Ryoan: A Distributed Sandbox for Untrusted Computation on Secret Data

Tyler Hunt, Zhiting Zhu, Yuanzhong Xu, Simon Peter, Emmett Witchel
The University of Texas at Austin

Abstract

Users of modern data-processing services such as tax preparation or genomic screening are forced to trust them with data that the users wish to keep secret. Ryoan protects secret data while it is processed by services that the data owner does not trust. Accomplishing this goal in a distributed setting is difficult because the user has no control over the service providers or the computational platform. Confining code to prevent it from leaking secrets is notoriously difficult, but Ryoan benefits from new hardware and a request-oriented data model.

Ryoan provides a distributed sandbox, leveraging hardware enclaves (e.g., Intel's software guard extensions (SGX) [15]) to protect sandbox instances from potentially malicious computing platforms. The protected sandbox instances confine untrusted data-processing modules to prevent leakage of the user's input data. Ryoan is designed for a request-oriented data model, where confined modules only process input once and do not persist state about the input. We present the design and prototype implementation of Ryoan and evaluate it on a series of challenging problems including email filtering, heath analysis, image processing and machine translation.

1. Introduction

Data-processing services are widely available on the Internet. Individual users can conveniently access them for tasks including image editing (Pixlr), tax preparation (TurboTax), data analytics (SAS OnDemand) and even personal health analysis (23andMe). However, user inputs to such services are often sensitive, such as tax documents and health data, which creates a dilemma for the user. In order to leverage the convenience and expertise of these services, she has to disclose sensitive data to them, potentially allowing them to disclose the data to further parties. If she wants to keep her data secret, she either has to give up using the services or hope that they can be trusted—that their service software will not leak data (intentionally or unintentionally), and that their administrators will not read the data while it resides on the server machines.

Companies providing data-processing services for users often wish to outsource part of the computation to third-party cloud services, a practice called "software as a service (SaaS)." For example, 23andMe may choose to use a general-purpose machine learning service hosted by Amazon. SaaS encourages the decomposition of problems into specialized pieces that can be assembled on behalf of a user, e.g., combining the health expertise of 23andMe with the machine learning expertise and robust cloud infrastructure of Amazon. However, 23andMe now finds itself a user of Amazon's machine learning service and faces its own dilemma—it must disclose proprietary correlations between health data and various diseases in order to use Amazon's machine learning service. In these scenarios, the owner of secret data has no control over the data-processing service.

We propose Ryoan[1], a distributed sandbox that allows users to keep their data secret in data-processing services, without trusting the software stack, developers, or administrators of these services. First, it provides a sandbox to confine individual data-processing modules and prevent them from leaking data; second, it uses trusted hardware to allow a remote user to verify the integrity of individual sandbox instances and protect their execution; third, the sandbox can be configured to allow confined code modules to communicate in controlled ways, enabling flexible delegation among mutually distrustful parties. Ryoan gives a user confidence that a service has protected her secrets.

A key enabling technology for Ryoan is hardware enclave-protected execution (e.g., Intel's software guard extensions (SGX) [15]), a new hardware primitive that uses trusted hardware to protect a user-level computation from potentially malicious privileged software. The processor hardware keeps unencrypted data on chip, but encrypts data when it moves into RAM. The hypervisor and operating system retain their ability to manage memory (e.g., move memory pages onto secondary storage), but privileged software sees only an encrypted version of the data that is protected from tampering by a cryptographic hash. Haven [21] and SCONE [19] are examples of systems that use enclaves to protect a user's computation from potentially malicious system software, including a library operating system to increase backward compatibility.

Ryoan faces issues beyond those faced by enclave-protected computation such as Haven [21]. Enclaves are intended to protect an application that is trusted by the user, which does not collude with the infrastructure, though it may unintentionally leak data via side channels. In Ryoan's model, neither the application nor the infrastructure is under the control of the user, and they may try to steal the user's secrets by colluding via *covert channels*—even if the application itself is isolated from

[1] Ryoan is a sandbox and its name is inspired by a famous dry landscape Zen garden that stimulates contemplation (Ryōan-ji).

the provider's infrastructure using enclave protection. Ryoan's goal is to prevent such covert channels and stop an untrusted application from intentionally and covertly using users' data to modulate events like system call arguments or I/O traffic statistics, which are visible to the infrastructure.

An untrusted application in Ryoan is confined by a trusted sandbox. For the Ryoan prototype we chose Native Client (NaCl) [64, 74], a state of the art user-level sandbox, as our basis (it can be built as a standalone binary, independent from the browser). NaCl uses compiler-based techniques to confine untrusted code rather then relying on address space separation, a property necessary to be compatible with SGX enclaves. The Ryoan sandbox safeguards secrets by controlling explicit I/O channels, as well as covert channels such as system call traces and data sizes.

The Ryoan prototype uses SGX to provide hardware enclaves. Each SGX enclave contains a NaCl sandbox instance that loads and executes untrusted modules. The NaCl instances communicate with each other to form a distributed sandbox that enforces strong privacy guarantees for all participating parties—the users and different service providers. Ryoan provides taint labels (similar to secrecy labels from DIFC [57]) defined by users and service providers, which allow them to ensure that any module that processes their secrets is confined by Ryoan. Confining untrusted code [47] is a longstanding problem that remains technically challenging, but Ryoan benefits from hardware-supported enclave protection. Also, Ryoan assumes a request-oriented data model, where confined modules only process input once and cannot read or write persistent storage after they receive the input. This model limits Ryoan's applicability to request-oriented server applications—but such servers are the most common way to bring scalable services to large numbers of users.

Ryoan's security goal is simple: prevent leakage of secret data. However, confining services over which the user has no control is challenging without a centralized trusted platform. We make the following contributions:

- A new execution model that allows mutually distrustful parties to process sensitive data in a distributed fashion on untrusted infrastructure.
- The design and implementation of a prototype distributed sandbox that confines untrusted code modules (possibly on different machines) and enforces I/O policies that prevent leakage of secrets.
- Several case studies of real-world application scenarios to demonstrate how they benefit from the secrecy guarantees of Ryoan, including an image processing system, an email spam/virus filter, a personal health analysis tool, and a machine translator.
- Evaluation of the performance characteristics of our prototype by measuring the execution overheads of each of its building blocks: the SGX enclave, confinement, and checkpoint/rollback. The evaluation is based on both SGX hardware and simulation.

2. Background and threat model

We assume a processor with hardware-protected enclaves, e.g., Intel's SGX-enabled Skylake (or later) architecture. SGX provides a cryptographic hash of code and initial data (called a measurement), allowing a program running in a protected enclave to verify code and data integrity and giving it access to private data encrypted by keys that the host software does not know and cannot find out. The address space of a protected enclave has its privacy and integrity guaranteed by hardware. Hardware encrypts and hashes memory contents when it moves off chip, protecting the contents from other users and also from the platform's privileged software (operating system and hypervisor). Code within an enclave can manipulate user secrets without fear of divulging them to the underlying execution platform. Code within an enclave cannot have its code or control manipulated by the platform's privileged software.

SGX's security guarantees are ideal for Ryoan's distributed NaCl-based sandbox. The sandbox confines the code it loads ensuring that the code cannot leak secrets via storage, network or other channels provided by the underlying platform. Ryoan instances communicate with each other using secure TLS connections. By collecting SGX measurements and by providing trusted initialization code, Ryoan can demonstrate to the user that their processing topology has been set up correctly.

2.1 Threat model

We consider multiple, mutually distrustful parties involved in data-processing services. A service provider is not trusted by the users of the service to keep data secret; if the service provider outsources part of the computation to other services, it becomes a user of them and does not trust them to provide secrecy, either. Each service provider can deploy its software on its own computational platform, or use a third-party cloud platform that is mutually distrustful of all service providers. We assume that users and providers trust their own code and platform, but do not trust each other's code or platforms. Everyone must trust Ryoan and SGX.

A service provider might be the same as its computational platform provider, and the two might collude to steal secrets from their input data. Besides directly communicating data, untrusted code may use covert channels via *software interfaces*, such as syscall sequences and arguments, to communicate bits from the user's input to the platform.

A user of a service does not trust the software at any privilege level in the computational platform. For example, the attacker could be the machine's owner and operator, she could be a curious or even malicious

administrator; she could be an invader who has taken control of the operating system and/or hypervisor; she might own a virtual machine physically co-located with the VM being attacked; she could even be a developer of the untrusted application or OS, and write code to directly record user input.

Ryoan takes no steps to prevent each party from leaking its own secrets intentionally (or via bugs). This model is suited for the case where the service provider deploys code on its own computational platform (see §4.2 for more discussion). When executing on a different platform provider, Ryoan provides protections against a malicious OS, e.g., system call validation to prevent Iago attacks [28] (similar to Haven [21], Inktag [40], and Sego [46]) and encryption to protect data secrecy. Orthogonal techniques [27, 31, 42, 61, 78] may be used to mitigate software bugs that unintentionally leak secret input data to a computation's output. Similarly, we assume a computational platform provider is responsible for protecting its own secrets (e.g., the administrator's password).

