UiT NORGES ARKTISKE UNIVERSITET

Faculty of Science and Technology
Department of Computer Science

SecureCached

Secure Caching with the Diggi frawework

—

Helge Hoff

INF-3981: Master in Computer Science
June 1 2018

“Slask!”
–Asle Hoff

Abstract

Caching services are vital for the performance of large-scale web services running in the cloud. However, placing sensitive data in caching services, implicitly includes all components of the cloud infrastructure that can be exploited. Therefore, end-users place their trust in the entire security stack of their service providers. In order to achieve confidentiality and integrity of sensitive data residing in cache services, the cloud infrastructure must be removed from the set of trusted components. This has led to a wide adoption of hardware-assisted Trusted Execution Environments (TEEs), protecting user-level software from higher-privileged system software.

The capabilities of TEEs do not support running legacy applications out-of-the-box. Many prominent frameworks for TEEs have been developed to achieve applicability through providing common programming abstractions. However, these frameworks focus on providing native Linux services for TEEs, which increases the probability for a trusted software component to be exploited. Diggi is one such framework, utilizing Intel's Software Guard Extension (SGX) trusted computing infrastructure to provide secure execution. Diggi differs from other framework for TEEs by implementing simplified abstraction for creating distributed cloud applications. Moreover, by employing logically separated tasks split into multiple units of application code, Diggi allows moving parts of the application code and data into a TEE, like, for instance, a caching service. This allows to drastically reduce the set of trusted components in a system, and only include the parts that require strong security guarantees.

This thesis describes the introduction of a modified memcached implementation, called SecureCached, to the Diggi framework. We demonstrate the feasibility of having a distributed cache deployed in a trusted execution environment.

Acknowledgements

First and foremost I want to thank my ever so patient supervisors, Lars Brenna and Anders Gjerdrum, for their help and guidance.

I want to thank my partners in crime at the office Kim Hardtvedt Andreassen, and Christoffer Hansen, for keeping me leveled throughout this endeavor. I also want to express my gratitude to Jon Foss Mikalsen for sticking with my nonsense throughout the years at uni, and especially for his unending appetite[1].

Lastly, a special mention to Vegard Sandengen for convincing us all to *Challenge the status quo!*.

1. www.reddit.com/r/picturesofjoneating

Contents

List of Figures

List of Tables

List of Code Listings

List of Abbreviations

ABI	Application binary interface
AES	AES
AESM	Application Enclave Service Manager
AEX	Asynchronous Enclave Exit
AID	Agent Identifer
API	application programming interface
CA	Certificate Authroity
CPU	central processor unit
DMA	direct memory access
EPC	Enclave page cache
EPCM	Enclave Page Cache Metadata
Glibc	GNU C Library
IOT	Internet of Things
IPI	Inter Processor Interrupt
ISA	Instruction Set Architecture
LKL	Linux Kernel Library
LOC	Lines of Code

MEE	Memory Encryption Engine
OS	operating system
POSIX	Portable Operating System Interface
PRM	Processor Reserved Memory
RPC	Remote Procedure Call
RSA	Rivest-Shamir-Adleman
RTT	Round-trip Time
SCONE	Secure Linux Containers
SDK	Software Development Kit
SE	Secure Element
SECS	SGX Enclave Control Structure
SEV	Secure Encrypted Virtualization
SGX	Software Guard Extension
smc	Secure Monitor Call
SME	Secure Memory Encryption
SMM	System Management Mode
SOC	System-On-Chip
SOF	Shared Object File
STL	Standard Template Language
SUVM	Secure User-managed Virtual Memory
SVM	Secure Virtual Machine
TC	Trusted Computing

TCB	Trusted Computing Base
TCG	Trusted Computing Group
TCP	Trusted Computing Platform
TCP	Transmission Control Protocol
TCS	Thread Control Structure
TEE	Trusted Execution Environment
TLS	Transport Level Security
TPM	Trusted Platform Module
TTL	Time To Live
TZ	TrustZone
UDP	User Datagram Protocol
VM	virtual machine
VMM	virtual machine monitor
YCSB	Yahoo! Cloud Serving Benchmark

/1 Introduction

Users entrust a wide array of cloud services to manage their sensitive data, and expect it to remain confidential. This implies that end-users place their trust in the entire security stack of their service providers. However, as the complexity of cloud-based software architecture increases, the confidentiality and integrity of data becomes harder to manage [1, 2, 3, 4].

Cloud-based software architectures such as web services are compromised of a hierarchy of services. In in order to meet the scalability demands of large-scale web services, caches are used to accelerate performance. However, placing sensitive data in caching services, implicitly includes all components of the cloud infrastructure in its Trusted Computing Base (TCB). Privileged software components such as virtual machine monitors (VMMs), hypervisors, and host operating systems (OSs), provide protection from other cloud services running on the hardware. However, they do not provide protection from potentially untrusted cloud providers with root access. This has led to a wide adoption of hardware-assisted Trusted Execution Environments (TEEs), protecting user-level software from higher-privileged system software. TEEs enable protection of code and data by running applications on a secure area of the main processor, allowing applications to exclude cloud infrastructures from the TCB. The applicability of these TEEs are further strengthened by commodity hardware support from many vendors: Arm's TrustZone [5], Intel's Software Guard Extension (SGX) [6], and AMD's Secure Encrypted Virtualization (SEV) [7].

Most TEEs implement the capabilities required for a fully trusted system:

verifiable execution of code snippets (remote attestation), protection of cryptographic keys, and sealed storage. Despite being useful properties for secure execution, they do not facilitate a versatile runtime for real world applications. To achieve applicability for TEEs they need to support a rich array of applications: e.g, language runtimes, distributed caches, and web servers. Many frameworks for TEEs have been developed to achieve applicability through providing common programming abstractions [8, 9, 10, 11]. However, these frameworks focus on providing native Linux services for TEEs, which arguably increases the probability for a trusted software component to be exploited. Reducing the TCB is therefore a pivotal part of guaranteeing safe and correct execution of a system.

Diggi is a TEE application framework for distributed cloud applications, like, for instance, caching services. As well as providing common programming abstractions, Diggi differs from previous frameworks by providing a simplified abstraction for application developers to create trusted distributed applications. By employing logically separated tasks split into multiple units of application code, Diggi allows moving parts of the application code and data into a TEE. Instead of executing monolithic legacy applications within a TEE, breaking an application into smaller logical pieces of software allows to drastically reduce the TCB.

Diggi implements the execution of sensitive code and data in trusted agents. These are supported by Intel's implementation of protected memory execution, referred to as enclaves. Enclaves are restricted to only run in user-mode (Ring 3), and are unable to issue system calls. To enable well know abstraction and functionality, Diggi implements user-level abstractions similar to those in an operating system: user-level scheduler, Portable Operating System Interface (POSIX) system call interface , communication primitives, and I/O encryption. Diggi agents must be implemented to accommodate the SGX architecture, and porting an application to the runtime does not work out-of-the-box. Diggi differs from other TEE frameworks, as it is specifically designed for distributed computiation. To evaluate the feasibility of running a distributed caching service on Diggi without sacrificing performance or functionality, this thesis will port a distributed caching service, Memcached, into the Diggi runtime. Using Memcached may allow Diggi to place sensitive inside a TEE, and also move cached data out of a security domain closer to computation. Thus, this thesis introduces SecureCached , a Memcached clone modified to run in Diggi. We also contribute several additions to the Diggi runtime to replace those OS services Memcached otherwise relies on.

1.1 Thesis Statement

This thesis shall investigate the properties and limitations of using the Diggi secure distributed application framework to accelerate performance in privacy sensitive web services. Specifically, the work will include evaluating the feasibility of porting an existing state-of-the-art caching system to run within the Diggi runtime.

We conjecture that Memcached falls within that category, therefore, our thesis is:

> *The Memcached codebase can be modified to run within the Diggi Library OS.*

1.2 Scope & Limitations

This thesis aims to address the performance of the primitives in the Diggi rungime and how its OS-services affects the performance of Memcached. Therefore, this thesis will not address the security of our solution.

While Memcached natively exposes a significant application programming interface (API), we shall evaluate functionality by requiring that the prototype can successfully run the Yahoo! Cloud Serving Benchmark (YCSB) benchmark.

1.3 Context

This thesis is written in the context of the Corpore Sano Center [1]. The center works interdisciplinary, in the cross-section of computer science, sport science and medicine, with life-sciences research and innovation. Specifically, it focuses on technological innovations in mobility, cloud computing, big medical data, and the Internet of Things (IOT).

Over the years, the Corpore Sano center has done extensive research on distributed systems. Mobile agents that are able to migrate between hosts, as a part of the TACOMA project [12, 13, 14], and StormCast [15], a distributed artificial intelligence application for weather monitoring. We also developed Fireflies [16], a Byzantine fault-tolerant full membership protocol, capable of operat-

1. http://www.corporesano.no/

ing in the presence of malicious members. Further, we built FirePatch [17], a secure software patch dissemination system, building on the intrusion tolerant network, Fireflies. With the goal of preventing an adversary from delaying the dissemination of critical security patches in distributed systems.

The center has also contributed to research in the security domain. [18] conducted extensive performance evaluation of Intel SGX, and provides recommendations for developing applications on the architecture. More into the privacy domain, the center shows a mechanism for flexible discretionary access control in computing infrastructures [19], and enforcing privacy policies for shared data [20].

Within sport science, the group has contributed to research in video analysis by creating a real-time system for soccer analytics, Bagadus [21], accommodated by a large-scale search based video system, DAVVI [22]. Corpore Sano is also further involved in soccer analytics, and have developed Muithu [23], an event-based tagging system where coaches can tag on-field events as they occur.

The Corpore Sano center also developed a novel omni-kernel architecture permitting fine-grained resource control by pervasive monitoring and scheduling [24].

1.4 Methodology

Task force on the Core of computer science [25] define three major paradims in the area of computing:

Theory rooted in mathematics. An iterative process consisting of four steps. First is to define an object of study, from that find and hypothesis, test it, and then interpret whether the hypothesis is true.

Abstraction construct a model from an hypothesis and make predictions, design experiments and analyze the result.

Design rooted in engineering. Construct a system to solve a given problem by stating the requirements of a system and its specifications. Then design and test the system.

This thesis follows the systems research methodology and is rooted, to some degree, in all the three paradigms. We build a system in order to solve a given problem that we seek to address. Prior to development, we state the requirements and specification. The system is tested throughout development to assure that it follows the requirements. If the constructed system is proven to meet the requirements, we design experiments to validate our hypothesis.

1.5 Outline

The remainder of this thesis is structured as follows.

Chapter 2 details the Intel SGX, its architecture and isolation mechanisms. In addition, the chapter describes previous work that aims to support applicable applications inside enclaves and how the different approaches tries to overcome the performance penalties that SGX imposes.

Chapter 3 presents the Diggi Library OS, its architecture, and it supports a simple programming abstraction for trusted components. Specifically, how Diggi achieves message passing, system call support, and POSIX compatibility.

Chapter 4 covers the choices of design taking into account the features in Diggi and the OS-service dependencies of Memcached, and describes the implementation of SecureCached.

Chapter 5 details the evaluation of SecureCached.

Chapter 6 concludes this thesis, and lists proposal for future work.

2 Background

This chapter presents the functionality and concepts that are relevant to trusted execution, and the architecture of memcached. Section 2.1 introduces the idea of trusted comuting. Section 2.2 describes implementations of a TEE. Section 2.3 presents the Intel SGX architecures. Section 2.4 focuses on application frameworks for intel SGX. Section 2.5 describes the architecture of memcached.

2.1 Trusted Computing

Trusted Computing (TC) is a technology developed and promoted by the Trusted Computing Group (TCG) consortium [26], as the successor to the Trusted Computing Platform (TCP). It is an effort to promote trust and security in the personal computing domain. The term is rooted from the field of *trusted systems*, and is a set of specifications that address the requirements for a trusted system. The initial goal of the consortium was to develop the Trusted Platform Module (TPM), a standardization for secure cryptoprocessors. The TPM is a secure chip isolated from the processing system with cryptographic capability. Features of a TPM includes secure generation of cryptographic keys, remote attestation and enabling computer programs to authenticate hardware devices. It can store passwords, certificates, and encryption keys to authenticate the platform, and ensure that the platform remains trustworthy. TPMs are not only used in the personal computing domain, but in other devices such as mobile phones, and network equipment. The Secure Element (SE) is a specification much like the TPM, targeting use-cases such as mobile payment by using the same technologies as the TPM. However, these features are formalized beyond the TPM, and extends beyond hardware support [27, 28].

The Trusted Computing Group (TCG) defines Trusted Computing (TC) by six capabilities required to have a fully trusted system:

Endorsement Key Immutable RSA key used for attestation and encryption of sensitive data.

Secure input and output Falls together with the former. Principle of securing communication channels in and out of a system.

Protected Execution/Memory Curtaining Provide fully isolated areas of memory on which code execute.

Remote Attestation Enabling authorized parties to remotely detect unauthorized changes to software.

Sealed Storage Coupling storage of sensitive data to hardware keys.

Trusted Third Party Attesting remote parties through a trusted third party, much like the role of a Certificate Authroity (CA).

Now most trusted hardware devices implement these concepts, to a varying degree, to enable a trusted system for multiple scenarios.

2.2 Trusted Execution Environment

A TEE is a set of hardware technologies for TC and was designed to enrich previously defined trusted platforms, such as the TPM. GlobalPlatform [1], an industry association initially devoloped the specifications for a TEE. There are multiple definitions of a TEE, however, we follow the definition from GlobalPlaform [29]. They define a TEE as a secure area of the main processor ensuring that sensitive data is stored, processed and protected in an isolated, trusted environment [30]. It enables secure execution for trusted applications by providing protected execution of authenticated code, integrity of runtime states (e.g CPU registers and memory), and remote attestation. As opposed to TPM or the SE, TEE is not physically isolated from the processing system, and offers a large amount of accessible memory and high processing speeds. There are

1. https://www.globalplatform.org/

many implementations of TEE, and their level of security and performance vary accross hardware vendors. This section will only address TEEs implemented in commodity hardware.

ARM TrustZone (TZ) is a security extension to the ARM System-On-Chip (SOC), which can be used to establish trusted components for mobile applications [31]. The processor can execute instructions in two different modes: *normal world*, and *secure world*. Unstrusted code runs in the *normal world*, while secure services are executed in the *secure world*. The two worlds have physically separate addressable memory regions and different privilege levels. The processor can only execute in one world at a time, and to execute code in another world is done by issuing a special instruction called the Secure Monitor Call (smc). System developers are able to instruct which devices are accessible from the two worlds. A special bit, the Non-Secure bit, which determines which world the processor is currently running in such that hardware interrupts and bus access to peripheral devices are trapped directly into the respective worlds.

Secure Encrypted Virtualization (SEV) was introduced by AMD to address the security of their Secure Virtual Machine (SVM) virtualization technology. SEV is an extension to AMD's memory encryption technology: Secure Memory Encryption (SME). It allows virtual machines (VMs) to obtain a unique AES (AES) encryption key from the SME which is used to encrypt the contents of the guest VM. This approach hides the contents of a guest VM from the hypervisor, enabling secure data transfer through the hypervisor to the guest VM. Moreover, the hypervisor will no longer be able to inspect or alter any guest VM's code or data. The SEV technology have been subject to attacks [32] in which execution context is disclosed by observing cache operations. Even though this has been addressed by AMD, the SEV does not support a TEE to the same extent as SGX or TZ.

2.3 Intel Software Guard Extensions (SGX)

Intel's SGX is an extention to the x86-x64 Instruction Set Architecture (ISA) designed to increase the security of applications [6]. SGX provides a sandbox for applications to create confidential, integrity preserving, and authenticated segments of code and data. Privileged system software such as the OS, hypervisor, and BIOS are all unable to interfere or access contents of an enclave. Enclaves are facilitated by a TEE which reduces the TCB by removing privileged system software, and only include the trusted hardware component and application. Specifically, an enclave is a protected area of execution in memory, in which all code and data is subject to encryption.

