud Computing*. ACM, 2019, pp. 158–164.

[101] Vishal Karande, Erick Bauman, Zhiqiang Lin, and Latifur Khan. "SGX-log: Securing system logs with SGX." In: *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*. ACM, 2017, pp. 19–30.

[102] Seongmin Kim, Juhyeng Han, Jaehyeong Ha, Taesoo Kim, and Dongsu Han. "Enhancing security and privacy of tor's ecosystem by using trusted execution environments." In: *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX, 2017, pp. 145–161.

[103] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. "Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors." In: vol. 42. 3. ACM, 2014, pp. 361–372.

[104] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. "Spectre attacks: Exploiting speculative execution." In: *2019 IEEE Symposium on Security and Privacy*. IEEE, 2019, pp. 1–19.

[105] David Kotz, Robert Gray, Saurab Nog, Daniela Rus, Sumit Chawla, and George Cybenko. "Agent Tcl: Targeting the needs of mobile computers." In: vol. 1. 4. IEEE, 1997, pp. 58–67.

[106] Robert Krahn, Bohdan Trach, Anjo Vahldiek-Oberwagner, Thomas Knauth, Pramod Bhatotia, and Christof Fetzer. "Pesos: Policy enhanced secure object store." In: *Proceedings of the 13th European Conference on Computer Systems*. ACM, 2018, pp. 1–17.

[107] Hugo Krawczyk. "SIGMA: The 'SIGn-and-MAc'approach to authenticated Diffie-Hellman and its use in the IKE protocols." In: *Annual International Cryptology Conference*. Springer, 2003, pp. 400–425.

[108] Alexander Krizhanovsky. "Lock-free multi-producer multi-consumer queue on ring buffer." In: vol. 2013. 228. Belltown Media, 2013, p. 4.

[109] Kubilay Ahmet Küçük, Andrew Paverd, Andrew Martin, N Asokan, Andrew Simpson, and Robin Ankele. "Exploring the use of Intel SGX for secure many-party applications." In: *Proceedings of the 1st Workshop on System Software for Trusted Execution*. ACM, 2016, pp. 1–6.

[110] Dmitrii Kuvaiskii, Oleksii Oleksenko, Sergei Arnautov, Bohdan Trach, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. "SGXBOUNDS: Memory safety for shielded execution." In: *Proceedings of the Twelfth European Conference on Computer Systems*. ACM, 2017, pp. 205–221.

[111] Åge Kvalnes, Dag Johansen, Robbert van Renesse, Fred B Schneider, and Steffen Viken Valvag. "Omni-kernel: An operating system architecture for pervasive monitoring and scheduling." In: vol. 26. 10. IEEE, 2014, pp. 2849–2862.

[112] Leslie Lamport et al. "Paxos made simple." In: vol. 32. 4. ACM, 2001, pp. 18–25.

[113] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. "Gradient-based learning applied to document recognition." In: vol. 86. 11. IEEE, 1998, pp. 2278–2324.

[114] 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)*. USENIX, Aug. 2017, pp. 557–574.

[115] B. Li, N. Weichbrodt, J. Behl, P. Aublin, T. Distler, and R. Kapitza. "Troxy: Transparent Access to Byzantine Fault-Tolerant Systems." In: *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2018, pp. 59–70.

[116] Joshua Lind, Christian Priebe, Divya Muthukumaran, Dan O'Keeffe, Pierre-Louis Aublin, Florian Kelbert, Tobias Reiher, David Goltzsche, David Eyers, Rüdiger Kapitza, et al. "Glamdring: Automatic Application

Partitioning for Intel SGX." In: *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX, 2017, pp. 285–298.

[117] Joshua Lind, Oded Naor, Ittay Eyal, Florian Kelbert, Emin Gün Sirer, and Peter Pietzuch. "Teechain: a secure payment network with asynchronous blockchain access." In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. ACM, 2019, pp. 63–79.

[118] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. "Unikernels: Library operating systems for the cloud." In: vol. 41. 1. ACM, 2013, pp. 461–472.

[119] Tarjei Mandt, Mathew Solnik, and David Wang. "Demystifying the secure enclave processor." In:

[120] Sinisa Matetic, Karl Wüst, Moritz Schneider, Kari Kostiainen, Ghassan Karame, and Srdjan Capkun. "BITE: Bitcoin Lightweight Client Privacy using Trusted Execution." In: *28th USENIX Security Symposium (USENIX Security 19)*. USENIX, 2019, pp. 783–800.

[121] Sinisa Matetic, Moritz Schneider, Andrew Miller, Ari Juels, and Srdjan Capkun. "Delegatee: Brokered delegation using trusted execution environments." In: *27th USENIX Security Symposium (USENIX Security 18)*. USENIX, 2018, pp. 1387–1403.

[122] Nicholas D Matsakis and Felix S Klock II. "The rust language." In: *ACM SIGAda Ada Letters*. Vol. 34. 3. ACM, 2014, pp. 103–104.

[123] Garrett McGrath and Paul R Brenner. "Serverless computing: Design, implementation, and performance." In: *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*. IEEE, 2017, pp. 405–410.

[124] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savagaonkar. "Innovative instructions and software model for isolated execution." In: vol. 10. 1. HASP ISCA, 2013.

[125] Frank McKeen, Ilya Alexandrovich, Ittai Anati, Dror Caspi, Simon Johnson, Rebekah Leslie-Hurd, and Carlos Rozas. "Intel® software guard extensions (Intel® SGX) support for dynamic memory management inside an enclave." In: *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016*. ACM, 2016, p. 10.

[126] Dominik Meissner, Benjamin Erb, Frank Kargl, and Matthias Tichy. "Retro-λ: An Event-sourced Platform for Serverless Applications with Retroactive Computing Support." In: *Proceedings of the 12th ACM International Conference on Distributed and Event-based Systems*. ACM, 2018, pp. 76–87.

[127] Pratyush Mishra, Rishabh Poddar, Jerry Chen, Alessandro Chiesa, and Raluca Ada Popa. "Oblix: An efficient oblivious search index." In: *2018 IEEE Symposium on Security and Privacy*. IEEE, 2018, pp. 279–296.

[128] Fan Mo, Ali Shahin Shamsabadi, Kleomenis Katevas, Soteris Demetriou, Ilias Leontiadis, Andrea Cavallaro, and Hamed Haddadi. "DarkneTZ: towards model privacy at the edge using trusted execution environments." In: 2020.

[129] Sonia Ben Mokhtar, Antoine Boutet, Pascal Felber, Marcelo Pasin, Rafael Pires, and Valerio Schiavoni. "X-search: revisiting private web search using Intel SGX." In: *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*. ACM, 2017, pp. 198–208.

[130] Sam Newman. *Building microservices: designing fine-grained systems*. O'Reilly Media, Inc., 2015.

[131] Audun Nordal, Åge Kvalnes, Joseph Hurley, and Dag Johansen. "Balava: Federating private and public clouds." In: *2011 IEEE World Congress on Services*. IEEE, 2011, pp. 569–577.

[132] Olga Ohrimenko, Felix Schuster, Cedric Fournet, Aastha Mehta, Sebastian Nowozin, Kapil Vaswani, and Manuel Costa. "Oblivious Multi-Party Machine Learning on Trusted Processors." In: *25th USENIX Security Symposium (USENIX Security 16)*. USENIX, Aug. 2016, pp. 619–636.

[133] Oleksii Oleksenko, Bohdan Trach, Robert Krahn, Mark Silberstein, and Christof Fetzer. "Varys: Protecting SGX Enclaves from Practical Side-Channel Attacks." In: *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX, 2018, pp. 227–240.

[134] *OP-TEE Documentation*. `https://optee.readthedocs.io/en/latest/index.html`. Accessed: 2020-03-17.

[135] *Open Source Serverless Computing*. `https://open.iron.io/`. Accessed: 2020-03-14.

[136] Meni Orenbach, Andrew Baumann, and Mark Silberstein. "Autarky: closing controlled channels with self-paging enclaves." In: *Proceedings of the 15th European Conference on Computer Systems*. ACM, 2020, pp. 1–16.

[137] Meni Orenbach, Yan Michalevsky, Christof Fetzer, and Mark Silberstein. "CoSMIX: a compiler-based system for secure memory instrumentation and execution in enclaves." In: *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX, 2019, pp. 555–570.

[138] Meni Orenbach, Pavel Lifshits, Marina Minkin, and Mark Silberstein. "Eleos: ExitLess OS services for SGX enclaves." In: *Proceedings of the Twelfth European Conference on Computer Systems*. ACM, 2017, pp. 238–253.

[139] Riccardo Paccagnella, Pubali Datta, Wajih Ul Hassan, Adam Bates, Christopher W Fletcher, Andrew Miller, and Dave Tian. "Custos: Practical tamper-evident auditing of operating systems using trusted execution." In: *Proceedings of the Symposium on Network and Distributed System Security (NDSS)*. 2020.

[140] Robert Pettersen, Steffen Viken Valvåg, Åge Kvalnes, and Dag Johansen. "Jovaku: Globally distributed caching for cloud database services using

DNS." In: *2014 2nd IEEE International Conference on Mobile Cloud Computing, Services, and Engineering*. IEEE, 2014, pp. 127–135.

[141] Sandro Pinto and Nuno Santos. "Demystifying arm trustzone: A comprehensive survey." In: vol. 51. 6. ACM, 2019, pp. 1–36.

[142] Rafael Pires, David Goltzsche, Sonia Ben Mokhtar, Sara Bouchenak, Antoine Boutet, Pascal Felber, Rüdiger Kapitza, Marcelo Pasin, and Valerio Schiavoni. "CYCLOSA: Decentralizing private web search through SGX-based browser extensions." In: *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2018, pp. 467–477.

[143] Rishabh Poddar, Chang Lan, Raluca Ada Popa, and Sylvia Ratnasamy. "Safebricks: Shielding network functions in the cloud." In: *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX, 2018, pp. 201–216.

[144] Donald E Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C Hunt. "Rethinking the library OS from the top down." In: *Proceedings of the 16th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2011, pp. 291–304.

[145] Christian Priebe, Kapil Vaswani, and Manuel Costa. "EnclaveDB: A secure database using SGX." In: *2018 IEEE Symposium on Security and Privacy*. IEEE. IEEE, 2018, pp. 264–278.

[146] Dhathri Purohith, Jayashree Mohan, and Vijay Chidambaram. "The dangers and complexities of SQLite benchmarking." In: *Proceedings of the 8th Asia-Pacific Workshop on Systems*. ACM, 2017, p. 3.

[147] Weizhong Qiang, Zezhao Dong, and Hai Jin. "Se-Lambda: Securing Privacy-Sensitive Serverless Applications Using SGX Enclave." In: *International Conference on Security and Privacy in Communication Systems*. Springer, 2018, pp. 451–470.

[148] Robbert van Renesse, Havard Johansen, Nihar Naigaonkar, and Dag Johansen. "Secure Abstraction with Code Capabilities." In: *Proceedings of the 2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. IEEE, 2013, pp. 542–546.

[149] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. "Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds." In: *Proceedings of the 16th ACM Conference on Computer and Communications Security*. ACM, 2009, pp. 199–212.

[150] Jordan Robertson and Michael Riley. *The Big Hack: How China Used a Tiny Chip to Infiltrate U.S. Companies*. URL: https://www.bloomberg.com/news/features/2018-10-04/the-big-hack-how-china-used-a-tiny-chip-to-infiltrate-america-s-top-companies.

[151] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. *Learning internal representations by error propagation*. Tech. rep. California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

[152] Signe Rüsch, Kai Bleeke, and Rüdiger Kapitza. "BLOXY: Providing Transparent and Generic BFT-Based Ordering Services for Blockchains." In: *38th International Symposium on Reliable Distributed Systems (SRDS 2019), Lyon, Campus La Doua, Lyon, France, October 1-4, 2019*. IEEE, 2019.

[153] Nuno Santos, Himanshu Raj, Stefan Saroiu, and Alec Wolman. "Using ARM TrustZone to build a trusted language runtime for mobile applications." In: *ACM SIGARCH Computer Architecture News*. Vol. 42. 1. ACM, 2014, pp. 67–80.

[154] Vasily Sartakov, Nico Weichbrodt, Sebastian Krieter, Thomas Leich, and Rudiger Kapitza. "STANlite–a database engine for secure data processing at rack-scale level." In: *2018 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 2018, pp. 23–33.

[155] Vasily A Sartakov, Stefan Brenner, Sonia Ben Mokhtar, Sara Bouchenak, Gaël Thomas, and Rüdiger Kapitza. "EActors: Fast and flexible trusted computing using SGX." In: *Proceedings of the 19th International Middleware Conference*. ACM, 2018, pp. 187–200.

[156] Sajin Sasy, Sergey Gorbunov, and Christopher W Fletcher. "ZeroTrace: Oblivious Memory Primitives from Intel SGX." In: vol. 2017. 2017, p. 549.

[157] 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: *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 38–54.

[158] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. "Malware guard extension: Using SGX to conceal cache attacks." In: *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2017, pp. 3–24.

[159] Mark Seaborn and Thomas Dullien. "Exploiting the DRAM rowhammer bug to gain kernel privileges." In: vol. 15. UBM, 2015, p. 71.

[160] *Serverless Functions, Made Simple.* `https://www.openfaas.com/`. Accessed: 2020-03-14.

[161] Alex Shamis, Matthew Renzelmann, Stanko Novakovic, Georgios Chatzopoulos, Aleksandar Dragojević, Dushyanth Narayanan, and Miguel Castro. "Fast general distributed transactions with opacity." In: *Proceedings of the 2019 International Conference on Management of Data*. ACM, 2019, pp. 433–448.

[162] Alex Shamis, Matthew Renzelmann, Stanko Novakovic, Georgios Chatzopoulos, Anders T Gjerdrum, Dan Alistarh, Aleksandar Dragojevic, Dushyanth Narayanan, and Miguel Castro. "Fast General Distributed Transactions with Opacity using Global Time." In: 2020.