Denial of service is outside of the scope of our threat model. Untrusted applications can simply refuse to run or the underlying untrusted operating system can simply refuse to schedule our code.

Although we consider covert channels based on software interfaces like system calls, in this paper we do not consider side or covert channels based on *hardware limitations* (§2.3) or *execution time*. Untrusted enclaves can leak bits by modulating their cache accesses, page accesses, execution time, etc. Such channels are themselves technically difficult and often require dedicated systems to address adequately [33, 35, 44, 49, 80]. Many well-regarded secure system designs factor out side/covert channels based on hardware limitations or execution time, at least to some degree [21, 52, 60, 69, 76], because doing so enables progress in designing and building secure systems. While we do not claim to prevent the execution-time channel, Ryoan does limit the use of this channel to once per request (§5.2).

2.2 Intel Software Guard Extensions

Software Guard Extensions (SGX), which is available in new Intel processors, allow processes to shield part of their address space from privileged software. Processes on an SGX-capable machine may construct an *enclave* which is an address region whose contents are protected from all software outside of the enclave (via encryption and hashing). Code and data loaded into enclaves, therefore, can operate on secret data without fear of unintentional disclosure to the platform. These guarantees are provided by the hardware [15].

SGX provides attestations of enclave identities. For our purposes it is enough to think of an enclave identity as a hash of the enclave's initial state, i.e. valid memory contents, permissions, and relative position in the enclave. Our trust of the hardware extends to these identities; particularly we assume that the initial state of an enclave cannot be impersonated under standard cryptographic assumptions. Ryoan uses SGX to attest that all enclaves have the same initial state and thus the same identity. Before passing sensitive data to Ryoan a user will request an attestation from SGX and verify that the identity is the Ryoan identity.

Knowing the initial state of an enclave ensures that Ryoan instances are not compromised. SGX restricts enclave entry to special offsets defined in the enclave preventing return-to-libc [26, 34] style attacks.

Enclave code may access any part of the address space which does not belong to another enclave. Enclave code does not, however, have access to all x86 features. All enclave code is unprivileged (ring 3), and any instruction that would raise its privilege results in a fault.

2.3 Hardware security limitations

We discuss some known security limitations in modern Intel processors. We believe these limitations must be addressed independently from Ryoan, and we hope they will be. Each of these limitations compromise Ryoan's security goals. If there are others, they also must be addressed independently from Ryoan.

SGX page faults. As currently defined, privileged software can manipulate the page tables of an enclave to observe a page-granularity trace of its code and data. Devastating attacks have been demonstrated where application-level information is used to recreate fine-grained secrets from these coarse addresses, e.g., words in a document and images [71]. If SGX enclaves serviced their own page faults, this leakage channel would disappear.

Cache timing. Two processes resident on the same core can use cache timing to obtain fine-grained information about each other. For instance Zhang et al. demonstrated (on an Amazon EC2 like plaform) the extraction of ElGamal keys from a non-colluding VM [81]. The problem is worse when processes can collude; others have demonstrated high-bandwidth covert channels using cache behavior [70, 73]. There are hardware proposals to address cache timing attacks [51].

Address bus monitoring. Although SGX encrypts data in RAM, if an attacker monitors the address bus via a sniffer or a modified RAM chip, it forms a cacheline-granularity side or covert channel. Ryoan cannot prevent such attacks without new architectural changes.

Processor monitoring. Processor monitoring units (PMUs) provide extensive performance counter information for on-chip events. If the PMU is updated about events that occur in enclave-protected execution, the operating system could use the information as a covert channel to learn secrets via untrusted code which could modulate

Module property	Enforce	Reason
OS cannot access module memory (§2.2).	SGX	Security
Initial module code and data verified (§2.2).	SGX	Security
Can only address module memory (§2.4).	NaCl	Security
Ryoan intercepts syscalls (§2.4,§4.3).	NaCl	Security
Cannot modify SGX state (§5).	NaCl	Security
User defines topology (§4.1).	Ryoan	Security
Data flow tracked by labels (§4.2).	Ryoan	Security
Memory cleaned between requests (§5).	Ryoan	Security
Module defines initialized state (§5.4).	Ryoan	Perf.
Unconfined initialization (§5).	Ryoan	Compat.
In-memory POSIX API (§5.1)	Ryoan	Compat.

Table 1: Properties Ryoan imposes on untrusted modules, the technology that enforces them, and the reason Ryoan imposes them.

its behavior to e.g., inflate certain event counts.

According to measurements on Skylake processors, certain monitoring facilities are turned off during enclave execution (e.g., Precise Event Based Sampling (PEBS)), however the uncore counters (e.g., cache misses, TLB misses) are enabled [32]. It is unknown at this time how effective attacks based on processor monitoring will be.

Part of the purpose in constructing the Ryoan prototype is to demonstrate the importance of addressing these hardware-based information leaks.

2.4 Native Client

Google Native Client (NaCl), is a sandbox for running x86/x86-64 native code (a NaCl module) using software fault isolation. NaCl consists of a verifier and a service runtime. To guarantee that the untrusted module cannot break out of NaCl's SFI sandbox, the verifier disassembles the binary and validates the disassembled instructions as being safe to execute.

NaCl executes system calls on behalf of the loaded application. System calls in the application transfer control to the NaCl runtime which determines the proper action. Ryoan cannot allow the application to use its system calls to pass information to the underlying operating system. For example, if Ryoan passed read system calls from the application directly to the platform, the application could use the size and number of the calls to encode information about the secret data it is processing. We discuss the details of the confinement provided by Ryoan in Section 5.1.

3. Design overview

Ryoan is a distributed sandbox that executes a directed acyclic graph (DAG) of communicating untrusted modules which operate on sensitive data. Ryoan's primary job is to prevent the modules from communicating any of the sensitive data outside the confines of the system (including external hosts and the platform's privileged software).

Ryoan prevents modules from leaking sensitive data by decoupling externally visible behaviors from the content of secret data. SGX hardware limits externally visible behaviors to explicit stores to unprotected memory and

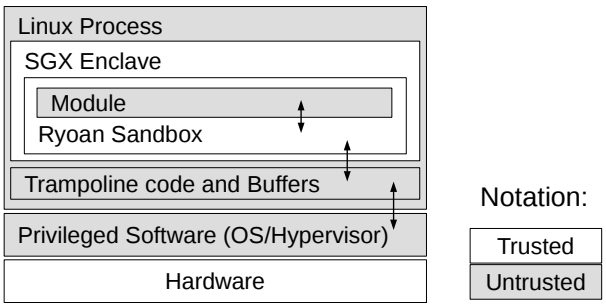

Figure 1: A single instance of Ryoan's distributed sandbox. The privileged software includes an operating system and an optional hypervisor.

use of system services (syscalls). Unprotected stores are eliminated by the NaCl tool chain and run time. Ryoan mostly eliminates system calls by providing their functionality from within NaCl. For example, Ryoan provides `mmap` functionality by managing a fixed-sized memory pool within the SGX enclave. However, untrusted modules must read input and write output so Ryoan provides a restricted IO model that prevents data leaks (e.g., the output size is a fixed function of input size). Table 1 summarizes the properties Ryoan imposes on untrusted code to achieve secure decoupling of observable behavior from secret input data.

Figure 1 shows a single instance of the Ryoan distributed sandbox. A principal (e.g., a company providing software as a service) can contribute a module which Ryoan loads and confines, enabling the module to safely operate on secret data. A module consists of code, initialized data, and the maximum size of dynamically allocated memory. The NaCl sandbox uses a load time code validator to ensure that the module cannot violate the sandbox by reaching outside of its address range or making syscalls without Ryoan intervention.

For backward compatibility, Ryoan modules support programs written for `libc`, which could include fully compiled languages and runtimes built on top of `libc`. To reduce memory use, our Ryoan prototype does not support a just-in-time compiler (JIT), though NaCl supports it [17]. Ring 0 execution is disallowed in enclaves so Ryoan cannot directly support an operating system or hypervisor. A Ryoan module can be a Linux program, or it could contain a library operating system [21].

Ryoan does not trust other software on the computational platform, including privileged software, i.e., operating system and hypervisor. Instead, Ryoan assures its own secrecy and integrity by executing in a hardware-protected enclave. Hardware attests to Ryoan's initial state becoming the anchor for Ryoan's chain of trust (Figure 2). SGX generates an unforgeable remote attestation for the user that an Ryoan instance is executing in an enclave on the platform. The user can establish an encrypted channel that she knows terminates within that Ryoan instance. SGX guarantees the enclave cryptographic secrecy and integrity against manipulation by privileged software.