An SGX-enabled application consists of two parts: untrusted code and a trusted `enclave`. The SGX-enabled processor isolates the enclave's code and data from the outside environment, including higher privileged software, i.e operating system and hypervisor, and hardware devices attached to the system bus. Enclaves are backed up by a region of memory separated at boot time, called PRM. The PRM is protected by the CPU such that no non-enclave memory accesses may happen. This includes the software kernel, the System Management Mode (SMM), and DMA accesses from peripheral devices. Enclave code and data are managed by the EPC, which in turn is protected by the PRM, depicted in Figure 2.1. The EPC is divided into 4 KB pages, and the assignment of pages to enclaves and page management is facilitated by a kernel module provided by Intel. Specifically, the contents of the EPC are encrypted upon being flushed from the L3 cache by the Memory Encryption Engine (MEE).

The integrity of the EPC is checked to ensure that no modifications have been done to it, resulting in a processor lock-down if the integrity of the EPC is violated. Enclaves are allowed to access other regions of memory that are located outside the EPC directly, whilst non-enclave code is not allowed to access enclave memory. Furthermore an enclave can copy data to and from the EPC, e.g, function call parameters and results, and it is the responsibility of the enclave to assert the integrity of data that originated from outside the protected memory region. SGX restricts the size of the EPC to 128 MB. There are no limits to how large an enclave may be, however, after creation the memory allocated for an enclave is finite and cannot be expanded. Exceeding the EPC size will cause the CPU to move the pages between the EPC and untrusted memory. Since the EPC is not accessible to any system mode, the OS handles page assignment through SGX instructions. An OS kernel module encrypts pages that are evicted from the PRM. Page faults targeting a particular enclave will cause the kernel to issue a Inter Processor Interrupt (IPI), affecting all logical cores running inside enclaves. This causes all threads in enclave mode to do an involuntary Asynchronous Enclave Exit (AEX), and trap down to the kernel page fault handler. The second generation of SGX has support for dynamically allocating new pages for an enclaves at runtime.

SGX supports multiple enclaves on a single machine; within the same process' address space or different processes. Enclaves are created by the intel kernel module in privileged mode on behalf of a process using the ECREATE instruction. The ECREATE instruction will allocate new pages in the EPC for code segments, stack, heap and data segment, the SGX Enclave Control Structure (SECS), and the Thread Control Structure (TCS). The SECS contains information that is used by SGX to identify the enclave and to hold any references to the memory resources of the enclave. A depiction of the contents of an enclave and its position within the process' address space is shown in Figure 2.2. When an enclave is loaded, its contents is cryptograpically hased by the CPU. This

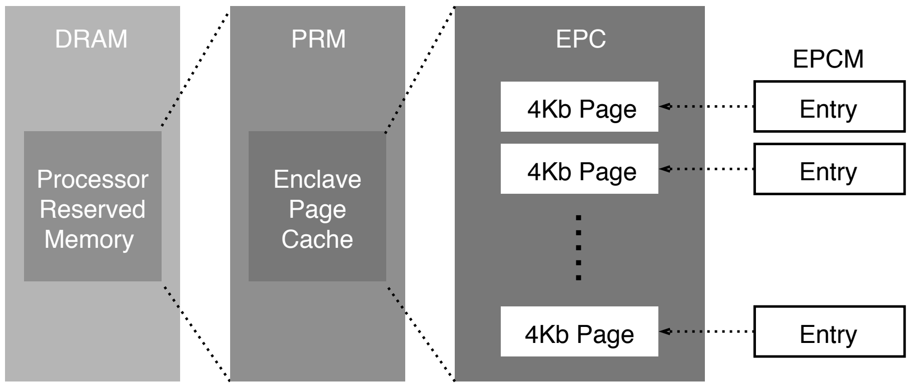

Figure 2.1: Shows the memory layout of the SGX memory architecture. The PRM contains the EPC which in turn consists of 4Kb pages that are managed by the Enclave Page Cache Metadata (EPCM)

becomes the enclave's measurement hash, used in the attestation process to uniquely identify the software running inside the enclave. After initialization, all code and data segments will be copied into the enclave, at which point no further allocations may be done by the enclave. When an enclave is destroyed the OS invalidates all pages that belonged to the enclave, and zero initializes them to ensure that no data is leaked.

After creation, threads transition into the enclave by calling the EENTER instruction. SGX allows multiple threads to enter the enclave. The only constraint is that the number of threads must be given prior to initialization. This is because every thread executing inside an enclave must have a TCS which stores the execution context. A thread can only transition into the enclave from user level (ring 3), and is disallowed from issuing software interrupts. Therefore, without the SYSCALL instruction enclaves cannot directly complete system calls. A thread must first exit the enclave and transition into ring 3 before issuing any software interrupts. Threads exit the enclave either through synchronous exits, issued by the EEXIT instruction, or involuntarily through asynchronous exits. Similarly, hardware interrupts are not handled by the enclave, as a mechanism to prevent leaking information from the central processor unit (CPU). A thread running in *enclave mode* will not directly service the interrupt or page fault. The CPU performs an AEX into ring 3 code before servicing the fault or interrupt. To avoid the OS from inferring the execution state of the evicted thread, all execution context is saved before being flushed on exit.

Intel provides application developers with a Software Development Kit (SDK)[2] for implementing applications. The SDK includes a stripped version of Glibc

2. https://github.com/intel/linux-sgx

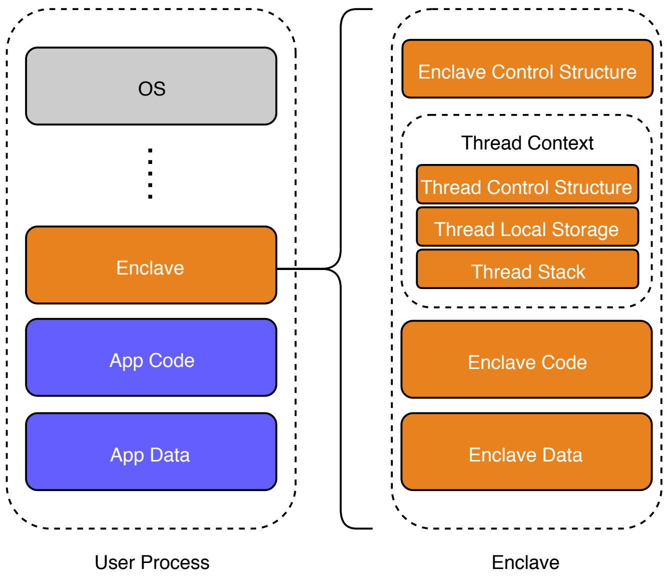

Figure 2.2: Depiction of an enclave in the virtual address space of a user process, along with the contents of an enclave.

and STL which includes memory allocation by emulating `brk()`, cryptographic primitives for software attestation, and support for secure communication. Notably, the Glibc and STL implementations exclude all system calls. The SDK also have support for running enclaves in simulation mode with support for standard debugging primitives, i.e GDB. Creating and running an enclave in hardware mode requires the presence of a kernel module, referred to as the SGX driver, and the the Application Enclave Service Manager (AESM) service [3]. The AESM service holds pre-provisioned enclaves by Intel, namely the Provisioning and Quoting enclave, which are used to verify other enclaves and sign them with an asymmetric key, and provide a *launch token*.

2.4 Frameworks for SGX

Prior application framework support for Intel SGX heavily focuses on the applicability, and addressing SGX performance restrictions. Specifically, the two restrictions in question are: 1) Entering and exiting an enclave to issue privileged instructions. 2) the limited EPC size (128 MB). The first may have a substantial performance impact on applications workloads frequently interacting with the operating system, such as I/O [18]. The second can incur expensive

3. https://github.com/intel/linux-sgx-driver

swapping if the application's working set exceeds the EPC size limit. To issue instructions that may require change in privilege levels such as SYSCALL and SYSENTER, the enclave is unable to directly execute system calls. Enclave code must first explicitly exit the enclave execution context, through an `OCALL`. Compared to a system call that takes about 150 cycles, an `OCALL` uses around 8000 cycles to complete [33]. The authors of [34] reported that the performance overhead for a synchronous implementation of system calls adds a significant overhead. Similarly, they discovered that memory accesses beyond the available EPC size, incurred performance overhead three orders of magnitude larger compared to memory accesses within the EPC boundary.

Initial work that pre-dates the availability of SGX hardware, Haven, showed that it is possible to run unmodified binaries by placing an entire library OS inside enclaves [35]. Haven builds on Drawbridge [36], a library OS based on windows 8, consisting of two core mechanisms: The *picoprocess* and a library OS. The Drawbridge Library OS is an Application binary interface (ABI) to the hardware services in the host OS, and is implemented by a *security monitor*. Picoprocesses is constructed in a hardware address space to create a secure isolation container, with no access to system calls. Together, they enable sandboxing of unmodified Windows applications. Haven locates the Drawbrigde library OS inside the enclave which interacts with an untrusted lower level ABI implementing 22 calls, such as thread management and encrypted I/O streams and virtual memory management. However, their results are not representative for real world performance as their evaluations were conducted on an Intel provided SGX emulator.

Secure Linux Containers (SCONE) compares an approach similar to that of Haven on SGX hardware [34], by using the Linux Kernel Library (LKL) to create a Linux library OS. They found that having an entire library OS inside an enclave increases the size of the TCB by 5x, the service latency by 4x, and halves the service throughput. In the same work, they proposed an alternate approach to that of Haven: placing Libc inside the SGX and shielding system calls by implementing a shielding layer between Libc and the host OS. Furthermore, to mitigate the incurred performance overhead of enclave transitions, SCONE implements asynchronous system calls, called m:n threading, in which M threads run inside the enclave and N threads run in a kernel module to service the system calls. Asynchronous system calls avoids the cost of unecessary enclave transitions. Their evaluation achieved at least 60% of the native throughput, and a comparable throughput for native Memcached.

Other approaches, however, contend that they are comparable to SCONE in terms of performance and with TCB of similar proportions. Graphene-SGX an open-source library OS for SGX [37], offering a wider range of functionality than that of SCONE, such as `fork`. Graphene-SGX is a port of the Graphene

Library OS [38], modified to run inside enclaves. The Graphene library OS implements most of the functionality of an OS in user-level, with the exception of an ABI exposing 18 system calls to the untrusted host OS. Graphene-SGX is able to support OS calls that SCONE does not. Most notably `fork`, `execve`, and dynamic loading of Shared Object File (SOF)s. The enclave code and data is measured before initialization and attested by the CPU, and is therefore unable to natively support dynamic loading. Graphene-SGX does this by creating a unique signature for any permutation of an executable and dynamically-linked libraries, by using the linux SGX driver. After initialization, their user-level bootloader checks the intergrity of each library that is copied onto the enclave heap; if a library does not match it will not be loaded into the enclave. They method in which they implement `fork` is by creating a new enclave, copying the execution state of the parent enclave, and establishing a secure channel between them for communication.

Eleos [39] is another effort which is also based on the graphene library OS. Their approach includes optimizations to tackle enclave thread transitioning and the memory footprint restrictions set by the EPC. Eleos implements what the authors refer to as the Secure User-managed Virtual Memory (SUVM) abstraction. By having a global allocator across enclaves, the SUVM mechanism avoids expensive page faults and associated enclave exits. The SUVM maintains a backing store in untrusted memory, allocated by the process owning the enclave. It implements paging in a similar manner to that of SGX, pages copied from the EPC to untrusted memory are encrypted, and when copied to the EPC the pages' intergrity is validated. Eleos also integrates an Remote Procedure Call (RPC) mechanism to enable exit-less system calls. They modify Graphene-SGX to support their SUVM and exit-less system calls, and use memcached to benchmark native Graphene-SGX with their two modifications. With memcached, they achieve a 2.2x throughput increase over native Graphene-SGX with the SUVM when memcached exceeds the EPC, and 2.5x when it does not.

Panoply [40] is another SGX frawework aiming to minimize the TCB, while providing a complete POSIX API within enclaves. Their solution prioritizes to minimize the TCB and trades API completeness over performance. In contrary to the aformentioned fraweworks, Panoply achieves system call support by implementing all functionality with `OCALLs`. Panoply offers much of the OS-services that Graphene-SGX does, with some additional features such as event handling and on-demand threading. However, what differntiates Panoply from other frameworks is that they place all libraries in an application outside the enclave. Argubly, this lowers the TCB but exposes an increased attack surface to the untrusted application.

Common for most of the SGX frameworks is that they address the performance restrictions that are inherent with the SGX architecture. The take-away is that in order for SGX to be applicable it needs the support of common OS services to accommodate real world applications.

2.5 Memcached

Memcached is an open-source distributed memory object caching system that was built to alleviate database load in dynamic web applications [41, 42, 43]. Specifically, memcached is an in-memory key-value store for generically typed data objects, e.g, rendered pages, database results, or any non-static data used in a web application. Its creator, Brad Fitzpatrick, originally built it to speed up LiveJournal's web servers in 2003 [4]. This chapter will introduce the architecture of memcached.

Memcached exposes an extensive API that is accessible by two protocols: accii and binary. Some of the functionality include the standard CRUD operations for a key-value store: Retrieve a value associated with a key, adding a value associated with a key, and deleting key-value pairs. Keys size is limited to 250 bytes and the maximum size for data value is default to 1MB. Most of its API runs in constant time $O(1)$, and having one element in the cache shall be as performant as a full cache. Additionally, memcached keeps track of hit-rate, eviction-rate, etc. Clients can query these statistics and alter the configurations to accommodate a usage pattern optimally.

In a multi-instance memcached deployment the abstraction the client is exposed to is a dictionary interface. However, Memcached instances are independent of each other, and do not communicate. Therefore, to consistently store key-value pairs memcached uses two-layer hashing. The first layer is implemented on the client side. By hashing the key, the client decides which memcached server to send the request to. The second layer is the selected server's hash table. Memcached instances process queries in parallel, thus the main reason for adding instances is to increase the total amount of memory.

Since memcached instances are generic in nature, clients implement extensive features. E.g, compression to reduce the memory footprint, mult-get to retrieve multiple keys at once, and weighting the key distribution among a cluster of memcached servers based on available memory. There exist client libraries for many programming languages: Perl, C++, C, Java, etc; the most used being libmemecached [5].

4. https://memcached.org/
5. http://libmemcached.org/libMemcached.html

2.5.1 Memcached Internals

The entire memcached codebase is written in C, and is maintained as an open-source project [6]. Internally, memcached implements a hash table that uses chaining to resolve collisions. The hash table can handle concurrent accesses, and the synchronization is done on the buckets in the hash table. This prevents elements from being accessed by multiple threads simultaneously whilst improving thread contention on the hash table itself. This is a key requirement of memcached, such that a client updating an item does not cause any other clients to wait. The hash table is expanded if the load factor exceeds a given threshold, 2/3 by default. In order to avoid slowing down queries while the hash table is expanded, memcached allocates a new hash table and lazily moves entries from the old table to the new. Entries that are not found in the new table are re-hashed and fetched from the old table.

To store the elements of the hash table, memcached uses a slab allocator. The slab allocator divides a memory pool into classes of increasing sizes and every slab class maintains a free-list of elements within each slab. The slab classes are power-of-two sizes from 64 bytes to 1MB, and fits an element in the smallest class possible. Every slab class, regardless of its size, allocates 1MB pages on-demand or prior to initialization to avoid memory fragmentation. If any of the slab classes fills up, memcached evicts items if they have not expired, the slab class is ouf of free chunks, and if there no more pages to allocate to a slab class. Memcached also maintains an LRU that evicts items based on their activeness. Items are moved between three categories: HOT, WARM, and COLD. New items enter the HOT category and are bumped down to COLD if they are not accessed as new items enter. This is one of the key designs in memcached: forgetting is a feature. To maintain this structure, memcached uses a separate thread to crawl three categories and move them accordingly.