[163] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. "Edge computing: Vision and challenges." In: vol. 3. 5. IEEE, 2016, pp. 637–646.

[164] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. "T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs." In: *Proceedings of the Symposium on Network and Distributed System Security (NDSS)*. 2017.

[165] Shweta Shinde, Dat Le Tien, Shruti Tople, and Prateek Saxena. "Panoply: Low-TCB Linux Applications With SGX Enclaves." In: *Proceedings of the Symposium on Network and Distributed System Security (NDSS)*. NDSS, 2017.

[166] Rohit Sinha, Sriram Rajamani, Sanjit Seshia, and Kapil Vaswani. "Moat: Verifying Confidentiality of Enclave Programs." In: *Proceedings of the 2015 ACM SIGSAC Conference on Computer and Communications Security*. CCS '15. Denver, Colorado, USA: ACM, 2015, pp. 1169–1184.

[167] Congzheng Song, Thomas Ristenpart, and Vitaly Shmatikov. "Machine learning models that remember too much." In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017, pp. 587–601.

[168] Emil Stefanov, Marten Van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. "Path ORAM: an extremely simple oblivious RAM protocol." In: *Proceedings of the 2013 ACM SIGSAC Conference on Computer & communications security*. ACM, 2013, pp. 299–310.

[169] Pramod Subramanyan, Rohit Sinha, Ilia Lebedev, Srinivas Devadas, and Sanjit A Seshia. "A formal foundation for secure remote execution of enclaves." In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017, pp. 2435–2450.

[170] Melanie Swan. *Blockchain: Blueprint for a new economy*. O'Reilly Media, Inc., 2015.

[171] Meysam Taassori, Ali Shafiee, and Rajeev Balasubramonian. "VAULT: Reducing paging overheads in SGX with efficient integrity verification structures." In: *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems*. 2018, pp. 665–678.

[172] *TEE Client API Specifications v1.0*. `https://globalplatform.org/specs-library/tee-client-api-specification/`. Accessed: 2020-03-17.

[173] Hongliang Tian, Yong Zhang, Chunxiao Xing, and Shoumeng Yan. "SGXKernel: A Library Operating System Optimized for Intel SGX." In: *Proceedings of the Computing Frontiers Conference*. CF 17. Siena, Italy: ACM, 2017, pp. 35–44.

[174] Bohdan Trach, Oleksii Oleksenko, Franz Gregor, Pramod Bhatotia, and Christof Fetzer. "Clemmys: Towards secure remote execution in FaaS." In: *Proceedings of the 12th ACM International Conference on Systems and Storage*. ACM, 2019, pp. 44–54.

[175] Bohdan Trach, Alfred Krohmer, Franz Gregor, Sergei Arnautov, Pramod Bhatotia, and Christof Fetzer. "Shieldbox: Secure middleboxes using

shielded execution." In: *Proceedings of the Symposium on SDN Research*. ACM, 2018, pp. 1–14.

[176] Chia-Che Tsai, Kumar Saurabh Arora, Nehal Bandi, Bhushan Jain, William Jannen, Jitin John, Harry A Kalodner, Vrushali Kulkarni, Daniela Oliveira, and Donald E Porter. "Cooperation and security isolation of library OSes for multi-process applications." In: *Proceedings of the 9th European Conference on Computer Systems*. ACM, 2014, pp. 1–14.

[177] Chia-Che Tsai, Donald E Porter, and Mona Vij. "Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX." In: *2017 USENIX Annual Technical Conference (ATC)*. USENIX, 2017, pp. 645–658.

[178] *Understanding serverless cold start*. `https://azure.microsoft.com/en-us/blog/understanding-serverless-cold-start/`. Accessed: 2020-03-10.

[179] Steffen Viken Valvåg, Dag Johansen, and Åge Kvalnes. "Cogset: a high performance MapReduce engine." In: vol. 25. 1. Wiley Online Library, 2013, pp. 2–23.

[180] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. "Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution." In: *27th USENIX Security Symposium (USENIX Security 18)*. 2018, pp. 991–1008.

[181] Sébastien Vaucher, Rafael Pires, Pascal Felber, Marcelo Pasin, Valerio Schiavoni, and Christof Fetzer. "SGX-aware container orchestration for heterogeneous clusters." In: *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2018, pp. 730–741.

[182] Jesse Walker and Jiangtao Li. "Key exchange with anonymous authentication using DAA-SIGMA protocol." In: *International Conference on Trusted Systems*. Springer, 2010, pp. 108–127.

[183] Huibo Wang, Pei Wang, Yu Ding, Mingshen Sun, Yiming Jing, Ran Duan, Long Li, Yulong Zhang, Tao Wei, and Zhiqiang Lin. "Towards Memory Safe Enclave Programming with Rust-SGX." In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2019, pp. 2333–2350.

[184] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. "Peeking behind the curtains of serverless platforms." In: *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX, 2018, pp. 133–146.

[185] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A Gunter. "Leaky cauldron on the dark land: Understanding memory side-channel hazards in SGX." In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017, pp. 2421–2434.

[186] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A. Gunter. "Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Hazards in SGX." In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS 17. ACM, 2017, pp. 2421–2434.

[187] Xiao Shaun Wang, Kartik Nayak, Chang Liu, TH Hubert Chan, Elaine Shi, Emil Stefanov, and Yan Huang. "Oblivious data structures." In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2014, pp. 215–226.

[188] Nico Weichbrodt, Anil Kurmus, Peter Pietzuch, and Rüdiger Kapitza. "AsyncShock: Exploiting synchronisation bugs in Intel SGX enclaves." In: *European Symposium on Research in Computer Security*. Springer, 2016, pp. 440–457.

[189] Samuel Weiser and Mario Werner. "Sgxio: Generic trusted I/O path for intel SGX." In: *Proceedings of the 7th ACM Conference on Data and Application Security and Privacy*. ACM, 2017, pp. 261–268.

[190] Ofir Weisse, Valeria Bertacco, and Todd Austin. "Regaining lost cycles with HotCalls: A fast interface for SGX secure enclaves." In: vol. 45. 2. ACM, 2017, pp. 81–93.

[191] Tom White. *Hadoop: The definitive guide*. O'Reilly Media, Inc., 2012.

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

[193] Yuval Yarom and Katrina Falkner. "FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack." In: *23rd USENIX Security Symposium (USENIX Security 14)*. USENIX, Aug. 2014, pp. 719–732.

[194] Fan Zhang, Ethan Cecchetti, Kyle Croman, Ari Juels, and Elaine Shi. "Town Crier: An Authenticated Data Feed for Smart Contracts." In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS 16. Vienna, Austria: ACM, 2016, pp. 270–282.

[195] Wenting Zheng, Ankur Dave, Jethro G Beekman, Raluca Ada Popa, Joseph E Gonzalez, and Ion Stoica. "Opaque: An oblivious and encrypted distributed analytics platform." In: *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX, 2017, pp. 283–298.

Paper I

Paper II

Performance of Trusted Computing in Cloud Infrastructures with Intel SGX

Anders T. Gjerdrum, Robert Pettersen, Håvard D. Johansen and Dag Johansen
UiT: The Arctic University of Norway, Tromsø, Norway

Keywords: Privacy, Security, Cloud Computing, Trusted Computing, Performance.

Abstract: Sensitive personal data is to an increasing degree hosted on third-party cloud providers. This generates strong concerns about data security and privacy as the trusted computing base is expanded to include hardware components not under the direct supervision of the administrative entity responsible for the data. Fortunately, major hardware manufacturers now include mechanisms promoting secure remote execution. This paper studies Intel's Software Guard eXtensions (SGX), and experimentally quantifies how basic usage of this instruction set extension will affect how cloud hosted services must be constructed. Our experiments show that correct partitioning of a service's functional components will be critical for performance.

1 INTRODUCTION

Sensors and mobile devices record ever more aspects of our daily lives. This is causing an influx of data streams that feeds into potentially complex analytical pipelines hosted remotely by various cloud providers. Not only are the sheer amounts of data generated cumbersome to store and analyze at scale; data might also be accompanied by strict privacy requirements, as is the case with smart home and health monitoring devices (Gjerdrum et al., 2016).

Processing of sensitive and personal data in the cloud requires the design of new Software-as-a-Service (SaaS) architectures that are able to enforce rigid privacy and security policies (Johansen et al., 2015) throughout the entire hardware and software stack, including the underlying cloud-provided Infrastructure-as-a-Service (IaaS) components. Although, commodity hardware mechanisms for trusted computing have been available for some time (TCG Published, 2011; Osborn and Challener, 2013), these are often poised with performance and functionality restrictions. Prior implementations by Intel, like Trusted Platform Modules (TPM) and Trusted Execution Technology (TXT), are able to establish trust and guarantee integrity of software, the latter also supporting rudimentary secure code execution.

Software Guard Extentions (SGX) (Anati et al., 2013) is Intel's new trusted computing platform that, together with similar efforts by both ARM and AMD, is quickly making general trusted computing a commodity. Fundamentally, SGX is an instruction set extension introduced with the Skylake generation of Intel's Core architecture, supporting confidentiality, integrity and attestation of trusted code running on untrusted platforms. SGX is able to counter a multitude of different software and physical attacks by the construction of secure enclaves consisting of trusted code and data segments. While SGX should be considered an iterative technology built on previous efforts, it surpasses previous iterations both in terms of performance and functionality. SGX is designed to provide general secure computing facilities allowing developers to easily port their existing legacy applications into SGX enabled enclaves. These properties make SGX an attractive technology for cloud-based SaaS architectures that handle person sensitive data.

SGX is a proprietary platform and prior knowledge is based on limited documentation describing its architecture. Furthermore, little is known about the performance of the primitives provided by the SGX platform and how to author software utilizing these primitives while maximizing performance.

In this paper we analyze the performance characteristics of the SGX technology currently available to better understand how such technologies can be used to enforce privacy policies in cloud hosted SaaS architectures. We analyze SGX primitives at a fine-grained level and provide detailed performance evaluation of the core mechanisms in SGX. The paper is structured as follows: Section 2 outlines the relevant parts of the SGX micro architecture while Section 3 outlines the

Gjerdrum, A., Pettersen, R., Johansen, H. and Johansen, D.
Performance of Trusted Computing in Cloud Infrastructures with Intel SGX.
DOI: 10.5220/0006373706960703
In *Proceedings of the 7th International Conference on Cloud Computing and Services Science (CLOSER 2017)*, pages 668-675
ISBN: 978-989-758-243-1

details of our microbenchmark. Section 4 provides an informed discussion of our findings and Section 5 detail relevant work before concluding remarks.

2 INTEL SOFTWARE GUARD EXTENTIONS (SGX)

SGX allows regular application threads to transition into secure enclaves by issuing the special EENTER special instructions to a logical processor. Entry is initiated by performing a controlled jump into the enclave code, analogous to how entry into virtual machine contexts occurs. A process can only enter an enclave from ring 3, i.e user level, and threads running in *enclave mode* are not allowed to trigger software interrupts, also prohibiting the use of system calls. An application which requires access to common Operating System (OS) provided services, like the file system, must be carefully designed so that its threads exit *enclave mode* through application defined interfaces before invoking any system calls. Since SGX's Trusted Computing Base (TCB) does not include the underlying OS, all such transitions, parameters, and responses, must be carefully validated by the application designer.

SGX allows multiple threads to execute inside the same enclave. For each logical processor executing inside an enclave a Thread Control Structure (TCS) is needed. These data structures must be provisioned before enclave startup, and are stored in the Enclave Page Cache (EPC) main-memory pages set aside for enclaves. Among other things, the TCS contains the OENTRY field which is loaded into the instruction pointer when entering an enclave. Before doing so, SGX stores the execution context of the untrusted code into regular memory, by using the XSAVE instruction, which then again is restored when exiting the enclave. Stack pointers are not modified when entering an enclave, however (Costan and Devadas, 2016) suggests that to avoid the possibility of exploits, it is expected that each enclave set their stack pointer to an area fully contained withing EPC memory. Parameter input to the enclave is marshalled into buffers, and once the transition is done, enclave code can copy data directly from untrusted DRAM memory. This is not part of the native SGX implementation, rather a convenience provided by the application SDK.

Threads exit enclaves either voluntarily through synchronous exit instructions, or asynchronously by service of a hardware interrupt occurring on the affected logical core. Synchronous exits, through the EEXIT instruction, causes the logical processor to leave enclave mode. The instruction pointer as well as the stack pointers are restored to their prior address before entering the enclave. SGX does not modify any instructions on enclave exit and so it is the authors' responsibility to clear them, to avoid leaking secret information. In the case of an Asynchronous Enclave Exit (AEX), a hardware interrupt such as a page fault causes the processor to exit the enclave and jump down to the kernel in order to service the fault. Prior to this, SGX saves the execution context into EPC memory for safekeeping, before clearing it so that the OS is not able to infer any execution state from the enclave. When the interrupt handler is done, SGX restores the execution context and resumes execution.

2.1 The Enclave Page Cache

Memory used by enclaves is separated at boot time from regular process DRAM memory into what is called Processor Reserved Memory (PRM). This contiguous region of memory is divided into 4 kb pages, collectively referred to as the Enclave Page Cache (EPC). EPC memory is only accessible inside the enclave or via the SGX instruction set. Neither system software running at protection ring 0 (kernel mode) or application code at ring 3 (user mode) are able to access its memory contents directly, and any attempt to read or write to it is ignored. Furthermore, DMA access to PRM memory is prohibited by hardware to guard against malicious peripheral devices attempting to tap the system bus. The confidentiality of the enclave is guarded by Intel's Memory Encryption Engine (MEE), which encrypts and decrypts memory at the CPU package boundary, on the system bus right after the L3 cache.

