2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks

Secure Tera-scale Data Crunching with a Small TCB

Bruno Vavala[1,2], Nuno Neves[1], Peter Steenkiste[2]
[1]*LaSIGE, Faculdade de Ciências, Universidade de Lisboa, Portugal* [2]*CSD, Carnegie Mellon University, U.S.*

***Abstract*—Outsourcing services to third-party providers comes with a high security cost—to fully trust the providers. Using trusted hardware can help, but current trusted execution environments do not adequately support services that process very large scale datasets. We present LAST$^{\mathrm{GT}}$, a system that bridges this gap by supporting the execution of self-contained services over a large state, with a small and generic trusted computing base (TCB). LAST$^{\mathrm{GT}}$ uses widely deployed trusted hardware to guarantee integrity and verifiability of the execution on a remote platform, and it securely supplies data to the service through simple techniques based on virtual memory. As a result, LAST$^{\mathrm{GT}}$ is general and applicable to many scenarios such as computational genomics and databases, as we show in our experimental evaluation based on an implementation of LAST$^{\mathrm{GT}}$ on a secure hypervisor. We also describe a possible implementation on Intel SGX.**

1. Introduction

Outsourced applications such as cloud services (databases, storage, etc.) are widely deployed but strong security guarantees are taken for granted. The de facto security model assumes that the service provider is fully trusted. In the real world, however, one third of the top threats listed by the Cloud Security Alliance [33] concern an attacker tampering with the integrity of computation or data, namely: (i) service hijacking [38], (ii) malicious insiders [1], (iii) system vulnerabilities [39], and (iv) shared technology issues [2]. This can raise suspicions on the trustworthiness of the results produced by a service.

The above threats stem from at least three issues:

- the lack of strong execution isolation, whereby a subverted OS, or hypervisor, or service application, can make threats affect other running software.
- a large TCB, which makes systems hard to verify; also, when it includes the OS—containing millions of lines of code [35]—a bug in the kernel [34] endangers security of all the applications and data.
- a complex OS interface—hundreds of system calls—which is difficult to secure [30] and whose malicious alteration can subvert an application [3].

Unfortunately, service owners and end-users have little or no means to distinguish between correct and compromised service code or input data by just looking at the results received from the cloud.

Trusted Computing (TC) technology is making progress towards allowing clients to verify results. The technology (e.g., Trusted Platform Modules (TPMs) [4] and Intel SGX [29]) is available in commodity platforms, and it is tied to a hardware root of trust certified by the manufacturer. This can be used by a service provider to isolate the service execution and to attest the identity of the executed code for remote verification.

Software support for such trusted hardware however is not (or just partially) suitable for many applications that process a huge amount of data (e.g., clinical decision support [5], predictive risk assessment for diseases [6], malware detection [7], workloads for sensitive financial records outsourced on public clouds [8], and genome analytics [9]). Previous systems support the execution of either small pieces of code and data [10], or large code bases [11], or specific software like database engines [12] or MapReduce applications [13]. Recent work [14] has shown how to support unmodified services. However, since "the interface between modern applications and operating systems is so complex" [30], it relies on a considerable TCB that includes a library OS. In addition, the above systems are specific for TPMs [10], [15], secure coprocessors [12], or Intel SGX [13]. Hence, porting them to alternative architectures (e.g., the upcoming AMD Secure Memory Encryption and Secure Encrypted Virtualization [36], [37]) requires significant effort. Clearly, it is desirable to design a generic system "not relying on idiosyncratic features of the hardware" [16].

We present LAST$^{\mathrm{GT}}$, a system that can handle a LArge STate on a Generic Trusted component with a small TCB. LAST$^{\mathrm{GT}}$ supports a wide range of applications and hardware because its design only relies on commonly available hardware features—mainly *paged virtual memory*. LAST$^{\mathrm{GT}}$ uses memory maps that allow the application to manage the placement of data in memory, and authenticated data structures for efficiently validating the data before it is processed. As most of the LAST$^{\mathrm{GT}}$'s mechanisms (e.g., data validation and memory management) are implemented at the application level, they can be optimized for different application requirements. LAST$^{\mathrm{GT}}$ ultimately delivers the following guarantee: *if the client can verify the results attested by the trusted component on the service provider platform, then the client request was processed by the intended code on the intended input state, so the received response can be trusted.*

We provide the following contributions.

- We describe LAST$^{\mathrm{GT}}$'s design, and show how it can protect large-scale data in memory efficiently and how it enables a client to verify the correctness of service code, data and results.
- We detail how LAST$^{\mathrm{GT}}$ has been implemented on XMHF-TrustVisor [10] using a commodity platform equipped with a TPM. Also, we discuss a possible implementation using the Intel SGX instruction set. In addition, we highlight important differences between the two architectures and how LAST$^{\mathrm{GT}}$ deals with them.
- We evaluate our XMHF-TrustVisor-based implementation for datasets up to one terabyte. We show that LAST$^{\mathrm{GT}}$ has a small TCB compared to state-of-the-art prototypes, and good performance. We also discuss expected improvements with an SGX-based implementation.

2. Related Work

We describe related work on trusted execution, trusted execution targeting large-scale data, and other solutions for ensuring the integrity of computation on large data.

Trusted Execution Environments. TrustVisor [10], Minibox [15] and Haven [14] all support secure execution. The first two focus on keeping the TCB small by removing the OS from the trust boundaries, thus supporting self-contained applications (i.e., with statically linked libraries and no OS dependencies). Haven instead bloats the TCB with a library OS—a refactored Windows 8—and can therefore run unmodified binaries. Security is achieved through hardware-based isolation mechanisms and by removing [10] or reducing [15], [14] the interface with the untrusted environment to protect against Iago attacks [3] (i.e., system calls that return values crafted by a malicious kernel).

These systems handle data I/O as follows. TrustVisor [10] transfers data to/from the secure environment at execution startup and termination only. The application thus receives all input data upfront. This can be inefficient if not all data is used, and the data size is also bounded by the physical memory. MiniBox [15] and Haven [14] implement additional system calls for dynamic memory and secure file I/O. Both construct a hash tree over the data, encrypt the data and handle I/O through the interface with the untrusted environment for disk access. However, working with a full hash tree in memory does not scale for applications that operate on a large state, since the hash tree itself can consume a large amount of memory. Also, their design introduces several system calls (thereby enlarging the interface), though fewer than an OS interface, that must be secured against Iago attacks.

Trusted execution on large data. M^{2}R [17] and VC3 [13] were designed for trusted execution of large-scale MapReduce applications. VC3 leverages Intel SGX for guaranteeing integrity and confidentiality of map and reduce functions. M^{2}R improves the level of privacy by hiding the memory access patterns through a secure data shuffler. Compared with the earlier platforms, these two systems achieve a small TCB at the expense of generality, since they only support MapReduce.

Other SGX applications. Graphene-SGX [41] can run unmodified Linux applications. As it includes a library OS, the same arguments that we used for Haven apply. Scone [18] secures Docker container applications while Panoply [19] secures Linux applications. The former supports multi-threaded container applications and has a larger TCB, mainly due to the libc library, while the latter is designed for multi-process applications and has a smaller TCB since it exposes a POSIX-level interface thus leaving the libc library outside the enclave. Ryoan [20] secures a distributed sandbox by leveraging SGX to ensure that possibly untrusted code can use but not leak sensitive data. These systems are orthogonal to LAST^{GT} since they focus on secure concurrent/distributed processing, not large-scale data. In addition, they expose from tens [18][20] to hundreds [19] of interface calls, many of which are related to data I/O from/to files. LAST^{GT} complements them with secure in-memory large-scale data handling that requires no additional interface, but instead relies on page faults, and handles data authentication in a scalable fashion.

Additional approaches. SecureMR [21] TrustMR [22] BFT-MR [23] use different replication technique for MapReduce computations to provide security guarantees against misbehaving execution replicas. VPR [24] proposes a secure passive replication scheme based on trusted hardware for improving the availability of generic services. LAST^{GT} does not use replication and is orthogonal to VPR.

3. LASTGT Overview

We give an overview of LAST^{GT}, presenting its key ideas, benefits and challenges.

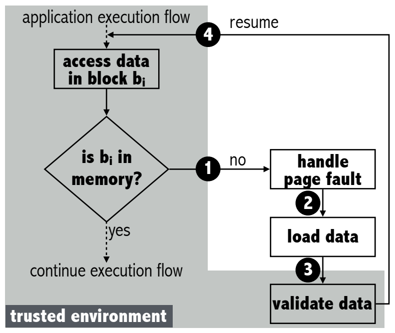


Fig. 1: Example of a program execution inside a trusted environment that offloads data I/O from storage or network devices to untrusted code.