Memcached delegates query processing to a user-defined amount of threads. Each thread maintains a connection structure holding the state of the current request and a list of all pending queries. Fueled by libevent, all worker threads use asynchronous I/O. Each worker is independent of each other, and all they share are references to the internal hash table. Memcached also uses libevent to update a global clock used for features such as Time To Live (TTL). Instead of having to explicitly run an update procedure, libevent executes a function that updates the global timer at a fixed time interval.

Memcached exposes a large set of configurable parameters. Many of these parameters have a direct impact on the data structures and performance. Memcached allows explicitly setting the maximum sized items which can be

6. https://github.com/memcached/memcached

stored, how large the slab's pages shall be and whether the memory used by the slab allocater shall be pre-allocated. Other parameters that may be more performance related are the degree of concurrency for request processing, dissallow expanding the hash table, and whether the maintaince threads shall run.

2.6 Summary

This chapter has presents the capabilities of Trusted Execution Environments (TEEs), implementations of TEEs with emphasis on the Intel SGX architecture. We also detailed the challenges of providing common OS services, and the performance restrictions, in SGX.

3 Diggi - A Framework for Trusted Execution

Diggi is a framework for TEEs, currently supporting intel SGX. This chapter will describe the motivation and design of Diggi, as well as the simple programming abstraction it exposes to application developers.

Section 3.1 give a brief description of the architecture of Diggi. Section 3.2 explains the Diggi programming model and the Diggi library OS. Section 3.3 describes Diggi's communication through message passing.

3.1 Diggi Architecture

Diggi is a distributed agent-based application framework for dissemination of privacy sensitive data and operations. It offers a simple abstraction for application developers to create trusted distributed applications. The trusted parts of the application that require strong security guarantees will be executed inside a TEE, while those that do not may run outside the TEE. Breaking an application into untrusted and trusted parts allows Diggi to reduce the TCB of the application. These components are units of execution referred to as *agents*, and all components in Diggi are implemented using this agent abstraction. The role of an agent is divided into two categories: system and application agents.

System agents perform tasks such as orchestration, IO, and message scheduling, while application agents perform application logic.

Diggi's design principles include ease of deployment and simplified abstractions. The key features of Diggi are: user-level scheduling, asynchronous system calls and non-blocking message queues for agent-to-agent communication. Moreover, diggi supports legacy applications by implementing user-level OS-services exposed through a POSIX API. Applications developers are exposed to the same programming abstractions regardless of the agent's execution environment. Currently, Diggi utilizes Intel's SGX trusted computing infrastructure to provide secure execution for its trusted components. As a consequence of using SGX, Diggi aims to tackle its inherent functionality and performance restrictions. Executing large codebases in SGX does not not only increase the TCB, but hurts performance due to complex abstractions and the current architecture of SGX [18]. Diggi's agent model of dividing an application into smaller logical applications, may reduce the overall runtime costs. Another performance restriction of SGX is the overhead of transitioning from trusted executing to non-trusted. Diggi solves this by performing all system call related operations asynchronously.

Figure 3.1 illustrates the overall architecture of Diggi. For simplicity of abstraction the runtime itself is also composed of agents. Each instance of the Diggi runtime includes an *agent-agent*, managing the agents for that particular Diggi instance. The agent-agent runs outside SGX, and is responsible for deploying new agents on-demand, agent discovery, and agent-to-agent communication. A Diggi instance is a process that contains all agents in its virtual address space, including the trusted components (enclaves).

Through service oriented applications, Diggi agent are also able to migrate between Diggi processes. This allows trusted parts of a Diggi application to be relocated, moving computation closer to the data and still guarantee strong isolation. In a classic three-tier architecture in which sensitive data resides in a key-value store, placing the cache closer to the web server in a TEE, will not diminish security guarantees.

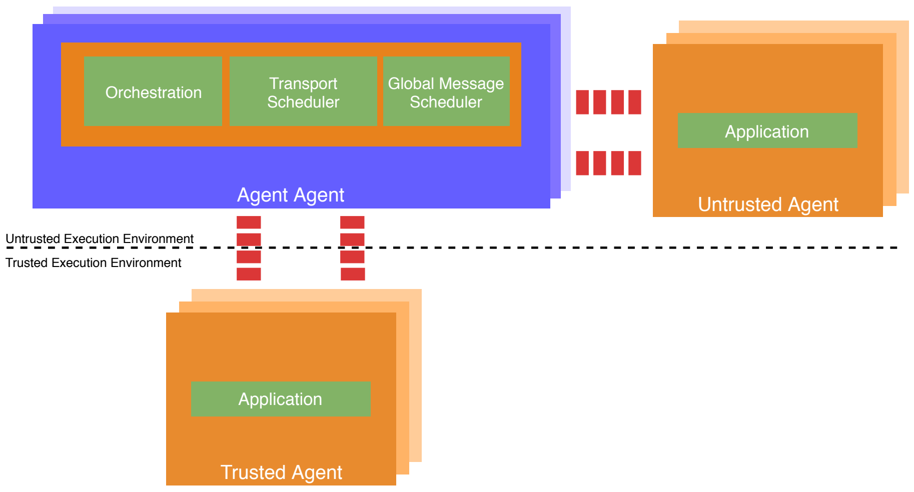

Figure 3.1: Shows the overall architecture of a Diggi instance running one trusted agent and one untrusted agent. The untrusted agent and the agent-agent both run outside the TEE. The agent-agent communicates with the two agents through the two red messagues queues.

3.2 Diggi Abstractions & Primitives

Diggi implements all operations asynchronously in the context of a single thread. Following the recommendation of [18], Diggi agents are single-threaded because pinning threads inside enclaves maximizes performance. That one thread handles everything within an agent: IO, messaging, application logic, etc. Diggi alleviates all these responsibilities for application developers, by only exposing asynchronous APIs. Although the core functionality of Diggi utilizes asynchronous operations, Diggi is able to hide asynchrony for POSIX-like compatibility. Specifically, to emulate a blocking call for a POSIX emulated system call, the POSIX-layer will use the thread scheduling API to execute other tasks while waiting for a response. Diggi's thread scheduler interface consists of the following: push tasks to the scheduler, and Yield. When pushing tasks to the scheduler, set task will not start executing until the function that pushed the subsequent task to the scheduler is done. The yield method is a way of supporting synchronous programming, dequeueing tasks from the scheduler until a condition is met. There is no priority scheme to distinguish the importance of task, that is, all tasks are dequeued in a FIFO manner.

A Diggi application is a collection of simple purpose-based agents that each do separate specific tasks, collectively creating an application. Applications in Diggi hold many similarities to the service oriented architecture paradigm, where single purpose units of operation are composed to implement a full application stack. E.g, an agent wanting to read data which is located on disk is able to read that data with the service of a second agent that implements file I/O. Each agent binary contains the Diggi library OS which implements messaging, thread scheduling, networking and encryption. The application code is compiled together with the library OS into a SOF, or shared library.

To illustrate how Diggi agents are developed, Listing 3.2 highlights the four functions all agents are required to implement. When the agent SOF is dynamically loaded into memory the agent-agent extracts the four functions from the symbol table. `agent_start` is the entry point for the agent application, similar to that of a `main` function in most programming languages. The function `agent_init` is called prior to the entry point such that the application may initialize the state of the application. Its counterpart, `agent_stop`, is called when the agent exits to deallocate state. The core features of Diggi, namely its scheduler message manager and logging, are exposed through a C++ interface. All POSIX functionality is C-compatible and build upon the C++ interface. The core services implemented by the library OS are passed through the `ctx` parameter holding the current execution state of the agent, and of the whole runtime. This state includes the agents own Agent Identifer (AID) and a list of other agents that are alive in the system.

Code Listing 3.1: Definition of the interface all Diggi agents are required to implement

```
// Called to initialize agent
void agent_init(void *ctx, int status);

// A Diggi agent's main function
void agent_start(void *ctx, int status);

// Default recieve callback
void agent_recieve(void *msg, int status);

// Deallocate all resources
void agent_stop(void *ctx, int status);
```

Figure 3.2 depicts the components that make up an agent binary in Diggi, where the application compiled together with the Library OS. Diggi agents are also written to be agnostic of architecture, allowing agents to run on any TEE and operation system. However, in its current form Diggi only supports intel SGX, but the techniques used here are expected to apply for similar systems. Application developers are exposed to an API that hides whether the application is running in trusted or unstrusted mode. When compiling the application together with the library OS. developers can either mark the application as trusted or untrusted. If the application is marked as trusted Diggi will transparently turn on security features such as encrypted communication and I/O.

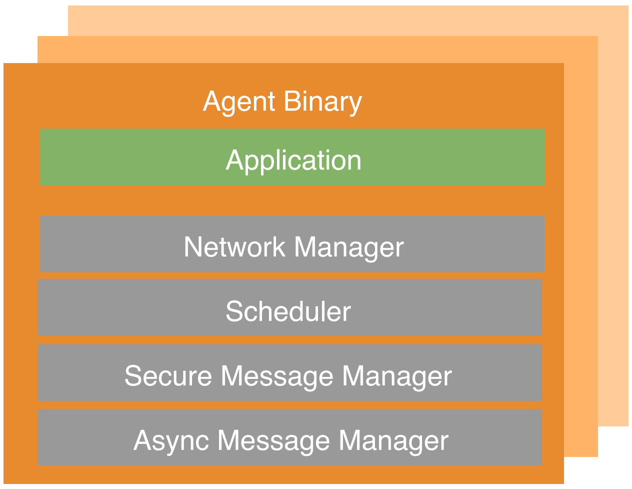

Figure 3.2: Depicts an agent binary that is composed of the application binary and the Diggi library OS. This particular example contains the services compiled for a trusted agent. All components are compiled into a SOF.

3.3 Agent Communication

Diggi agents communicate with the system, and other agents through an asynchronous lock-free message queue. Diggi's message manager is versatile and used as a building block to implement more complex features. It implements transparent agent-to-agent communication, either within or between Diggi instances (inter-node). Upon initialization each agent allocates one outbound and one inbound queue. To schedule message between agents, Diggi uses the Global Message Scheduler which is a part of the agent-agent. The message scheduler keeps references to all input/output queue pairs of every agent running on the same Diggi instance. To keep track of the source and destination of a message, the message scheduler uses Diggi's Agent Identifers (AIDs). Therefore, sending a message to an agent only requires the identifier of the recipient agent.

Messages bound for another agent are appended to the queue with a destination identifier. The global message scheduler continuously polls for incoming messages, and copies messages from the outbound queue of the source agent to the inbound queue of the recipient agent. The message manager also transparently implements message passing between Diggi instances with Transmission Control Protocol (TCP). If the message scheduler receives a message with an AID that is not running on the local Diggi instance, the scheduler maintains a map of agents residing on other instances of Diggi.

One of the design principles of Diggi is runtime agnostic agents, allowing agents not running in SGX to communicate with those that do. To adhere to that principle, Diggi implements two methods of sending messages: securely, and insecurely. Insecure agents are able to establish a TLS channel between agent-agents, or directly targeting a secure agent. Communication between trusted agents are encrypted by default, thus all communication between trusted agents are always subject to encryption. All message queues reside in untrusted memory, because processes outside SGX are not allowed to access the EPC. Therefore, messages that are sent out of the enclave must be copied into unstrusted memory. This scenario is depicted in Figure 3.3.

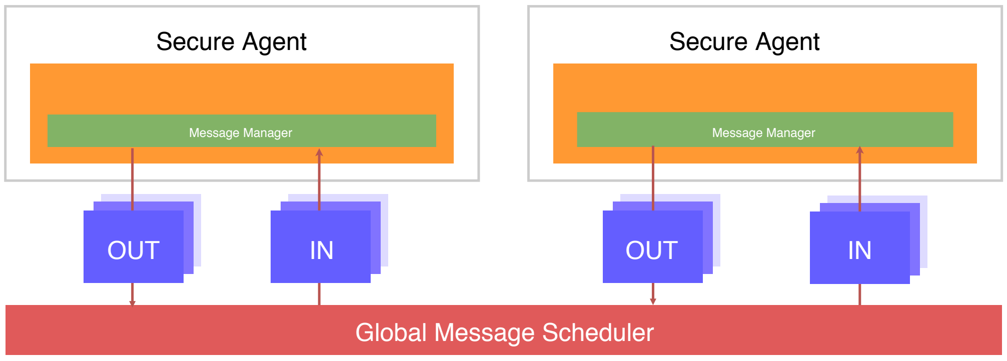

Figure 3.3: Shows the communication between two agents where the Global message scheduler handles message passing. Both agents run in a enclaves, where messages are subject to encryption. Note that all message queues reside outside EPC.

The message manager is also a building block for asynchronous operations. As Diggi composes an application of smaller logical pieces, agents must have the possibility to issue operations to other agents. Therefore, the message manager supports multiple types of messages: regular message, operations, or system calls. This enables a system agent to receive operations from an application agent asynchronously.

3.4 Summary

This chapter has presented, Diggi, a framework for creating trusted distributed applications, currently utilizing Intel SGX. Specifically, it has described the simple programming abstractions in Diggi, how Diggi splits application into smaller parts in order to reduce the size of trusted components, and how Diggi tackles the performance restrictions of intel SGX.

4 Design & Implementation

This thesis introduces SecureCached, a port of the popular in-memory distributed key-value store Memcached to Diggi. Memcached is an OS-intensive application requiring many features provided by conventional operating system. Diggi does not yet support the set of features required in order to run legacy applications such as memcached. Therefore, this chapter will first detail the capabilities memcached requires from Linux, followed by the modifications to memcached that are results of Diggi's architecture. Third, we detail feature extensions to Diggi that emulate a small portion of the POSIX abstraction. Lastly, we describe how we enabled memcached to run within the Diggi runtime.

Section 4.1 discusses the design choices made when porting memcached to the Diggi runtime, and summarizes the required extensions to Diggi. Section 4.2 gives a brief overview of SecureCached. Section 4.3 explains how we altered Memcached to fit the single-thread Diggi agent programming model. Section 4.4 and 4.6 describes the implementation of a shim layer to support linux primitives in Diggi. Section 4.8 briefly discusses the implementation of a memcached client supporting a minimal client API.

4.1 Design trade-offs

Memcached is an OS-intensive application written in C (18k Lines of Code (LOC)) with an extensive use of system calls. The system call dependencies include socket operations, UNIX-domain sockets, event polling, pthreads, and inter-thread communication. This section will focus on which of these features Diggi can support, and which constraints the Diggi runtime imposes on memached. For future reference we will refer to the unmodified memcached implementation as memcached, and refer to our modified version as *SecureCached*.

As described in Chapter 3, Diggi does not support running unmodified binaries inside enclaves. Enabling memcached to run in Diggi will therefore require extensive modifications and recompilation. First, a set of constraints must be accounted for in order to make the right design choices. Memcached is implemented in C and Diggi is implemented in C++, therefore all extensions to Diggi that are required by memcached must made be C-compatible. None of the modifications made to memcached can break its API, that is, the API memcached exposes to its clients. Furthermore, by the design principles of Diggi, an agent shall be agnostic of its execution environment, therefore, implementing SecureCached as a Diggi agent requires it to be able to run in both trusted and untrusted Diggi agents.

Having defined a set of constraints, we also consider multiple solutions to implement the required features. The three things to concider while making the right design choices are: 1) Which OS-services does Diggi support, 2) and what are the consequences of implementing them in user mode 3) Which features are Diggi unable to support. The next subsection will discusses the design trade-offs individually.

4.1.1 Sockets

Memcached uses the client-server model and exposes its services over either User Datagram Protocol (UDP) or TCP using POSIX sockets [44]. An illustration of how clients interact with memcached is shown in Figure 4.1. Diggi does not yet implement a POSIX socket interface, and we propose two solutions for implementing a POSIX compliant socket API in Diggi:

1. Implementing an external POSIX-socket `OCALL` interface, where the system calls are serviced by the untrusted part of the Diggi process.

2. Implement a POSIX socket API on top of Diggi's message manager.

Solution 1 conflicts with the Diggi asynchronous system call model, and its CPU thread utilization model. Transitioning from enclave mode to issue a system call that may block will deprive or block other Library OS routines from executing.