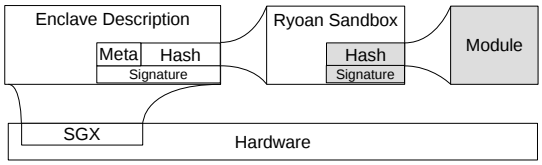

Figure 2: The Ryoan chain of trust. SGX hardware attests that a valid instance of Ryoan is executing (Hash) with an intended SGX configuration (Meta). Ryoan ensures that the expected binary is loaded with a signed hash from the software provider (grey).

A master enclave creates all Ryoan instances and they establish cryptographically protected communication channels among themselves as specified by the user. Once the distributed topology has been established, the master forwards the attestations for each node in the topology to the user who verifies that the configuration matches her specification. Then the user inputs her secret data. Ryoan provides simple labels to protect secret data added by modules in the DAG. All of Ryoan's instances together form a distributed sandbox that protects secret input data from being leaked by the untrusted code modules that operate on it.

Ryoan identity and module identity. SGX attests to the Ryoan sandbox using processor hardware and the Ryoan sandbox attests to the module's initial state (Figure 2) using software cryptography. SGX supports two forms of identity, one based on a hash of the module's initial state (MRENCLAVE) and one based on a public key, product identifier and security version number (MRSIGNER). SGX can verify Ryoan using either form of identity; our prototype uses MRENCLAVE. Ryoan can support software analogs of either identity for untrusted modules; the prototype identifies modules by the public key that signs them.

In the next section, we will describe Ryoan's distributed properties and how they are enforced, followed by a more detailed explanation of how individual instances confine modules.

4. The Ryoan distributed sandbox

The Ryoan sandbox is distributed, with different instances confining untrusted modules while all instances communicate to enforce global properties like the communication topology and secrecy labels for code and data.

4.1 Enforcing Topology

The user either defines the communication topology of confined modules or explicitly approves it. A topology is a DAG of modules with unidirectional links (see §5.2 for why Ryoan requires a DAG). The DAG specification is first passed to an initial enclave which we call the *master*. It contains standard, trusted initialization code provided by Ryoan. The master requests that the operating system start enclaves that contain Ryoan instances for modules listed in the specification. These enclaves can be hosted on different machines. The master uses SGX to perform local or remote attestation to verify the validity of individual Ryoan enclaves, then lets neighbor enclaves in the DAG establish cryptographically protected communication channels via key exchange using the untrusted network or local inter-process communication as a transport. The user can verify the validity of the master via attestation, and ask it whether a desired topology has been initialized. After that, the user establishes secure channels with the entry and exit enclaves of the DAG, and starts data processing.

The master enclave is convenient but not essential to our design. We could instead append a DAG specification to each user request, and have each enclave verify the identities of its neighbors according to the specification before sending its output.

Figure 3 shows an example of Ryoan processing input from user Alice whose sensitive data is processed by both 23andMe and Amazon. Each Ryoan instance executes in an enclave on the same or different machines. The host machine(s) might be provided by 23andMe, Amazon, or a third party. In all cases, Ryoan assures no leakage of the user's secrets and also prevents leakage of any trade secrets used by 23andMe and Amazon.

4.2 Label-based model for communication

Ryoan labels. Ryoan adapts previous label-based systems to enable multiple mutually distrustful modules to cooperate on sensitive data. Ryoan uses secrecy labels to mark secret data and enclaves which have seen that secret data. Ryoan's labels are similar to a DIFC system [45, 52, 60, 69, 76], but far simpler. Ryoan labels could also be thought of as taint tracking [30] at enclave-level granularity, with per-principal classes of taint. Taint is attached to data at unit of work granularity (where units of work are application defined). Conceptually, *a label is a set of tags*, where each tag is an opaque identifier drawn from a large universe that identifies a principal, indicating secrets from this principal.

In our prototype, a user's tag is his/her public key. A company can use its private key to sign its module binaries, and use the associated public key as the tag for those modules. The company can also use different key pairs to sign its module binaries to make them different principals, enabling privilege separation.

Label manipulation rules. Each module has the ability to add or remove a single tag that corresponds to its principal — each module can declassify its own secrets. When a module reads data with a non-empty label (e.g., from a user or another module's output), it merges the data's label with its current label which becomes the new label for both the module and the data. Ryoan marks a module's output data with the module's label.

In Figure 3, input from Alice is labeled with her tag, and the first 23andMe module adds the 23andMe tag, to make sure that its secrets cannot flow back to the user after

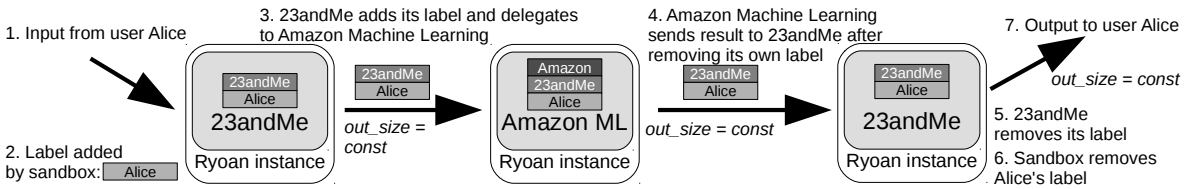

Figure 3: Ryoan's distributed sandbox. Ryoan instances manage labels on data and modules. The user's tag is propagated to all modules, making them confined after receiving input; For example, 23andMe's tag is kept when it outsources to Amazon Machine Learning to prevent leaking secrets from 23andMe.

handing them off to Amazon's machine learning module. This control is important since the user is in control of the topology. The second 23andMe module removes its tag from its output's data label. In a sense, the public key of 23andMe creates a group and both of its modules are members of the group—verified by Ryoan because both are signed with that key. Ryoan is trusted to remove the user's tag when it communicates over a protected and authenticated connection to the user.

4.2.1 Non-confining labels

A Ryoan instance is created with an empty label, and the module can add the tag that corresponds to its principal at any time. If the label does not contain tags from other principals, the module is not confined and may perform any file system operation, network communication, or address space modification permitted by Ryoan and NaCl. For example, it can freely initialize its state by reading from the network or file system. Ryoan allows unfettered access to external resources because the principal's own tag means that the module may have seen secrets only from itself. In Ryoan's threat model, principals trust their own module not to leak its own secrets (§2.1).

In many DIFC systems [45, 52, 60, 69, 76], principals are independent from the application code, e.g., multiple users (principals) use the same wiki Web application, and the users do not trust the application. Ryoan allows application owners (service providers) to be principals who trust their own code, which is different from the standard DIFC model. Although a service provider's code may have bugs that cause it to release its own secrets in its output, that is not within the threat model for Ryoan and can be mitigated using orthogonal techniques (§2.1). Ryoan protects a principal's data when that data is processed by modules that are not under control of the principal.

A service provider can host its modules and secret data on its own machines to protect them. However, if it chooses to use a third-party computational platform that it does not trust, its modules containing non-confining labels need encryption to protect persistent secrets from the platform. Ryoan uses the SGX sealing feature to store secret data on behalf of modules. Sealing provides an encryption key only accessible to enclaves with the same identity executing on the same processor. For Ryoan, all enclaves are Ryoan instances and have the same identity. Any data that the module wants to persist securely is passed to Ryoan, which adds its own metadata, including the public key of the requesting module. Ryoan seals the data and metadata and writes the result into a file. The metadata allows Ryoan to persist data on behalf of different modules and allows it to restrict any module's access to its own data.

4.2.2 Confining labels

When a module's label contains tags of other principals (as a result of receiving secrets from a user or another module's output), it enters a confined environment strictly enforced by Ryoan. Ryoan must prevent confined modules from leaking data that belong to other principals. Such labels are called *confining labels*.

Modules with confining labels are disallowed to persist data. As a result, Ryoan's label system is far simpler than DIFC systems [45, 57, 60, 69, 76]. Confined modules have seen secret data from other principals, so allowing them persistent storage violates Ryoan's "one-shot" request-oriented data model—a module processes a request's data once and only once.

4.2.3 Ryoan data audit trail

As data traverses the DAG of modules, Ryoan tracks which modules process each piece of user work. The audit trail for each work unit is available to the user as part of a DAG's output. While Ryoan cannot verify that modules are performing their intended or claimed function, an audit trail can still be useful. For example, a given piece of data might have been processed by a version of a module which is known to be faulty. Whether a user wants the audit trail and for what purpose is dependent on the application and the user.

4.3 Data oblivious communication

One of the primary safety functions of Ryoan is to prevent the computational platform from inferring secrets about the input data by observing data flow among modules. Therefore, data flow must be independent from the contents of the input data: Ryoan never moves data in response to activity under the control of the untrusted module once the module has read its input data. This safety property is sometimes called being data oblivious [58].