Similar to virtual memory, EPC page management is handled entirely by the OS. However, EPC memory is not directly accessible to any system mode and each page assignment is done through SGX instructions. The OS is responsible for assigning pages to particular enclaves and evict pages to regular DRAM. The current generation of SGX hardware only supports a maximum PRM size of 128 MB, but through swapping, there are no practical limits to the size of an enclave. The integrity of pages swapped out is guaranteed by always checking an auxiliary data structure also residing in PRM, called the Enclave Page Cache Map. This datastructure contains the correct mappings between virtual addresses and Physical PRM memory, as well as integrity checks for each page. Each page can only belong to one enclave, and as a consequence, shared memory between enclaves is prohibited. They are however able to share DRAM memory if residing inside the same process'

address space, and enclave memory is allowed to read and write to untrusted memory inside that process. The page eviction instruction also generates a liveness challenge for each page, storing them in special EPC pages for later comparison. These precautions guard against a malicious OS trying to subvert an enclave by either manipulating the address translation, explicitly manipulating pages, or serving old pages back to the enclave (replay attacks).

In order to guard against stale address translations for executing enclaves, the processor does a coarse-grained TLB shootdown for pages being evicted. Page faults targeting a particular enclave will cause the kernel to issue a Inter Processor Interrupt (IPI) for all logical cores running inside of the enclaves in question. This will cause each thread to do an AEX, as mentioned above, and trap down to the kernel page fault handler. Moreover, the lowermost 12 bits of the virtual address at fault, stored in the CR2 registry, is cleared so that the OS cannot infer any access pattern. To amortize the cost of interrupting all cores executing inside a particular enclave for each page eviction, the SGX implementation supports batching up to 16 page evictions together at a time.

2.2 Enclave Creation

SGX supports multiple mutually distrusting enclaves on a single machine either within the same process' address space or in different processes. Enclaves are created by system software on behalf of an application, issuing an ECREATE instruction. This will cause SGX to allocate a new EPC page for the SGX Enclave Control Structure (SECS) which stores metadata for each enclave. This is used by SGX instructions to identify enclaves, and among other things map enclaves to physical EPC pages via the EPCM structure. Before the enclave is ready for executing code, each initial code and data segment must be added to enclave memory via the OS issuing specially crafted instructions to the SGX implementation for each page. The same instruction is also used to create the TCS for each expected thread inside the enclave. In addition, the OS driver issues updates for enclave measurements used for software attestation. We refer to the SGX developer manual for a description of the SGX attestation process. When all pages are loaded, the enclave is initialized and the enclave receives a launch token from a special pre-provisioned enclave entrusted by Intel. At this point, the enclave is considered fully initialized and no further memory allocations may happen. Intels revised specifications for SGX version 2 includes support for expanding enclaves after initial creation by dynamic paging support. However, we refrain from further explanation as hardware supporting these specifications has not been released at this point.

When an enclave is destroyed, the inverse happens, as the OS marks each page used by the enclave as invalid by the EREMOVE instruction. Before freeing the page, SGX makes sure that no logical processor is executing inside the enclave that owns the particular page. Finally, the SECS is deallocated if all pages in the EPCM referring to that particular enclave are deallocated.

3 EXPERIMENTS

To gain experience in how the next generation cloud-based SaaS systems should be architected to best take advantage of the SGX features in modern processors, we ran a series of micro benchmarks on SGX-enabled hardware. Our experimental setup consists of a Dell Optiplex workstation with an Intel Core i5-6500 CPU @ 3.20 GHz with four logical cores and 2×8 GB of DDR3 DIMM DRAM. To avoid inaccuracies caused by dynamic frequency scaling, Intel Speedstep and CStates were disabled in all our experiments. To measure the peak performance of the architecture, we also altered the PRM size in hardware setup to be the maximum allowed 128 MB. We run the experiments on Ubuntu 14.04 using the open source kernel module for Intel SGX.[1] We instrumented the SGX kernel module to record the operational costs. Based on our understanding of the system we derived different benchmarks testing various features of the platform. Common for all experiments is the observation that more iterations did not yield a lower deviation. This may be attributed to noise generated by the rest of the system. This noise is subtle, but significant since we are measuring at fine-grained time intervals.

Note that the current iteration of SGX prohibits use of the RDTSC instruction inside of enclaves, and as such there are no natively timing facilities available inside enclaves. A later release reveals that the updated specifications for SGX version one does support RDTSC inside enclaves. Hints suggests that this might be distributable by means of an update to the microcode architecture. We were, however, unsuccessful in obtaining this update. Time measurements performed throughout this experiment must therefore exit the enclave before being captured. As a consequence, we can only measure the total time taken between entering and exiting an enclave described as the sequence of events depicted in Figure 1.

[1] https://github.com/01org/linux-sgx-driver

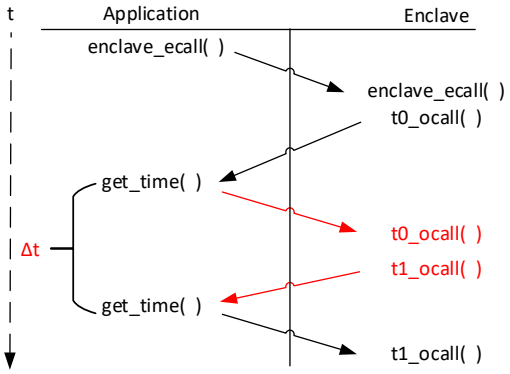


Figure 1: Sequence of events involved in measuring time spent inside enclaves.

3.1 Entry and Exit Costs

In our first experiments, we look at the cost of entering and leaving an enclave. Understanding this cost is important as it dictates how SGX enabled SaaS services can partition its functionality between enclaved and non-enclaved execution to minimize TCB size. A prohibitively large cost of entry would necessitate a reduction in the number of entry calls, and thus increasing the amount of code and data residing inside of the enclave, increasing the required TCB. The extreme case being a full library OS that include almost all the functionality an application requires within the enclave (Baumann et al., 2014). The Intel Software Developer Manual[2] suggests that the cost of entering an enclave is also a function of the size of the data copied into the enclave as a part of the entry. Thus, if experiments show that the cost of large amounts of data entering the enclave is prohibitively large, only data requiring confidentiality should be copied into the enclave.

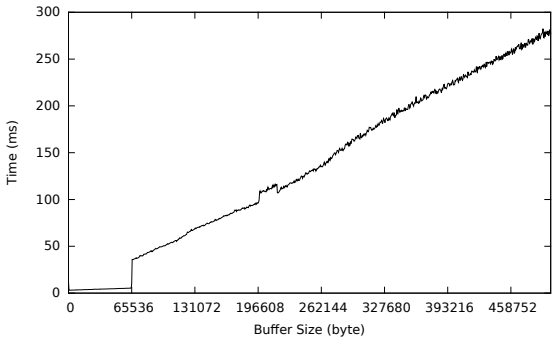


Figure 2: Enclave transition cost as a function of buffer size.

Figure 2 shows the measured cost as a function of increasing buffer sizes. As shown in the figure, the cost of transitioning into enclaves increases linearly with the buffer size. This experiment only uses buffers as parameter while transitioning into the enclave. To be able to host the buffer inside the enclave, its heap size must be sufficiently large. The observed baseline cost with no buffer is the bare transition cost for entering enclaves. This cost quickly becomes insignificant as the buffer size increases. This behavior is expected as this cost includes copying the buffers into enclave memory on transitions, which invokes the MEE for memory written to the enclave. To our surprise, however, we observed that the baseline cost only increased above 64 kb. One possible explanation for this is that the pages may already be present in EPC memory for buffer sizes smaller than 64 kb

For larger buffers the increased cost can also be attributed to page faults caused by enclave memory previously evicted to DRAM. This issue is further explored in the next experiment.

3.2 Paging

Another probable architectural trade-off is the logical assumption that an increase in TCB would reduce enclave transitions but requires more PRM. As mentioned in Section 1, the PRM is a very limited resource in comparison to regular DRAM and the system has a total of 128 MB of it. Moreover, any enclave is subject to the system software evicting EPC pages when PRM resources becomes scarce. Any system using SGX should factor in the cost of swapping pages between PRM and regular DRAM. Figure 3 illustrates this cost in enclaves as observed by both the kernel and the user level enclave.

The y-axis is the discrete cost in nano seconds, while the x-axis is time elapsed into the experiment. We instrumented the OS kernel driver to measure the time taken to evict pages out of EPC into DRAM denoted by red dots, as well as the total time spent inside the page fault handler, shown by the black line.

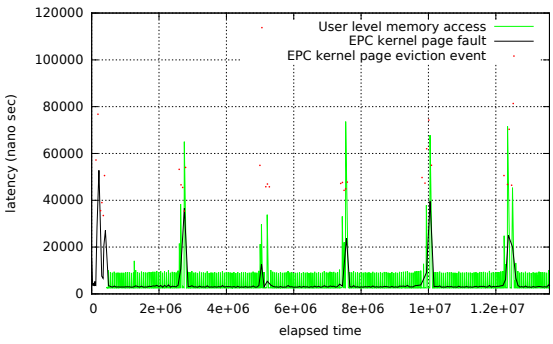


Figure 3: Paging overhead in nano seconds as a function of time elapsed while writing sequentially to enclave memory.

[2]https://software.intel.com/en-us/articles/intel-sdm

The green line denotes user level instrumentation measuring the time it takes to write to a particular address in EPC memory. Similarly to the prior experiment, we are prohibited to make timing measurements inside enclaves. Therefore, the user level measurements include the baseline cost of entry and exit of an enclave, notably with 4 byte buffers transitioning each way.

To induce enclave page faults we set the total enclave heap size to 256 MB, which is larger than the total amount of available EPC memory. Furthermore, to hit each page we invoke write operations to each address within the 4 kb page size sequentially along the allocated memory space inside the enclave. As mentioned in Section 2, the only time enclaves are able to allocate memory is before the EINIT instruction is called by issuing EADD. Therefore, all memory must be allocated before enclave execution begins. We can clearly see at the beginning of the experiment an increase in page faults occurring when trying to fit 256 MB of enclave memory into potentially 128 MB of physical EPC memory.

Correlating the different events happening at user level and kernel level we observe a strong relationship between eviction events and increase in write time at user level. One property of the system that might increase this cost is the fact that evicting pages causes AEX events for any logical processor executing within an enclave, as explained in Section 2.

We also observe that the kernel driver is operating very conservatively in terms of assigning EPC pages to enclaves by the amount of page faults occurring during execution. Moreover, as mentioned in Section 2, the 12 lower bits of the virtual page fault address is cleared by SGX before trapping down to the page fault handler. Therefore, the driver is not able to make any assumptions about memory access patterns inside enclaves. Moreover, as Section 2 explains, liveness challenge data might also be evicted of EPC memory, causing a cascade of page loads to occur from DRAM. It is worth noting that our experiment only uses one thread, and that all page evictions issuing IPI only interrupt this single thread.

It is clear that high performance applications might want to tune the OS support for paging to their needs. If an application can predict a specific access pattern, the kernel paging support should adapt to this. Moreover, by optimizing towards exhaustive use of the EPC memory, applications running inside enclaves might be subject to fewer page faults.

Furthermore, initial setup will keep large amounts of the enclave in memory, which might eliminate the overhead of paging for some enclaves. This furthermore reduces overhead caused by IPI interrupts triggering AEX from the given enclave. Initially, the creation of large enclaves trigger memory allocations by the kernel, and it might be necessary for application developers to offset this initial cost by previsioning enclaves.

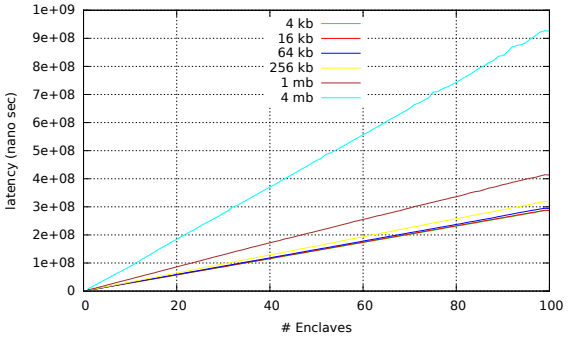


Figure 4: Latency as a function of number of enclaves created simultaneously, for differing sizes of enclaves

3.3 Enclave Provisioning

Modern distributed system architectures increasingly rely on modular programing paradigms and multi-component software with possibly differing trust domains. Such distributed systems often consist of several third-party open source components, both trusted and not. Moreover, separating both the unit of failure and trust of such systems is often a good idea.

SGX supports the creation of multiple mutually distrusting enclaves which can be used in such a modular design. As mentioned in Section 2 the SGX programming model allows enclaves to communicate with the outside using well defined interfaces, which lends itself to an architecture where trust is compartmentalized into separate enclaves. Figure 4 illustrates the additional cost in terms of provisioning latency as a function of enclaves created simultaneously, and we can clearly observe that the added cost in enclave creation increases linearly. Through multiple iterations of this experiment we observe the added cost by increasing enclave sizes. As demonstrated, this added cost becomes increasingly significant when provisioning multiple enclaves exceeding 256 kb in size. As mentioned in Section 2, enclaves are created by issuing specially crafted functions for each page of code and data being allocated inside enclave memory. It is worth mentioning that we observed a significant amount of page faults occurring during enclave creation, and it is reasonable to assume that this is also contributing to the cost. Furthermore, the observations made about entry cost for buffer sizes less than 64 kb shown in Figure 2, is further corroborated by the fact that for enclave sizes less than 64 kb the provisioning costs are nearly identical.

For application software requiring low latency operation it might be necessary to pre-provision enclaves to offset this cost in latency. However, this approach might cause additional problems with collocating them in EPC memory if the individual enclaves are sufficiently large.