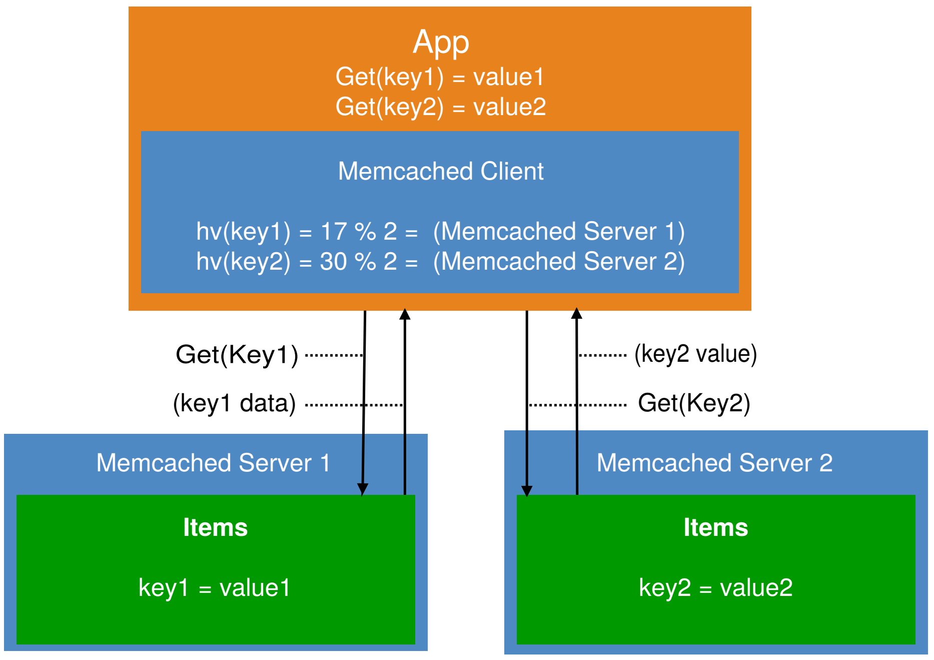

Figure 4.1: Illustration of the interaction between a memcached client and two memcached server. The keyspace is distributed among the two servers, and keys are located accordingly.

As described in section 3.3 Diggi's message manager transparently implements reliable agent-to-agent communication over TCP, and communication between trusted agents also involves encrypting the contents of message, which is not a feature in memcached. Therefore, implementing a POSIX socket API by using the message manager has the following advantages:

1. Requests sent to a memcached agent will adhere to the secure messaging protocol in Diggi while not requring any changes to the memcached codebase.

2. Avoid thread transitions for agent-to-agent communication within the same Diggi process.

The first advantage also adheres to our contraints that memcached must be agnostic to whether it is running in a trusted or untrusted agent. However by using the message manager we set constraints to the network protocol in

SecureCached, that is, functions such as `recvfrom`, and UDP or protocol specific POSIX functionality. This is because all cummunication to Diggi instance on a different machine has to be reliable, and is therefore always TCP.

4.1.2 Libevent

Memcached uses `libevent`, an event framework enabling callback notifications set on file descriptors to avoid polling, to implement asynchronous I/O. Libevent uses system calls such as `poll()`, and implements a range of functionality, e.g, rate-limiting, filters, and zero-copy file transmission. Implementing libevent support for Diggi would require the Diggi library OS to support a majority of the POSIX API and signal handling. Enclaves are unable to handle software interrupts, which would require an `OCALL` interface in which threads transition out of the enclave to wait for interrupts. Implementing that approach would break with thread utilization in Diggi. Memcached only uses 2% of the libevent API, and we therefore deem that porting libevent to memcached is unnecessary [1]. Instead, we implement the event handling as a part of the Diggi library OS, with a small shim-layer as the interface to SecureCached.

Memcached also uses libevent for updating its global timer: It registers a clock handler callback with libevent which is expected to be called at a fixed time interval. The global timer is an imperative part of the core functionality of memcached, and elements that are stored in a memcached instance becomes associated with a Time To Live (TTL); a feature which is exposed through the memcached client API. If the global timer never gets updated, all items will remain in the cache until memcached runs out of memory; similarly achieved by setting the TTL to zero. Diggi does not implement timer interrupts, which is due to SGX restrictions. SGX does support secure time through the SDK function `sgx_get_trusted_time`. However, Diggi's thread scheduler does not have the feature to execute callbacks with fixed time intervals because SGX does not support interrupts; diggi does not support preemption and thus not timed events. Removing this feature will impair our initial constraint of not breaking the memcached API. However, we deem that by removing the TTL feature from SecureCached we still have the feature set to support relevant benchmarks [45].

1. The percentage of the libevent API utilized by Memcached was obtained from counting all API functions of the library.

4.1.3 Pipes

POSIX pipes is another OS-service that is utilized by memcached. Pipes are used for communication between threads or processes, e.g, between a parent and a child process. However, memcached only uses pipes for inter-thread communication, allowing us to exclude supporting inter-process communication. The POSIX API we need to fulfill only consists of three functions, and will therefore be implemented in the Diggi POSIX API. Memcached uses pipes in combination with libevent to notify threads in the system with additional arguments, hence, the implementation must support event handling.

4.1.4 Threading

As described in Section 2.5.1, Memcached is a multi-threaded application relying on separate threads to handle specific tasks, e.g, maintenance threads to keep the invariants in their data structures. The clock event handler calls a method that resizes the hash table if the amount of elements in memcached exceeds a given threshold. We could implement a scheme in which the method is called every N request. That would not be an optimal approach because it will cause the operation to block all other requests while the table is resized. We ammortize this effect by adjusting the hash table size to fit with a hard element threshold, removing the need to re-size.

As a side-effect of bypassing high transition cost for threads, Diggi allocates dedicated threads to each agent. To ensure many agents can be located on a single host, each agent is given a single thread, and must ensure high thread utilization by exclusively using asynchronous operation. We therefore have to modify SecureCached to run all routines within the context of a single thread. Memcached uses the `pthread` library for multi-threading. As we aim to run memcached single-threaded, we may remove pthreads entirely from the codebase.

4.1.5 Design Choice

To summerize, we list what the Diggi library OS must support and the modifications to the memcached codebase.

Extensions to the Diggi library OS

POSIX Pipes: Support for inter-thread communication. Specifically, emulating message passing between threads in an agent, including support for the

API to allow both non-blocking and blocking operations.

POSIX Sockets: The functions that memcached uses and must be supported are: `Read`, `write`, `listen`, `bind`, `sendmsg`. Additionally for `Read`, `write`, and `sendmsg` Diggi must support non-blocking and blocking operations.

Event Framework: Implement a subset of the functionality in libevent: register an event on a file descriptor of either type socket or pipe (read and write); delete a prevously registered event; Since libevent uses the file descriptor abstraction, the framework must be compatible with all file descriptor primitives.

Memcached Client: There are many implementations of memcached client in numerous languages, the most used being *libmemcached* [2]. However, porting an existing client API is outside the scope of this thesis. Instead, we need to implement a small subset of a memcached client to support the most basic operations: load memcached with a key-value pair, and retrieve a value associated with a key. The client also have to implement consistent hashing in order to utilize multiple memcached servers.

Logging: Memcached uses standard in and out (*stdin, stdout, stderr*) as the destination of log messages. Diggi must support a C-compatible logging interface for debugging purposes.

Alterations to the memcached codebase

Remove threading: Modify the memcached codebase to run in the context of a single thread, effectively this will involve removing multi-threading in memcached entirely. This will also result in switching off memcached's maintenance threads.

Remove Threading primitives: As we aim to remove threading in memcached entirely, all synchronization primitives most notably pthreads must be removed. This will not change the behaviour in any way as we have already removed concurrency from memcached, there will be no need for synchronization.

Glibc & Linux: The intel SGX SDK only provides a subset of the GNU C Library (Glibc) functionality. Therefore, any side functionality that does not directly affect the core features of memcached must either be removed or supported in Diggi.

2. http://libmemcached.org/libMemcached.html

Remove TTL feature: since Diggi can not support libevent timing, we remove the TTL feature of memcached. We conjecture that this is viable as memcached can still support the feature set needen to complete evaluation. Memcached will still evict items, just not based on the activeness of the item.

Remove signaling: Diggi agents have to be written such that they are able to run inside SGX. Calling functions such as `assert()` will call the `SIGABRT` signal which is an illegal instruction in SGX. Therefore, we remove all signal handling and Glibc code that utilizes privileged calls.

Remove excessive features: Unix domain socket support, UDP protocol functionality, and ip-address specific functionality such as `getaddr`.

4.2 Architecture

Porting memcached to the Diggi runtime involved modifying 1500 LOC, most of which were deleted linux-specific features listed in the previous section. Figure 4.3 depicts a high level illustration of how memcached interfaces with Glibc and Libevent. Much of the Glibc API implicitly calls the linux kernel to service system calls. This also includes libevent, which is heavily dependent on Glibc. Compare that to SecureCached, shown in Figure 4.2. Glibc and Libevent are replaced with a POSIX shim-layer that binds SecureCached with the Diggi library OS. The OS-services provided by the linux kernel have been replaced with our implementation of sockets, and pipes built on the core primitives of Diggi.

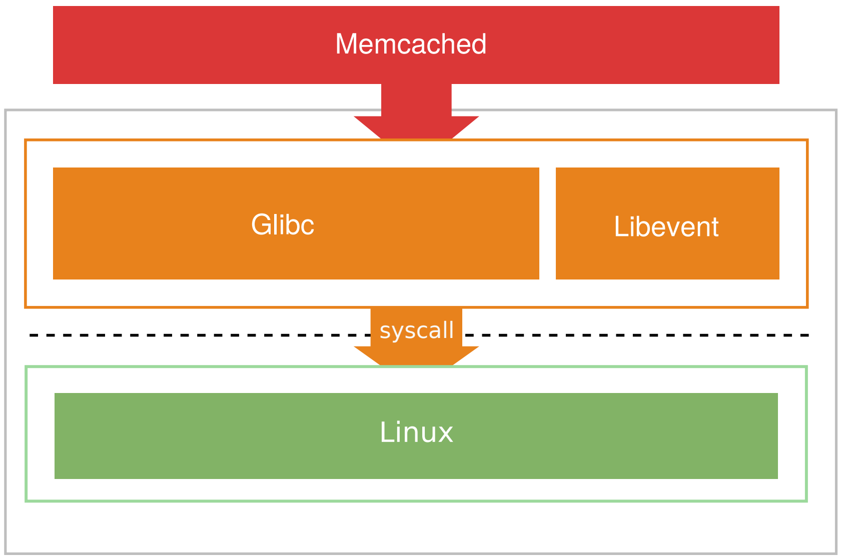

Figure 4.2: Depiction of how memcached interfaces with linux. Memcached uses Glibc which implementis a POSIX API. Memcached also uses libevent which depends on functionality from Glibc. Most of the libraries that memcached uses issue system calls to the linux kernel.

The handle-layer, under the POSIX-layer, unifies all extensions to Diggi that emulate the POSIX file descriptor primitive. By utilizing a shim-layer on top of Diggi's functionality, we are able to minimize the modifications to legacy applications. Diggi socket and pipes are specialized implementaions of the Handle abstractions. Diggi sockets in turn utilize some of the core Diggi library OS primitives, namely the massage manager and thread scheduler. The remaining sections of this chapter will describe the implementation details of SecureCached, the handle abstraction, Diggi sockets, and Diggi pipes.

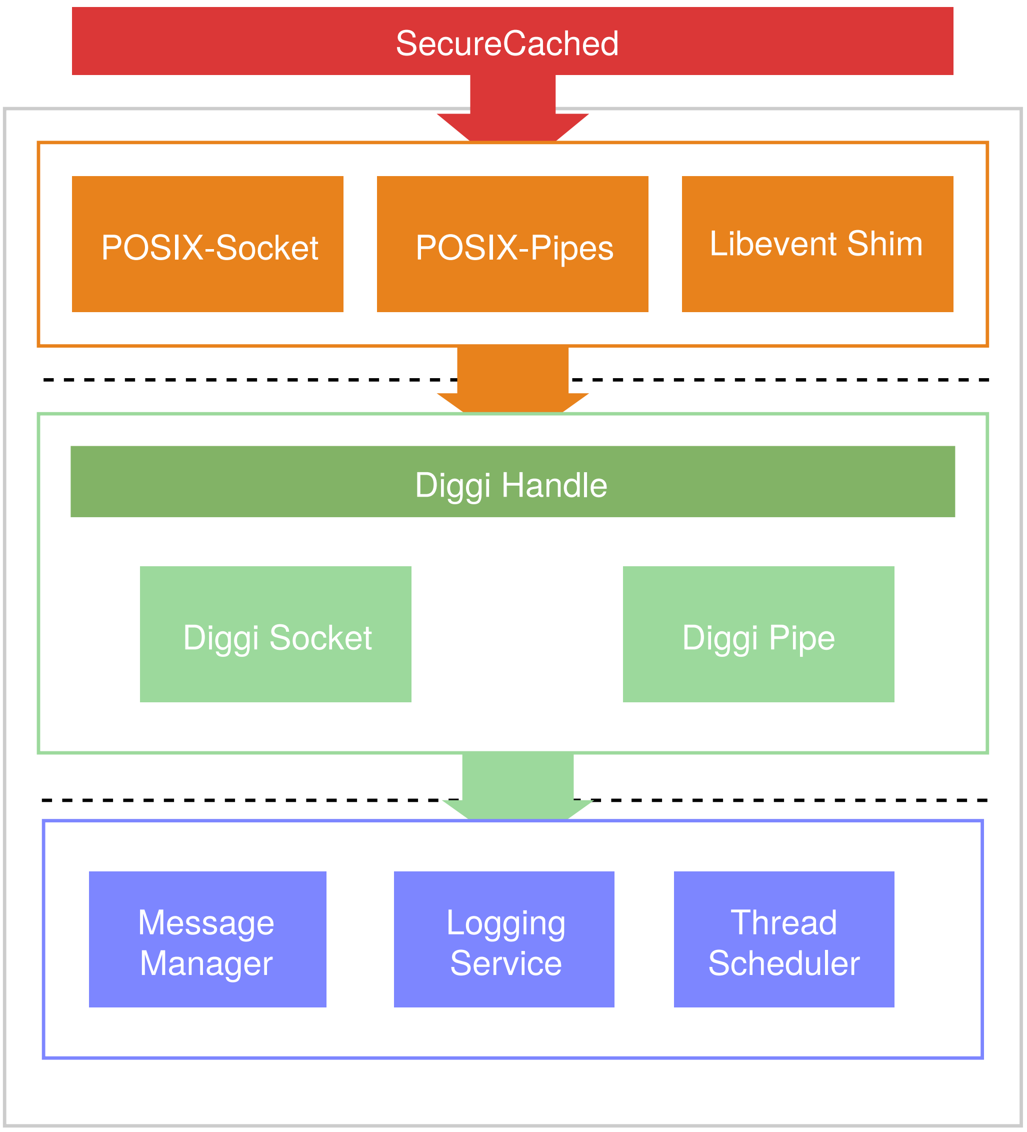

Figure 4.3: Illustration of how the SecureCached agent interfaces with the Diggi library OS, The SecureCached is exposed to a POSIX C shim-layer running on top of Diggi primitives. The socket and pipe implementation is unified through an event framework, Diggi Handle. Diggi Handle in turn utilizes the core services from the Diggi library OS. All components listed in the figure run in user-space (ring 3).

4.3 Single-Threaded Execution

By turning off many of memcached's maintenance threads, and modify the request logic, SecureCached is able to run within the context of a single thread. Memcached implements request handling by using the co-routines design pattern: a single thread is responsible for accepting new connections and delegate requests to worker threads. Memcached achieves this through the libevent library.