Operation. LAST^{GT} allows a client to send requests and verify the results produced by a large-state service that runs on an untrusted platform. The execution of the self-contained service application is secured by means of a trusted hardware component. This component enables the establishment of an isolated execution environment, where the trusted code is identified and later attested. The service executes requests received from the client by reading and writing local state of up to a tera-byte (in our implementation) maintained in a set of files. LAST^{GT} ensures the integrity of the data used during the secure execution, exploiting the key ideas described below. Next, the service generates an attested reply for the client. The reply binds together the identities of the service code, the local state used by the service, the client's request, and the reply. Finally, the client verifies and accepts (if valid) the reply.

Key ideas. The core of LAST^{GT} is a secure and efficient data loading technique based on paged virtual memory and asynchronous handlers (Fig. 1). LAST^{GT} presents the large state to the service code as a memory region in its address space, thus allowing it to access the data directly (shaded area, left-side) without requiring explicit calls to privileged code. Data however is not preloaded for efficiency reasons and possible memory constraints. Instead, accesses result in page faults ❶ that LAST^{GT} handles transparently by moving the data from the untrusted part of the system ❷ into the secure environment. While the service application remains interrupted, the data is cryptographically validated ❸ by a trusted application-level handler (shaded area, right-side) whose execution is asynchronous (i.e., independent from the service application). Only if the loaded data is valid, is the interrupted service allowed to resume ❹.

Benefits offered by this design include:

- First, it reduces the problem of large-scale data handling to a virtual memory management problem. As virtual memory is *widely* supported, LAST^{GT} can be implemented on pretty much any TC-capable systems, enabling hardware diversity.
- Second, it keeps the TCB small by not including the OS. Moreover, it does not involve system calls to privileged or untrusted code, that may create vulnerabilities.
- Third, it does not need alterations to the service code since data validation and integrity protection is done transparently.
- Fourth, data validation, integrity protection and (un)loading are handled by customizable application-level code. This allows tuning the authenticated data structures to data access patterns, upgrading deprecated cryptographic algorithms, or devising application-specific data eviction policies.

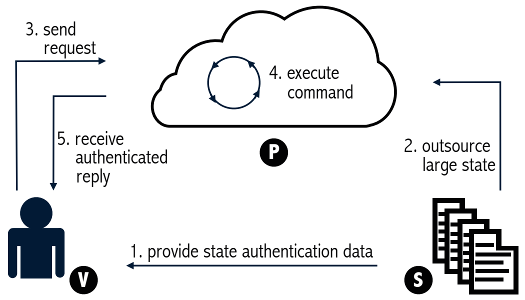

3. send request
4. execute command
5. receive authenticated reply
P
2. outsource large state
1. provide state authentication data
V
S

Fig. 2: Three-party (Verifier, Prover, Source) system model.

Challenges. Implementing LaSt^{GT} has several challenges:

Offloading I/O securely to untrusted code. Transferring data (e.g., disk I/O) is not useful computation for the application, but can increase the TCB size and put peripherals in the trust boundaries. Offloading such operations to untrusted code reduces the TCB but requires means to validate or protect data when it crosses the trust boundaries.

Transparency. Services should only have to access data and perform computation, without dealing with orthogonal issues such as data loading and state management. Hence, making such mechanisms fully transparent simplifies service development and the use of LaSt^{GT}.

Securely overcoming memory constraints. Physical memory is limited, especially for secure executions. For example, on a recent Dell Optiplex 7040, SGX is constrained to use up to 128MB of memory (out of 32GB). Hence, efficient memory management is needed for handling a terabyte-scale state and the associated authenticated data structure in memory.

Dealing with architectural differences. The hardware platforms for secure execution have very different architectures, making it hard to hide their differences through abstraction. For example, XMHF-TrustVisor and SGX use completely different mechanisms for secure execution and for paging. Devising a single design that works for both platforms is challenging.

4. System Model

We assume three parties (Fig. 2): a trusted source S producing lots of data (*user state*); an untrusted service provider P with significant computational resources; a trusted verifier V that uses P's resources. S gives authentication data to V (Step 1) and the user state to P (Step 2). V sends requests to P (Step 3), who applies them to the data (Step 4) and returns replies (Step 5) that V checks. We focus on a single request/reply exchange with the client. Extensions can be devised to deal with subsequent requests and to authenticate state updates.

Assumptions. We make common assumptions [10], [14], [11].

P is untrusted as it may be subject to internal/external attacks. For example, P's system administrator could compromise the software running on the platform, including the OS and running services/applications.

P is equipped with a trusted component that provides hardware-based security services such as isolated code execution, code identification (cryptographic hash of code) and attestation—a digital signature, computed by the trusted hardware with its embedded (or securely provisioned [31]) keys, of one or more identities (assertions [31], or cryptographic hash values). The trusted component is assumed to be robust against software attacks and free from physical tampering. Also any code, even malicious, is allowed to (be loaded, identified and) run in the trusted execution environment.

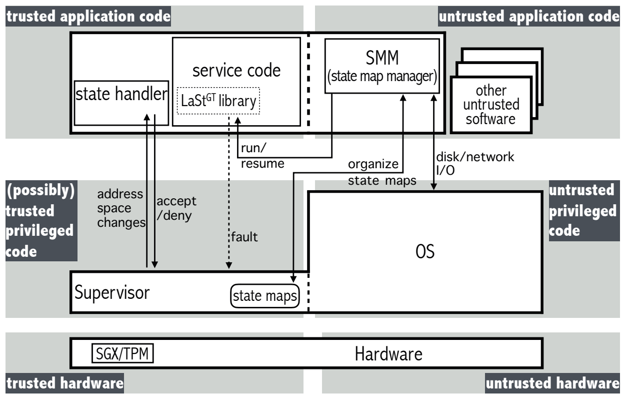


Fig. 3: LaSt^{GT}'s system architecture.

It is up to V to validate replies by verifying the attestation of the identities of the executed service code and the input data, which are assumed to be known to V, and of the output data which is assumed to be ultimately delivered to V. Also, V is able to determine whether the attestation has been issued by a trusted component. Under the implementations that we consider, this is equivalent either to knowing the public attestation key of the trusted component certified by the manufacturer, or to contacting a trusted Attestation Verification Service [31]. The content of the signed message (which includes the code identity) is part of LaSt^{GT}'s design (§5.4).

DoS attacks are not considered, since untrusted code can deny the use of the trusted component. The complexity to break cryptographic algorithms is assumed to exceed the adversary's capabilities. Side-channel attacks are not considered. Finally, the correctness of the executed trusted code is out of scope and up to the service developers.

5. Design of LaSt^{GT}

We introduce the architecture of LaSt^{GT} (§5.1) and describe how data is protected at the source (§5.2), processed by the service provider (§5.3), and verified by the client (§5.4). In §6 we detail implementations on two TC architectures.

5.1 Architecture

In the architecture (Fig. 3), we distinguish between three types of components, namely from the bottom up: hardware (CPU, memory, security chips, disk, TC hardware, etc.), privileged code (OS, drivers, etc.) and user-level (or application-level, non-privileged) code. We also distinguish between two types of code execution: trusted code (left) and untrusted code (right) run respectively inside and outside of the trusted environment. Depending on the TC component, the Supervisor's privileged code must be trusted (e.g., in XMHF-TrustVisor) or can be untrusted (e.g., in SGX), as discussed in §6. The two hardware/software stacks represent the trusted and the untrusted execution environments. A hardware-based (e.g., a TPM and Intel TXT, Intel SGX) isolation mechanism prevents untrusted code from tampering with its trusted counterpart.

User-level code includes both untrusted and trusted code. The trusted user-level code is transferred and identified at runtime within the trusted environment. The untrusted code

Component	Functionality	Trusted	TC arch.	Implem.
service code	self-contained general purpose service	✓	any	app-level code
state handler	check modifications to secure address space; data validation and protection	✓	any	app-level code
SMM	organization of state in memory; proposal of new memory maps	✗	any	app-level code
Supervisor	(un)map pages in trusted application address space; switch among components	✓	XMHF-TrustVisor	hypervisor
	proposal of modifications to secure address space	✗	SGX	OS-level driver

TABLE I: LASTGT's software components.

organizes data and memory for the trusted user-level code, which validates the data before using it. Table I lists key software modules in LASTGT, where they execute, and what implementation they apply to. We discuss how they work together to bring data securely into the trusted environment in §5.3.

5.2 *From User Data to* LASTGT*-compatible State*

The data produced by the data source has to be protected for client verification and structured for secure and efficient processing on LASTGT. How data is structured and protected impacts the efficiency of computing the metadata at the data source, the performance of verifying the data at the untrusted provider (e.g., how much data has to be loaded to verify data that is used) and the verification effort at the client.

This leads to the following requirements for protecting the data. First, it should be efficient and incremental, so the metadata can be computed as data is produced. Second, it should enable piece-wise data validation, so to handle only subsets of data in memory. Third, it should enable constant time verification by the client.

LASTGT provides these features by pre-processing the user data into a LASTGT *state hierarchy* of sub-components, consisting of blocks of user data at the leaves and structural and authentication metadata higher up in the tree (Fig. 5). Each component has a cryptographic identity that depends on its sub-components' identities, and ultimately on the user data. These identities form a cryptographic hash tree optimized for a large state, as described in §6.1.1. The structure is built using an incremental procedure and allows piece-wise data loading and validation through the concepts of data *chunks* and *blocks*. Also, it enables constant time verification at the client by checking the state identity, i.e., the identity of the root.