Units of work can be any size, but Ryoan ensures that data flow patterns do not leak secrets from input data by making module output size a fixed, application-defined

function of the input size. Ryoan protects communication with the following rules: (1) each Ryoan instance reads its entire input from every input-connected Ryoan instance before the module starts processing. (2) the size of the output is a polynomial function of the input size, specified as part of the DAG. Ryoan pads/truncates all outputs to the exact length determined by the polynomial and the size of the input. (3) Ryoan is notified by the module when its output is complete , and it writes the module's output to all output-connected Ryoan instances. Ryoan encapsulates module output in a message that contains metadata which describes what is module output and what is padding (if any). The metadata is interpreted, and any padding is stripped away by the next Ryoan instance before exposing the data to its module. These rules are sufficient because they ensure that output traffic is independent from input data (though there are possible alternatives, for example, each request could specify its output size).

Consider the scenario in Figure 3. Each input comes from a user. The user can choose to leak the size of the input, or she can hide the size by padding the input. The description of the DAG specifies that (1) the output of 23andMe's first module is padded to a fixed size defined by 23andMe which can hold the largest possible user input, (2) the output of Amazon Machine Learning's classifier module is padded to a fixed size to encode the classification result, and (3) the response to the user from 23andMe's second module is also padded to a fixed size that can hold the largest possible result. Each Ryoan instance must receive the complete input of a work unit before executing its module.

Ryoan ensures that output size is a fixed function of the input, so it is a module's mistake if the output is not large enough. Ryoan will truncate outputs that are too large and pad outputs that are too small. However, a module author should be able to describe the maximum possible output for a given sized input request. For example, a spam detector's output will be the size of the input mail message (which is just copied) plus a constant size sufficient to hold the spam rating for the email.

5. Module confinement

Ryoan relies on instances of its sandbox to prevent modules from leaking sensitive data to an adversary, including the platform's privileged software. To that end, a sandbox instance enforces the life cycle, system service restrictions, and input-output behavior of the module.

Module validation Ryoan module validation ensures that modules are safe to execute by enforcing a set of constraints on the code being loaded. Ryoan uses NaCl's load-time code validator to ensure that the module's code adheres to a strict format. NaCl's code format is designed to be efficiently verified and efficiently sandboxed, restricting control flow targets and cleanly separating code from data. Memory accesses are confined to remain within the address space occupied by the module, including fetches for execution. The detailed guarantees of NaCl are available as prior work [64, 74] and Ryoan does not change the base guarantees of the NaCl sandbox. Ryoan adds the constraints that modules may not contain any SGX instructions, and that control flow is constrained to the initial module code; i.e., Ryoan disallows dynamic code generation.

Module life cycle A Ryoan instance enforces the following life cycle on its module: creation, initialization, wait, process, output, destruction/reset. The sandbox begins by validating its module and verifying that its identity matches the DAG specification. It allows the module to initialize with an empty label and the module can give itself a non-confining label (§4.2.1). In both cases the module has full access to the system services exposed by vanilla NaCl. Non-confined initialization makes module creation more efficient and it makes porting easier because initialization code can remain unchanged.

Modules signal Ryoan when initialization is complete by calling `wait_for_work`, a routine implemented by Ryoan. Once a module is initialized, it processes a request, generates its output, and then is destroyed or reset to prevent accumulating secret data. Ryoan instances are request oriented: input can be any size, but each input is an application-defined "unit of work." For example, a unit of work can be an email when classifying spam, or a complete file when scanning for viruses. Each module gets a single opportunity to process its input data.

5.1 Ryoan's confined environment

Any module with a confining label is executed in Ryoan's confined environment. Ryoan's confined environment is intended to prevent information leakage while reducing porting effort. When a module receives the secret data contained within a request, it enters the confined environment and loses the ability to communicate with the untrusted OS via any system call. Therefore, Ryoan must provide a system API sufficient for most legacy code to perform their function. Ryoan provides these services.

- The most important service is an *in-memory virtual file system*. First, Ryoan allows files to be preloaded in memory, and the list of preloaded files must be determined before the module is confined; e.g., they can be listed in the DAG specification, or requested by the module during initialization. Ryoan presents POSIX-compatible APIs to access preloaded files that are available even after the module is confined. Second, a confined module can create temporary files and directories (which Ryoan keeps in enclave memory). When the module is destroyed or reset, all temporary files and directories are destroyed, and all changes to preloaded files are reverted.
- `mmap` calls are essential to satisfy dynamic memory allocation, so Ryoan supports anonymous memory map-

pings by returning addresses from a pre-allocated memory region. The maximum size of the region must be decided before the module becomes confined.

Ryoan's confined environment is sufficient for many data-processing tasks. For example, ClamAV, a popular virus scanning tool loads the entire virus database during initialization; when scanning the input such as a PDF file, it creates temporary files to store objects extracted from the PDF. Ryoan's in-memory file system satisfies these requirements.

However, if an application needs a large database that does not fit in memory when processing data, Ryoan cannot support it as a single module. A workaround would be to partition the database, and use multiple modules to load different partitions and perform different parts of the task, if that is feasible for a particular application.

Any design alternative that allows access to persistent files (as opposed to Ryoan's in-memory files) must cope with the covert channel created by allowing the OS to see file reads, which might occur based on computation within the untrusted module. Ryoan eliminates this channel by executing from memory only. All Ryoan modules must fit into memory for their entire lifetime because any "swapping" done by Ryoan will create a covert channel between the module and the operating system. File access techniques based on oblivious RAM (ORAM [50, 62]) can hide data access patterns, but at a performance and resource cost we deem too high for Ryoan.

5.2 Processing-time channels

A confined module cannot communicate with the untrusted OS via system calls, but it determines when its data processing is finished, which can be a channel to the OS to leak secrets by choosing different processing times depending on the data.

Fixed processing time. Timing channels can be eliminated by forcing a fixed processing time whose length is determined before the module has seen any data. The OS cannot directly determine when the module completes, so the Ryoan runtime can pad execution time by busy waiting. However, controlling its timing without the cooperation of the operating system is a challenge. Fixed processing time can be quite expensive for computations with widely variable run times, because all runtime would be padded to the worst case. However, fixed processing time can be quite modest for computations with highly predictable run times (e.g., evaluating certain machine learning models like decision trees) or with light throughput requirements. Fixed time execution does not leak information, though we defer to future work building a Ryoan instance that supports it. To add some flexibility with no loss of security, execution time could also be a fixed function of input length.

Quantized processing time. Processing time channels are mitigated by reducing the granularity of potential processing times by padding execution to a fixed number of quantized, pre-defined values [20, 67, 79, 80]. Because Ryoan only allows modules to see sensitive data once, individual modules can only leak a number of bits that is proportional to the logarithm of the number of distinct execution durations (e.g., if the code terminates after one of eight different statically determined intervals, it leaks three bits).

Randomness. Users can specify whether confined modules need access to randomness. If the user allows, a module can access randomness via the processor, e.g., Intel's RDRAND instruction. Ryoan does not allow confined modules to get randomness from the operating system. Access to randomness means a malicious module can leak random bits from an input, for example choosing an input bit at random and leaking it using its processing time. If the user repeats input data, a malicious module with access to randomness can eventually leak the entire input over its processing-time channel, even though it only leaks once for each input unit of work. Using a fixed processing time eliminates this channel.

One shot at input data. Ryoan is designed to allow each module a single opportunity to process its input data, with no opportunity to carry forward state from one input to the next. This one-shot policy limits data leakage. Therefore, Ryoan must prevent a module from accessing the same input again after reset. Ryoan enforces the one-shot policy by 1) requiring that the data processing topology is a DAG to avoid cycles; 2) Ryoan's reset mechanism deletes all data dependent on state modified after secret data is read; 3) Ryoan prevents input replay attacks by reinitializing all secure connections if any connection is ever broken. Secure communication protocols contain protection against replay attacks [75], so reinitializing broken links prevents input replay. Note that the OS can pause or stop the execution of an SGX enclave, but it cannot rollback its state [15], which means the state of a secure connection cannot be rolled back. Ryoan itself uses high-quality randomness available via the processors RDRAND instruction to establish secure connections, which does not rely on the OS.

5.3 Protecting Ryoan from privileged software

A Ryoan instance requires services provided by the untrusted operating system and possibly the hypervisor. The Ryoan instance must check the results coming from the untrusted operating system to make sure the OS is not being misleading. Most of these checks can be transparently inserted into `libc`, the lowest level of software that communicates with the operating system. Ryoan-libc is Ryoan's replacement for `libc` and it manages system call arguments and checks their return values. The Ryoan sandbox code invokes Ryoan-libc through standard `libc` functions, such as the wrappers for system calls (e.g., `read`).

Iago attacks. Ryoan-libc guards against all known Iago attacks [28] by keeping state in enclave memory and carefully checking the results of system calls e.g., making sure that addresses returned from `mmap` do not overlap with previously allocated memory (like the stack). For Linux, the system call interface can be secured e.g., by maintaining semaphore counts in enclave memory and duplicating futex [37] memory inside and outside the enclave. Ryoan shares the need for this type of checking with all systems distrustful of the operating system [29, 40, 46], though some check at a lower level than system calls [21].