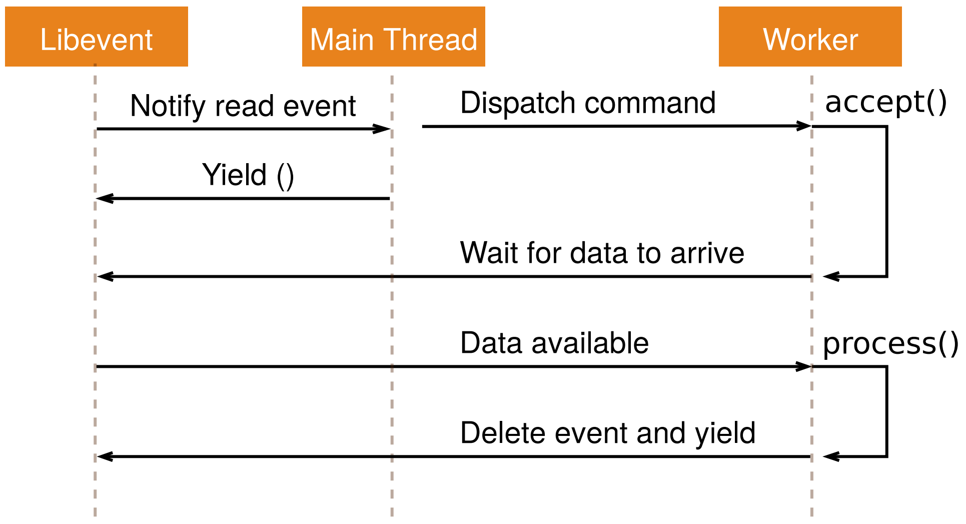

Figure 4.4: Illustration of how memcached uses libevent and worker threads to implement request processing. First libevent notifies the orchestration thread (main thread), after which it writes to a pipe to notify one of the worker thread. The worker thread starts a new session by calling `accept()` and then yields to the libevent to wait for incoming data. When data arrives on the socket the worker will process the request and yield to the libevent main loop to wait for a notification from the main thread.

Figure 4.4 states how memcached uses libevent to implement request processing. First libevent executes the registered callback when data arrives on the listening socket in the context of the main thread. The main thread chooses a thread and delegates the request by sending the connection object for the listening socket to that thread. Moreover, upon initialization all threads subscribe a read even on the receiving end of a POSIX pipe. The main thread notifies that thread by writing to the thread thread's pipe. After the thread is notified it will accept the connection and instantiate a client session before it yields to libevent, waiting to be scheduled when data arrives on the connection. When data arrives, the request will be processed in the context of that worker thread. When the thread completes the request it yields to libevent, waiting to process another request. Diggi only supports running single threaded agents, and is not able to construct muli-threaded co-routines such as query processing is implemented in memcached. Conversely, this thesis shows that this is achievable in a single-threaded context.

Libevent's usage-pattern involves deferring the execution of a function until an event occurs on the registered file descriptor. When a worker thread is scheduled to complete a request, it will register an event on the socket prior to reading a request. This means that each worker thread is driven by executing callbacks that are registered, waiting for a specific event. This callback-based pattern can be utilized to implement an event-based Diggi framework, running only in the context of a single thread. By removing threading entirely in memcached's implementation of co-routines, and taking advantage of the event-based nature of the worker threads, we are able to support co-routines in a single-threaded manner.

A pseudo-code of how memcached implements the callback-based pattern is presented in Code Listing 4.1 The implementation involves the three routines : `memcached_worker_entry_point`, `thread_process`, and `process_request`. Notoably, `thread_proccess` and `process_request` bot have the libevent callback signature. By that, we can internally register the subscriptions for those events in Diggi, and use its scheduler to execute the routines.

Every memcached worker thread is initialized by setting the function `memcached-_worker_entry_point` as argument to `pthread_create`. The entry point function will set up the worker context, and subscribe for read events on the allocated pipe before the thread yields to libevent; the state of a worker contains a pipe set and a structure to hold requests. We modified the memcached codebase to call that method in the context of the main thread, and return after initialization in order to complete the event initialization for a worker. Memcached supports multiple worker threads, SecureCached also support multiple workers by allowing memcached to create and register states to the libevent shim-layer. The `thread_process` function is a function that alters the state of a worker, and accept new client connections. Specifically, after the worker has initiated a new session, it will register a read event on the acquired socket to start processing the request when there is available data to read. The worker sets the function `process_request` to be activated on incoming data for the newly established session. Note that `process_request` also follows the libevent callback signature, allowing it to be called in the context of any of the worker threads. However, in SecureCached that routine will be executed in the context of the same thread. Diggi will still be able to logically support multiple workers as the third callback argument will contain the connection state, even if that routine is called in the context of the same thread.

Code Listing 4.1: Implementation of Memcached request pattern - some details are omitted for brevity.

```
// Entry point for a request process thread
static void *memcached_worker_entry_point(void *arg)
{
  // Setup logic...
  event_thread *me = arg;

  // Register a read event on the recieving end of
  // pipe (notify_fd) for thread_process to be invoked
  event_set (&me->notify_event, me->notify_fd, EV_PIPE_READ,
             thread_process, me);

  event_base_set (me->base, &me->notify_event);

  // Yields to the event loop
  event_base_loop(me->base, 0);
}

// Callback that is invoked when the request
// process thread is scheduled when a new request comes
// in
static void thread_process (int fd, short which, void *args)
{
  char buf[1];

  // Read command
  read (me->notify_fd, buf, 1);

  switch (buf[0])
  {
  // new connection
  case 'c':
  {
        int sfd = setup_connection(fd);
        // Register a new event on the session
        event_register (sfd, SOCKET_READ, process_request);
        me->state = CONN_READ;
  }
  }
}

// State machine Callback - invoked when there
// is data on the socket
void process_request (int fd, short which, void *args)
{
  conn *c;
  c = (conn*)args;

  switch(operation)
  {
  case CONN_READ:
  case CONN_WRITE:
        .
        .
        .
  }
}
```

As memcached implements its request processing using a state-machine pattern that is based on callbacks, Diggi event will queue these operations in the Diggi thread scheduler. Therefore, by running memcached single-threaded and transparently queuing operations based on calls to libevent, Diggi is able to emulate the correct runtime behavior of memcached. Doing so allows us to support features needed by memcached without extensively changing the codebase.

4.4 Diggi Handle

This thesis introduces the *handle* abstraction to Diggi, which implements event scheduling, polling, and provides a unified interface for file descriptor types. To achieve that, all file descriptor types introduced to Diggi must satisfy the interface. Due to the lack of OS primitives in Diggi, we need to take advantage of Diggi's design to provide the support for a POSIX API, and Libevent. The interface of Diggi handle is depicted in Listing 4.2 and its purpose is to provide a `poll` interface in the Diggi library OS. As previously discussed, agents in Diggi are composed of only a single thread, eliminating the option of having designated threads to poll for events in the system.

Code Listing 4.2: Details the Handle abstract class methods that must be implemented for a stream type.

```
//! Register an event on a filedescriptor that will be scheduled
//! for every incoming packet
virtual int registerEvent (int fd, async_cb_t handler,
                           bool persistent, void *arg) = 0;

//! Unregister event
virtual int unregisterEvent (int fd) = 0;

//! Polls for incoming packets on the filedescriptor
virtual enum handle_status poll (int fd) = 0;

// Returns the state of fds in *fds*
int unix_poll (struct pollfd *fds, nfds_t);

// Invokes register callback (async_cb_t) until
// the buffer assiciated with the fd is drained
static void messageDrain (void *arg, int status);

// Return the fd type (Socket, Pipe, etc.)
virtual enum handle_type handle_type () = 0;
```

Calling `registerEvent` will activate the callback argument for that particular file descriptor, and its counterpart `unregisterEvent` will deactivate it. Events notified by the handle will include all events that may happen on the descriptor: read, writes, etc; Retrieving the exact event requires a subsequent call to `poll`.

Since this is an interface that must be satisfied by any file descriptor type, events may vary or be triggered differently between implementations. The `poll` method is a simplified version of POSIX poll as it only returns one event for one file descriptor. The interface also includes a generic poll function implemented to emulate the original POSIX call.

4.5 Diggi Event

Libevent provides a mechanism to dispatch events, or execute program logic through callbacks for events that occur on a filedescriptor. This functionality is dependent on the ability to continuously checking for state on a file descriptor such as `poll(), select()`, and unix filedescriptors. The subset of libevent that we implemented is the following:

Event_set: identify an event with a non-negative file descriptor.

Event_add: add an event coupled with a combination of the flags READ, WRITE, or PERSIST. This activates the event if the file descriptor is ready for reading or writing depending on the flag. Since libevent can operate on several types of file descriptor, e.g pipes or sockets, we must support the same operations for all those types. Calling `event_add` sets the callback provided in the subscription to active. When an event occurs the callback will become non-pending right before the callback is invoked. A subsequent call to `event_update` will reactivate the callback. However, if the event is flagged as persistent, the event will remain pending even if the callback is executed once.

Event_del: called after `event_add` to make the callback non-pending and the event non-active.

The parts of libevent we were unable to support were: `evtimer_set`, `evtimer-_add`, and `evtimer_update`. This is because there are no primitives in the Diggi library OS allowing timers. As previously mentioned, this is an inherent restriction based on the architectural limitations of Intel SGX.

Diggi event mirrors the subset of functions that memcached require from libevent, and the libevent shim-layer acts as a C-interface to specific implementations of the handle abstraction. When an event is registered for a file descriptor, e.g a read event, Diggi event chooses the corresponding handle implementation. It then activates the provided callback through the handle implementation. If the activation of the callback is marked persistent, it registers that callback to call the `messageDrain` routine.

The function `messageDrain` is a side effect of Diggi's inherent single-threaded runtime, as there are no primitives for continuously polling for events in the system. To circumvent this inadequacy, the handle abstraction supports rescheduling incomplete events until completion. The protocol involved with receiving large messages over network firstly reads a prepended header in the message that contains its size - as is the case with memcached. However, after reading the header memcached will yield to the libevent main-loop, waiting to be invoked when more data arrives. Recall that libevent has the *persistent* flag denoting that an event is always pending, creating the need for the event scheduler to be persistent The routine of rescheduling incomplete events is shown in Listing 4.3. If the `persistent` flag is set upon calling the `registerEvent` method, `messageDrain` will be called. This is a libevent-specific feature, however, we want to keep complexity of the shim-layers to the minimal. After each invocation of the event callback the function polls for incoming messages, and if there are pending messages in the queue `messageDrain` schedules itself again. By doing this, we can ensure that a process finishes processing the request for which it has registered.

Code Listing 4.3: Implements rescheduling of an event for all Handle type until a message is drained.

```
void
IHandle::messageDrain (void *arg, int status)
{
  auto handle_ctx = (event_context_t*)arg;
  auto _this = (IHandle*)handle_ctx->ec_handle;

  // Call the registered callback
  handle_ctx->ec_handler (handle_ctx->ec_arg, 0);

  // After we return from the callback we check if the message
  // has been consumed, if not we schedule this method again
  if (_this->poll (handle_ctx->ec_fd) == HANDLE_MSG_IN)
  {
    // Queue messageDrain to the scheduler
    _this->acont->GetThreadPool()->Schedule (IHandle::messageDrain,
                                             arg, __PRETTY_FUNCTION__);
  }
  else
  {
    delete handle_ctx;
  }
}
```

4.6 Diggi Sockets

The socket operations that are paramount to implement support for communication in SecureCached are receiving and sending data on a file descriptor. We omit the parts of the socket API that is bound to the IP protocol such as `getaddrinfo`, and `bind`. Specifically, the POSIX-socket functionality that we need to implement is:

Socket: allocate a POSIX file descriptor that we externally expose to the application, while identifying it with the internal Diggi socket implementation.

Listen: set the socket file descriptor state to a listening state; that is, activate the input stream as ready to accept incoming connections.

Accept: extracts the first connection on the queue of pending connections, and return a new file descriptor.

Write: write to a file descriptor - blocking and non-blocking.

Poll: unix version of poll. Query the state of multiple file descriptors.

Read: read a message from a file descriptor - blocking and non-blocking.

Sendmsg: similar to `write` but takes in a structure of type `msghdr` that contains the content to send.

From Section 3.3, Diggi's message manager is built as a messaging service between agents. It is also agnostic to where an agent resides, which can either be on the same instance of the Diggi runtime, or on another machine. If the destination of a message is an agent residing on different computer, the Diggi runtime will transparently forward that message through the network to the message manager of the respective machine. Therefore, the only abstraction a developer needs to know is the message manager. As mention in the start of this chapter, we implement a POSIX-compatible socket layer on top of the message manager to avoid unnecessary system calls for communication between agents residing within the same Diggi instance. Arguably, this adds an extra layer of abstraction since Diggi's message manager uses POSIX sockets for inter-node communication. However, if a socket is set to *blocking* and a read call yields to the Diggi scheduler until data arrives, we achieve high thread utilization for the agent. This is because Diggi's global message scheduler runs in a separate thread, and therefore achieves high thread utilization even if that thread blocks on a read call.

The Diggi socket abstraction is built using the Diggi handle abstraction in order for the socket layer to support events. The socket implementation in Diggi is verbatim POSIX sockets with the POSIX socket shim layer as the C interface to SecureCached. Since the interface for communication in Diggi is its message manager, there is no support for listening on unix devices. Diggi is built to only communicate between its logical applications component with a simple interface, and have no need for exposing hardware or protocol specific interfaces. Therefore, the Diggi socket interface will listen to any incoming messages with a specific message type that denotes a socket-type message and subsequently de-multiplex messages for delivery to requesting services on top. Diggi-socket also supports some of the socket-specific option which may be set on a file descriptor, such as blocking and non-blocking. If the socket is set to *blocking*, each call made to the POSIX socket API will yield until the operation can be completed.

To illustrate how Diggi sockets are implemented, we present a figure illustrating the same request pattern as shown in Figure 4.4. The requests flow between SecureCached and the components of the Diggi library OS involved in setting up a socket connection and instantiating a client session is shown in Figure 4.5. We omit the notification of a worker thread on incoming requests to only focus on Diggi sockets. SecureCached first allocates a socket in Diggi-socket through the POSIX API. It then starts listening for incoming connections by calling `listen`. Listen will subscribe to the *NETIO* messsage type from the massage manager, and provide a callback. The callback routine resides in Diggi-socket, and takes the file descriptor as argument. Whenever the message manager receives a message of that type it calls that routine which append the message to a queue identified with the specific file descriptor. After the file descriptor is ready to receive connections, SecureCached subscribes for read events on that file descriptor by calling `event_add`, which in turn will register that subscription to Diggi-socket.

When the message manager receives a `NETIO` message type, it calls subscribed routine in Diggi-socket with the file descriptor as argument. Diggi-socket finds a subscription on that particular file descriptor and calls the routine subscribed by the libevent shim-layer, which will notify SecureCached of the incoming connection. On every unique connection from a client, SecureCached will call `accept()` to initiate a session with the client. The function `accept()` will call Diggi-socket which extracts the first Diggi message from a queue of pending receives, create a new queue to which all messages from that session will be appended.

As shown in listing 4.4, a network session in Diggi is identified by a socket file descriptor, the AID of the client agent, and an id used internally by the message manager to create callback sessions. Diggi socket uses all three identifiers such that a client agent may have multiple sessions to a server. Moreover, the message id will be provided to the active callback routine handling incoming messages and connections. The POSIX socket API uses the void argument of `async_handler_t` callback type to pass the correct file descriptor of that particular session. The member `so_sfd`, is the non-negative integer that is externally exposed to application code.

Code Listing 4.4: Structure for the socket abstraction on top of the message manager in Diggi.

```
typedef struct diggi_socket
{
   //! External socket identifier
  uint64_t so_sfd;
  union
  {
     unsigned int so_attributes;
     struct
     {
         unsigned int SO_BLOCKING    : 1,
                      SO_NONBLOCKING : 1,
                      SO_SESSION     : 1,
                                     : 0;
    }so_flag;
  };

   //! Client identifier
   aid_t so_dest;
   //! Id of message session
   unsigned long so_id;
   async_cb_t so_handle;
   void *so_handle_args;
}diggi_socket_t;
```

Diggi's message manager exposes multiple methods for receiving messages: registering a callback method per message type, and one for a given message response. Internally, our socket implementation subscribes to the *NETIO* message type. Messages delivered by the message manager transient, meaning that a message will be deallocated by the message manager after delivery. Therefore, Diggi-socket persists messages by copying them to the internal queue. For I/O intensive applications such as memcached where requests can be as large as 1MB, having to copy each request may arguably impose an overhead.