5.3 *Data Processing at the Untrusted Provider*

We describe how LASTGT manages the service code execution (§5.3.1), how data is read from disk (§5.3.2), loaded into the trusted environment (§5.3.3) and reclaimed (§5.3.4).

5.3.1 *Service Execution*

A LASTGT execution begins by registering the state root identity (provided by the data source) with the state handler in the trusted execution environment (§6.1.3). This is a one-time procedure that must be secure since the integrity of the entire state hierarchy, including the user state, depends on the correctness of the state root identity. It is not necessary to load the full state upfront, since the root is sufficient to validate any data loaded during execution. When the service terminates, the registered (root) identity is also included in the attestation so that a client can verify it.

The service code is then executed and it uses regular I/O calls to access user data, though without issuing any system call. I/O calls use the LASTGT library to get access to the user data. The library has a memory-mapped view of the state hierarchy, which it traverses beginning from the registered state root to access the data. As the data, including the metadata, is not available upfront in the isolated memory, its execution is interrupted by page faults, which are handled by the Supervisor, as described in the next section.

In-Memory Embedded Locators (IMELs). A naive loading would allocate (for example) 2^{40} virtual addresses upfront for a state of 1TB, and each page access would trigger a page fault. This is feasible on 64-bit architectures, but the SMM would have to ask the OS for many virtual address mappings, most of which may not be used. Also, the OS may be required to remap physical pages and maybe do some paging to disk. This would occur similarly for the Supervisor while managing pages in the trusted execution environment, thus adding overhead. In addition, platforms like TrustVisor use 32-bit addresses.

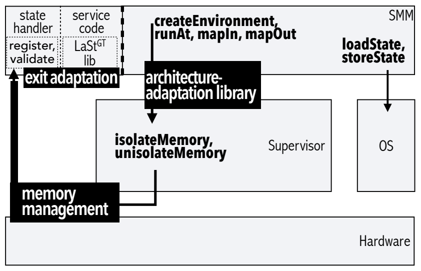
To deal with these overheads and constraints, LASTGT uses IMELs to reuse addresses and memory. IMELs are memory pages embedded in the state hierarchy at runtime between a parent and a child component (e.g., a master-chunk and a chunk). Specifically, a parent points to an IMEL that contains the address of its child, rather than pointing to its child directly. By not loading the IMEL in isolated memory when the parent component is first loaded, this makes the service code raise a page fault on a memory page that contains an address, rather than the child component. So IMELs just provide positions in memory. They can be filled at runtime and loaded in isolated memory together with the child components they reference. Similarly, they can be unloaded together with the component, so to reuse the allocated memory with other data.

5.3.2 *Loading state from disk into untrusted memory*

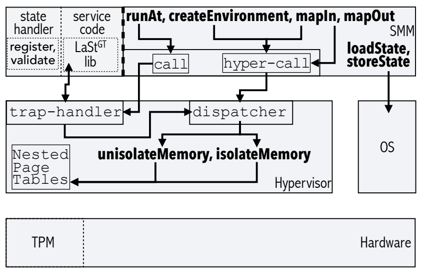
The Supervisor offloads to the untrusted SMM the handling of page faults related to data that is not yet in main memory. In particular, it does so by transferring control and providing the fault address to the SMM (§6.1.5). Offloading such task to user-level code moves the code out of the Supervisor's TCB, which is important in architectures where the Supervisor must be trusted. The SMM uses the fault address to figure out what state component (see hierarchy in Fig. 5) should be loaded from disk. It then loads the component from disk and places it into untrusted memory. For any component that is in memory (e.g., chunks, IMELs, etc.), the SMM maintains a map item in a map list (Fig. 7, §6.1.2). Such list is updated before the SMM returns control to the Supervisor. These maps will be used by the Supervisor for moving pages into the trusted environment, and by the state handler for validation.

5.3.3 *Authenticated lazy loading from untrusted memory*

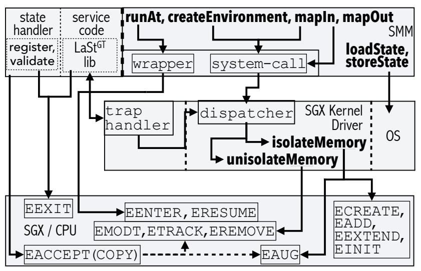
LASTGT optimizes loading data from untrusted memory into the isolated memory using authenticated lazy loading, i.e., pages or blocks are loaded on demand (§6.1.4). This is done



(a) LASt^{GT} generic implementation.



(b) Trustvisor-specific implementation.



(c) SGX-specific implementation.

Fig. 4: LASt^{GT} abstraction of non-common mechanisms (4a, black boxes). TC-architecture-specific implementations (4b and 4c).

either after a page fault on data already in untrusted memory or after data has been fetched from disk. In particular, the Supervisor handles page faults by simply using the memory maps to locate the pages and to map them. The state handler is then invoked to validate them.

Page and data validation are performed within the trusted environment by the state handler before the service code can access the data. This ensures that the library (and thus the service) can only access valid data—so there is no data validation performed within the service code. The procedure is performed at the user-level because the Supervisor may be untrusted depending on the TC-architecture.

5.3.4 Reclaiming memory

The untrusted SMM can reclaim state components from isolated memory by updating the map list (§6.1.5). The reclaim is validated and accepted (or denied) by the state handler and, only when it is accepted, the Supervisor is allowed to withdraw the reclaimed pages from the trusted execution environment.

5.4 Client Verification of a Remote Execution

The verification of a remote execution is equivalent to verifying a hardware-based execution attestation. It can include the identities of: 1) the executed code, 2) the input and 3) output data, 4) a client-provided nonce (§6.1.6). A successful verification validates the signature, using a manufacturer-certified public key (or an Attestation Verification Service [31]), and makes sure that the attested identities are the intended ones.

Verification of state updates. If the service code modifies the state, the state handler can update a separate runtime version of the state root. This represents the output state identity and can be included in the attestation and returned to the data source (or the client) so it can verify the modifications.

6. IMPLEMENTATION OF LAST^{GT}

Overview. Fig. 4 zooms into the architecture (Fig. 3) detailing the implementation of LASt^{GT}. In particular, Fig. 4a abstracts the details of the hypervisor-based implementation (Fig. 4b) and of the SGX-based implementation (Fig. 4c). This helps identifying common parts of LASt^{GT} whose code can be shared across different TC-architectures.

The primitives in bold are common across implementations. Inside the SMM, they allow the untrusted user-level code to set up the environment (**createEnvironment**) for the execution of the trusted user-level code, to run it (**runAt**), to map state components data and metadata in and out (**mapIn, mapOut**) of the the isolated address space—though the state handler validates them first—and to manage state components on disk (**loadState, storeState**) through the untrusted OS. At the privileged level inside the Supervisor, the primitives (**isolateMemory, unisolateMemory**) allow the Supervisor to provide memory pages to (or to withdraw them from) the isolated address space of the trusted application code according to the maps configured by the SMM. The attestation primitive that is used by the state handler to validate data and metadata is not shown to simplify the description and figures.

Several primitives need to interact with hardware and software that are specific to the used TC-architecture (abstracted by the black boxes in Fig. 4a). First, the architecture-adaptation library includes code to execute a hyper-call or a system-call handled by a dispatcher, or to execute an instruction wrapper, or a call that traps into a trap-handler. The memory management and the exit adaptation boxes have very specific functions. The former touches the state handler due to the EACCEPT(-COPY) SGX instructions (§6.3) that the state handler runs to accept changes to enclave pages. The latter instead hides the EEXIT instruction in SGX, or a simple return of a function in XMHF-TrustVisor, to terminate the execution. Finally, LASt^{GT} simply uses the OS and standard libraries for functions such as managing state on disk.

Inside the trusted application code, the LASt^{GT} library mainly navigates the state hierarchy, while the register (§6.1.3) and validate (§6.1.4) primitives hide the details of the memory maps and of the authenticated data structure embedded in the LASt^{GT}-compatible state. Also, in XMHF-TrustVisor, the trusted application code just directly calls the hypervisor for the attestation (not shown). In SGX, instead, it has to run dedicated instructions to terminate, to accept pages and to attest, making the code more dependent on the TC architecture.

To summarize, LASt^{GT} can deal with the architectural differences between XMHF-TrustVisor and SGX through small adaptations within a single design. Next, we describe the architecture-independent components of LASt^{GT} (§6.1), our implementation for XMHF-TrustVisor (§6.2), and a design for SGX (§6.3) in more detail.