Controlling an enclave's address space. SGX provides user control of memory mapping, including permissions. Ryoan-libc maintains a data structure that is equivalent to the kernel's list of virtual memory areas (VMAs). It knows about each mapped region and its permissions. Map requests are fulfilled eagerly and verified by Ryoan-libc at the time of the request (i.e., as part of the `mmap` call), not at page fault time.

SGX dictates a very specific procedure for verifying enclave mappings. A typical new mapping proceeds as follows: (1) Untrusted code notifies the kernel of a new desired mapping via a system call made by Ryoan-libc. (2) The kernel selects new enclave page frames to satisfy the mapping and modifies the page tables to map the frames at the requested virtual address with the requested permissions. (3) Untrusted user code resumes and passes control to enclave code. (4) Enclave code verifies that the mapping completed as expected by invoking the SGX instruction `EACCEPT` on every new page. The `EACCECPT` instruction accepts a virtual address and protection bits and verifies that the current address space maps that page to a valid, SGX protected 4KB physical frame. New pages added to the enclave always start out with read and write permissions and their contents are zeroed by hardware.

If the user wants something other than read and write permission, SGX provides the `EMODPE` instruction to make them more permissive and the `EMODPR` instruction which makes them less permissive. `EMODPE` is only available to enclave code while `EMODPR` is only available to privileged software (ring 0, outside of the enclave). If an enclave desires less permissive page access rights, it must signal privileged software to request the restriction, but can validate that it was done correctly through another use of the `EACCEPT` instruction. Note that the OS can always restrict page permissions against the enclave's wishes, which will create more permission faults.

Ryoan-libc emulates `mmap` behavior by doing work required by SGX on behalf of the user. For instance, if the user expects new pages to have particular contents (e.g., she privately mapped a file) and to be read-only, Ryoan-libc can copy the requested portion of the file into enclave memory and ensure those pages have read-only permissions before returning.

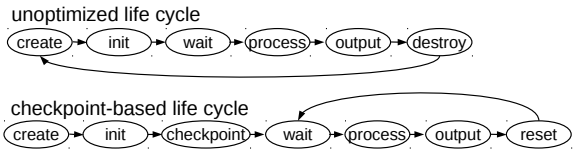

Figure 4: Instance life cycle: unoptimized vs checkpoint-based.

5.4 Optimizing module reset

The restrictions necessary to confine modules create execution time and memory space overheads. In this section we discuss strategies for mitigating these overheads.

Checkpoint-based enclave reset. Creating and initializing modules often requires far more CPU time than processing a single request (see Section 8 for measurements). For instance, loading the data necessary for virus scanning takes 24 seconds; orders of magnitude greater than the ≈0.124 seconds it takes to process a single email. Ryoan manages the module life cycle efficiently using checkpoint-based enclave reset.

Creating and initializing a hardware protected enclave is slow (e.g., we measured 30 ms for a small enclave). Compounding the problem is that applications often do not optimize their own initialization sequence on the assumption that it is not frequently executed. But Ryoan does not allow any data from one input to be carried forward to the next, so each input requires that computation begins from the same, non-secret state, making initialization a bottleneck.

Ryoan provides a checkpoint service that allows the application to be rolled back to an untainted, but initialized, memory state (Figure 4). In our prototype this state is at the first invocation of `wait_for_work`. Ryoan does not allow an enclave that has seen secret input to be checkpointed, because its data model is request-oriented: modules should not depend on past requests to operate. Checkpointing a module that has seen secret data would (potentially) give that module multiple execution opportunities on a single request's data.

Checkpoint restore allows Ryoan to save the cost of tearing down and rebuilding the SGX enclave and it saves the cost of executing the application's initialization code. Ryoan checkpoints are taken once, but restored after each request is processed. Therefore, Ryoan makes a full copy of the module's writeable state and simply tracks which pages get modified (avoiding a memory copy during processing). Only the contents of pages that were modified during input processing are restored (§6.6). SGX provides a way for enclave code to verify page permissions and be reliably notified about memory faults, which is necessary to track which pages are written.

Batch requests before reset. A user might want more efficiency by allowing a module to process several input units of work before it is reset. Whether batching multiple

inputs within a single request constitutes a threat is user and application dependent. But if a module can process more than one unit for the same data source, it can accumulate secrets across two wait-process-output cycles. Having access to more secret data for longer exacerbates the problem of slow leaks (e.g., timing channel leaks). For example, an email-filtering module allowed to process multiple emails without resetting could leak a password contained in one email by using the processing-time channel across multiple wait-process-output cycles.

6. Implementation

The Ryoan instance prototype is based on NaCl version 2d5bba1 with the last upstream commit on Jan 19 2016. We leverage NaCl's existing sandboxing guarantees to control the module's access to the platform. NaCl ensures that the module in the sandbox has no direct access to OS services. We ported NaCl for use in SGX with the introduction of the Ryoan-libc layer. NaCl depends on libc to interface with the platform. Ryoan-libc makes system calls on behalf of a Ryoan instance after checking that the system call is allowed. We modified eglibc's dynamic linker to support loading Ryoan into enclaves, but all modules must be statically linked. We base Ryoan-libc on eglibc 2.19 which is compatible with our version of NaCl.

6.1 Constraints of current hardware

Ryoan relies on features from version 2 of the SGX hardware, while only version 1 is currently available. Version 2 adds the ability to modify enclaves dynamically, i.e., augmenting an executing enclave with new memory and changing protections on existing enclave memory. Furthermore, our first generation SGX-capable machine makes only a limited amount of physical memory available to SGX (128MB on our machine).

6.2 Ryoan-libc

Ryoan-libc manages interactions with the untrusted operating system. It is impossible for the OS to read enclave memory; so Ryoan-libc marshals system call arguments into the process' untrusted memory and copies back results. Interposition from libc is common for applications that do not trust the operating system [29, 40, 46], while Haven protects a smaller system interface [21].

Fault handling. Signals allow user-level code to be interrupted by the system. The source of most signals is unreliable when the OS is untrusted, but SGX allows us to get reliable information about memory faults; this allows Ryoan-libc to expose this information to Ryoan instances though the normal signal handler registration interface. Ryoan-libc signal support is currently limited to the memory fault signal (SIGSEGV).

After any fault, exception, or interrupt the OS returns control to untrusted trampoline code contained within the process. In the case of a memory fault, rather than simply resuming the enclave where it was paused (as in the normal case), our trampoline code enters the enclave where it can read reliable information about the fault from SGX and make necessary arrangements to fix the fault (e.g., change permissions). After handling the fault, the enclave exits and then our trampoline resumes the enclave at the instruction that caused the memory fault. We cannot protect the trampoline code from the OS, but it can only enter the enclave using the EENTER instruction, which will transfer control to our fault-checking entry point, or resume the enclave using the ERESUME instruction which will re-execute the instruction that faulted. If the enclave is resumed without calling the enclave fault handler, the instruction will simply refault.

6.3 Module address space

x86-64 NaCl allocates a 84 GB region for a NaCl module with 4 GB of module address space flanked above and below by 40 GB of inaccessible guard pages, but current SGX hardware only allows enclaves with 64 GB of virtual address space. Fortunately, the original x86-64 NaCl design [64] overestimated the amount of guard pages needed to allow for future changes in the architecture. A detailed analysis [6] indicates we can remain safe by keeping the upper guard region unchanged but reducing the lower region from 40 GB to 4 GB. A Ryoan instance therefore requires 48 GB of virtual address space which fits into current SGX hardware.

6.4 I/O control

A Ryoan instance controls its module's access to files and request (work unit) buffers when it is confined, preventing the module from leaking data via direct syscalls.

In-memory virtual file system. A confined module cannot access the file system, but Ryoan implements POSIX-compatible APIs for in-memory virtual files, including preloaded files and temporary files. An in-memory file is backed by a set of 4 KB blocks that are indexed by a two-level tree structure (similar to a page table). The blocks of a file are allocated on demand as the file grows. The maximum size of an in-memory file is 1 GB. An in-memory directory is backed by a hash table, and we use reference counts to track the lifetime of files. This virtual file system supports standard APIs including `open`, `close`, `read`, `write`, `stat`, `lseek`, `unlink`, `mkdir`, `rmdir` and `getdents`. When the module writes a preloaded file, the Ryoan instance keeps the original file blocks. When the module is reset, preloaded files are restored to their original versions and temporary files are deleted.

Input/output buffers. For each unit of work, a module calls `wait_for_work` (a system service implemented by Ryoan), and the Ryoan instance reads its entire input from all input channels into memory buffers before returning to the module. After processing the work unit, the module's output is written to a buffer, and in the

next `wait_for_work` call, the Ryoan instance flushes the buffer to output channels after padding or truncating the output to a size calculated using a polynomial function of input size according to the DAG specification. The module accesses these buffers via file descriptors using APIs implemented in the virtual file system, just like using regular pipes or sockets.