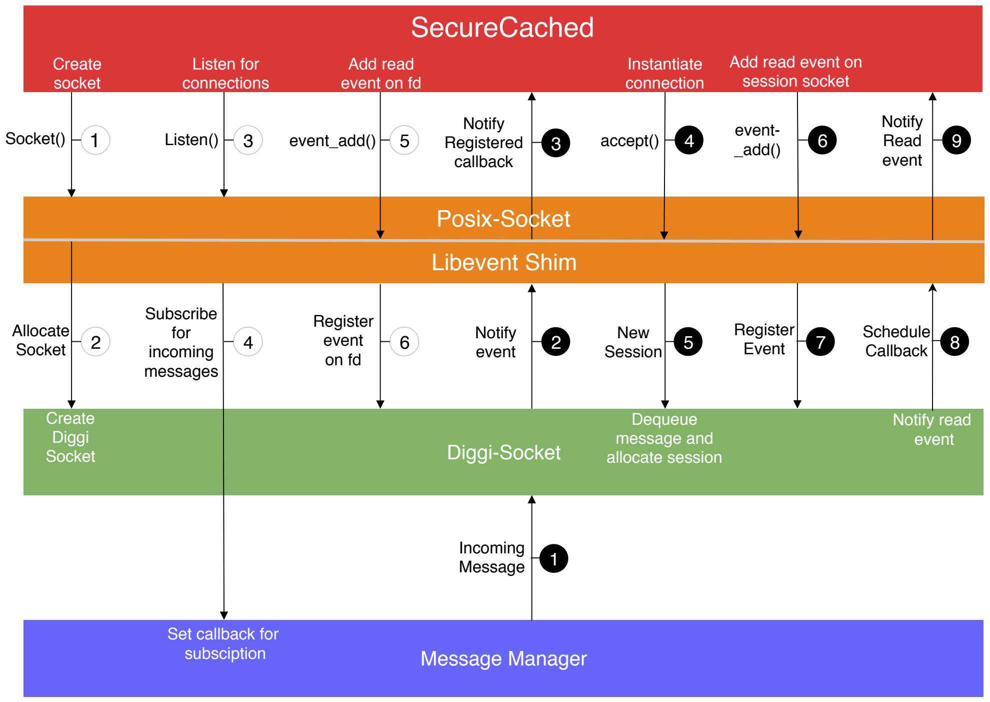

Figure 4.5: The figure show the steps involved in creating a file descriptor in Diggi, and subscribing a notification for that descriptor. Also the figure details the steps that happens when Diggi Socket receives a connection until SecureCached is notified of the read event. The steps of the former is highlighted in white circles, while the steps of the latter is highlighted in black circles.

4.7 Diggi Pipes

Memcached uses a combination of unix pipes and libevent to implement thread notification. Diggi supports this by extending the Diggi handle abstraction to support pipe semantics, with a POSIX pipe shim-layer as the interface to SecureCached. Specifically, the functions we need to implement are:

Pipe: generate two file descriptor - one for receiving end and one sending a data.

Read: extend the read method to support a file descriptor of type pipe.

Write: similar to read, extend this method to support a file descriptor of type pipe.

The difference between pipes and sockets is that we can not rely on the message manager to supply events on incoming messages. Instead, we schedule events based on writes to a pipe that have registered events through libevent. Incoming messages that are directed to a socket will invoke callbacks that are registered on the socket. For pipes, however, callbacks are scheduled to run whenever a notification pipe is written to. When a thread writes to a pipe that has a event subscription active, the callback will be pushed to the Diggi scheduler task queue. Internally, Diggi implements this with a circular ring buffer for storing the data sent through the pipe. The pipe implementation only supports inter-thread communication, as Diggi does not have the primitives for communicating between processes.

4.8 Memcached Client

Memcached clients support various features, and implements support for the entire memcached API. Diggi agents are light-weight, therefore, porting client libraries such as libmemcached will bloat the library OS. Instead we implement a minimal interface to the memcached API. In order to have agent-to-agent queries, we implemented a subset of the memcached operations and features which are:

Set: Unconditionally store a value associated with a given key. As opposed to Add which will fail if the item already exists.

Get: Retrieve a value associated with a key.

Delete: Delete the item with the specific key.

Multiple Servers: Allow the client to located keys in multiple SecureCached instances.

Consistent hashing: Consistently distribute the key space among SecureCached instances.

Our memcached client implementation exclusively uses the memcached binary protocol [3].

The client follows Diggi's asynchronous callback model, with the exception of the support for asynchronous operations combined with blocking response calls. That is, to support Yahoo! Cloud Serving Benchmark (YCSB) the mem-

3. https://github.com/memcached/memcached/wiki/BinaryProtocolRevamped

cached client has to support synchronous operations. An example of a set operation followed by a blocking call to retrieve the response is depicted in listing 4.5. The put operation will complete the sending of the request, and `getResponse` will yield to the Diggi scheduler until a the response for that request is received.

Code Listing 4.5: Example of an asynchronous put operation to memcached followed by a synchronous retrieval.

```
enum memcached_status status;

// Non-blocking call
status = client->put("key", 3, "value", 5);

struct memcached_response *resp;
// Blocking call - waits until response is received.
status = client->getResponse (&resp);
if (status != MEMCACHED_STATUS_NO_ERROR)
{
  // query failed
}
```

Since message delivery in Diggi deletes message after the registered delivery callback returns, the synchronous `getResponse` copies the message internally.

The memcached client also supports querying multiple memcached servers. Since half the memcached logic is placed on the client side, where the client is responsible for discovering and choosing the distribution of the key space. When a client is set up to support multiple memcached servers, every request has to be issued to the memcached instance responsible for a particular key. The client supports two methods of choosing the key space, both involving hashing the key. The first method uses the key hash modulo the number of memcached servers.

To identify memcached instances in a Diggi deployment, as we have no support for ip or ports, Diggi maintains a list of its agents and where they are located from which we can detect the memcached instances.

4.9 Summary

This chapter has presented SecureCached, a modified version of memcached to Diggi. This was achieved by making modifications to memcached and implementing OS-services in Diggi. Notably, SecureCached may run in single-threaded context by emulating the runtime behaviour of memcached. The extentions to the Diggi library OS were implementations of POSIX sockets and pipes, a subset of libevent, and an event framwork that unifies file descriptor types.

5 Evaluation

In this chapter we evaluate the SecureCached. Before we assess SecureCached in a distributed setup, we evaluate the communication throughput of Diggi to find how Diggi compares to native Linux sockets. We evaluate SecureCached in three different set-ups: between agents in the same Diggi process, between Diggi processes on a single machine, and between Diggi processes located on separate machines.

5.1 Experimental Setup

Four different machines were used to generate load, and one to run Secure-Cached. **Machine 1 & 2** have identical hardware specifications: Intel Core i5-6500 3.20 GHz Quad-Core processor with 4 logical cores, each core has a separate 32 x 4 way 32KB L1 data and instruction cached and 4 x 256 KB way L2 caches, and all core share a 6MB 12-way cached. Each processor is connected to 16 GB of DDR3 RAM with a front bus of 1600MHz. Both machines run Ubuntu 16.04 LTS with Linux kernel version 4.13.0-47.

Machine 3 & 4 is a Dell PowerEdge R330. Is is equipped with a Intel Xeon E3-1270 v5 processor with a base frequency of 3.6 GHz. The processor has 4 physical cores and 8 hyperthreads. Each core has a separate 64 x 8 way 32K L1 data and instruction cached, a 4 x 256 KB 4 way L2 cache, and all cores share a 8MB L3 cache. The machine runs Ubuntu 16.04 LTS with Linux kernel version

Machine	RTT
1	0.380 ms
2	0.369 ms
3	0.475 ms
4	0.461 ms

Table 5.1: Measure latencies from all machines used for load generation to the machine that runs SecureCached

4.4.0-119.

Machine 5 is used to run SecureCached and is an Intel server blade S1200SP. It is equipped with a Intel Xeon E3-1270 v6 processor running at 3.8GHz. Each core has a seperate 64 x 8 way 32K L1 data and instruction cached, a 4 x 256 KB 4 way L2 cache, and all cores share a 8MB L3 cache. It has 64 GB of DDR4 RAM running at 2133 MHz. The machine runs Ubuntu 16.04 LTS with Linux kernel version 4.13.0-37.

All machines are connected by a 1Gbps Ethernet link, however, they are connected to different network typologies. We therefore measure the Round-trip Time (RTT) latency from **machine 1-4** to **machine 5**. The measured RTT latency is listed in the Table 5.1. The RTT is measure with the *ping* commandline utility. The packet size for the measurents is set to 1Kb since all evaluation of SecureCached is done with value sizes of 1Kb.

The enclaves are created using Intel's open source kernel module [1]. Unless specified, all enclaves are compiled and run in hardware mode with the SGX_PRERELEASE flag. All code is compiled with GCC version 5.4.0. Note that all components used in the experiments, namely SecureCached and client agents, run on SGX-enabled hardware, and thus, within enclaves.

5.1.1 YSCB

YCSB is an open source benchmarking tool by Yahoo [45]. Its purpose is to create a reference benchmark for popular data serving systems, ranging from databases to in-memory key-value stores, by defining a set of core workloads that covers a wide range of use cases. We evaluate SecureCached using the popular YCSB load generator implementet as a Diggi agent. The agent first pre-loads the memcached instances with key-value pairs, and then issues *update* or *get* requests. The workloads are according to the pre-defined YCSB

1. https://github.com/intel/linux-sgx

get/update ratios, however, for our experiments we only use workload b: 95% GET requests and 5% UPDATES. The value size for all experiments is 1Kb. All experiments conducted which includes SecureCached have mean latencies for 100K measurements per client, and include a 96% confidence interval.

The YCSB Diggi agent used for load generation differs from the standard implementation of the framework. YCSB measures applications by incrementally increasing the frequency of operations. However, since the YCSB Diggi agent runs within an enclave fine-grained frequency control is difficult to implement. This is because enclave threads exit the enclave in order to access Linux time. Moreover, YCSB also increases the concurrency of the benchmark to increase load. Our approach to that is increasing the number of Diggi client agents.

5.2 Single thread Performance

Firstly, we assess the single threaded performance of the SecureCached by locating the client agents and the SecureCached agent within the same Diggi process. We want to assess the maximum requests that SecureCached is able to complete, without involving network overheads. Recall Diggi's global message scheduler from 3, which runs in the context of a single thread. Therefore, in addition to assessing the performance of SecureCached we also want to evaluate at which point the global message scheduler is saturated.

The experiment was conducted by increasing the the number of clients that query a single server agent; all running inside enclaves, and therefore messages sent between them will be subject to encryption. The experiment does not issue any system calls either, as agent-to-agent communication is within the same Diggi process.

Table 5.2 shows the parameters for the setup, which includes enclave sizes for the number of threads utilized in the experiment. The experiment was conducted on **machine 5** as it provides 8 hyperthreads, allowing us to test with up to six clients.

The results are are shown in Figure 5.1. SecureCached achieves a maximum throughput of 150K reqs/s with four clients. When more than four clients generate requests, the throughput decreases and latency increases. This can be explained by threads not being affinitized to a logical core. As the global thread scheduler polls for messages to be sent and received we expected to reach a cap, and not degrading performance. However, due to threads being scheduled across logical cores, the latency increases. Note that from two to three clients

	Enclave Size	Threads
Clients	10Mb	1-6
SecureCached	50Mb	1
Diggi Process	-	1
Total	110 MB	2-8

Table 5.2: Parameters for inter-enclave benchmark.

there is a drop in throughput, which increases again for four clients. During testing, the third client agent spends longer to finish its requests than the first two. Since the throughput is measure by dividing the total amount of requests for all client agents by the time of the longest running client agent, this affects our results. However, this happens because the third client agent will introduce the fifth thread to the experiment. As there are four physical cores on the machine, the fifth thread will compete with the other four threads for a physical core. This is then amortized when the sixth thread is introduced to the experiment (client agent). To conclude, this benchmark shows that SecureCached yields a maximum throughput of 150K req/s in Diggi, when network overhead is removed from the equation.

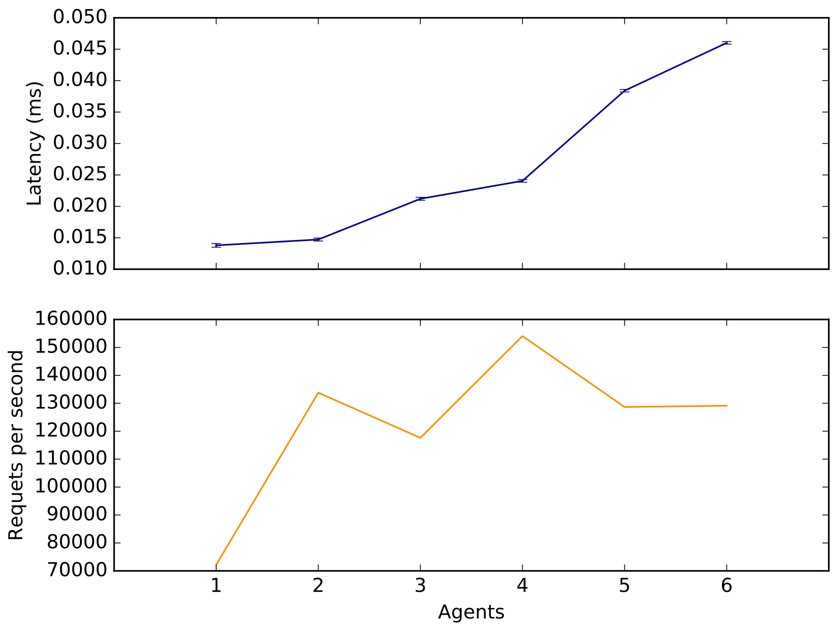

Figure 5.1: Measured throughput and latency for SecureCached where the clients are located within the same Diggi process. The uppermost plot shows the measured latency in miliseconds and the bottom most plot shows measured throughput in requests per second.

5.3 Inter-node communication baseline

As we explore to assess inter-node performance of SecureCached, we conduct experiments to find the theoretical maximum for inter-node communication in Diggi. To be able to compare the baseline performance to the SecureCached experiments, we include a value size of 1Kb. Evaluating the cap for communication allows us to assess the overhead of SecureCached queries as well as having a theoretical maximum for queries between agents located on different physical machines.

We first use `iperf` to define the maximum throughput of a regular application. Packet sizes of 1KB yielded 550 mbit/s maximal throughput.

We conduct the experiments by setting up one server agent and increasing the number of client agents. The client agent only uses Diggi's message manager to send packages to the server agent. We increase packets sizes from 16 to 4096 bytes in power-of-twos. Each run for a given packet size include 100K requests

	Enclave Size	Threads
Client	10Mb	1-3
Server	10Mb	1

Table 5.3: Parameters for communication baseline benchmark.

to the server agent.

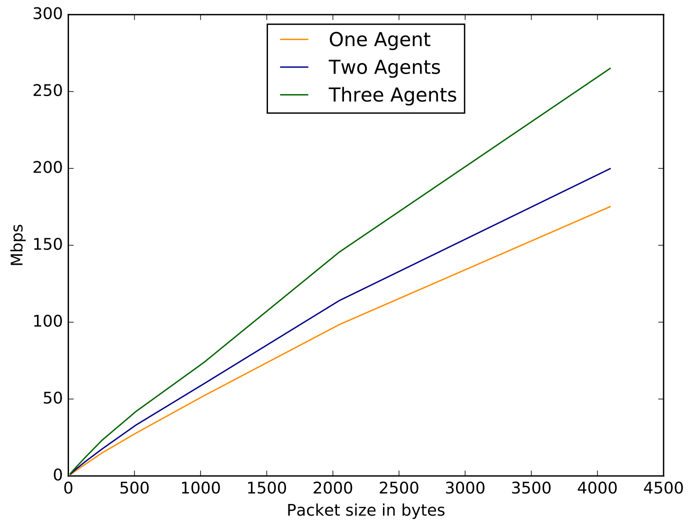

Figure 5.2: Baseline commincation throughput in megabit per second for agent-to-agent communication, between two Diggi process, each of which located on a separate machine. The number of agents issuing requests are increase from one to three, where each iteration includes 100K requests per power-of-two packet size.