6.1 TC-architecture-independent Details

6.1.1 Building the state

As discussed in §5.2, the user state and its meta data is organized in a state hierarchy (Fig. 5). The *state root* contains only one hash value. Both the service and client code rely on it to validate the authenticity of the data. A *directory* is

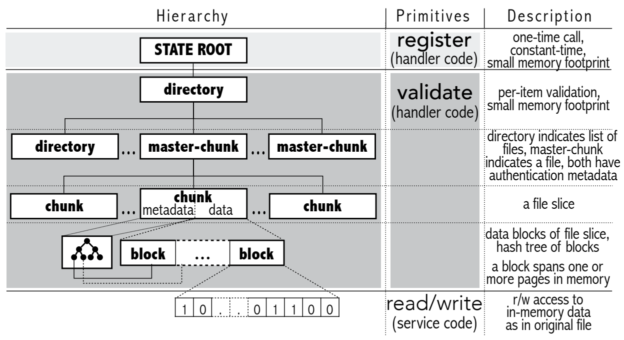

Hierarchy
Primitives
Description
STATE ROOT
register (handler code)
one-time call, constant-time, small memory footprint
directory
validate (handler code)
per-item validation, small memory footprint
directory
master-chunk
master-chunk
directory indicates list of files, master-chunk indicates a file, both have authentication metadata
chunk
chunk metadata data
chunk
a file slice
block
block
data blocks of file slice, hash tree of blocks
a block spans one or more pages in memory
1 0 . . 0 1 1 0 0
read/write (service code)
r/w access to in-memory data as in original file

Fig. 5: State hierarchy. A directory can contain several master chunks and (sub-)directories. The relevant primitives (register, validate) build a chain of trust between the service reads/writes and the state root verified by a client.

a set of (sub-)directories and master chunks. A *master chunk* maps one-to-one to a file in the filesystem. Each master chunk includes a set of chunks, each of which corresponds to a contiguous sequence of user data in the file. The user data in a chunk is further logically divided into a set of *blocks*. Each chunk also includes metadata, called a chunk descriptor. It contains a static hash tree built from the chunk's data. The leaves of the tree are computed by hashing contiguous bytes of the block in the chunk. The root of the tree represents the identity of the chunk. The identities of the chunks are hashed to form the identity of their parent master-chunk, and so on up the root, which is the identity of the entire state.

Only a configuration file and two parameters (chunk and block size) are required to build a LAST^{GT}-compatible state. This information is defined by the user. The configuration file is a list of files (each one producing a master chunk) and directories to be included in the state.

This design of the state hierarchy allows efficient management of data in memory. We can load fixed-size chunks from disk as needed, without dealing with the entire state data at once. Then we can load blocks from untrusted memory into the trusted execution environment, possibly batching the transfer of the pages spanned by a block. Also, as the authentication metadata is distributed across the state hierarchy, we can easily and locally validate data blocks in a chunk and update the hash tree when a block is modified. In fact, the hash trees of other chunks are not required, so they can remain on disk.

Distributing the authentication data is important for our target state sizes. A single file-wide (as in Minibox) or disk-wide (as in Haven) hash tree has several drawbacks in comparison. First, a single tree can take up to gigabytes (Fig. 6, top-right of shaded area). Second, this adds complexity to cache it in secure memory and in untrusted memory. Finally, one could opt to load a data block together with a short membership proof (linear in the height h of the tree). However, when using a single tree for 1TB of user data (2^{40} bytes) and small blocks (2^{10} bytes), the hash tree is tall $h = 31$, so the proof is large, i.e., $(h-1)\ nodes \times 32\ bytes/node = 960\ bytes$ or 93% of block size, and verifying it takes many hashes (i.e., $h-1$).

6.1.2 Maps for State Organization and Memory Management

LAST^{GT} uses memory maps for state and memory management. Fig. 7 shows some entries in an example map list that the SMM uses to store type, address and size (in pages) of the memory it allocates. The SMM uses different map types for metadata and data (15 types, including those for IMELs,

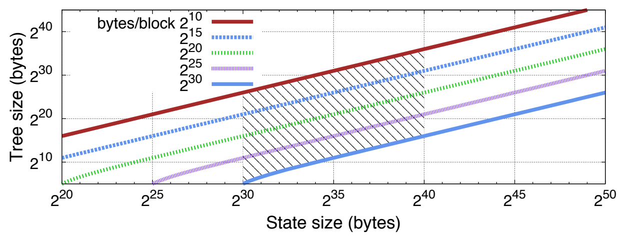


Fig. 6: Hash tree size (y) as a function of the state size (x) for different block sizes, with a 32 bytes hash (e.g., SHA256). Shaded area is our target.

debugging and performance measurements), including a special one to reclaim a map. The maps are also shared with the Supervisor which manages the physical pages (in SGX, or the memory access permissions in XMHF-TrustVisor), and with the state handler that validates changes to the address space.

Memory accesses to data that is not in the trusted environment trigger a page fault and the Supervisor is invoked. The Supervisor looks up the page fault address into the memory maps. If the address points to data mapped in memory, this is a *map hit* and the Supervisor performs lazy loading in the secure environment (§6.1.4). It the address instead points to non-mapped data, this is a *map miss* and the Supervisor triggers the procedure for loading state from disk or for shutting down the execution in the case of an illegal access (§6.1.5).

The state handler uses the maps to locate metadata for validation and to ensure that pointers to supposedly non-mapped components do not incorrectly dereference mapped components—they must produce a fault. Notice that the initial maps (if any) must be embedded in the trusted user-level code and so included in the code identity eventually verified by a client. So, initial and subsequent maps can be trusted. To avoid tampering by the SMM, who might maliciously swap map types (e.g., the state root map type for a IMEL type), the state handler maintains a secure copy of the map list.

map list		
type	address	pages (=length)
D	0x0b000000	2^{18} (=1GB)
H	0x4b000000	1 (=4KB)
C	0x4b001000	2^{15} (=128MB)

Fig. 7: Example of a map list. Items describe type (D=dynamic memory, not included in state hierarchy; H=in-memory embedded locator; C=chunk), location and size of a map. Other types (and entries) are available for: root, directories, master-chunks, chunk metadata, input and output buffers.

6.1.3 State Registration

State registration is the first code executed in the trusted environment. It allows the state handler to receive the state root map. The map contains a single hash value for authenticating the metadata and data in the state hierarchy. The state handler uses the register primitive to copy the root hash to a static variable representing the input state—once set, it cannot be overwritten nor reset (without beginning a new trusted execution).

6.1.4 Normal Execution and Lazy Loading

The service execution is triggered by the LAST^{GT} library that begins the execution in the trusted environment. The library initializes the state base pointer to the state root map set up at registration-time. From the state root it then walks the state hierarchy by following the pointers between parent and child components.

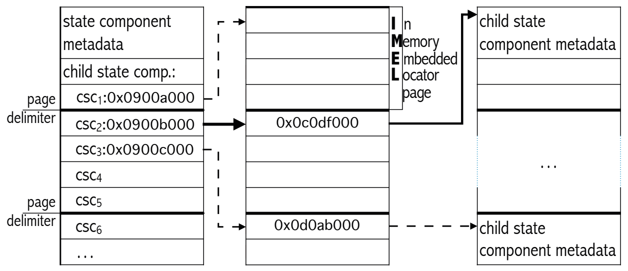


Fig. 8: IMEL pages contain positions of data in memory. They allow the LASt^{GT} library to access a child state component (e.g., a chunk, csc_2), and they help the untrusted code to locate such component and load it. Other IMELs (e.g., at csc_1, csc_3) that are not accessed are not loaded.

Correct access of a state component. LASt^{GT} has to ensure that the library: (i) produces a page fault when it accesses a non-validated state component; (ii) will find valid content in memory and hence (iii) can be resumed only in this case. The state handler ensures (i) by peeking into a state component being loaded to check that it has a pointer to an IMEL page that is not yet mapped in the isolated memory (e.g., csc_3 in Fig. 8). The handler ensures (ii) by cryptographically validating the state component using the validate primitive. For example, in the case of a chunk, the primitive checks the chunk's hash tree and whether its root matches that stored in the parent master-chunk. For an IMEL, instead, the state handler simply checks that it contains the address of valid state component being loaded jointly. Ensuring (iii) is TC-architecture-dependent so deferred to §6.2 and §6.3.

Loading metadata vs. Loading data. Except for the original user data in a chunk, the rest of the state hierarchy is considered metadata. Metadata (directories, master chunks, chunk metadata) and data have different types of maps. This allows the Supervisor to behave differently when the library produces a page fault while walking through the state components. If the fault address is map-hit by a data-typed map (only containing data), one data block is loaded. In this case, the state handler just performs cryptographic validation. If the address is instead map-hit by a metadata-typed map (only containing metadata), then the entire map is loaded. The rationale is that metadata is small (compared to data, see evaluation §7) and can best be validated immediately. This later allows validation of a data block in constant time—check if a block's hash matches the associated hash tree leaf.