4 DISCUSSION

From our experiments in Section 3, we have identified several important performance idiosyncrasies of SGX that should be considered when constructing SGX enabled cloud services: the cost of entering and exiting enclaves, the cost of data copying, the cost of provisioning new enclaves and the cost of memory usage.

As mentioned in Section 2, entry and exit procedures do similar amounts of work in terms of cost. As our experiments show, the most significant cost factor of transitioning is the buffer size input as argument to the transition either through entry or exit. In particular, we observed a steep rise in data copy cost when buffer sizes are larger than 64 kb. Our recommendation is therefore that:

Recommendation 1. *Applications should partition its functional components to minimize data copied across enclave boundries.*

One possible component architecture that follows the guideline of Recommendation 1 would be to co-locate all functionality into one single enclave, making it largely self-sufficient. An example a system following such an approach is Haven (Baumann et al., 2014), which reduces the interface between trusted and untrusted code by co-locating a larger part of the system software stack inside a single enclave by means of a library OS. The efficiency of this approach, however, directly contradicts the observation we made in Section 3.2, where we measured the overhead associated with enclave memory being paged in and out to regular DRAM. Because the EPC is a scarce resource, system software aggressively pages out enclave memory not being used. However, as our experiments show, the page fault handler is overeager, and fails to fully utilize EPC memory exhaustively. Because of security concerns, the kernel is not given the exact faulting address of each enclave page fault, and therefore does not make any assumptions as to the memory access patterns. We therefore recommend that:

Recommendation 2. *The size of an enclave should not exceed 64 kb.*

Recommendation 3. *Prior knowledge about application's memory consumption and access pattern should be used to modify the SGX kernel module in order to reduce memory page eviction.*

As our experiment shows, enclave creation is costly and time consuming. To hide some of this cost, the underlying OS can pre-provision enclaves whenever usage patterns can be predicted. However, once used, an enclave might be tainted with secret data. Recycling used enclaves to a common pool can therefore potentially leeks secrets from one process to the next: invalidating the isolation guarantees. We therefore recommend that:

Recommendation 4. *Application authors that can accurately predict before-the-fact usage of enclaves should pre-provision enclaves in a disposable pool of resources that guarantees no reuse between isolation domains.*

The cost of enclave creation must also factor in the added baseline cost of metadata structures associated with each enclave. Provisioning an enclaves must at least account for its SECS, one TCS structure for each logical core executed inside an enclave, and one SSA for each thread performing AEX. (Costan and Devadas, 2016) explains that to simplify implementation, most of these structures are allocated at the beginning of a EPC page, wholly dedicated to that instance. Therefore, it is not out of line to consider an enclave with 4 logical cores, having 9 pages (34 kb) allocated to it, excluding code and data segments. Applications should consider the added memory cost of separate enclaves in conjunction with the relative amount of available EPC. Furthermore, to offset the cost of having multiple enclaves, application authors should consider security separation at a continuous scale. Some security models might be content with role based isolation, rather than call for an explicit isolation of all users individually. We therefore recommend that:

Recommendation 5. *Application authors should carefully consider the granularity of isolation required for their intended use, as a finer granularity includes the added cost of enclave creation.*

At the time of writing, the only available hardware supporting SGX are the Skylake generation Core chips with SGX version 1. As our experiments show, paging has a profound impact on performance, and a natural follow-up would be to measure the performance characteristics of the dynamic paging support proposed in the SGX V2 specifications. However,

as mentioned earlier, Intel has yet to release any information regarding the arrival of SGX V2 enabled chips. The imminent 8th generation Kaby Lake chips do not include support, and the earliest likely release will therefore as part of Cannon Lake in Q4 2017.

SGX supports attestation of software running on top of untrusted platforms, by using signed hardware measurements to establish trust between parties. These parties could be either locally with two distinct enclaves executing on the same hardware, or remotely by help of Intel's attestation service. In the future, it would be interesting, in light of the large cost of enclave transition demonstrated above, to examine the performance characteristics of a secure channel for communication between enclaves.

5 RELATED WORK

Several previous works quantifies various aspects of the overhead associated with composite architectures based on SGX. Haven (Baumann et al., 2014) implements shielded execution of unmodified legacy applications by inserting a library OS entirely inside of SGX enclaves. This effort resulted in architectural changes to the SGX specification to include, among other things, support for dynamic paging. The proof of concept implementation of Haven is only evaluated in terms of legacy applications running on top of the system. Furthermore, Haven was built on a pre-release emulated version of SGX, and the performance evaluation is not directly comparable to real world applications. Overshadow (Chen et al., 2008) provide similar capabilities as Haven, but does not rely on dedicated hardware support.

SCONE (Arnautov et al., 2016) implements support for secure containers inside of SGX enclaves. The design of SCONE is driven by experiments on container designs pertaining to the TCB size inside enclaves, in which, at the most extreme an entire library OS is included and at the minimum a stub interface to application libraries. The evaluation of SCONE is, much like the evaluation of Haven, based on running legacy applications inside SCONE containers. While (Arnautov et al., 2016) make the same conclusions with regards to TCB size versus memory usage and enclave transition cost as (Baumann et al., 2014), they do not quantify this cost. Despite this, SCONE supplies a solution to the entry exit problem we outline in Section 3, where threads are pinned inside enclaves, and do not transition to the outside. Rather, communication happens by means of the enclave threads writing to a dedicated queue residing in regular DRAM memory. This approach is still, however, vulnerable to theads being evicted from enclaves by AEX caused by IPI as part of a page fault.

(Costan and Devadas, 2016) describe the architecture of SGX based on prior art, released developer manuals, and patents. Furthermore, they conduct a comprehensive security analysis of SGX, falsifying some of its guarantees by explaining in detail exploitable vulnerabilities within the architecture. This work is mostly orthogonal to our efforts, however, we base most of our knowledge of SGX from this treatment on the topic. These prior efforts lead (Costan et al., 2016) to implement Sanctum, which implements an alternative hardware architectural extension providing many of the same properties as SGX, but targeted towards the Rocket RISC-V chip architecture. Sanctum evaluates its prototype by simulated hardware, against an insecure baseline without the proposed security properties. (McKeen et al., 2016) introduce dynamic paging support to the SGX specifications. This prototype hardware were not available to us.

Ryoan (Hunt et al., 2016) attempts to solve the same problems outlined in the introduction, by implementing a distributed sandbox for facilitating untrusted computation on secret data residing on third party cloud services. Ryoan proposes a new request oriented data-model where processing modules are activated once without persisting data input to them. Furthermore, by remote attestation, Ryoan is able to verify the integrity of sandbox instances and protect execution. By combining sandboxing techniques with SGX, Ryoan is able to create a shielding construct supporting mutually distrust between the application and the infrastructure. Again, Ryoan is benchmarked against real world applications, and just like other prior work, does not correctly quantify the exact overhead attributed to SGX primitives. Furthermore, large parts of its evaluation is conducted in an SGX emulator based on QEMU, which have been retrofitted with delays and TLB flushes based upon real hardware measurements to better mirror real SGX performance. These hardware measurements are present for EENTRY and EEXIT instructions, however do not attribute the cost of moving argument data into and out of enclave memory. Moreover, Ryoan speculate on the cost of SGX V2 paging support, although strictly based on emulated measurements, and assumptions about physical cost.

ARM TrustZone is a hardware security architecture that can be incorporated into ARMv7-A, ARMv8-A and ARMv8-M on-chip systems (Ngabonziza et al., 2016; Shuja et al., 2016). Although the underlying hardware design, features, and interfaces differ substantially to SGX,

both essentially provide the same key concepts of hardware isolated execution domains and the ability to bootstrap attested software stacks into those enclaves. However, the TrustZone hardware can only distinguish between two execution domains, and relies on having a software based trusted execution environment for any further refinements.

6 CONCLUSION

Online services are increasingly relying on third-party cloud providers to host sensitive data. This tendency brings forth strong concerns for the security and privacy of data owners as cloud providers cannot fully be trusted to enforce the restrictive usage policies that often govern such data. Intel SGX provides hardware support for general trusted computing in commodity hardware. These extensions to the x86 instruction set establish trust through remote attestation of code and data segments provisioned on non-trusted infrastructure, furthermore guaranteeing the confidentiality and integrity of these from potentially malicious system software.

Prior efforts demonstrate the capabilities of SGX through rigorous systems capable of hosting large legacy applications securely inside enclaves. These systems, however, do not quantify the exact cost associated with using SGX. This paper evaluates the micro architectural cost of entering and exiting enclaves, the cost of data copying, the cost of provisioning new enclaves and the cost of memory usage. From this, we have derived five recommendations for application authors wishing to secure their cloud-hosted privacy sensitive data using SGX.

ACKNOWLEDGMENTS

This work was supported in part by the Norwegian Research Council project numbers 231687/F20. We would like to thank the anonymous reviewers for their useful insights and comments.

REFERENCES

Anati, I., Gueron, S., Johnson, S., and Scarlata, V. (2013). Innovative technology for cpu based attestation and sealing. In *Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy*, volume 13.

Arnautov, S., Trach, B., Gregor, F., Knauth, T., Martin, A., Priebe, C., Lind, J., Muthukumaran, D., O'Keeffe, D., Stillwell, M. L., Goltzsche, D., Eyers, D., Kapitza, R., Pietzuch, P., and Fetzer, C. (2016). Scone: Secure linux containers with intel sgx. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 689–703, GA. USENIX Association.

Baumann, A., Peinado, M., and Hunt, G. (2014). Shielding applications from an untrusted cloud with Haven. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)*. USENIX – Advanced Computing Systems Association.

Chen, X., Garfinkel, T., Lewis, E. C., Subrahmanyam, P., Waldspurger, C. A., Boneh, D., Dwoskin, J., and Ports, D. R. (2008). Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems. In *Proc. of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, pages 2–13, New York, NY, USA. ACM.

Costan, V. and Devadas, S. (2016). Intel sgx explained. In *Cryptology ePrint Archive*.

Costan, V., Lebedev, I., and Devadas, S. (2016). Sanctum: Minimal hardware extensions for strong software isolation. In *USENIX Security*, volume 16, pages 857–874.

Gjerdrum, A. T., Johansen, H. D., and Johansen, D. (2016). Implementing informed consent as information-flow policies for secure analytics on eHealth data: Principles and practices. In *Proc. of the IEEE Conference on Connected Health: Applications, Systems and Engineering Technologies: The 1st International Workshop on Security, Privacy, and Trustworthiness in Medical Cyber-Physical System*, CHASE '16. IEEE.

Hunt, T., Zhu, Z., Xu, Y., Peter, S., and Witchel, E. (2016). Ryoan: A distributed sandbox for untrusted computation on secret data. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 533–549, Berkeley, CA, USA. USENIX Association.

Johansen, H. D., Birrell, E., Van Renesse, R., Schneider, F. B., Stenhaug, M., and Johansen, D. (2015). Enforcing privacy policies with meta-code. In *Proceedings of the 6th Asia-Pacific Workshop on Systems*, page 16. ACM.

McKeen, F., Alexandrovich, I., Anati, I., Caspi, D., Johnson, S., Leslie-Hurd, R., and Rozas, C. (2016). Intel® software guard extensions (intel® sgx) support for dynamic memory management inside an enclave. In *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016*, page 10. ACM.

Ngabonziza, B., Martin, D., Bailey, A., Cho, H., and Martin, S. (2016). Trustzone explained: Architectural features and use cases. In *Collaboration and Internet Computing (CIC), 2016 IEEE 2nd International Conference on*, pages 445–451. IEEE.

Osborn, J. D. and Challener, D. C. (2013). Trusted platform module evolution. *Johns Hopkins APL Technical Digest*, 32(2):536–543.

Shuja, J., Gani, A., Bilal, K., Khan, A. U. R., Madani, S. A., Khan, S. U., and Zomaya, A. Y. (2016). A survey of mobile device virtualization: taxonomy and state of the art. *ACM Computing Surveys (CSUR)*, 49(1):1.

TCG Published (2011). TPM main part 1 design principles. Specification Version 1.2 Revision 116, Trusted Computing Group.

Paper III

Performance Principles for Trusted Computing with Intel SGX

Anders T. Gjerdrum(✉), Robert Pettersen, Håvard D. Johansen, and Dag Johansen

Department of Computer Science, UIT The Arctic University of Norway, Tromsø, Norway
anders.t.gjerdrum@uit.no

Abstract. Cloud providers offering Software-as-a-Service (SaaS) are increasingly being trusted by customers to store sensitive data. Companies often monetize such personal data through curation and analysis, providing customers with personalized application experiences and targeted advertisements. Personal data is often accompanied by strict privacy and security policies, requiring data processing to be governed by non-trivial enforcement mechanisms. Moreover, to offset the cost of hosting the potentially large amounts of data privately, SaaS companies even employ Infrastructure-as-a-Service (IaaS) cloud providers not under the direct supervision of the administrative entity responsible for the data. Intel Software Guard Extensions (SGX) is a recent trusted computing technology that can mitigate some of these privacy and security concerns through the remote attestation of computations, establishing trust on hardware residing outside the administrative domain. This paper investigates and demonstrates the added cost of using SGX, and further argues that great care must be taken when designing system software in order to avoid the performance penalty incurred by trusted computing. We describe these costs and present eight specific principles that application authors should follow to increase the performance of their trusted computing systems.

Keywords: Privacy · Security · Cloud computing
Trusted computing · Performance

1 Introduction

Pervasive computing and the ongoing Internet of Things (IoT) revolution have led to many new mobile recording and sensory devices that record ever more facets of our daily lives. Captured data is often analyzed and stored by complex ecosystems of cloud hosted services. Storing and analyzing large amounts of data are non-trivial problems. Handling personal data such as smart home monitoring systems and health tracking, only adds the to this complexity as data processing might be governed by strict privacy requirements [1].