6.5 Key establishment between enclaves

Ryoan instances implement protected channels using an authenticated encryption algorithm (AES-GCM [55]) provided by the libsodium [9] library. Encryption keys are agreed on at runtime using Diffie-Hellman key exchange. SGX allows you to embed the key parameters in attestations, accelerating a Diffie-Hellman key exchange between enclaves [16]. On our hardware (§8), SGX key exchange takes 1.78ms while OpenSSL takes 1.90ms. Randomness is required for key exchange and Ryoan uses the x86 instruction RDRAND to obtain it.

6.6 Checkpointing confined code

Ryoan uses page permission restriction and fault information to detect module writes. Recall that SGX provides reliable memory page permissions and information about memory faults; Ryoan does not trust the OS (§6.2). The entire module is write protected by the OS when it is confined. Ryoan verifies that the protection was done using `EACCECPT`. As the module writes, the Ryoan instance catches permission faults and records the page's address before changing the permissions to allow writes and resumes the module. However, it still needs the OS to change permissions in the page table, which requires ring-0 privilege. In fact, the page fault causes an exit from the enclave; the kernel catches it and invokes the Ryoan instance's signal handler. The handler first executes outside enclave mode and makes an `mprotect` syscall to change page permissions, then enters enclave mode to update SGX page permissions with `EMODPE`. After that, the handler returns and normal execution resumes.

To reset the enclave, all written pages are restored to their initial value and made unwritable again. In our prototype, before an untrusted module is confined for the first time, the Ryoan instance creates a checkpoint by copying the module's complete writable memory state. This copy-on-initialize strategy optimizes the case where Ryoan instances are created once and then used and reset for many requests. If the copy-on-initialize cost is too high, Ryoan could incrementally create the checkpoint by doing copy-on-write for each request, gradually accumulating and preserving unmodified versions of any page modified during any execution.

In our prototype the checkpoint is taken when the module is blocking on `wait_for_work`, and restore occurs the next time the module blocks on `wait_for_work`. This gives module writers clear semantics about what state will not persist across invocations, and allows the Ryoan instance to purge any secrets kept in registers.

Restoring a checkpoint does incur additional page faults which could be used as a channel to leak data. We find these additional faults acceptable as even normal page accesses by the module are a channel between module and OS that SGX does not close [71]. Page faults will continue to leak information about enclave execution until future generations of hardware enclaves can service their own page faults (§2.3). To make Ryoan execution on current SGX hardware more secure, we could save/restore all writable regions of the module instead of tracking individual pages using write protection. This strategy is less efficient but does not leak additional per-page information.

7. Use cases

This section explains four scenarios where Ryoan provides a previously unattainable level of security for processing sensitive data. For all examples, the Ryoan instances could execute on the same platform or on different platforms, e.g., the entire computation might execute on a third-party cloud platform like Google Compute Engine, or a provider's module might execute on its own server. Ryoan's security guarantees apply to all scenarios.

7.1 Email processing

A company can use Ryoan to outsource email filtering and scanning while keeping email text secret. We consider spam filtering and virus scanning, using popular legacy applications — DSPAM 3.10.2 and ClamAV 0.98.7.

The computation DAG for this service contains four Ryoan instances, each confining a data processing module (see Figure 5). An email arrives at the entry enclave over a secure channel, which simply distributes the email text and attachments to the enclaves containing DSPAM and ClamAV, respectively. The results of virus scanning and spam filtering are sent to a final post-processing enclave which constructs a response to the user over a secure channel.

7.2 Personal health analysis

Consider a company (e.g., 23andMe) that provides customized health reports for users based on a variety of health data. 23andMe accepts a user's genetic data, medical history, and physical activity log as input, extracts important health features from these data, and predicts the likelihood of certain diseases [1]. Since genetic and health information is extremely sensitive, users may not feel comfortable with the company keeping their data. To encourage use of the service, 23andMe can deploy it with Ryoan, assuring users that the code that processes their data cannot retain or leak their secrets.

23andMe owns its research results about the associations between diseases and health features, but it may want to use a third-party cloud machine learning service

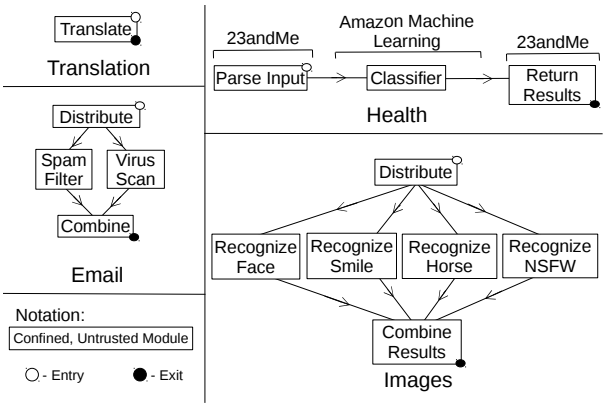

Figure 5: Topologies of Ryoan example applications. Nodes in the graph are Ryoan instances, though we identify them by their untrusted module. Users establish secure channels with trusted Ryoan code for the source and sink nodes to provide input and get output respectively.

(e.g., Amazon Machine Learning [2]), to train its model and generate predictions. 23andMe's trade secret would be how to map a user's complex, multi-modal health data onto machine learning features. Amazon Machine Learning provides a way to train models based on unlabeled features and software (a classifier) which queries that model. After training a model this way 23andMe want to keep the input to the classifier a secret from parties which have the means to map the inputs back to secret heath data: users of their service. Ryoan enables 23andMe to outsource machine learning tasks to Amazon while protecting its proprietary transformation from user data to health features.

Secrecy for both users and 23andMe is protected with a DAG shown in Figures 3 and 5. 23andMe compiles a training data set which it transmits to Amazon to construct a model. Amazon provides the classifier which queries that model as a Ryoan module. Users provide their genetic information, medical history, and activity log in a request. Upon receiving a user's request, 23andMe's first module constructs a boolean vector of health features and forwards it to Amazon's module. Amazon's module generates predictions based on the model and forwards the result to 23andMe's second enclave, which then forwards the result back to the user.

The user's label is kept throughout the entire pipeline, so that all the enclaves are confined when they receive the user's input, and prevented from leaking information about the input. Further, 23andMe keeps its label with the request sent to Amazon, so that Amazon cannot leak data about 23andMe's heath features to other parties (in particular the user), since they cannot remove 23andMe's label in order to release data out of Ryoan's confinement. Amazon's module passes the results of classification to another module owned by 23andMe which verifies its proprietary transformations are not being leaked before removing the 23andMe label and allowing results to be returned to the user.

The actual prediction model is unknown to us and out of scope for this paper; but our workload uses our knowledge of best practices. We train a support vector machine (SVM), and choose 20 well studied diseases and the top 500 genes that have correlations with the them, according to a database provided by DisGeNet [14]. The SVM models are trained using synthetic data based on that database. Our prototype uses stochastic gradient descent as the training algorithm [24] which allows incremental updates to existing models.

7.3 Image processing

Image classification as a service is an emerging area that could benefit from Ryoan's security guarantees (e.g., Clarifai [3] or IBM's Visual Recognition service [5]). We envision a scenario where a user wants different image classification services based on their expertise. For example, one service might be known for accurate identification of adult content [53] while another might do an excellent job recognizing and segmenting horses. The image processing DAG in Figure 5 shows an example where an image filtering service outsources different subtasks to different providers and then combines the results. The user's label is propagated to all processing enclaves, causing Ryoan to confine their execution. Our prototype implements all of these detection tasks using OpenCV 3.1.0, and each detection task loads a model that is specialized to the detection task and would represent a company's competitive advantage.

7.4 Translation

A company uses Ryoan to provide a machine translation service while keeping the uploaded text secret. Users upload text to the translation enclave and get the translated text back. Our prototype uses Moses [10], a statistical machine translation system. We train a phrase-based French to English model using the News Commentary data set released for the 2013 workshop in machine translation [13].

8. Evaluation

We quantify the time and space costs of Ryoan and its components by measuring the execution of the use cases described in the previous section using a combination of real hardware and emulation.