6.1.5 *Loading Data From Disk and Reclaiming Maps.*

If the Supervisor cannot find a map that covers the page fault address, then that is a map miss. A map miss only occurs on IMEL pages (§5.3.1) unless bugs result in illegal map misses. The Supervisor transfers control to the SMM to handle the map miss. The SMM uses the fault address to locate the IMEL and needs some metadata in untrusted memory to locate the data on disk. For this reason the SMM maintains shadow copies of parent state components—treated later (§6.2, §6.3) due to architectural differences in the trusted address space configuration. So the child state component is loaded from disk into an arbitrary free memory range and the associated IMEL page is updated. Then the SMM creates map entries for both the IMEL page and the child component and informs the Supervisor of the new maps. The Supervisor retries handling the fault whose address should result now in a map hit.

When a map miss does not reference a IMEL page (e.g., the null 0×0 address), this is considered a software bug. The SMM thus triggers an execution shutdown (segmentation fault).

Reclaiming maps. The SMM reclaims a map $m = (type, address, pages)$ by inserting in the list another entry $m' = (\text{RECLAIM}, address, pages)$ with a special reclaim-type. Finally, the handler checks the reclaim and accepts to give the map back to the SMM.

Reclaiming maps in the presence of modified data. While there is a modified state component mapped in (e.g., a chunk), the state handler never accepts the reclaim of any map of that component's ancestors (e.g., a master chunk). The hash tree in fact may not be up to date with the modifications. If a reclaimed component has no child components in the secure environment, the state handler updates the hash tree (if necessary) and accepts the reclaim. At attestation-time, the state handler similarly updates the hash tree root, so clients can know the identity and verify the integrity of the output state.

6.1.6 *Attestation and Remote Verification*

LASt^{GT} combines in the attestation a client-provided nonce and the identities of the registered state, the output state (if any), the client request, the reply, the trusted application code. The state handler initially receives from the SMM specially-typed maps that contain the request, the reply and the nonce. It accepts them and, respectively, it hashes the request map to save the request identity, it saves the nonce, it zeroes the reply map that is later filled by the service code. After the service code execution terminates, when the state handler runs again to perform the attestation, the data in the reply map is also hashed to get the identity. The state handler then hashes the identities of the state, the request and the reply together with the nonce. The resulting hash can be either hash-chained to the trusted application code identity and attested (in XMHF-TrustVisor) or provided in input for the cryptographic report (in SGX) which will include the trusted application code identity.

Assuming that the client receives the reply (and recalling our model §4), the client knows all the attestation parameters and can therefore verify the attestation. In particular, the client establishes trust in the reply only if the identities that are combined in the attestation match the expected ones.

6.2 *Implementation in XMHF-TrustVisor*

6.2.1 *Background*

XMHF-TrustVisor [25], [10] is a hypervisor that provides efficient isolated execution and attestation of self-contained code (Fig. 4b). The trusted execution environment is created by registering the trusted application code in the hypervisor using a hypercall. Code memory pages are isolated from the untrusted OS using nested page tables (e.g., Intel EPT) to forbid access to the physical pages. Any access from untrusted code traps into the hypervisor. Only when the program counter points to the registered code's entry point, the hypervisor switches to secure mode to execute the registered code. The code executes until it terminates—it is never preempted and input data is provided upfront. Termination occurs when the code attempts to return to code outside the isolated region. Hence, it traps into the hypervisor which switches to non-sensitive mode to run (and makes the output available to) the untrusted application code.

At verification time, the client checks the correctness of two attestations: one from the hardware-TPM (a low-power chip) that vouches for the correct execution of the hypervisor; and one from the hypervisor, that vouches for the correct execution of the registered code. We refer to [25], [10] for further details.

6.2.2 *Implementation*

The architecture of XMHF-TrustVisor simplifies (w.r.t. SGX) LASt^{GT}'s implementation in two ways. First, the hypervisor can keep the service interrupted (after a page fault) until successful validation by the state handler. So the service accesses valid data when it is resumed. Second, the hypervisor can modify the trusted application code's page tables. So it can be trusted to load (and similarly unload) data at the correct position in the trusted execution environment.

The extended hypervisor orchestrates a LASt^{GT} execution based on the feedback received from the application-level. The hypervisor begins by running the state handler supplying the maps of the state root, the heap memory, the request, the reply and the nonce. Feedback from the state handler is a return value: 0, registration (or validation) unsuccessful; 1, registration (or validation) successful. If successful, the service code can be executed until termination, or until a page fault occurs by accessing a non-isolated state component as shown in Fig. 9 (left-side). On termination, the hypervisor asks the state handler to protect the integrity of modified pages in the state hierarchy up to the root and attest the result.

On page fault, the hypervisor checks the maps. In the map-hit case, it provides the data and the page fault address to the state handler for validation and, if successful, it resumes the service code. In the map-miss case, the hypervisor provides the page fault address to the SMM. The SMM uses it to locate the metadata shadow copy of the (isolated) state component, which is used to load the missing (child) state component in memory. The SMM then returns the new maps to the hypervisor.

The hypervisor switches control between the service code, the state handler and the SMM by alternating the execution of the VM with the untrusted OS and of the VM with the trusted application code. Namely, it updates the instruction pointer to one of the entry points (service code, state handler, SMM) or to the faulted service code instruction. In the case of an entry point, the hypervisor pushes an instruction pointer on the stack so the code traps back into the hypervisor when it returns.

The hypervisor isolates a map by ensuring that: the map pages have been (lazily §6.1.4) inserted in the page tables of the trusted application code; the nested page tables are configured to grant trusted application code access to the associated physical pages, while denying the SMM and OS access to them. Un-isolating a map works in the opposite way.

6.3 *On the feasibility of* LASt^{GT} *using Intel SGX*

6.3.1 *Background*

Intel SGX [29] is an instruction set extension available on the Intel Skylake microarchitecture that enables trusted remote code execution and verification. It uses an area in main memory, encrypted by the CPU, where secured code and data reside. It does not require external hardware for attestation. So the CPU package delimits the physical security boundary.

A secured area called Enclave can be created to set up an execution environment for trusted application code

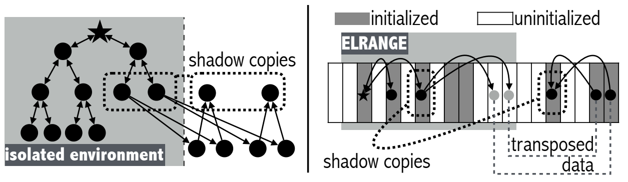


Fig. 9: In the trusted environment, parent state components can reference child ones in untrusted main memory either *directly* using their addresses (in XMHF-TrustVisor, left-side), or *indirectly* through their transposed addresses in uninitialized pages within the enclave's secure range, where they will be loaded (in SGX, right-side). Referenced components, that are not yet in untrusted memory, are located and loaded using metadata shadow copies.

(Fig. 4c). At enclave-creation time, a range of logical addresses (ELRANGE) defines the enclave secure region. The enclave can access memory inside and outside the secure region, but it cannot execute code outside it. The enclave can include one or more entry points where the execution starts. If the enclave is interrupted, the sensitive processor state is saved within the secure region and restored when the enclave is resumed. Adding and removing enclave memory pages at runtime requires cooperation between untrusted privileged code (the Supervisor, i.e., an OS driver) and the enclave's trusted application code. We refer to [29], [31] for further details.

6.3.2 *Implementation*

In SGX, the memory management and the secure control flow management are slightly more complex (compared with XMHF-TrustVisor) for three reasons. First, addresses and content of enclave memory pages to be added or removed at runtime must be checked and accepted by the enclave at the application-level, without help from trusted privileged code. Second, enclave code can access untrusted memory outside ELRANGE. While this is useful to load (and then validate) data from untrusted memory, it opens the risk of using incorrect data inadvertently. Third, untrusted code can run/resume enclave code at any time. Hence, concurrency issues may arise within the enclave, particularly when resolving a page fault or performing an attestation. LASt^{GT} can deal with these three challenges as follows.

1. Since untrusted code cannot start the enclave execution at an arbitrary instruction, we build the enclave with separate entry points for the service code and the state handler. This allows running the state handler while the service code is interrupted.

A mechanism is required to ensure that the service code can only be resumed, and not re-executed, after an interruption. As untrusted code can behave arbitrarily, it may restart the enclave at the service code entry point. We thus build the enclave with a single area to save the processor state on interruption (e.g., due to a page fault). As an interruption consumes one such area and the CPU requires one to be available to start the enclave, this prevents multiple executions at the service code entry point, before the service code terminates.

When the state handler is executed, the handler has to validate the position at which memory pages are supplied, and the content these pages should have. The position is the address of the page where the fault occurred during the service code execution. Such address is found using the CR2 control register where the CPU stores the fault address. However, reading CR2 requires privileges, so the application-level enclave code cannot do it. Also, the state handler cannot trust the (untrusted) SGX driver to supply it correctly, although it expects the driver to supply the memory pages. Fortunately, the CPU includes the value of CR2 in the enclave's secure region when the execution is interrupted. So the state handler has access to a trusted address and can check that a map (in the list) covers it.