D. Ferguson et al. (Eds.): CLOSER 2017, CCIS 864, pp. 1–18, 2018.
https://doi.org/10.1007/978-3-319-94959-8_1

The curation and analysis of privacy sensitive personal data on third-party cloud providers necessitate the design of a new Software-as-a-Service (SaaS) architecture that is able to enforce rigid privacy and security policies [2] throughout the entire software stack, including the underlying cloud provided Infrastructure-as-a-Service (IaaS). Commodity hardware components for trusted computing have been available for some time [3,4], but the functionality of existing solutions has been limited to establishing trust and guarantees on the integrity of running software, and rudimentary support for secure code execution (e.g., Intel Trusted Execution Technology).

In 2015, Intel introduced the Software Guard Extensions [5] as part of their sixth generation Intel Core processor micro architecture (codenamed Skylake). Together with complementary efforts by ARM and AMD, SGX is making general trusted computing a commodity, providing confidentiality, integrity and attestation of code and data running on untrusted third-party platforms. SGX is able to deter multiple different software and physical attacks by establishing secure execution environments, or enclaves, of trusted code and data segments inside individual CPUs. While SGX is an iterative technology building upon previous efforts, it is more general in functionality allowing code execution inside enclaves at native processor speeds, a significant performance improvement over previous efforts. SGX is designed with backwards compatibility in mind, allowing developers to port sensitive logic from existing legacy applications into secure enclaves. These properties make SGX a compelling technology for cloud based SaaS hosting privacy sensitive data on untrusted third-party cloud providers. SGX is a proprietary technology and prior knowledge of its characteristics is mostly based on limited documentation by Intel. In particular, little is known about the performance of the computing primitives comprising SGX and how developers should best utilize these to maximize application performance.

This paper provide an in-depth investigation into key performance traits of the Intel SGX platform. We provide a performance analysis of its low-level mechanisms and primitives, and describe several non-obvious idiosyncrasies related to threading, context switching, and memory footprint. From our observations, we derive 1 principles for developing more efficient software on this platform.

The remainder of this paper is structured as follows: Sect. 2 outlines the relevant parts of the SGX micro architecture while Sect. 3 outlines the details of our micro benchmarks. Section 4 provides an informed discussion of our findings and a set of derived principles intended for developers of trusted computing systems. Section 5 details relevant work before concluding remarks.

2 Intel Software Guard Extensions (SGX)

Intel's new general trusted computing platform enables the execution of code on untrusted third-parties at native processor speed. Moreover, the platform preserves the confidentiality and integrity of code and data segments running inside what is referred to as *enclaves*. This section details the core mechanisms comprising SGX, building a foundation for the performance analysis detailed in Sect. 3.

2.1 Enclave Creation

Enclave code and data are distributed to runtime systems in form of a shared library which is bundled together with what the developer reference refers to as the SIGSTRUCT data structure. During the compilation of an enclave, a hash, or measurement, of each code and data segment executable within the shared library is computed and stored together with a signature generated by the developers private key. This bundle is then distributed to the target third-party platform together with the corresponding public key. During initialization, the signature is verified against the public key and the measurement is recalculated and compared with the corresponding value inside the SIGSTRUCT. If the signature matches that of the public key and the integrity of the code and data segments are preserved, the enclave is allowed to execute. This establishes a guarantee that only the expected enclave code and data from the expected enclave author are successfully able to run on the third-party.

2.2 Entry and Exit

Regular application threads are able to enter secure enclaves by invoking the EENTER instruction on a particular logical core. The thread then performs a controlled jump into the enclave code, similar in operation to a call-gate. Threads can only enter enclaves from privilege level 3 (user level).

Software interrupts are prohibited when running in *encalve mode*. As a consequence, no system calls are allowed within enclaves. Applications requesting access to common Operating System (OS) resources such as IO, must therefore explicitly exit the enclave prior to invocation. The application developer explicitly defines these transitions and, in the presence of a potentially malicious OS, all such transitions, parameters to these and responses must be carefully validated.

Although threads cannot be instantiated in enclave mode, SGX allows multiple threads to enter the same enclave and execute concurrently. For each logical core executing inside a particular enclave, a Thread Control Structure (TCS) is required to keep track of thread specific context. Before instantiation, these data structures must be provisioned and stored in the Enclave Page Cache (EPC), comprising pages explicitly set aside for enclaves. The TCS contains an OENTRY field specifying the entry point for the thread, loaded into the instruction pointer upon entry. Stack regions are not explicitly handled by the SGX microcode, however, as Costan and Devadas [6] state, the stack pointer is expected to be set to a region of memory fully contained within the enclave during entry transition. Parameters input to the developer-specified entry points are marshaled and, once the transition is done, copied into enclave memory from untrusted memory. Although not handled by SGX directly, parameter marshaling and stack pointer manipulation are managed under the hood by the SDK implementation which most application authors will use for enclave development.

Threads may transition out of enclaves by means of two different mechanisms, either synchronously trough the explicit EEXIT instruction, or asynchronously

by service of a hardware interrupt. Synchronous exists will cause the thread to leave enclave mode, restoring the execution context to its content prior to enclave entry. Asynchronous Enclave Exit (AEX) is caused by a hardware interrupt such as a page fault event. In this case all threads executing on the logical core affected by the interrupt must exit the enclave and trap down to the kernel in order to service the fault. Before exit, the execution context for all logical cores executing within the enclave is saved and subsequently cleared to avoid leaking information to the untrusted OS. When the page fault has been serviced, the ERESUME instruction restores the context and the enclave resumes execution.

2.3 Enclave Memory

During boot-up of the CPU, a contiguous region of memory called Processor Reserved Memory (PRM) is set aside from regular DRAM. Divided into 4 kB pages, only accessible inside the enclave or directly by the SGX instructions, this region of memory is collectively referred to as the EPC. Any attempts to either read or write EPC memory from both privileged level system software or regular user level applications are ignored. Moreover, any Direct Memory Access (DMA) request to this region is explicitly prohibited, deterring physical attacks on the system bus by potentially malicious peripheral devices. Confidentiality is achieved through Intels Memory Encryption Engine (MEE), further preventing physical memory inspection attacks as enclave data is encrypted at the CPU package boundary on the system bus right after the L3 cache.

Much the same as regular virtual memory, EPC pages are also managed by the OS. However, these are handled indirectly through SGX instructions as EPC memory is not directly accessible. The OS is responsible for assigning pages to enclaves and evict unused pages to regular DRAM. Through memory management, the physical limit of 128 MB is evaded by swapping EPC pages and as such there is no practical limit to the size of enclaves. The integrity and liveness of pages being evicted are guarded by an axillary data structure also contained within the PRM, called the Enclave Page Cache Map (EPCM). The EPCM maintains the mappings between virtual and physical addresses of PRM memory. Moreover, it maintains for each page an integrity check and a liveness challenge vector. These precautions guard against a malicious OS trying to subvert an enclave by either manipulating the address translation, explicitly manipulating pages, or serving old pages back to the enclave (replay attacks). In this memory model, only one enclave can claim ownership of a particular page at one given moment, and as a consequence shared memory between enclaves is prohibited. Enclaves are however allowed to read and write directly to untrusted DRAM inside the host process' address space, and therefore two enclaves residing within the same host process are able to share untrusted memory.

Because stale address translations may be exploited to subvert enclave integrity, the processor performs a coarse-grained Translation Lookaside Buffer (TLB) shootdown for each page subject to eviction. Given a page fault event on a particular thread executing inside an enclave, all threads executing on that same logical core must perform an AEX, as described in Sect. 2.2. In order to avoid

information leakage stemming from memory access patterns inside enclaves, the lowermost 12 bits of the faulting address, stored in the CR2 registry are cleared. SGX instructions explicitly support batching up to 16 page evictions together at a time, thus curtailing the cost of AEX for each page fault inside an enclave.

2.4 Enclave Initialization

SGX allows the creation of multiple, mutually distrusting enclaves, on the same hardware instance. These can reside in either a single process' address space or multiple. To instantiate enclave system software the OS, on behalf of the application, invokes the ECREATE instruction. This causes the underlying microcode implementation to allocate a new EPC page for the SGX Enclave Control Structure (SECS), identifying each enclave and storing per-enclave operational metadata. Moreover, physical pages are mapped to enclave SECS through the EPCM structure. Before initialization is complete, each separate code and data segment must be added to enclave memory explicitly through the EADD instructions. Similarly, each TCS is added for each logical core expected to execute within the enclave. Once this process is complete the OS issues the EINIT instruction which finalizes initialization and compares the enclave measurement observed to the contents of the SIGSTRUCT. Upon completion, a launch token is generated by a special pre-provisioned enclave trusted by Intel, at which point the enclave is considered fully initialized. Once this process is completed, no further memory page allocations may happen. Intels revised specifications for SGX version 2 includes the possibility for dynamic paging support by means of the EEXTEND command. However, we refrain from further comment, as hardware supporting these features have not yet been released at the time of writing.

Inversely, during teardown of an enclave, the opposite operation is performed. The OS tags each page as invalid, by issuing the EREMOVE instruction. Prior to this, SGX verifies for each page that no threads attributed to that page are executing inside the enclave. Lastly, the SECS is destroyed once all pages referring to it through the EPCM are themselves deallocated.

2.5 Enclave Attestation

In order for applications to securely host privacy-sensitive software components on platforms outside of their administrative domain, we need to establish trust. This can be achieved through remote attestation, a process in which the remote party proves its correctness to the initiator. Assuming an enclave has been created and initialized as outlined above on an untrusted platform, the entity wishing to establish trust with this enclave issues a request for proof. The code inside this enclave then requests a *Quote* from the hardware, which consists of the enclave measurement, in addition to a signature from the hardware platform key. This quote is then sent to the requesting party which can themselves validate the measurement compared to the expected provisioned enclave. Lastly, the quote is sent to Intel for verification through their *Intel Attestation Service*, which validates the signature against their own private key. These two in

combination prove to the requesting party that the expected code and data segments are running on a valid SGX-enabled platform.

3 Experiments

The next generation of SaaS systems should be designed from the ground up to utilize trusted computing features in a performance optimal way. Therefore, we conduct a series of micro benchmark experiments on a SGX-enabled CPU to fully understand the micro architectural cost of trusted computing on commodity hardware. Our experimental setup consists of a Dell Optiplex workstation with an Intel Core i5-6500 CPU @ 3.20 GHz with four logical cores and 2×8 GB of DDR3 DIMM DRAM. Dynamic frequency scaling, Intel Speedstep and CStates are disabled throughout our experiments to avoid inaccuracies. We set the PRM size to its maximum allowed 128 MB to measure the peak theoretical performance of the platform. Our experiments ran on Ubuntu 14.04 using the open source kernel module by Intel implementing OS support for SGX[1]. Furthermore, this module has been modified with instrumentation in order to also capture the operational cost from the system perspective. Based on our knowledge regarding SGX, we have derived a set of benchmarks conjectured to capture core aspects of the trusted computing platform. It is worth noting that for all our experiments, more iterations did not yield a lower deviation. We attribute this to noise generated by the rest of the system that while subtle, becomes significant at fine-grained time intervals.

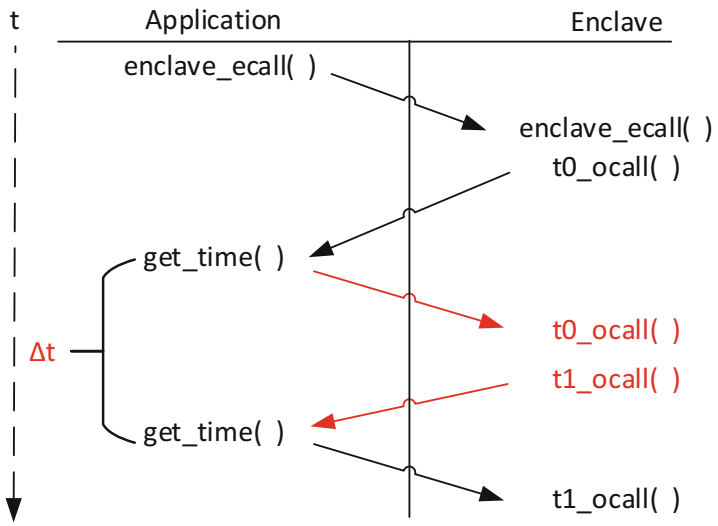


Fig. 1. Sequence of events involved in measuring time spent inside enclaves [7].

[1] https://github.com/01org/linux-sgx-driver.

The current generation of SGX does not support the use of the RDTSC instruction or any other native timing facilities inside enclaves. Intel has later released a microcode update to counter this problem, allowing for the RTDSC instruction to execute inside enclaves. We are however unsuccessful, at the time of writing, in obtaining a firmware update specific to our SKU through the correct OEM. Measurements performed throughout the experiments must therefore exit the enclave for each point in time. Consequently, all measurements therefore include the time taken to enter and exit the enclave, described as the sequence of events detailed in Fig. 1.

3.1 Entry and Exit Costs

With SGX, SaaS applications are able to influence the size of their Trusted Computing Base (TCB) by partitioning application logic between trusted and untrusted execution domains. In order to quantify any potential performance trade-off, we examine the associated cost of enclave transitions. An optimal application arrangement should conciser the following trade-off depending on the transition cost: A high cost of transition would necessitate a reduction in the overall amount of transitions and mediating this cost will increase the amount of logic residing within the enclave, thus expanding the TCB. A prominent example at one end of the spectrum is Heaven [8], in which an entire library OS is placed within a secure enclave. Furthermore, details in the Intel Software Developer Manual[2] suggest that the cost of entering an enclave should also factor in the cost of argument data copied as part of the transition into the enclave. Therefore, if the cost of data input to an enclave is high, only data requiring explicit confidentiality and trust should be placed within the enclave.

Figure 2 depicts the measured cost in millisecond latency, as a function of increasing buffer sizes. The cost of entering an enclave is observed to increase linearly with the size of the buffer input as the argument. It is worth mentioning that only buffer input to the enclave is considered. The experiment does not include output buffers or return values from enclaves.