All benchmarks are measured on a Dell Inspiron 7359 laptop with Intel Core i5-6200U 2.3 GHz processor (with Skylake microarchitecture and SGX version 1) and 4 GB RAM. We use a laptop because it contains the first SGX-enabled processor we could purchase. We use Intel's SGX Linux Driver [8] and SDK [7] to measure the costs of SGX instructions. We inserted delays based on those measurements, and appropriate TLB flushes consistent with Intel's SGX specification to measure the performance of Ryoan. To test our implementation and overcome the limitations of our hardware, we built an SGX emulator based on QEMU [11] (full emulation

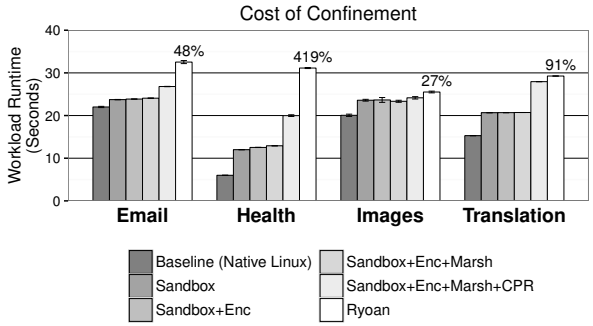

Figure 6: Runtimes of applications with Ryoan sources of overhead broken out. Each bar represents the mean of 5 trials annotated with the 95% confidence interval. Ryoan bars show percent slowdown over native. (Enc: encryption; Marsh: syscall marshaling; CPR: checkpoint restore; Ryoan: Sandbox+Enc+Marsh+CPR+SGX)

mode), augmented with SGX version 2 instructions. We could not use OpenSGX [41], because it lacked 64-bit signals. Our emulator can run a complete software stack including an SGX-aware Linux kernel.

Figure 6 shows a breakdown of the various sources of overheads for Ryoan. The baseline is to run applications built for a native Linux environment and then add sandboxing, encryption, syscall marshaling, checkpoint restore and SGX (where SGX overheads are a mix of emulation and measurements, see the discussion below). Table 2 shows the inputs for each of the workloads, as well as detailed measurements for each module in the DAG and counts of important events (the workloads are explained in Section 7).

Inputs. Workload inputs are designed to be realistic. Email bodies are taken from a spam training set [4]. Email attachments are a set of PDFs randomly attached to 30% of emails (figure taken from a study of corporate email characteristics [12]). Images are a mix of photographs, computer generated patterns, and logos. Gene data was synthesized based on DisGeNet [14]. Translation text comes from the News Commentary dataset [13].

Confinement overhead. In Figure 6, the Sandbox and Sandbox+Enc overheads are necessary for confinement, and across all workloads encryption does not add significant overheads. For Genes, the confinement overhead is high (100%) because it runs a very simple SVM classifier and the actual data processing time is small, which amplifies the effect of Ryoan's data buffering/padding and serves as a worst-case scenario. For Images, the workload involves heavy computation with OpenCV and the confinement overhead is 18%.

Checkpoint restore overhead. The CPR Size column in Table 2 shows the amount of memory copied/zeroed on checkpoint restore. Figure 6 (the difference between the Sandbox+Enc+Marsh and Sandbox+Enc+Marsh+CPR columns) shows that checkpoint restore's impact on performance is significant (55%) for Genes, because it has

		Load Size (MB)	Inited Size (MB)	Init Time (sec)	CPU Time (sec)	CPR Size	Sys. Calls	PF	Intrp
Email	Distribute	18.0	18.1	0.59	1.32	11.6MB	47k	60k	473
	DSPAM	19.6	273.5	11.15	22.10	45.3MB	1.29m	1.81m	6k
	ClamAV	21.1	403.9	24.96	29.17	83.3MB	247k	423k	7k
	Combine	18.0	18.1	0.59	0.11	16KB	12k	2k	77
Health	LoadModel	19.3	19.4	0.58	12.52	28KB	82k	280k	56k
	Classifier	19.3	19.4	0.58	18.23	36KB	1.84m	359k	151k
	Return	18.0	18.1	0.59	6.77	16KB	668K	162k	3k
Images	Distribute	18.0	18.1	0.59	0.42	632KB	2k	2k	36
	Recognize	26.6	27.1	0.63	24.79	83.2MB	88k	174k	6k
	Combine	18.0	18.1	0.59	0.36	2.5MB	14k	3k	129
	Translation	25.3	386.9	2.34	26.65	29.1MB	303k	248k	8k
	Email Input	250 emails, 30% with 103KB-12MB attachment							
	Health Input	20,000 1.4KB Boolean vectors from different users							
	Images Input	12 images, sizes 17KB-613KB							
	Trans. Input	30 short paragraphs, sizes 25-300B, 4.1KB total							

Table 2: For each workload, report counts of significant events during one execution of each module. Load Size: the size of the loaded module before execution, Inited Size: module size after initialization. Init Time: module initialization time. CPU Time: Processing time of enclave (seconds), CPR size: data copied/zeroed on checkpoint restore, Sys. Calls: system calls, PF: page faults, Intrp: interrupts. "Images: Recognize" reports the maximum of all 4 image recognition enclaves.

the lightest per-unit workload (≈1ms) and the relative cost of page fault handling is high; in contrast, its impact on Images is only 3%, which has the heaviest per-unit workload (≈2s).

SGX overhead. Executing code in an SGX-protected enclave imposes several overheads. We simulate SGX hardware overheads by using delays to model the performance of SGX instructions and we flush the TLB on all enclave exits (we could not measure execution on our hardware because it lacks SGX version 2 features (§6.1)). Besides explicit EEXIT instructions, we also model exits due to events like exceptions and interrupts (Table 2). The amount of delay for `EENTER` and `EEXIT` is based on our hardware measurement (3.9μs for each `EENTER`/`EEXIT` pair); in kind, the amount of delay added for each `ERESUME`/Async-Exit pair is based on our hardware measurement (3.14μs).

Version 2 instructions `EACCEPT`, `EMODPE`, `EMODPR` are simpler, so we model their cost at one-tenth of one `EENTER`/`EEXIT` pair. Figure 7 explores the effect of varying this cost on the runtime of our workloads. If the version 2 instruction turn out to be as costly as an `EENTER`/`EEXIT` pair (3.9μs), for instance, the running times of our email, health, images, and translation workloads increase by 25%, 14%, 7%, and 4% respectively. Every checkpoint-related page fault requires one `EMODPE` to extend page permissions. Every page reverted after checkpoint requires one `EMODPE` followed by one `EACCEPT`. Unfortunately version 2 of SGX also imposes extra synchronization (via extended behaviors of `ETRACK`) when modifying enclave page state [56]. We believe the performance effect on these workloads will be negligible, given that our applications only have one thread per

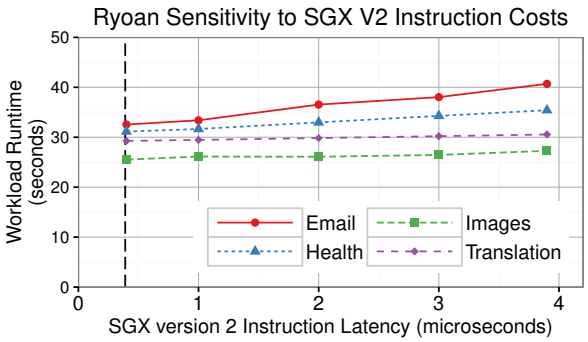

Figure 7: Ryoan application workloads' sensitivity to emulated instruction cost. The dashed vertical line denotes the delay (0.39μs) used to compute the Ryoan bars in figure 6.

enclave. SGX execution also requires syscall marshaling to copy system call arguments and results to and from untrusted memory, but the overhead of marshalling is negligible. All results are shown in Figure 6.

Checkpoint restore vs initialization. Creating an enclave and loading a module takes less than 0.5s for all our cases, but Table 2 shows application-level initialization times are over 20 seconds for DSPAM and ClamAV because they need to load and parse databases. As a result, for this workload it is preferable to use Ryoan's checkpoint-based reset rather than reinitialize the modules for every work unit. Enclave construction imposes further overheads on reinitialization. Even creation of small enclaves (e.g., 298KB) incur a penalty of 30 milliseconds. In comparison, Ryoan's checkpoint-based reset is much more efficient, and the per-unit cost is under 10ms.

9. Related work

Haven [21] allows a trusted program and its library operating system to execute in an SGX enclave that protects them from attack by host software. VC3 [63] secures trusted MapReduce using SGX. MiniBox [48] uses Native Client and a trusted computing module (TPM) to protect an application and the OS from each other. Systems like Overshadow [29], InkTag [40] and Sego [46] use a trusted hypervisor to protect trusted applications from an untrusted operating system, and InkTag/Sego also allow a trusted application to verify untrusted operating system services (e.g., a file system) with help from the hypervisor. These systems are designed to protect trusted applications in an untrusted environment, while Ryoan confines untrusted code that processes sensitive data.

Attempts to use late launch and TPMs (e.g., Flicker [54]) for user assurance suffer from poor usability due to the restricted execution environment (even in their modern incarnations such as Ironclad [39]). Code executing in an enclave can be more complex than what is practical to execute on a TPM. SGX also encrypts data in RAM to keep them secret to an enclave, thus preventing a malicious administrator from monitoring the memory bus to steal secrets, which cannot be guaranteed by TPMs.