From 5.2 we see that for packet sizes of 1Kb, Diggi achieves a throughput of approximately 74 mbps, Diggi therefore only achieves 13 % of native throughput, compared to measurements from *iperf*. Relating this to requests per second, packet sizes of 1KB yields around 9K. By that the maximum throughput a machine with four physical cores may achieve for packet sizes of 1Kb is 9K. We therefore deem that in order to impose greater load to SecureCached must utilize more machines in order to saturate a single-threaded SecureCached setup.

5.4 Inter-process performance

Diggi supports inter-process communication via POSIX sockets. We want to assess the throughput between two Diggi processes to find at which point Diggi's agent-to-agent communication handler experiences saturation. Recall that Diggi has a dedicated thread handling all outgoing and incoming requests.

We set up the benchmark on **machine 5**. All the details is shows in Table 5.4

	Enclave Size	Threads
Client	10Mb	1-5
SecureCached	50Mb	1
Diggi Process 1	-	1
Diggi Process 2	-	1
Total	100 MB	8

Table 5.4: Parameters for inter-process benchmark.

The plot in Figure 5.3 shows the results of the experiment. We increased the client agents issuing requests from one until five clients. Machine 5 only has four physical cores (8 hyperthreads), and we allocate eight logical threads for the highest number of client agents. This will cause expensive thread transition whenever a thread is relocated to a different core. However, we aim to assess the theoretical maximum load for the Diggi message scheduler.

The results support our claim to a certain extent, latency increases drastically from two clients. This could also be an effect of the single-threaded communication handler in Diggi. However, comparing that to the performance penalty of enclave thread transition we believe that it is the main cause for the added latency. The overall throughput does not rise above 40K req/s. After adding more than two clients it stabilizes to around the maximum. The maximum throughput is therefore capped to 40K req/s for SecureCached requests with value size 1Kb.

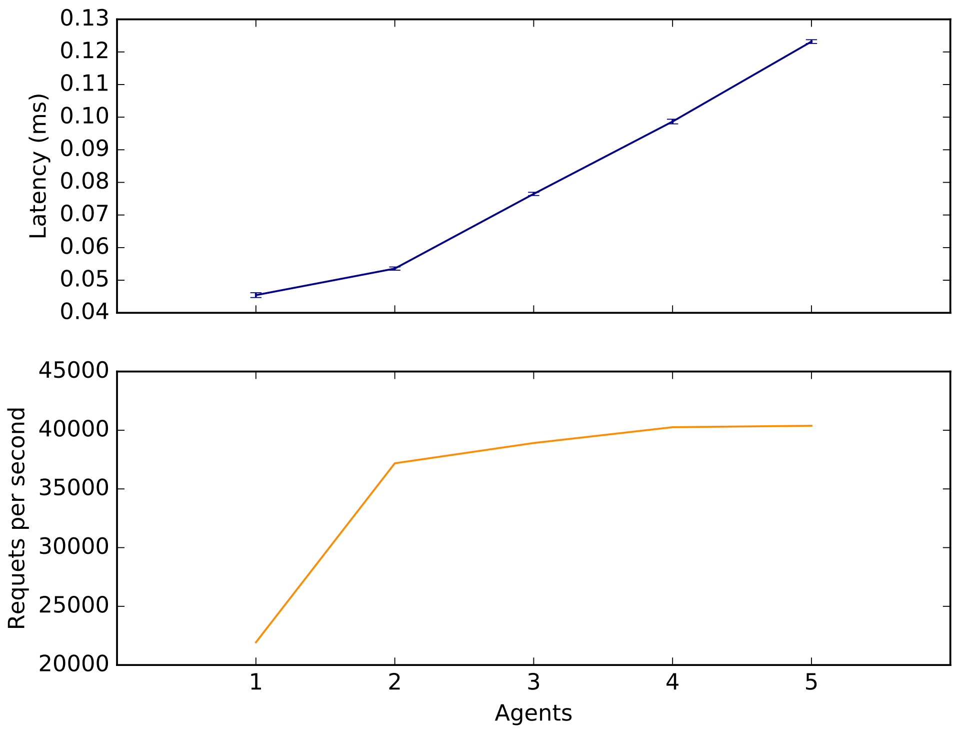

Figure 5.3: Throughput for SecureCached with inter-process agent-to-agent communication on a single machine. The throughput is shown in the lowermost plot, whilst the latency is the uppermost plot.

5.4.1 Exceeding the EPC size

As explained in Chapter 2, causing the OS to swap pages from the EPC is a performance penalty. We want to see how this affects the overall throughput for SecureCached. The parameters of the experiment is identical to the previous inter-process setup, only that we increase the size of the enclaves to exceed the EPC. The size of the SecureCached agent remains 50Mb, whilst the client agents are increased to 20Mb. We could have increased the size of the SecureCachedagent, however, as long as the total size of all enclaves exceeds the EPC boundary memory accesses will incur paging.

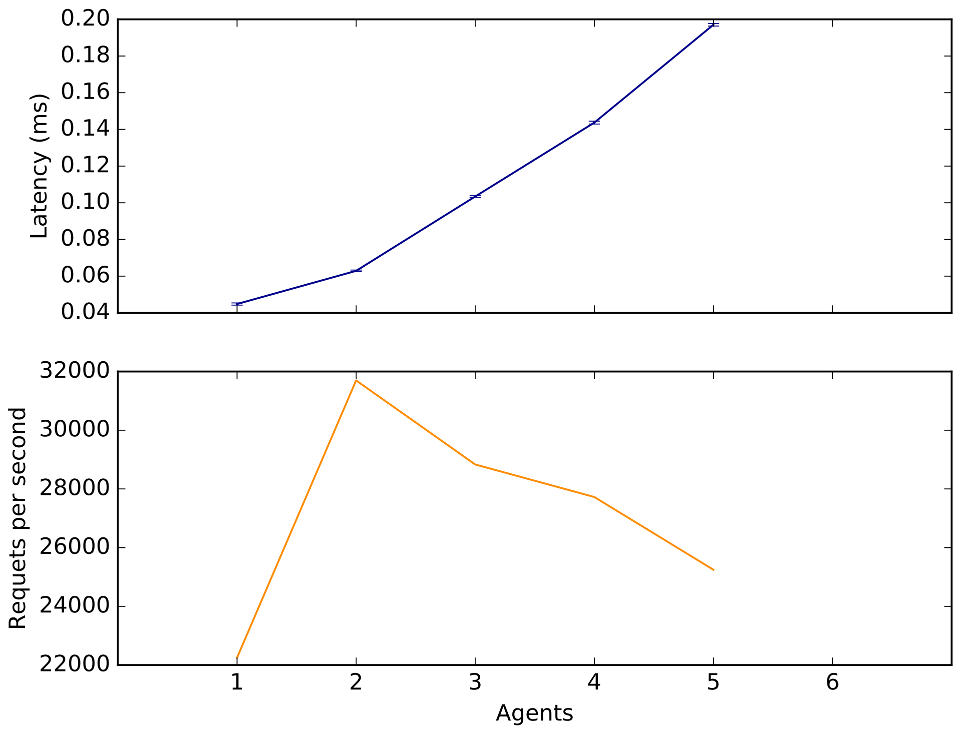

Figure 5.4: Throughput and latency for a inter-process setup of SecureCached and client agents where the size of the client agents are increased to exceed the EPC.

From the plot in Figure 5.4 we can infer that exceeding the EPC does degrade performance. Comparing it to Figure 5.3, latency is comparable up to two clients. For three clients and onwards, latency increases more than if the EPC boundary is not exceed. The throughput drastically decreases as well for number of clients above three.

5.5 Inter-node performance - Single Memcached Instance

To fully benchmark SecureCached in a distributed setup, we conduct experiments where we deploy one SecureCached agent on **machine 5** and place the YCSB clients on the four load generation machines, **machine 1-4**. Note that to avoid expensive enclave thread eviction during the experiments, we can

Machine	Enclave Size	Threads
1	10Mb	1-3
2	10Mb	1-3
3	10Mb	1-3
4	10Mb	1-3
SecureCached	50Mb	1

Table 5.5: Parameters for inter-node benchmark.

only run as many client agents as there are physical cores on the machine's processor, minus the thread dedicated for communication.

We want to assess at which point the server experiences saturation. Therefore, we start by deploying one client agent on one machine and increasing the number of clients by one at a time on that machine. When the amount of threads exceeds the physical cores on that machine we continue increasing clients by adding more to the next machine, and so forth. The experiment starts by loading the SecureCached agent with key-value pairs, before issuing requests. YCSB does this by increasing the amount of threads doing requests. We are unable to replicate that scenario as a Diggi agent can only run in the context of a single thread. Instead we delegate the responsibility of pre-loading the memcached server to one client agent which notifies the other client agents in the experiment to start issuing requests after it finishes. The key space is distributed among all the clients in the experiment in slices. Since our adoption of memcached is single-threaded, there will be no contention for keys.

The machines used for load generation all have different hardware specifications and connected to the server machine through varying network topologies. To that end, we measured the latency for each request done by a client agent and accumulate them for each unique machine. We also calculate the total requests per second for each machine. Note that for this experiment the memcached server enclave does not allocate more than the size of the EPC.

Measured mean latencies for the experiment is plotted in Figure 5.5. The total mean latency for all client agents residing on the same physical machine are plotted separately. That is, for every increase in client agents on a machine all latency is calculated together. We do this because of the heterogeneity of our machines and their interconnections. The introduction of a new machine to the experiment is marked by gray dotted line in plot.

From the results there are several things we are able to infer. The latency for all clients on a single machine increases by adding clients to that machine. This shows that Diggi's agent-to-agent communication is subject to contention

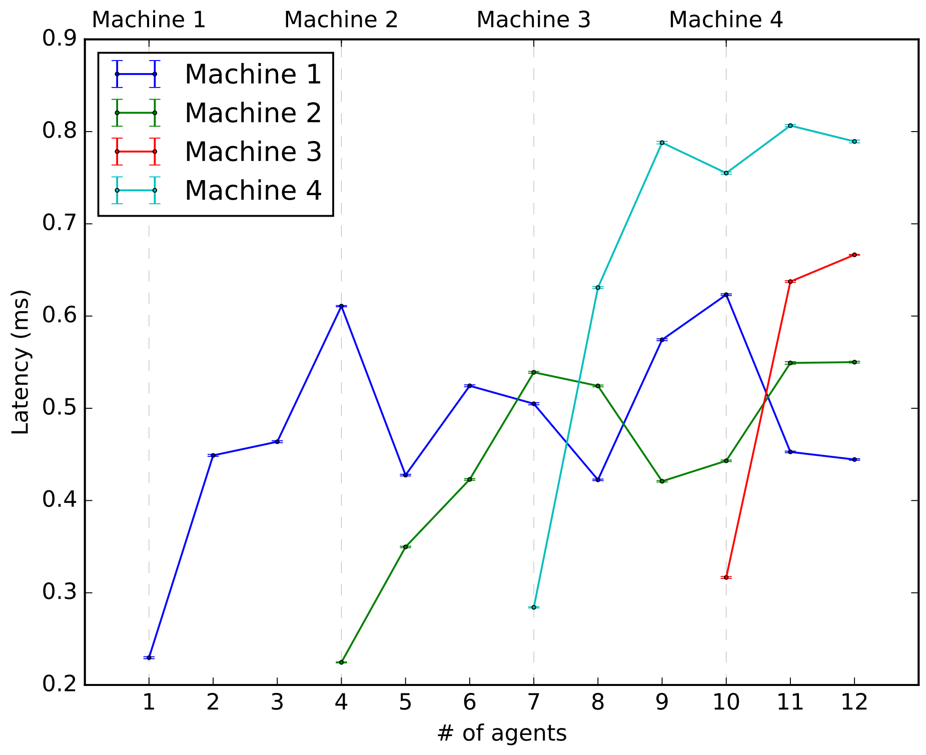

Figure 5.5: End-to-end latency between client agents and the memcached agent server. Latencies for client agents on the same machine are plotted together as the total number of client agents are increase in the experiment.

when there are several agents using it. Introducing the total amount of client agents does not affect the latency of the system. This is substantiated by the stable latency of machine one and machine two when the third and fourth mahine is introduce to the experiment. Machine one and two have unstable latencies for each addition of a client agent. Especially, when machine two is added, and comes down to the fact that a machine running only one client will issue requests faster than with a machine running three clients; that is, a single client will deprive other clients from issuing requests.

Machine three and four have a larger RTT for packets send to the server node, which the trends from the plot is showing. Overall, however, we are no able to reach the point saturation in the system. Even if the latency climbs up to 0.6 and 0.8 ms, respectively, for the machine three and four we do no consider a mean latency of such a small magnitude as an indication of a breaking point.

We also measure the overall throughput of the system during the same benchmark. In Figure 5.6, we plot the throughput collectively and individually for the machines. The YCSB benchmark runs load generation on a single machine, since we divide to load to multiple machines there is no point of coordination for testing. That is, all client agents each run 100K requests in the key space that is loaded into the SecureCached agent. This causes the clients to finish at different times, hence, to calculate the overall throughput in the system we divide the total number of requests by all clients with the time spent for the longest running client agent.

The results show that the single-threaded SecureCached agent handles approximately 16K requests per second when all four machines run three clients each (12 client agents). The throughput scales linearly until with the number of clients, however, there is a drop in throughput when machine three is introduced. The reason why there is a drop in throughput at that point comes down to two reasons. The RTT time from that machine to the server node is slightly higher and when the third client agents is added to that machine it uses longer time to finish than machine one and two. However, when the fourth machine is added it contributes to more requests in the same time-span, increasing the throughput again. Although the measurements are colored by the benchmark setup it shows that we are able to scale to 12 clients. Overall, we were not able to find at which point SecureCached becomes subject to saturation, and there are two reasons why:

1. The throughput measurement shows a trend of increasing throughput when clients are added, and the results does not show any decrease in throughput.

2. Latency does not increase significantly for machine 1 an 2 when the other machine contribute to the load. Arguably, a maximum latency of 0.8 ms does not show that there is a point of saturation.

The configuration setup may not be comparable or realistic to that of data center with homogeneous hardware and interconnections, however, the setup allowed us to evaluate the performance of SecureCached in Diggi.