2. Validating the content of the memory pages is tricky because they are not available, otherwise the service code could be resumed. The problem is to enable the resumption of the service code only after the pages are available with the right content. This is solved by validating the content elsewhere and leveraging SGX to fill the pages appropriately as follows. We include in the enclave (at creation-time) a buffer large enough (e.g., 4MB) to contain a state component metadata or a data block. We program the state handler to copy the data from non-enclave memory to the internal buffer and to validate it. Besides validating the integrity of the data, the state handler also checks that any address referencing an IMEL or a child state component falls within the secure region. This prevents the service code from accessing untrusted memory. We discuss later where the data is placed in untrusted memory.

Assuming the data is valid, the next step is placing it correctly so that the service code can be resumed and access it. We resort to the EACCEPTCOPY SGX instruction to do it. The instruction allows to copy an available enclave page into an uninitialized enclave page and to initialize it. So the state handler executes the instruction to copy a page of the buffer containing the data into a still unavailable page that the interrupted service code cannot access. After this step, the service code can make progress since the page is available and contains valid data. The procedure can be easily extended to batch the validation and acceptance of a set of pages. Similarly, enclave memory is reclaimed with a two-phase protocol as in [29, 3.5.9]. We mention that SGX provides another instruction for accepting memory pages, i.e., EACCEPT. This is useful for dynamic memory allocations (e.g., our dynamic memory map), that are supposed not to contain sensitive data initially—in fact they are zeroed. However, using it in the previous step would not be secure, because it initializes the memory pages (which become accessible) before they are filled with valid data.

We explain how our maps are used as the data is to be transferred from untrusted memory to the trusted buffer and then copied elsewhere in the enclave's memory. In XMHF-TrustVisor, since we can (un)isolate as single memory page, one map per component is sufficient. In SGX, instead, the SMM cannot load data directly into the enclave region, and the state handler should have some means to locate data in untrusted memory. Our solution is mapping each state component into two map lists M_1, M_2, thereby having two maps. M_1 follows our original description: it expresses where memory and data are or should be placed within the enclave region—so the addresses belong to the enclave region. M_2 is logically derived from M_1 by transposing the address of each map into an address in untrusted memory. So, the SMM uses M_2 to arrange in untrusted memory the state components loaded from disk, and M_1 to arrange them in trusted memory. Instead, the state handler uses M_2 to (un)load to/from untrusted memory, and uses a private copy of M_1 to ensure the correct position of the maps in the enclave's secure region.

3. Finally, as the attestation is performed by the state handler, the handler has to make sure that the service has indeed terminated and will not modify the state (e.g., if re-executed) during the attestation. We address this concurrency problem by synchronizing the service code and the state handler using shared variables transparently inside our linked library. Notice that in a multi-core environment, such shared variables can be managed in transactional regions with Intel TSX (Transactional Synchronization Extensions), which is available on the Skylake microarchitecture and compatible with SGX [29, 6.14].

	VC3	Haven	LaSt$^{\text{GT}}$*		
			hypervisor	library	SQLite
SLoC $\times$ 10^3	9.2 [13]	23.1+ $\mathcal{O}(10^3)$# [14]	15.1 [25] +1.9‡	7.7	92.6
			$\underbrace{\qquad\qquad}_{24.7}$		
				$\underbrace{\qquad\qquad}_{100.3}$	

** based on XMHF-TrustVisor* ‡ LaSt$^{\text{GT}}$ *core code and headers*
# LibOS contains millions of lines of code

TABLE II: TCB size breakdown (in source lines of code) and comparison. SQLite is included as an example real-world application ported to LaSt$^{\text{GT}}$.

7. Evaluation

This section analyzes LaSt$^{\text{GT}}$'s Trustvisor-based implementation, quantifying its TCB, comparing it with the original TrustVisor, and using it to run various application.

Experimental setting. We use a Dell PowerEdge R420 Server equipped with: a 2.2GHz Intel Xeon E5-2407 CPU; 16GB of DDR3 memory; a TPM v1.2; a primary 300GB, 15Krpm hard-disk; a secondary 2TB, 7.2Krpm hard-disk. The server runs Ubuntu 12.04 32-bit with a Linux kernel 3.2.0-27. The resources are fully dedicated to our experiments. LaSt$^{\text{GT}}$ uses the secondary disk to ensure uniform experimental conditions.

Data is organized in chunks of 128MB and blocks of 256KB. Our micro-benchmarks justify these values for sequential workloads (typical of data analytics) and suggest optimizations for random workloads (§7.5). We assume the worst-case scenario that requires LaSt$^{\text{GT}}$ to maintain a low memory footprint. Hence, we set up the SMM to reclaim old chunk maps when the service code tries to access a new one.

7.1 TCB Size

We quantify LaSt$^{\text{GT}}$'s TCB size using the lines of source code metric (SLoC), as calculated by the SLOCCount tool [32] and compare it with previous work (Tab. II). At the hypervisor level, the TCB size increases by 12%. The LaSt$^{\text{GT}}$ library adds an additional 7.7 SLoC of user-level code, which is relatively small. For example, it increases the size of the SQLite service code by only 8.3%.

The table also compares the TCBs of LaSt$^{\text{GT}}$, VC3 and Haven. Haven's TCB is notably large due to the library OS. LaSt$^{\text{GT}}$'s TCB is larger than that of VC3. However, LaSt$^{\text{GT}}$ is not application-specific and can run generic self-contained applications.

We expect LaSt$^{\text{GT}}$'s SGX implementation to have a smaller TCB than the current one. As SGX keeps privileged code out of the enclave, the hypervisor functionality—to manage the VMs, protect the isolated memory from the untrusted OS, and schedule the execution of trusted and untrusted

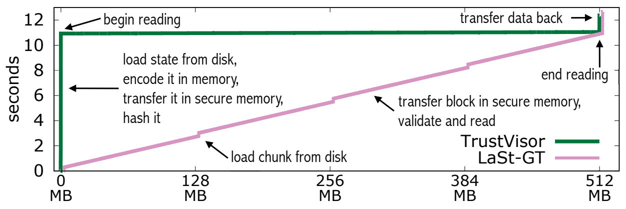


Fig. 10: Comparison between LAST$^{\mathrm{GT}}$ and XMHF-TrustVisor.

applications—will be moved out of the TCB and implemented in untrusted code, while retaining similar security guarantees.

7.2 *Comparing* LAST$^{\mathrm{GT}}$ *and TrustVisor*

As a baseline experiment, we compare LAST$^{\mathrm{GT}}$ with the original TrustVisor implementation. As displayed in Fig. 10, LAST$^{\mathrm{GT}}$ can be much faster than TrustVisor depending on how the applications access memory. The experiment uses a 512MB dataset that fits in memory; TrustVisor cannot scale to larger sizes. The data is read sequentially, allowing a comparison of the associated overheads according to the memory accesses. TrustVisor always exhibits a large startup time (dependent on the data size) as it reads everything upfront into memory. In contrast, LAST$^{\mathrm{GT}}$ exhibits a performance that is related to the parts of memory that are actually read/written thanks to its ability to do incremental data loading and validation. So, in an execution that only touches half of the dataset, TrustVisor would roughly end up taking twice as much time as LAST$^{\mathrm{GT}}$.

7.3 *Micro-benchmarks*

LAST$^{\mathrm{GT}}$ incurs overhead when it needs access to additional data through page faults. The primary sources of overhead include switching control between software components, (un)loading maps in isolated memory, and disk accesses by the SMM. Since the latter is the same for trusted and untrusted executions, we focus on the quantifying the overhead of the first two. We also quantify the overhead of preparing the state hierarchy by the content source. We present results that are the average of 1000 experiments with a 95% confidence interval.

Context-Switching. We measure the overhead to switch between the Supervisor, the state handler and the SMM (§5.3.2).

The Supervisor invokes the SMM when data is needed from disk. This involves switching from the trusted to the untrusted environment, and back (see table below). This time is mostly used to transfer the memory map list between the untrusted and the trusted execution environments. This requires inspecting the nested page tables of the virtual machine to check permissions for the data transfer, and then modifying both the nested page tables with the new permissions and the sensitive environment page tables to add (or remove) the pages from the isolated virtual address space (§6.2).

A second source of overhead is that associated with resuming the state handler (after some data has been brought into the secure environment) and the SMM (for disk access). These resumption times (table below) include the overheads of the XMHF-TrustVisor software stack[1], of virtualization to resume the trusted VM or the untrusted VM, and of the VM interruption to return back into the hypervisor. The slightly higher and more variable overhead for the SMM can be attributed to scheduling delays (time slicing, preemption) caused by the OS. This does not occur in the isolated (and dedicated) execution environment where the state handler runs.

[1]TrustVisor is an application running within XMHF [25].