Hosting a buffer inside enclave memory requires that the enclave heap is sufficiently large. Since enclave sizes are final after initialization, we set the heap size to be equally large for all iterations of the experiment. From the graph, we observe that the baseline cost of entering an enclave quickly becomes insignificant as the buffer size increases. This behavior is not surprising, as the overall cost includes the cost of copying memory into the enclaves which invokes the MEE for each page written to the enclave. A curious observation, however, is the fact that the baseline cost only increases linearly for buffers larger than 64 kB. This could be explained by enclaves less than 64 kB being fully provisioned into EPC memory at startup. Whereas for large buffers the cost may be attributed to lazily loaded enclave memory, triggering page faults during the buffer copy operation. This aspect is explored in detail in the following experiment.

[2] https://software.intel.com/en-us/articles/intel-sdm.

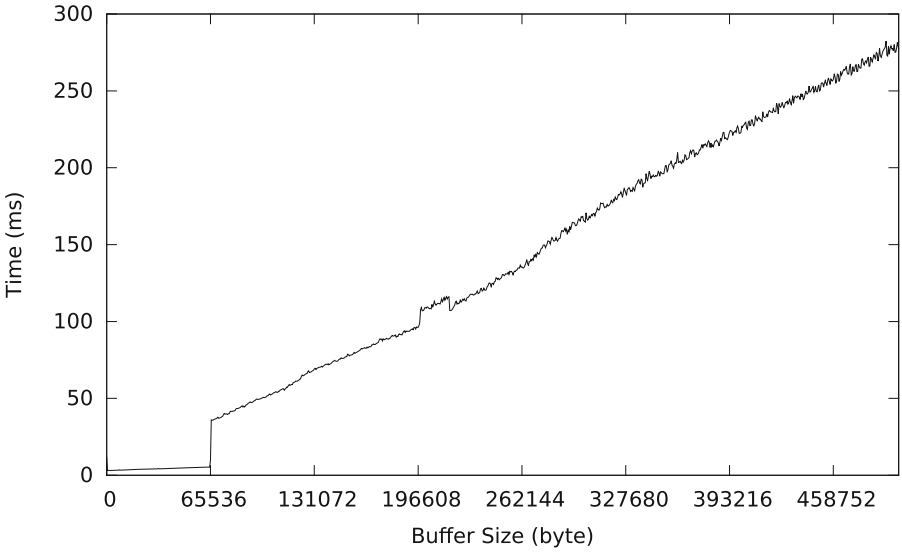


Fig. 2. Enclave transition cost as a function of buffer size [7].

3.2 Paging

Another aspect to consider in the application trade off between TCB and enclave transition cost, is the fact that an increase in TCB would cause an increase in PRM consumption. Moreover, as stated in Sect. 2.3, PRM is a fairly limited resource compared to regular memory and the depletion of this resource will cause system software to evict EPC pages more aggressively. As such, any application utilizing SGX should consider carefully the cost of enclave memory management, more specifically the cost of page swapping between EPC and regular DRAM. Figure 3 illustrates this overhead as observed by both the OS kernel and inside the enclave.

The y-axis is the discrete cost in nano seconds, while the x-axis is time elapsed into the experiment. The SGX kernel module has been instrumented to measure the latency of page eviction denoted by the red dots, and the total time spent in the page fault handler, represented by the black solid line.

From the enclave perspective, the green line denotes the user level instrumentation and represents time spent writing to a particular address in enclave memory. As mentioned in the experimental introduction, measurement primitives are unavailable inside enclaves, and all user level measurements therefore include the cost of entry and exit, including a 4 byte word as parameter input each way.

To induce page faults, the experimental enclave heap size is set to 256 MB, double that of the of the physical PRM size made available by hardware. Moreover, we invoke write operations on addresses located within each 4 kB page sequentially along the allocated memory address space inside the enclave.

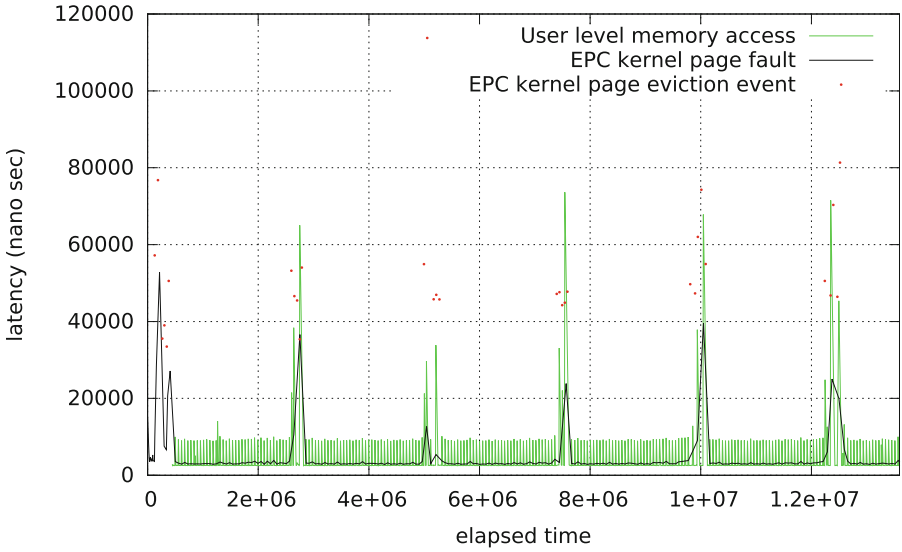


Fig. 3. Paging overhead in nano seconds as a function of time elapsed while writing sequentially to enclave memory [7]. (Color figure online)

Recall from Sect. 2 that all memory for a particular enclave must be allocated prior to initialization. We observe from Fig. 3 that prior to enclave startup, a cluster of page fault events occur at the beginning of the experiment, corresponding with our prior observations. The system is attempting to allocate memory for an enclave of 256 MB while only being physically backed by 128 MB of EPC memory.

The events occurring at user level can easily be correlated with the observations made in the page fault handler. For each increase in latency observed from inside the enclave, a corresponding cluster of evictions occur in the page fault handler. Moreover, the total time spent in the page fault handler coincides with the write overhead observed at user level. Parts of the overhead can be attributed to the fact that page faults cause AEX events to occur for each logical core executing within the enclave, as detailed in Sect. 2.

Moreover, we observe that the SGX kernel module is behaving conservatively in terms of page evictions, and is not exhausting EPC memory resources. As detailed in Sect. 2, the 12 lower bits of the virtual page fault address are cleared by SGX before exiting the enclave and trapping down to the page fault handler. Hence, system software is not able to make any algorithmic assumptions about memory access patterns to optimize page assignment. Furthermore, liveness challenge vector data might also be evicted out of EPC memory, causing a cascade of page loads to occur from DRAM. As a side note, this experiment only uses a single thread, and all page evictions only interrupt this single thread.

In light of the prior discovery, high performance applications should consider tuning the SGX page fault handler to their particular use case, given that the

application is able to predict a specific access pattern. Moreover, regardless of access pattern the SGX page fault handler should be optimized to allow exhaustive use of EPC, such that applications running inside enclaves may be less affected by page faults in high memory footprint scenarios.

The initial setup of enclaves will retain large amounts of the pages in EPC memory, alleviating the overhead of paging in certain situations. Moreover, this reduces the execution overhead caused by threads performing AEX. Given that the cost of enclave setup is still a large factor, by the prior statements, it might be advantageous for application developers to pre-provision enclaves.

3.3 Enclave Provisioning

Modular programming and componentized system organization are paradigms commonly used in modern distributed systems. Applications consisting of possibly multiple trust domains and third-party open source components should separate the unit of failure and trust to reduce the overall system impact.

By enabling the creation of mutually distrusting enclaves, SGX is able to support a modular application architecture. Section 2 explains how enclaves might communicate with the untrusted application through well defined interfaces, lending itself to compartmentalization of software into separate enclaves. To capture the cost of using SGX through the scope of a modular software architecture, Fig. 4 illustrates the cost in terms of provisioning latency as a function of enclaves created simultaneously for differently sized enclaves. We observe that the added cost of enclave creation increases linearly for all sized enclaves,

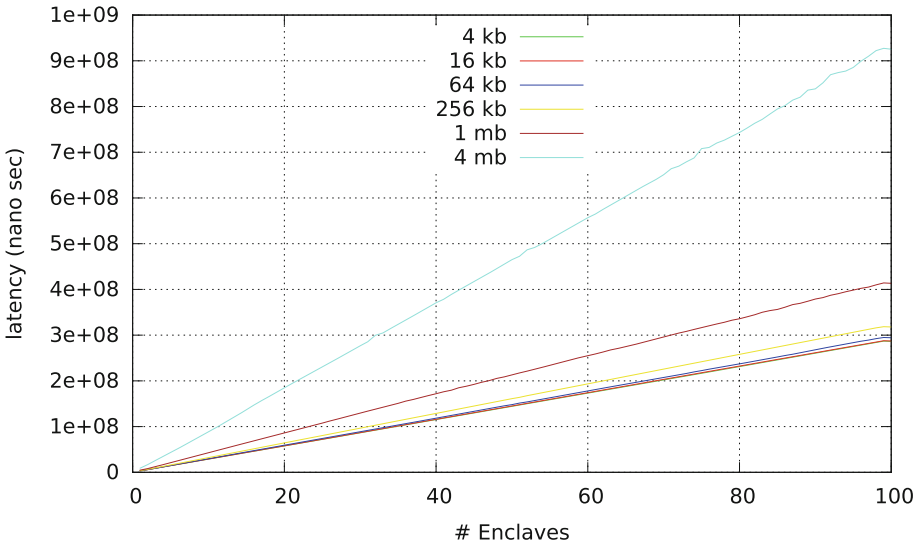


Fig. 4. Latency as a function of number of enclaves created simultaneously, for differing sizes of enclaves [7].

becoming significant for enclaves larger than 256 kB. As detailed in Sect. 2, enclaves are created by allocating each page of code and data to the enclave prior to initialization. During this experiment we observed a significant amount of page faults further attributing to the creation cost. This is expected as the size of enclaves combined with number of instances increases above that of the physically available PRM. Our observations about buffers less than 64 kB from Sect. 3.1 still stands, as we observe that the provisioning cost for enclaves less than 64 kB is nearly identical.

To offset the latency of creation for enclave instances, real-time applications should consider pre-provisioning them. However, as prior experiments show co-locating multiple enclaves in EPC memory might result in additional cost if the memory footprint is large enough.

3.4 Multithreading

The curation and analysis of large amounts of data use concurrency as a measure to speed up processing of data elements. This is especially true for *embarrassingly parallel* computations. One example is the *distinct count* aggregate operation, where a large corpus of data is sectioned into buckets and where each can be counted in parallel. Such computations require parallelism built into the runtime. Fortunately, SGX provides the ability to run multithreaded operations inside the same enclave. However, implementation details reveal that applications with high memory footprint might suffer from extensive page faults, which can act as a barrier and in the worst cases degrade performance significantly. Furthermore, as we argued earlier, applications with multiple tenants might want to isolate analytics execution into separate enclaves, and it is therefore important to consider how threads are delegated inside of enclaves.

To induce a high memory footprint we use the same technique as in Sect. 3.2, where we create an enclave which exceeds in size the amount of available physical PRM. We expect some performance degradation for multiple threads running on the same logical core executing within the enclave. When a page fault occurs, all threads running on the particular core must exit the enclave and block until the page fault is serviced. Our experiment therefore consists of two modes, one where we pin all threads to separate logical cores, and one where we pin all threads to a single core. Both experiments dedicate a single thread to interrogating every 4 kB page of the heap memory causing regular page faults to occur. Our test bench has 4 logical cores so our experiment runs a total of 4 threads simultaneously for both experiments. The remaining threads are just busy-waiting in a loop, measuring the time taken in each iteration. Figure 5 illustrates 4 threads pinned to 4 different cores where core 0 is interrogating memory and causing page faults to occur as illustrated in the green spikes. We observe that there is no co-dependency between threads, and the 3 remaining threads are not impacted by interrupts occurring on the former (Fig. 6).

Our second experiment demonstrates the opposite. We force all 4 threads to be pinned to a single logical core, and as a consequence we observe that thread

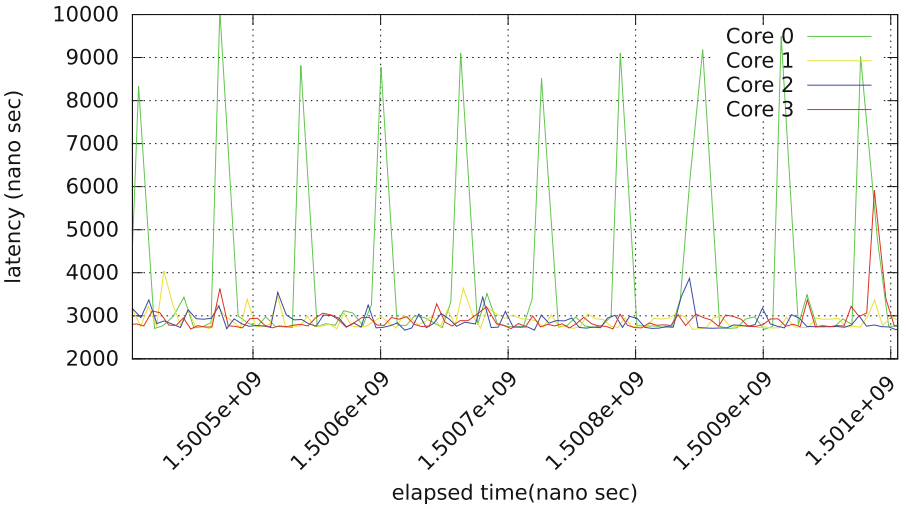


Fig. 5. Execution overhead for multiple threads running on separate logical cores, with page fault events occurring. (Color figure online)

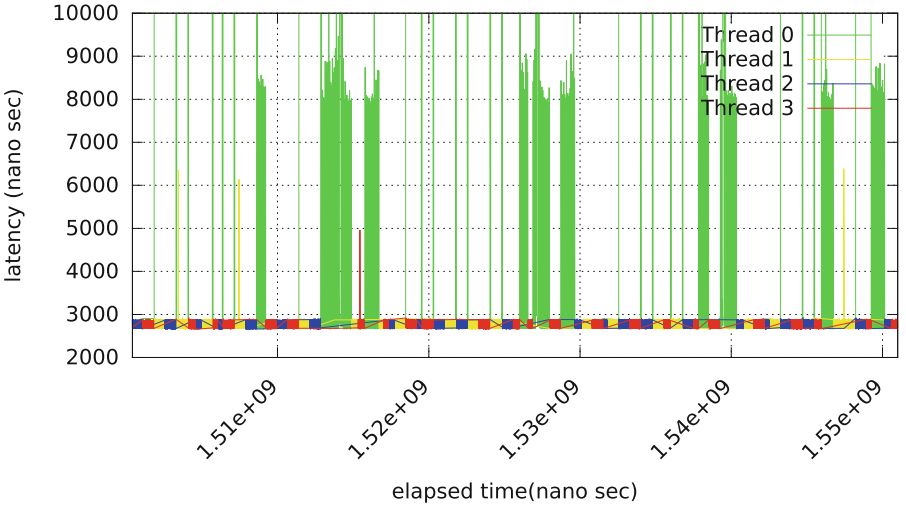


Fig. 6. Execution overhead for multiple threads pinned to a single core, with page fault events occurring.