5.6 Discussion

We have shown that it is feasible to run memcached on trusted hardware by using the Diggi framework. Memcached is a distributed caching service, and we implemented the support for deploying multiple instances of SecureCached across Diggi processes that are located on separate machines. We did not include the performance of such a setup, as the number of machines that

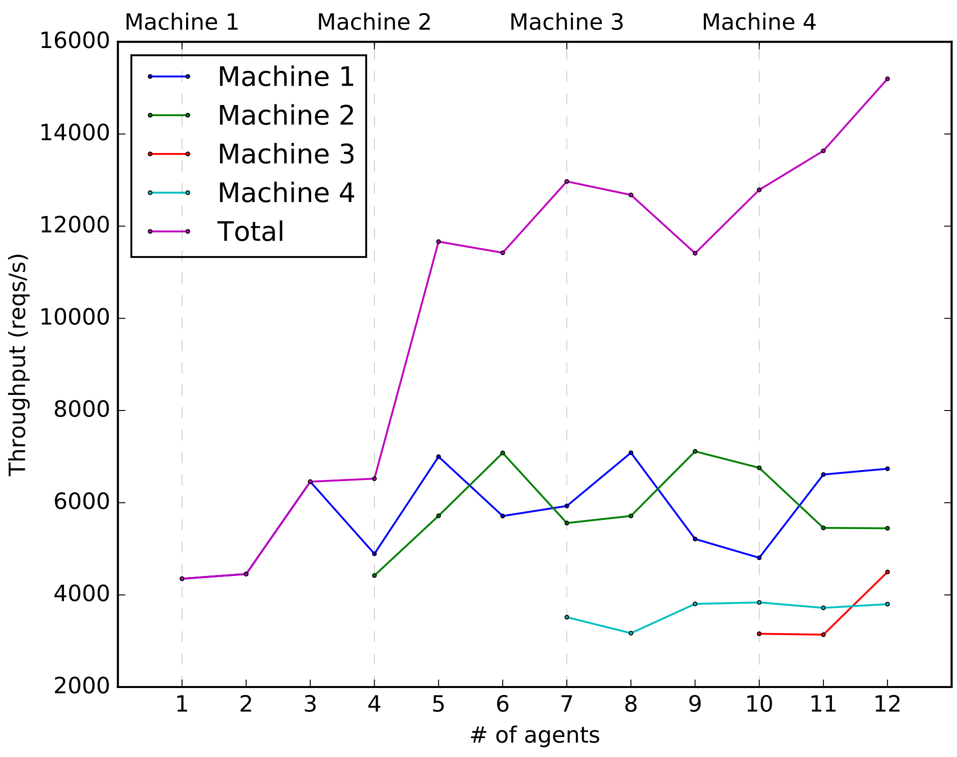

Figure 5.6: Measured throughput for a single SecureCached agent. Througput is measured overall for each machine, and the gray dotted line shows when a client is introduces on a new machines. The overall throughput of the SecureCached agent is measured by dividing the total amount of requests done by all clients, by the longest time spent to finish the experiment.

were available was not enough to utilize a multi-instance setup. Memcached instances are typically added to a system in order to increase the memory availability. This would be the case for the memcached agent as well due to the limited memory resources of SGX. Our memcached agent is single-threaded, and increasing the number of instances could also help performance. Reducing the size of each SecureCached agent, trading memory for concurrency.

In large-scale cloud services where the amount of data residing in caches can exceed several gigabytes, placing those caching service in SGX will incur too large performance overheads. As caches are built to speed up software system, such a performance overhead will be counterproductive. Therefore, in order for SGX-enabled caching services to be viable, the amount of data residing in the cache should be kept small.

5.7 Comparison to Other Frameworks

Porting memcached to run on trusted hardware has been investigated in research. Our implementation is not comparable to the performance others report, however, we will compare the framework that have been utilized to run memcached on SGX.

The first was scone [34], achieving near vanilla memcached throughput. Compared to our implementation, scone implements multi-threaded support for memcached, and increased concurrency for network handling. Diggi only uses a single thread to service network packets, whilst scone utilizes a greater amount of threads to service network traffic than the amount of threads used for memcached. In addtion to that, scone service the system calls in a kernel module, and can provide network packets directly to the enclave without switching privileged levels.

Eleos compares Graphene-SGX and a modified version of it that has asynchronous system calls and a user-level virtual memory manager [39]. The baseline Graphene-SGX framework running single-threaded memcached reached 20K req/s. With the two extensions, eleos achieved twice the throughput. They report that the speedup is caused by avoiding expensive enclave thread transitions. They also tested their virtual memory manager by setting the memcached memory pool to 500Mb. According to their results they were able to obtain a performance that is comparable to a memcached instance that fits the epc, 2.5x speedup over Graphene when the EPC is not exeeded and 2.0x when it is.

5.8 Summary

This chapter has conducted experiments to evaluate SecureCached. The experiments showed that a single-instance SecureCachedscales to 12 clients. However, the hardware resources used to conduct the experiments were not enough to determine the maximum amount of requests one SecureCached agent is able to handle.

6 Concluding Remarks

This chapter will conclude this thesis, summarize our contributions and results, relating them to our thesis statement.

6.1 Conclusion

This thesis aimed to evaluate the feasibility of running a distributed caching service on Diggi without sacrificing performance or functionality.

Specifically, our thesis is:

> *The Memcached codebase can be modified to run within the Diggi Library OS.*

We successfully ported a modification of Memcached, SecureCached to run in Diggi with the following contributions:

1. SecureCached - a port of Memcached to run on Diggi and SGX.
2. Shim extensions to the Diggi library OS that replaces legacy OS services required by Memcached.

Our evaluation showed that SecureCached is able to provide the legacy API of

Memached by succesfully running the Yahoo! Cloud Serving Benchmark (YCSB) benchmark. Therefore, we demonstrated the feasibility of having a distributed cache deployed in a secure execution environment. Diggi is now able to move cached data into a security domain without diminishing the confidentiality and integrity of sensitive data. To that end, our thesis holds.

6.2 Future Work

We propose future work for our solution, with regards to security and memcached functionality.

6.2.1 Multi-Threading

Memcached is implemented to take advantage of the paralellism of the cpu. Many of the side features of memcached also require threading. The only restriction to that is that the number of threads that may execute inside an enclave must be given prior to creating it. For an application such as memcached that start threads at startup, depending on the configurated parameters, that can be challenging. Therefore, dedicated threads that service a log stream through the network, and the number threads handling requests must be given. We therefore propose a future implementation of multi-threading in Diggi.

6.2.2 Feature Rich Memcached Client

The authors are unaware of the extent to which all the client-side memcached features are used. We implemented a minimal client API to service the YCSB testing framework. For a secure caching service, implementing features such as compression that allows a memcached agent residing in SGX to have a smaller memory footprint. Taking into account the memory restrictions of the SGX architecture, one additional client feature would improve on that. E.g, allow application developers to mark an item as sensitive that requires to be storage in secure hardware. This would require the client to be aware of which memcached agents run on trusted hardware and which instances run on untrusted system software. Application developers can choose to store only the most sensitive data in a memcached agent residing in trusted hardware to reduce the memory footprint.

Bibliography

[1] J. Tang, Y. Cui, Q. Li, K. Ren, J. Liu, and R. Buyya, “Ensuring security and privacy preservation for cloud data services,” *ACM Comput. Surv.*, vol. 49, pp. 13:1–13:39, June 2016.

[2] S. Luo, Z. Lin, X. Chen, Z. Yang, and J. Chen, “Virtualization security for cloud computing service,” in *2011 International Conference on Cloud and Service Computing*, pp. 174–179, Dec 2011.

[3] M. Almorsy, J. C. Grundy, and I. Müller, “An analysis of the cloud computing security problem,” *CoRR*, vol. abs/1609.01107, 2016.

[4] D. Puthal, B. P. S. Sahoo, S. Mishra, and S. Swain, “Cloud computing features, issues, and challenges: A big picture,” in *2015 International Conference on Computational Intelligence and Networks*, pp. 116–123, Jan 2015.

[5] A. ARM, “Security technology building a secure system using trustzone technology (white paper),” *ARM Limited*, 2009.

[6] I. Corp, “Software guard extensions programming reference, ref. 329298-002us..” . https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf, Oct. 2014.

[7] D. Kaplan, J. Powell, and T. Woller, “Amd memory encryption,” *White paper*, 2016.

[8] T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel, “Ryoan: A distributed sandbox for untrusted computation on secret data,” in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, (Savannah, GA), pp. 533–549, USENIX Association, 2016.

[9] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich, “Vc3: Trustworthy data analytics in the cloud using sgx,” in *Security and Privacy (SP), 2015 IEEE Symposium on*, pp. 38–

54, IEEE, 2015.

[10] S. Brenner, C. Wulf, D. Goltzsche, N. Weichbrodt, M. Lorenz, C. Fetzer, P. Pietzuch, and R. Kapitza, “Securekeeper: Confidential zookeeper using intel sgx,” in *Proceedings of the 17th International Middleware Conference*, Middleware ’16, (New York, NY, USA), pp. 14:1–14:13, ACM, 2016.

[11] L. Guan, P. Liu, X. Xing, X. Ge, S. Zhang, M. Yu, and T. Jaeger, “Trustshadow: Secure execution of unmodified applications with arm trustzone,” in *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys ’17, (New York, NY, USA), pp. 488–501, ACM, 2017.

[12] D. Johansen, K. Marzullo, and K. Lauvset, “An approach towards an agent computing environment,” in *Proceedings. 19th IEEE International Conference on Distributed Computing Systems. Workshops on Electronic Commerce and Web-based Applications. Middleware*, pp. 78–83, 1999.

[13] D. Johansen, R. van Renesse, and F. B. Schneider, “Operating system support for mobile agents,” in *Proceedings 5th Workshop on Hot Topics in Operating Systems (HotOS-V)*, pp. 42–45, May 1995.

[14] D. Johansen, H. Johansen, and R. van Renesse, “Environment mobility: Moving the desktop around,” in *Proceedings of the 2Nd Workshop on Middleware for Pervasive and Ad-hoc Computing*, MPAC ’04, (New York, NY, USA), pp. 150–154, ACM, 2004.

[15] G. Hartvigsen and D. Johansen, “Co-operation in a distributed artificial intelligence environment—the stormcast application,” *Engineering Applications of Artificial Intelligence*, vol. 3, no. 3, pp. 229 – 237, 1990.

[16] H. D. Johansen, R. V. Renesse, Y. Vigfusson, and D. Johansen, “Fireflies: A secure and scalable membership and gossip service,” *ACM Trans. Comput. Syst.*, vol. 33, pp. 5:1–5:32, May 2015.

[17] H. D. Johansen, D. Johansen, and R. van Renesse, “Firepatch: Secure and time-critical dissemination of patches,” 2006.

[18] A. T. Gjerdrum, R. Pettersen, H. D. Johansen, and D. Johansen, “Performance of trusted computing in cloud infrastructures with intel sgx,” in *Proceedings of the 7th International Conference on Cloud Computing and Services Science. Porto, Portugal: SCITEPRESS*, pp. 696–703, 2017.

[19] R. v. Renesse, H. Johansen, N. Naigaonkar, and D. Johansen, “Secure

abstraction with code capabilities," in *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pp. 542–546, Feb 2013.

[20] H. D. Johansen, E. Birrell, R. van Renesse, F. B. Schneider, M. Stenhaug, and D. Johansen, "Enforcing privacy policies with meta-code," in *Proceedings of the 6th Asia-Pacific Workshop on Systems*, APSys '15, (New York, NY, USA), pp. 16:1–16:7, ACM, 2015.

[21] H. K. Stensland, V. R. Gaddam, M. Tennøe, E. Helgedagsrud, M. Næss, H. K. Alstad, A. Mortensen, R. Langseth, S. Ljødal, O. Landsverk, C. Griwodz, P. Halvorsen, M. Stenhaug, and D. Johansen, "Bagadus: An integrated real-time system for soccer analytics," *ACM Trans. Multimedia Comput. Commun. Appl.*, vol. 10, pp. 14:1–14:21, Jan. 2014.

[22] D. Johansen, P. Halvorsen, H. Johansen, H. Riiser, C. Gurrin, B. Olstad, C. Griwodz, Å. Kvalnes, J. Hurley, and T. Kupka, "Search-based composition, streaming and playback of video archive content," *Multimedia Tools and Applications*, vol. 61, pp. 419–445, Nov 2012.

[23] D. Johansen, M. Stenhaug, R. B. A. Hansen, A. Christensen, and P. M. Høgmo, "Muithu: Smaller footprint, potentially larger imprint," in *Seventh International Conference on Digital Information Management (ICDIM 2012)*, pp. 205–214, Aug 2012.

[24] Kvalnes, D. Johansen, R. van Renesse, F. B. Schneider, and S. V. Valvag, "Omni-kernel: An operating system architecture for pervasive monitoring and scheduling," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, pp. 2849–2862, Oct 2015.

[25] P. J. Denning, D. E. Comer, D. Gries, M. C. Mulder, A. Tucker, A. J. Turner, and P. R. Young, "Computing as a discipline," *Computer*, vol. 22, pp. 63–70, Feb 1989.

[26] C. Mitchell and I. of Electrical Engineers, *Trusted Computing*. Computing and Networks Series, Institution of Engineering and Technology, 2005.

[27] S. Kim, Y. Shin, J. Ha, T. Kim, and D. Han, "A first step towards leveraging commodity trusted execution environments for network applications," in *Proceedings of the 14th ACM Workshop on Hot Topics in Networks*, HotNets-XIV, (New York, NY, USA), pp. 7:1–7:7, ACM, 2015.

[28] J. E. Ekberg, K. Kostiainen, and N. Asokan, "The untapped potential of trusted execution environments on mobile devices," *IEEE Security Privacy*,

vol. 12, pp. 29–37, July 2014.

[29] M. Sabt, M. Achemlal, and A. Bouabdallah, "Trusted execution environment: What it is, and what it is not," in *2015 IEEE Trustcom/BigDataSE/ISPA*, vol. 1, pp. 57–64, Aug 2015.

[30] "Trusted execution environment (tee) guide." `https://www.globalplatform.org/mediaguidetee.asp`. Accessed: 2018-05-19.

[31] N. Santos, H. Raj, S. Saroiu, and A. Wolman, "Using arm trustzone to build a trusted language runtime for mobile applications," *SIGARCH Comput. Archit. News*, vol. 42, pp. 67–80, Feb. 2014.

[32] Z. Du, Z. Ying, Z. Ma, Y. Mai, P. Wang, J. Liu, and J. Fang, "Secure encrypted virtualization is unsecure," *CoRR*, vol. abs/1712.05090, 2017.

[33] O. Weisse, V. Bertacco, and T. Austin, "Regaining lost cycles with hotcalls: A fast interface for sgx secure enclaves," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pp. 81–93, June 2017.

[34] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. Stillwell, *et al.*, "Scone: Secure linux containers with intel sgx.," in *OSDI*, vol. 16, pp. 689–703, 2016.

[35] A. Baumann, M. Peinado, and G. Hunt, "Shielding applications from an untrusted cloud with haven," *ACM Trans. Comput. Syst.*, vol. 33, pp. 8:1–8:26, Aug. 2015.

[36] D. E. Porter, S. Boyd-Wickizer, J. Howell, R. Olinsky, and G. C. Hunt, "Rethinking the library os from the top down," *SIGARCH Comput. Archit. News*, vol. 39, pp. 291–304, Mar. 2011.

[37] C. che Tsai, D. E. Porter, and M. Vij, "Graphene-sgx: A practical library OS for unmodified applications on SGX," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, (Santa Clara, CA), pp. 645–658, USENIX Association, 2017.

[38] C.-C. Tsai, K. S. Arora, N. Bandi, B. Jain, W. Jannen, J. John, H. A. Kalodner, V. Kulkarni, D. Oliveira, and D. E. Porter, "Cooperation and security isolation of library oses for multi-process applications," in *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, (New York, NY, USA), pp. 9:1–9:14, ACM, 2014.

[39] M. Orenbach, P. Lifshits, M. Minkin, and M. Silberstein, "Eleos: Exitless os services for sgx enclaves," in *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, (New York, NY, USA), pp. 238–253, ACM, 2017.

[40] S. Shinde, D. Tien, S. Tople, and P. Saxena, "Panoply: Low-tcb linux applications with sgx enclaves," in *Proceedings of the Annual Network and Distributed System Security Symposium (NDSS)*, p. 12, 2017.

[41] B. Fitzpatrick, "Distributed caching with memcached," *Linux J.*, vol. 2004, pp. 5–, Aug. 2004.

[42] Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Characterizing facebook's memcached workload," *IEEE Internet Computing*, vol. 18, pp. 41–49, Mar 2014.

[43] J. Jose, H. Subramoni, K. Kandalla, M. Wasi-ur Rahman, H. Wang, S. Narravula, and D. K. Panda, "Scalable memcached design for infiniband clusters using hybrid transports," in *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*, pp. 236–243, IEEE Computer Society, 2012.

[44] I. of Electrical and E. Engineers, "Ieee standard for information technology-portable operating system interface (posix): approved september 15, 1993: Ieee standards board; approved april 14, 1994: American national standards institute," Inst. of Electrical and Electronics Engineers.

[45] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, (New York, NY, USA), pp. 143–154, ACM, 2010.