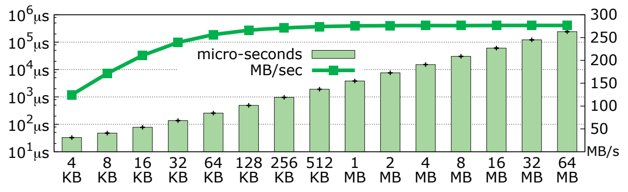


Fig. 11: Log-scale average time (bars, left y-axis) and speed (line, right y-axis) for mapping data (x-axis) inside/outside the isolated trusted environment. The right y-axis shows the attained speed.

❶ trusted-untrusted env. switching	❷ state handler resumption	❸ SMM resumption
$191.68\mu s \pm 0.12$	$36.11\mu s \pm 0.08$	$39.93\mu s \pm 0.5$

This means that the overheads for invoking the SMM and the state handle from the Supervisor after a page fault are ❶+❸ and ❶+❷ respectively, in addition to the cost of the processing (e.g., accessing disk or validating data).

I/O data mapping overhead. The overhead for transferring memory maps between the execution environments (§6.2.2) is shown in Fig. 11. Larger maps can be transferred at higher speed, suggesting that if the application has to process all data in a map, it is advantageous to transfer more data per fault to reach high-speed (e.g., using a 256KB block size as we did).

From user data to LAST$^{\mathrm{GT}}$-compatible state. Fig. 12 shows the cost of building the state hierarchy (§5.2) for user data sizes from 1MB to 2GB. This is sufficient since disk bottlenecks (for reading data and writing back metadata) show up at 64MB. For larger states, the throughput stabilizes at $\approx$ 60MB/s.

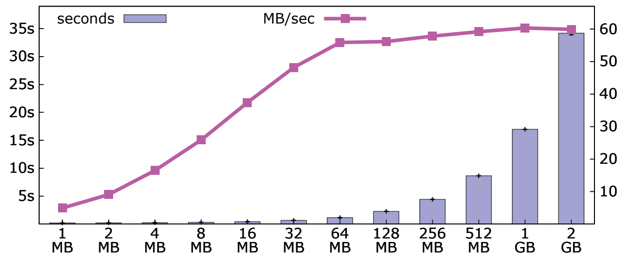


Fig. 12: Time (bars, left y-axis) and speed (line, right y-axis) to build the LAST$^{\mathrm{GT}}$-compatible state for different state sizes.

Building the hash tree also incurs a high cryptographic cost. It needs to hash $2^9 \times 256$KB-sized blocks and $2^{10} - 2$ tree nodes (i.e., all nodes except the root). The procedure is optimized to take linear time in the size of the hash tree. We chose SHA-256 as the hash function and carefully optimized it.

Different applications can leverage the incremental construction and parallelize the operations, particularly considering that data chunks can be built separately.

7.4 *End-to-end Application Performance*

We ran experiments with three applications with different data access patterns. They include: a simple application that sequentially walks 1TB of data, checking the first page of each data chunk; a nucleobase search application [26, 3.2] that accesses data sequentially, requiring all blocks of all chunks; and SQLite [28], which accesses random blocks of chunks.

Tera-scale data processing. We use a synthetic state of 1TB. The LAST^{GT}-compatible state contains (in addition to the state root and one directory) one master chunk of 2.5MB that carries a list of 8192 chunks. This master chunk size is much larger than the 256KB hash list contained in the master chunk due to additional metadata (e.g., size and name) relative to the chunks that we maintain. Each chunk has 33KB of metadata (32KB due to the static hash tree, so 97%) and 128MB of data.

The execution environment is initially composed by a heap map of 262K memory pages (1GB) loaded lazily—though just a few are used in this application—and a state root map that fits in 1 page. Directory and IMELs are loaded as they are accessed, and they also fit in 1 page each. The master chunk instead fits in 641 pages. As chunks are accessed, two additional maps are included in the environment: the chunk metadata that fits in 9 pages and is loaded; and the chunk data that fits in 32768 pages and is lazily loaded. Only about 15 maps are present at a time due to the state hierarchy and the environment constraints.

Figure 13 provides the progress of the application. It takes roughly 13 hours to process the terabyte of data, rather steadily at 23MB/s—the zoom in shows a slight variability.

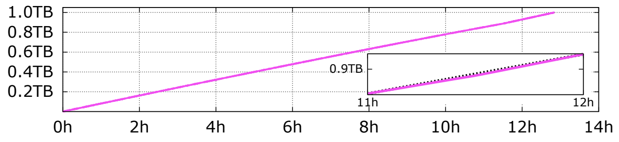


Fig. 13: Progress in hours to process 1TB of data. In the zoom-in, the black straight dashed line highlights a negligible variability in processing speed.

We emphasize that the experiment uses a dataset between 1 and 2 orders of magnitude larger than previous work on secure data analytics [13], [17]. Also, it is unprecedented on the XMHF-TrustVisor software stack [10], [25], [15]—designed for small applications—and on a bare-metal hypervisor.

Nucleobase search. This application [26, 3.2] searches for a nucleobase sequence among the billion-scale reads (i.e., fragments obtained from DNA sequencing machines) present in the FASTQ format[2] of the human genome in [40]. The application is relevant to investigate protein-coding mRNA sequence or to assemble sequences to reconstruct a contiguous interval of the genome [27].

Figure 14 show the outcome of our experiment on a human genome of roughly 0.3TB. The experiment produces about 1.15 million page faults that are handled directly by the hypervisor, and about 2.24 thousand are forwarded to the SMM for grabbing data from disk. The Nucleobase search is slower than the first application because it uses all the data and also involves actual processing.

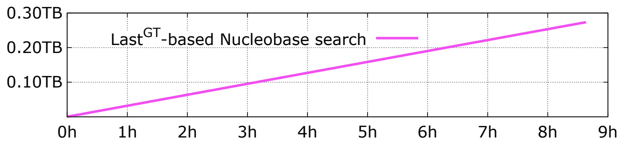


Fig. 14: Progress in hours to process a 0.3TB large human genome.

Database engine. The final application is the full SQLite (v.3.8.7.2) [28]—a real-world database engine with a non-trivial code base of 92.6K lines of code—that was executed on LAST^{GT} without modifying its source code. We compiled it with a virtual file system module that uses the LAST^{GT}'s library to access the state, and with an abstraction layer that uses LAST^{GT}'s functionality for memory management and I/O. The benchmark measures the time to query key-value stores of different sizes to get the value associated to a specific key.

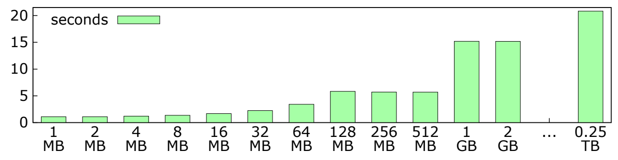


Fig. 15: Time (y) to query a SQLite-based key-value store of different sizes (x). The state is built using 128MB chunk and 256KB block sizes.

The results are shown in Fig. 15. The query time grows slowly up to $x = 128\text{MB}$ due to the larger data that is loaded in untrusted memory and to the larger hash tree to be validated in the trusted execution environment. At $x = 128\text{MB}, 256\text{MB}, 512\text{MB}$ the query time stabilizes. This is due to the data access pattern of SQLite which only involves the first chunk. At $x = 1\text{GB}, 2\text{GB}$, however, SQLite also accesses the 7th chunk before going back to the first one. This forces LAST^{GT} to load and validate the metadata of several chunks and to maintain their data in untrusted memory in the case it is accessed. This happens similarly with larger databases. For instance, at $x = 0.25\text{TB}$, SQLite requires to access one more chunk (the 420th) in addition to those listed before. Hence, the overhead scales linearly with the number of accessed chunks.

A glimpse of chunk & block size optimization. In order to show the benefits of optimizing LAST^{GT} for specific application requirements, we repeated the previous SQLite-based experiments by first protecting the database using a smaller chunk size (1MB) and block size (4KB). The results in Fig. 16 show about an order of magnitude better performance. This is chiefly due to the smaller chunks read from disk and smaller data blocks transferred in secure memory whenever the engine performs a random memory access. The performance with a terabyte-scale state is in the order of a few seconds. This is due to a large master-chunk that contains metadata for many (about 280K) small chunks, in contrast with the few (about 2K) large chunks in the previous experiment.

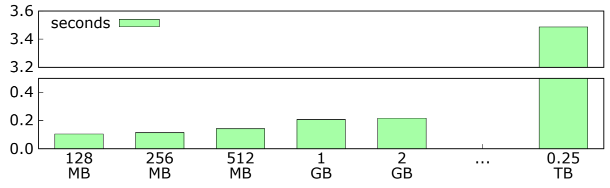


Fig. 16: Time (y) to query a SQLite-based key-value store of different sizes (x). The state is built using 1MB chunk and 4KB block sizes.