0, who is causing interrupts to occur, is blocking all other threads from executing while servicing the costly page faults. It is worth noting that this is how threads behave in regular process address space when faced with a hardware interrupt. However, page faults are more costly to perform in enclave memory and more frequent as previous experiments show due to memory footprint constraints. Secondly, we observe that thread scheduling behaves differently as well.

Context switches between threads executing on the same logical core happens multiple magnitudes more infrequently than regular threads executing outside of enclaves. We theorize that this is a design choice when implementing enclave support, because interrupts in enclaves are especially costly. Any context switch would have to be induced by the timer hardware interrupt triggering the thread to exit the enclave, and so it makes sense increasing the scheduler time slices to amortize this cost.

4 Discussion

From the micro benchmarks detailed in Sect. 3, we pinpoint several performance traits of SGX that should be taken into consideration when designing trusted computing-enabled cloud services. We classify these individually as the cost of entering and exiting enclaves, the cost of data copying, the cost of provisioning new enclaves, the cost of memory usage and the cost of multithreaded execution.

Section 2 explained that the transitioning cost is uniform in terms of cost with respect to direction. Moreover, the most significant cost is attributed to the buffer size input as argument to the transition. More specifically, from Fig. 2 we observe a sharp rise in cost when buffer sizes are larger than 64 kB. We conjecture that this is an architectural boundary, where encalves are pre-provisioned, by default, with a given number of pages. Future iterations of SGX may alter this behavior, opting for an increase in pre-provisioned pages. Our principles therefore state:

The Size Principle. *The size of an enclave should not exceed the architecturally determined pre-provisioned memory resources.*

The Cohesion Principle. *Applications should partition its functional components to minimize data copied across enclave boundaries.*

Following the latter principle, a possible component architecture would be to co-locate all application logic into a single, self-sufficient enclave. Haven [8], is a prominent example of this approach. By means of a library OS, a large part of the system software stack is placed within a single enclave, reducing the interface between trusted and untrusted code. However, this approach directly contradicts the observation made in Sect. 3.2 regarding the cost of having a large memory footprint. Since the EPC is a limited resource, the SGX page fault handler promptly pages out enclave memory not being used. However, the paging experiment demonstrates that the available pool of EPC memory is not exhausted, even in the presence of high memory contention. As detailed in Sect. 3.2, the faulting address is not provided as part of the page fault event and the page fault handler is therefore not able to make any assumptions about the memory access patterns. We therefore state that:

The Access Pattern Principle. *Prior knowledge about application's memory consumption and access pattern should be used to modify the SGX kernel module in order to reduce memory page eviction.*

Our experiments have demonstrated that enclave creation is costly in terms of provisioning latency. By pre-provisioning enclaves whenever usage patterns can be predicted, the application is able to hide some of this cost. However, once used, an enclave might be tainted with secret data. Recycling used enclaves to a common pool can therefore potentially leek secrets from one domain to the next; invalidating the isolation guarantees. We therefore state that:

The Pre-provisioning Principle. *Application authors that can accurately predict before-the-fact usage of enclaves should pre-provision enclaves in a disposable pool of resources that guarantees no reuse between isolation domains.*

The cost of enclave creation must also factor in the added baseline cost of storing metadata structures associated with each enclave in memory. Provisioning enclaves must at least account for its SECS, one TCS structure for each logical core executed inside an enclave, and one SSA for storing secure execution context for each thread. [6] details that to simplify implementation, most of these structures are allocated at the beginning of an EPC page wholly dedicated to that instance. Therefore, enclaves executing on 4 logical cores may have 9 pages (34 kB) in total allocated to it, excluding code and data segments. Applications should consider the added memory cost of separate enclaves in conjunction with the relative amount of available EPC. Furthermore, to offset the cost of having multiple enclaves, application authors should consider security separation at a continuous scale. Some security models might be content with role based isolation, rather than call for an explicit isolation of all users individually. We therefore state that:

The Isolation Principle. *Application authors should carefully consider the granularity of isolation required for their intended use, as a finer granularity includes the added cost of enclave creation.*

Executing multiple threads from the same core inside a single enclave degrades the concurrent performance by blocking execution when servicing a page fault. Although regular non-enclave execution behaves similarly, the overhead associated with enclave page faults becomes significant when memory footprint increases. Moreover, latency critical applications will suffer because of the increased time slices of thread interrupts initially thought to amortize the cost of exiting enclaves when switching contexts. From this we deduce that the number of threads executing inside enclaves should never exceed the logical core count for a given system. We therefore establish the following principle:

The Affinity Principle. *Applications should not affinitize multiple threads to the same core.*

Section 3.1 demonstrates the cost of transitioning into and out of an enclave, and it becomes evident that to reduce the transitioning overhead threads should be pinned inside enclaves. Enclave threads should rather transport data out of the enclave through writing to regular DRAM and similarly poll for incoming data. We therefore state:

The Pinning Principle. *Application authors should pin threads to enclaves to avoid costly transitions.*

The prior statements lead us to the following principle:

The Asynchrony Principle. *All execution inside enclaves should be asynchronous.*

Threads should be pinned inside enclaves to amortize transition cost and total thread count should not exceed logical core count. Application authors must therefore be diligent in terms of assigning threads to enclaves. Applications might further isolate contexts based on either user or tenant in different mutually distrusting enclaves, each of which requires a dedicated thread. Core logic executing inside enclaves should remain responsive at all time, servicing both incoming requests and processing data. We therefore state that rather than allocating multiple threads to the same enclave, all execution should be fully asynchronous. This furthermore has the added benefit of high resource utilization improving overall application performance.

At the time of writing, the only available hardware supporting SGX are the Skylake generation Core chips with SGX version 1. Our experiments demonstrate that paging has a profound impact on performance and a natural follow-up would be to measure the performance characteristics of dynamic paging support proposed in the SGX version 2 specifications.

SGX supports attestation of software running on top of an untrusted platform, by using signed hardware measurements to establish trust between parties. For future efforts it would be interesting, in light of the large cost of enclave transition demonstrated above, to examine the performance characteristics of a secure channel for communication between enclaves.

5 Related Work

Several previous works quantify various aspects of the overhead associated with composite architectures based on SGX. Haven [8] implements shielded execution of unmodified legacy applications by inserting a library OS entirely inside of SGX enclaves. This effort resulted in architectural changes to the SGX specification to include, among other things, support for dynamic paging. The proof-of-concept implementation of Haven is only evaluated in terms of legacy applications running on top of the system. Furthermore, Haven was built on a pre-release emulated version of SGX, and the performance evaluation is not directly comparable to real world applications. Overshadow [9] provides similar capabilities as Haven, but does not rely on dedicated hardware support.

SCONE [10] implements support for secure containers inside of SGX enclaves. The design of SCONE is driven by experiments on container designs pertaining to the TCB size inside enclaves, in which, at the most extreme an entire library OS is included and at the minimum a stub interface to application libraries. The evaluation of SCONE is much like the evaluation of Haven, based on running legacy applications inside SCONE containers. While Arnautov et al. [10] make the same conclusions with regards to TCB size versus memory usage and enclave transition cost as Baumann et al. [8], the paper does not quantify this cost. Despite this, SCONE supplies a solution to the entry-exit problem we outline in Sect. 3, where threads are pinned inside the enclave, and do not transition to the outside. Rather, communication happens by means of the enclave threads writing to a dedicated queue residing in regular DRAM memory. This approach is still, however, vulnerable to threads being evicted from enclaves by AEX caused by an Inter Processor Interrupt (IPI) as part of a page fault.

Costan and Devadas [6] describe the architecture of SGX based on prior art, released developer manuals, and patents. Furthermore, they conduct a comprehensive security analysis of SGX, falsifying some of its guarantees by explaining in detail exploitable vulnerabilities within the architecture. This work is mostly orthogonal to our efforts, yet we base most of our knowledge of SGX from this treatment on the topic. These prior efforts lead Costan et al. [11] to implement Sanctum, an alternative hardware architectural extension providing many of the same properties as SGX, but targeted towards the Rocket RISC-V chip architecture. This paper evaluates its prototype by simulated hardware, against an insecure baseline without the proposed security properties. McKeen et al. [12] introduce dynamic paging support to the SGX specifications. This prototype hardware was not available to us.

Ryoan [13] attempts to solve the same problems outlined in the introduction, by implementing a distributed sandbox facilitating untrusted computing on secret data residing on third-party cloud services. Ryoan proposes a new request oriented data-model where processing modules are activated once without persisting data input to them. Furthermore, by remote attestation, Ryoan is able to verify the integrity of sandbox instances and protect execution. By combining sandboxing techniques with SGX, Ryoan is able to create a shielding construct supporting mutually distrust between the application and the infrastructure. Again, Ryoan is benchmarked against real world applications, and just like other prior work, does not correctly quantify the exact overhead attributed to SGX primitives. Furthermore, large parts of its evaluation is conducted in an SGX emulator based on QEMU, which has been retrofitted with delays and TLB flushes based upon real hardware measurements to better mirror real SGX performance. These hardware measurements are present for EENTRY and EEXIT instructions, but do not attribute the cost of moving argument data into and out of enclave memory. Moreover, Ryoan speculates on the cost of SGX V2 paging support, although strictly based on emulated measurements and assumptions about physical cost.

ARM TrustZone is a hardware security architecture that can be incorporated into ARMv7-A, ARMv8-A and ARMv8-M on-chip systems [14,15]. Although the

underlying hardware design, features, and interfaces differ substantially to SGX, both essentially provide the same key concepts of hardware isolated execution domains and the ability to bootstrap attested software stacks into those enclaves [16]. However, the TrustZone hardware can only distinguish between two execution domains, and relies on having a software based trusted execution environment for any further refinements.

6 Conclusion

SaaS providers are increasingly storing personal privacy-sensitive data about customers on third-party cloud providers. Moreover, companies monetize this data by providing personalized experiences for customers requiring curation and analysis. This dilution of responsibility and trust is concerning for data owners as cloud providers cannot be trusted to enforce the, often government mandated, restrictive usage policies which accompany privacy-sensitive data.

Intel SGX is part of a new wave of trusted computing targeting commodity hardware and allowing for the execution of code and data in trusted segments of memory at close to native processor speed. These extensions to the x86 ISA guarantee confidentiality, integrity and correctness of code and data residing on untrusted third-party platforms.

Prior work demonstrates the applicability of SGX for complete systems capable of hosting large legacy applications. These systems, however, do not quantify the exact micro architectural cost of achieving confidentiality, integrity and attestation for applications through the use of trusted computing. This paper has evaluated the cost of provisioning, data copying, context transitioning, memory footprint and multi-threaded execution of enclaves. From these results we have distilled a set of principles which developers of trusted analytics systems should use to maximize the performance of their application while securing privacy-sensitive data on third-party cloud platforms.

Acknowledgments. This work was supported in part by the Norwegian Research Council project numbers 263248/O70 and 250138. We would like to thank Robbert van Renesse for his insights and discussions, and anonymous reviewers for their useful insights and comments.