7.5 *Discussion*

The performance of LAST^{GT} is influenced by different factors and can be optimized by leveraging from better performing trusted components. First, the chunk and block sizes that we fixed for all experiments can instead be tuned for specific applications. This optimization problem is left for future work. However, intuitively, smaller chunk and block sizes reduce unnecessary data validation and data loading. Second, when

[2]Common text format used for storing sequences and quality score.

a memory map is created, LASt^{GT} optimizes for lazy loading in isolated memory, but XMHF-TrustVisor still requires the data to be present in untrusted main memory. This means writing data in memory bypassing critical optimizations such as lazy loading from disk. In this case, we believe that the SGX implementation can be beneficial to take advantage of the highly optimized kernel software stack, in addition to avoiding expensive virtualization operations such as VMEXITs and maintaining nested page tables.

8. Conclusions

We have described the design, implementation and evaluation of LASt^{GT}, showing that it is possible to build a secure system using a generic trusted component and supporting generic large-scale data applications. Our experiments with applications such as databases and genome analytics show that generic large-scale applications can run on systems with a small TCB. Overhead is significant, but we also show that the overhead in XMHF-TrustVisor is mostly due to expensive data I/O and context switches, which we expect can be heavily reduced on Intel SGX.

Acknowledgments

This work was partially supported by the EC through project H2020-643964 (SUPERCLOUD), by national funds of Fundação para a Ciência e a Tecnologia (FCT) through project UID/CEC/00408/2013 (LaSIGE). We thank Vinicius Cogo for interesting discussions and support with the cluster, and André Nogueira for fruitful discussions on kernel-level development.

References

[1] W. R. Claycomb and A. Nicoll, "Insider Threats to Cloud Computing: Directions for New Research Challenges," in *Proc. of the 36th Computer Software and Applications Conf. (COMPSAC)*, 2012.

[2] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds," in *Proc. of the 16th Conf. on Computer and Communications Security (CCS)*, 2009.

[3] S. Checkoway and H. Shacham, "Iago attacks," in *Proc. of the 18th Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, vol. 41, no. 1, 2013.

[4] Trusted Computing Group, "TPM Specs v2.0 rev. 00.99," 2013.

[5] A. F. Simpao, L. M. Ahumada, J. A. Gálvez, and M. A. Rehman, "A Review of Analytics and Clinical Informatics in Health Care," *Journal of Medical Systems*, vol. 38, no. 4, 2014.

[6] K. Srinivas, B. Rani, and A. Govrdhan, "Applications of Data Mining Techniques in Healthcare and Prediction of Heart Attacks," *Int. Journal on Computer Science and Engineering (IJCSE)*, vol. 02, no. 2, 2010.

[7] D. H. Chau, C. Nachenberg, J. Wilhelm, A. Wright, and C. Faloutsos, "Polonium: Tera-Scale Graph Mining and Inference for Malware Detection," in *Proc. of the SIAM Conf. on Data Mining (SDM)*, 2011.

[8] K. Ren, C. Wang, and Q. Wang, "Security Challenges for the Public Cloud," *IEEE Internet Computing*, vol. 16, no. 1, 2012.

[9] P. Muir and Et-al., "The real cost of sequencing: scaling computation to keep pace with data generation," *Genome Biology*, vol. 17.1, 2016.

[10] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig, "TrustVisor: Efficient TCB Reduction and Attestation." in *Proc. of the IEEE Symp. on Security and Privacy (S&P)*, 2010.

[11] B. Vavala, N. Neves, and P. Steenkiste, "Secure Identification of Actively Executed Code on a Generic Trusted Component," in *Proc. of the IEEE Conf. on Dependable Systems and Networks (DSN)*, 2016.

[12] S. Bajaj and R. Sion, "TrustedDB: A Trusted Hardware-Based Database with Privacy and Data Confidentiality," *IEEE Transactions on Knowledge and Data Engineering*, vol. 26, no. 3, 2014.

[13] F. Schuster and M. Costa, "VC3: Trustworthy data analytics in the cloud," in *Proc. of the Symp. on Security and Privacy (S&P)*, 2015.

[14] A. Baumann, M. Peinado, and G. Hunt, "Shielding applications from an untrusted cloud with Haven," in *Proc. of the 11th USENIX Conf. on Operating Systems Design and Implementation (OSDI)*, 2014.

[15] Y. Li, J. McCune, J. Newsome, A. Perrig, B. Baker, and W. Drewry, "MiniBox: a two-way sandbox for x86 native code," in *Proc. of the USENIX Annual Technical Conf. (ATC)*, 2014.

[16] A. S. Tanenbaum, "Lessons learned from 30 years of MINIX," *Communications of the ACM (CACM)*, vol. 59, no. 3, 2016.

[17] T. T. A. Dinh, P. Saxena, E.-C. Chang, B. C. Ooi, and C. Zhang, "M 2 R: enabling stronger privacy in mapreduce computation," in *Proc. of the 24th USENIX Security Symp. (SEC)*, 2015.

[18] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, M. L. Stillwell, D. Goltzsche, D. Eyers, P. Pietzuch, and C. Fetzer, "SCONE: Secure Linux Containers with Intel SGX," in *Proc. of the 12th USENIX Conf. on Operating Systems Design and Implementation (OSDI)*, 2016.

[19] S. Shinde, D. L. Tien, S. Tople, and P. Saxena, "PANOPLY: Low-TCB Linux Applications with SGX Enclaves," in *Proc. of the Annual Network and Distributed System Security Symp. (NDSS)*, 2017.

[20] T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel, "Ryoan : A Distributed Sandbox for Untrusted Computation on Secret Data," in *Proc. of the 12th USENIX Conf. on Operating Systems Design and Implementation (OSDI)*, 2016.

[21] W. Wei, J. Du, T. Yu, and X. Gu, "SecureMR: A Service Integrity Assurance Framework for MapReduce," in *Proc. of the Computer Security Applications Conf. (ACSAC)*, 2009.

[22] H. Ulusoy, M. Kantarcioglu, and E. Pattuk, "TrustMR: Computation integrity assurance system for MapReduce," in *Proc. of the IEEE Conf. on Big Data (Big Data)*, 2015.

[23] M. Correia, P. Costa, M. Pasin, A. Bessani, F. Ramos, and P. Verissimo, "On the Feasibility of Byzantine Fault-Tolerant MapReduce in Clouds-of-Clouds," in *Proc. of the 31st IEEE Int. Symp. on Reliable Distributed Systems (SRDS)*, 2012.

[24] B. Vavala, N. Neves, and P. Steenkiste, "Securing Passive Replication Through Verification," in *Proc. of the 34st IEEE Symp. on Reliable Distributed Systems (SRDS)*, 2015.

[25] A. Vasudevan, S. Chaki, L. Jia, J. McCune, J. Newsome, and A. Datta, "Design, Implementation and Verification of an eXtensible and Modular Hypervisor Framework," in *Proc. of the IEEE Symp. on Security and Privacy (S&P)*, 2013.

[26] B. Haubold and T. Wiehe, *Biological Sequences and the Exact String Matching Problem*. Birkhäuser Verlag, Basel (Switzerland), 2006.

[27] J. C. Venter and Et-al., "The Sequence of the Human Genome," *Science*, vol. 291, no. 5507, 2001.

[28] SQLite. www.sqlite.org

[29] Intel. Intel Software Guard Extensions. https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf

[30] A. Baumann, M. Peinado, G. Hunt, K. Zmudzinski, C. V. Rozas, M. Hoekstra. "Secure execution of unmodified applications on an untrusted host". Poster/Work-in-Progress. SOSP, 2013. http://research.microsoft.com/pubs/204758/sosp13-abstract.pdf

[31] Intel Software Guard Extensions: EPID Provisioning and Attestation Services. https://software.intel.com/sites/default/files/managed/ac/40/2016%20WW10%20sgx%20provisioning%20and%20attesatation%20final.pdf

[32] David A. Wheeler. SLOCCount. http://www.dwheeler.com/sloccount/sloccount.html

[33] Cloud Security Alliance. The Treacherous 12 – Cloud Computing Top Threats in 2016. https://downloads.cloudsecurityalliance.org/assets/research/top-threats/Treacherous-12_Cloud-Computing_Top-Threats.pdf

[34] Common Vulnerabilities and Exposures. CVE-2016-3841. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-3841

[35] Kernel Statistics. http://linuxcounter.net/statistics/kernel

[36] AMD. Secure Memory Encryption. http://developer.amd.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf

[37] AMD. Secure Encrypted Virtualization. http://support.amd.com/TechDocs/55766_SEV-KM%20API_Spec.pdf#search=Secure%2520Encrypted%2520Virtualization

[38] Amazon. Amazon Linux AMI Security Advisory: ALAS-2016-653. https://alas.aws.amazon.com/ALAS-2016-653.html

[39] Rackspace. QEMU "VENOM" Vulnerability (CVE-2015-3456). https://community.rackspace.com/general/f/53/t/5187

[40] DNAnexus Sequence Read Archive. Homo Sapiens SRR622458-NA12891. http://sra.dnanexus.com/runs/SRR622458

[41] C. Tsai, D. Porter. "Graphene / Graphene-SGX Library OS - a library OS for Linux multi-process applications, with Intel SGX support". https://github.com/oscarlab/graphene