References

1. Gjerdrum, A.T., Johansen, H.D., Johansen, D.: Implementing informed consent as information-flow policies for secure analytics on eHealth data: principles and practices. In: IEEE Conference on Connected Health: Applications, Systems and Engineering Technologies: The 1st International Workshop on Security, Privacy, and Trustworthiness in Medical Cyber-Physical System, CHASE 2016. IEEE (2016)
2. Johansen, H.D., Birrell, E., Van Renesse, R., Schneider, F.B., Stenhaug, M., Johansen, D.: Enforcing privacy policies with meta-code. In: 6th Asia-Pacific Workshop on Systems, p. 16. ACM (2015)

3. Osborn, J.D., Challener, D.C.: Trusted platform module evolution. Johns Hopkins APL Tech. Digest **32**, 536–543 (2013)
4. TCG Published: TPM main part 1 design principles. Specification Version 1.2 Revision 116, Trusted Computing Group (2011)
5. Anati, I., Gueron, S., Johnson, S., Scarlata, V.: Innovative technology for CPU based attestation and sealing. In: 2nd International Workshop on Hardware and Architectural Support for Security and Privacy, vol. 13 (2013)
6. Costan, V., Devadas, S.: Intel SGX explained. In: Cryptology ePrint Archive (2016)
7. Gjerdrum, A.T., Pettersen, R., Johansen, H.D., Johansen, D.: Performance of trusted computing in cloud infrastructures with Intel SGX. In: 7th International Conference on Cloud Computing and Services Science, CLOSER 2017. SCITEPRESS (2017)
8. Baumann, A., Peinado, M., Hunt, G.: Shielding applications from an untrusted cloud with Haven. In: 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2014). USENIX Advanced Computing Systems Association (2014)
9. Chen, X., Garfinkel, T., Lewis, E.C., Subrahmanyam, P., Waldspurger, C.A., Boneh, D., Dwoskin, J., Ports, D.R.: Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. In: 13th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XIII, pp. 2–13. ACM, New York (2008)
10. Arnautov, S., Trach, B., Gregor, F., Knauth, T., Martin, A., Priebe, C., Lind, J., Muthukumaran, D., O'Keeffe, D., Stillwell, M.L., Goltzsche, D., Eyers, D., Kapitza, R., Pietzuch, P., Fetzer, C.: Scone: secure Linux containers with Intel SGX. In: 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2016), GA, pp. 689–703. USENIX Association (2016)
11. Costan, V., Lebedev, I., Devadas, S.: Sanctum: minimal hardware extensions for strong software isolation. In: USENIX Security, vol. 16, pp. 857–874 (2016)
12. McKeen, F., Alexandrovich, I., Anati, I., Caspi, D., Johnson, S., Leslie-Hurd, R., Rozas, C.: Intel® software guard extensions (Intel® SGX) support for dynamic memory management inside an enclave. In: Hardware and Architectural Support for Security and Privacy 2016, p. 10. ACM (2016)
13. Hunt, T., Zhu, Z., Xu, Y., Peter, S., Witchel, E.: Ryoan: a distributed sandbox for untrusted computation on secret data. In: 12th USENIX Conference on Operating Systems Design and Implementation, OSDI 2016, pp. 533–549. USENIX Association, Berkeley (2016)
14. Ngabonziza, B., Martin, D., Bailey, A., Cho, H., Martin, S.: Trustzone explained: architectural features and use cases. In: 2016 IEEE 2nd International Conference on Collaboration and Internet Computing (CIC), pp. 445–451. IEEE (2016)
15. Shuja, J., Gani, A., Bilal, K., Khan, A.U.R., Madani, S.A., Khan, S.U., Zomaya, A.Y.: A survey of mobile device virtualization: taxonomy and state of the art. ACM Comput. Surv. (CSUR) **49**, 1 (2016)
16. Pettersen, R., Johansen, H.D., Johansen, D.: Trusted execution on ARM TrustZone. In: 7th International Conference on Cloud Computing and Services Science (CLOSER 2017) (2017)

Paper IV

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.

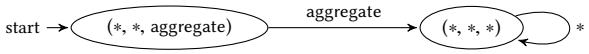


Figure 1: Example Avenance policy.

```
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.

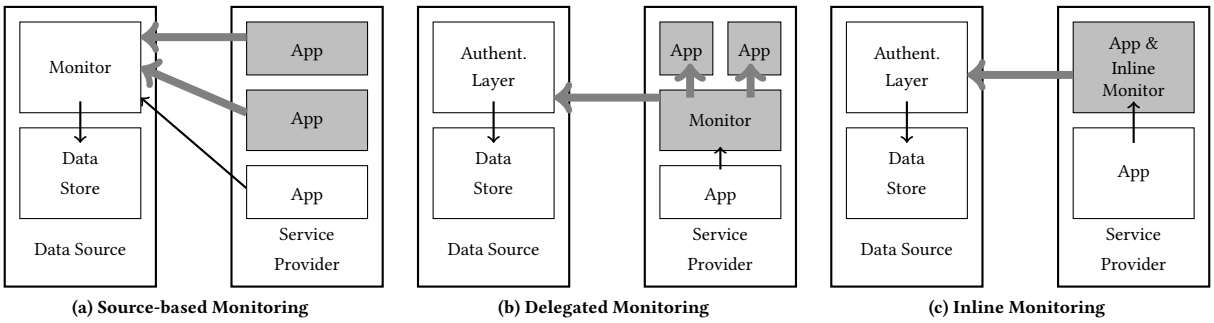


Figure 3: An overview of the different architecture designs. The direction of each arrow indicates which principal instigates communication between two components of the system. SGX enclaves are shown in gray; wide gray arrows indicate that a program has authenticated using an SGX report (local attestation)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

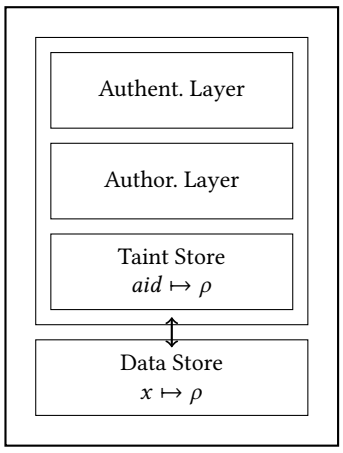


Figure 4: Detailed design for a data source that implements source-based monitoring.

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

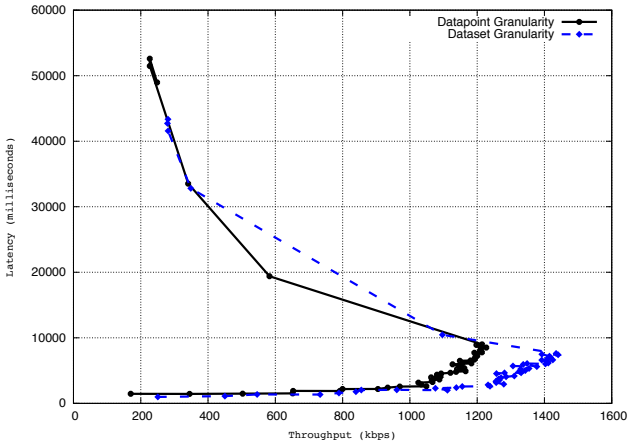


Figure 5: Latency and throughput of a data source with source-based monitoring as a function of the number of concurrent requests, ranging from 1 to 50 concurrent requests. The solid black line shows performance when the data source enforces datapoint-granularity policies and the dashed blue line shows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

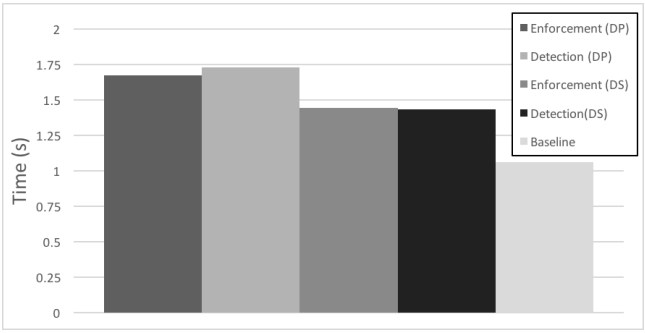


Figure 6: Performance of the PMSys averaging function with source-based monitoring.

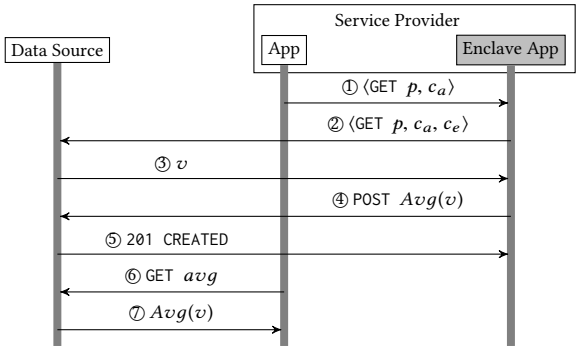


Figure 7: Protocol for the PMSys averaging function in a source-based monitoring architecture.

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.

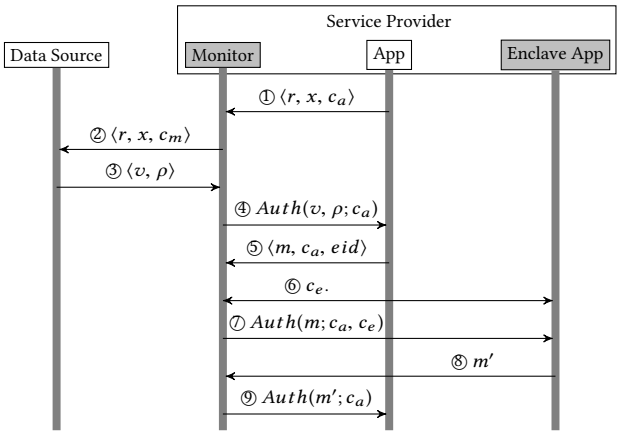


Figure 8: Example interactions with a delegated monitor.

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.

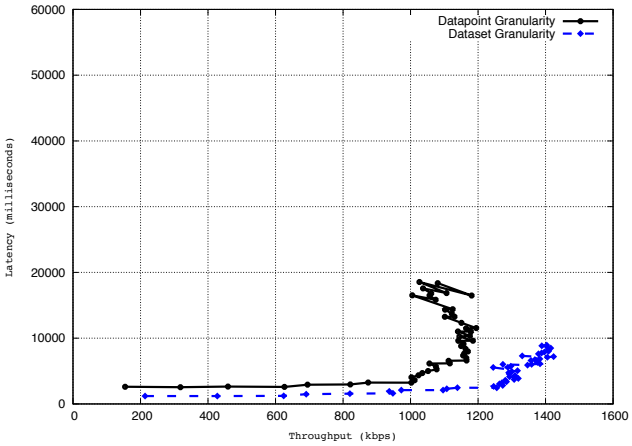


Figure 9: Latency and throughput of a data source with client-side monitoring (delegated monitoring or inline monitoring).

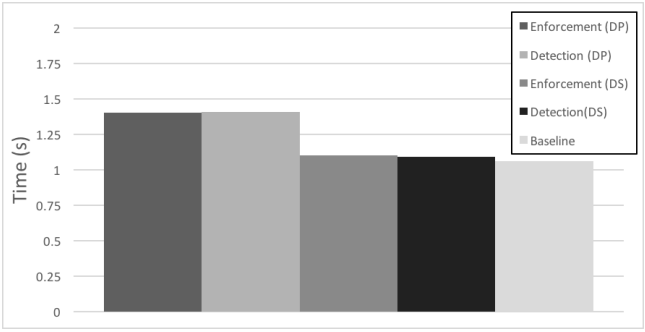


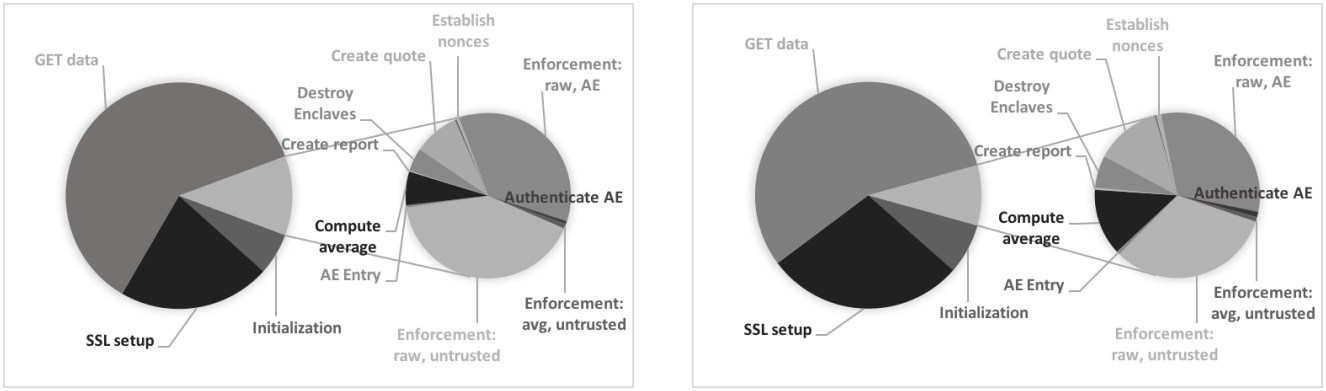
Figure 10: Performance of the PMSys averaging function with delegated monitoring.

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,

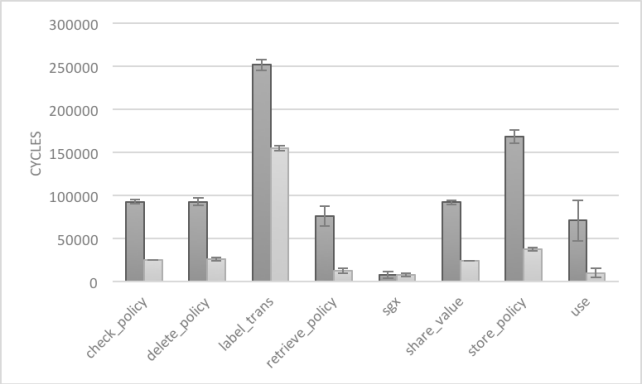


Figure 13: Performance of inline monitoring library calls. Results with logging are in dark gray; results with prevention-based enforcement are in light gray.

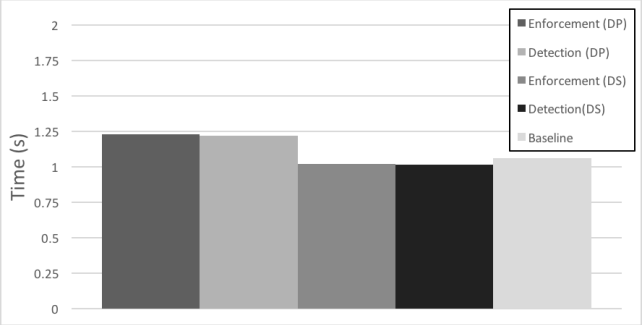


Figure 14: Performance of the PMSys averaging function with inline monitoring.

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.

Paper V