Decentralized information flow control (DIFC) allows untrusted applications to access secret data but prevents them from leaking data to unauthorized parties. However, most DIFC systems require that all trusted code is deployed in a centralized platform or administrative domain under a trusted, privileged reference monitor [18, 45, 52, 60, 69, 76]; similar enforcements have also been realized in a browser (COWL [66]) and a mobile device (Maxoid [72]). An exception is DStar [77], which does not have a centralized reference monitor; however, although the user does not need to trust all machines involved in the system, she has to trust the machine that she wants to process her data, which means a correct reference monitor (the OS that supports DIFC) is properly installed on the machine, and that the machine's administrator does not use root privilege to steal secret data. Such trust is not required in Ryoan. Systems that track information flow down to the hardware gate level [67, 68] form a basis for strong information flow guarantees, but such hardware is not available and as designed does not include the privacy and integrity guarantees provided by SGX.

Timing and termination channels are studied in previous work [36, 43] in the context of information flow control. In Ryoan, a module has to terminate for each unit of work, and the processing-time channel can only be used once per unit; different units will not interfere due to module reset.

Homomorphic encryption [25, 38] and order-preserving encryption [22] share similar motivations with Ryoan. They allow untrusted code to perform certain operations directly on encrypted data, in order to protect secrecy. There are also systems built on these primitives [23, 59, 65]. However, these techniques usually suffer from limited application scenarios, weak security guarantees, or significant performance overhead. In comparison, Ryoan's confinement does not require domain-specific knowledge about the applications.

10. Conclusion

Ryoan allows users to safely process their secret data with software they do not trust, executing on a platform they do not control, thereby benefiting users, data processing services, and computational platforms.

11. Acknowledgments

We would like to thank Mark Silberstein, Christopher J. Rossbach, Bennet Yee and Petros Maniatis, the anonymous reviewers and our shepherd Jeff Chase for comments on early revisions of this work. We also thank Shane Williams for background on mail servers. We acknowledge funding from NSF grants CNS-1228843 and CFC-1333594, and a Google research award.

References

[1] 23andMe compares family history and genetic tests for predicting complex disease risk. http://mediacenter.23andme.com/blog/23andme-compares-family-history-and-genetic-tests-for-predicting-complex-disease-risk/. (Accessed: September 2016).

[2] Amazon machine learning. https://aws.amazon.com/machine-learning/. (Accessed: September 2016).

[3] Clarifai. https://www.clarifai.com. (Accessed: September 2016).

[4] CSMINING Group: Spam email datasets. https://csmining.org/index.php/spam-email-datasets-.html. (Accessed: April 2016).

[5] IBM visual recognition service. http://www.ibm.com/smarterplanet/us/en/ibmwatson/developercloud/visual-recognition.html. (Accessed: September 2016).

[6] Implementation and safety of NaCl SFI for x86-64. https://groups.google.com/forum/#!topic/native-client-discuss/C-wXFdR2lf8. (Accessed: September 2016).

[7] Intel(R) Software Guard Extensions for linux* OS, linux-sgx. https://github.com/01org/linux-sgx. (commit:d686fb0).

[8] Intel(R) Software Guard Extensions for linux* OS, linux-sgx-driver. https://github.com/01org/linux-sgx-driver. (commit:0fb8995).

[9] libsodium: A modern and easy-to-use crypto library. https://github.com/jedisct1/libsodium. (Accessed: September 2016).

[10] Moses. http://www.statmt.org/moses/. (Accessed: September 2016).

[11] QEMU: open source processor emulator. http://wiki.qemu.org/Main_Page. (Accessed: September 2016).

[12] The Radicati group, inc: Email statistics report 2009-20013 (summary). http://www.radicati.com/wp/wp-content/uploads/2009/05/email-stats-report-exec-summary.pdf. (Accessed: September 2016).

[13] Shared task: Machine translation. http://www.statmt.org/wmt13/translation-task.html. (Accessed: September 2016).

[14] The DisGeNET Database. http://www.disgenet.org/ds/DisGeNET/files/current/DisGeNET_2016.db.gz. (Accessed: February, 2016).

[15] Intel(R) Software Guard Extensions programming reference, 2014. https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf.

[16] Intel software guard extensions evaluation SDK users guide: Diffie-Hellman key exchange. https://software.intel.com/sites/products/sgx-sdk-users-guide-windows/Default.htm, 2015. (Accessed: September 2016).

[17] Jason Ansel, Petr Marchenko, Úlfar Erlingsson, Elijah Taylor, Brad Chen, Derek L. Schuff, David Sehr, Cliff L. Biffle, and Bennet Yee. Language-independent sandboxing of just-in-time compilation and self-modifying code. In *PLDI*, 2011.

[18] Owen Arden, Michael D George, Jed Liu, K Vikram, Aslan Askarov, and Andrew C Myers. Sharing mobile code securely with information flow control. In *IEEE S&P*, 2012.

[19] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Daniel O'Keeffe, Mark L. Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. SCONE: Secure linux containers with Intel SGX. In *OSDI*, 2016.

[20] Aslan Askarov, Danfeng Zhang, and Andrew C. Myers. Predictive black-box mitigation of timing channels. In *CCS*, 2010.

[21] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding applications from an untrusted cloud with Haven. In *OSDI*, 2014.

[22] Alexandra Boldyreva, Nathan Chenette, Younho Lee, and Adam Oneill. Order-preserving symmetric encryption. In *EuroCrypt*, 2009.

[23] Raphael Bost, Raluca Ada Popa, Stephen Tu, and Shafi Goldwasser. Machine learning classification over encrypted data. In *NDSS*, 2015.

[24] Lèon Bottou. Stochastic gradient SVM. http://leon.bottou.org/projects/sgd#stochastic_gradient_svm. (Accessed: September, 2016).

[25] Zvika Brakerski and Vinod Vaikuntanathan. Fully homomorphic encryption from ring-lwe and security for key dependent messages. In *CRYPTO*. 2011.

[26] Erik Buchanan, Ryan Roemer, Hovav Sacham, and Stefan Savage. When good instructions go bad: Generalizing return-oriented programming to risc. In *CCS*, 2008.

[27] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, 2008.

[28] Stephen Checkoway and Hovav Shacham. Iago attacks: Why the system call API is a bad untrusted RPC interface. In *ASPLOS*, 2013.

[29] Xiaoxin Chen, Tal Garfinkel, E. Christopher Lewis, Pratap Subrahmanyam, Carl A. Waldspurger, Dan Boneh, Jeffery Dwoskin, and Dan R. K. Ports. Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems. In *ASPLOS*, 2008.

[30] Jim Chow, Ben Pfaff, Tal Garfinkel, Kevin Christopher, and Mendel Rosenblum. Understanding data lifetime via whole system simulation. In *USENIX Security*, 2004.

[31] Bart Coppens, Ingrid Verbauwhede, Koen De Bosschere, and Bjorn De Sutter. Practical mitigations for timing-based

side-channel attacks on modern x86 processors. In *IEEE S&P*, 2009.

[32] Victor Costan and Srinivas Devadas. *Intel SGX Explained*. 2016.

[33] Victor Costan, Ilia Lebedev, and Srinivas Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *USENIX Security*, 2016.

[34] Solar Designer. "return-to-libc" attack. Bugtraq, 1997.

[35] Andrew Ferraiuolo, Yao Wang, Danfeng Zhang, Andrew C. Myers, and G. Edward Suh. Lattice priority scheduling: Low-overhead timing-channel protection for a shared memory controller. In *HPCA*, 2016.

[36] Bryan Ford. Plugging side-channel leaks with timing information flow control. In *HotCloud*, 2010.

[37] Hubertus Franke, Rusty Russell, and Matthew Kirkwood. Fuss, futexes and furwocks: Fast userlevel locking in linux. In *Ottawa Linux Symposium*, 2002.

[38] Craig Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009. `https://crypto.stanford.edu/craig`.

[39] Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Arjun Narayan, Bryan Parno, Danfeng Zhang, and Brian Zill. Ironclad apps: End-to-end security via automated full-system verification. In *OSDI*, 2014.

[40] Owen S. Hofmann, Sangman Kim, Alan M. Dunn, Michael Z. Lee, and Emmett Witchel. InkTag: Secure applications on an untrusted operating system. In *ASPLOS*, 2013.

[41] Prerit Jain, Soham Desai, Seongmin Kim, Ming-Wei Shih, JaeHyuk Lee, Changho Choi, Youjung Shin, Taesoo Kim, Brent Byunghoon Kang, and Dongsu Han. OpenSGX: An Open Platform for SGX Research. In *NDSS*, San Diego, CA, February 2016.

[42] Min Gyung Kang, Stephen McCamant, Pongsin Poosankam, and Dawn Song. DTA++: Dynamic taint analysis with targeted control-flow propagation. In *NDSS*, 2011.

[43] Vineeth Kashyap, Ben Wiedermann, and Ben Hardekopf. Timing-and termination-sensitive secure information flow: Exploring a new approach. In *IEEE S&P*, 2011.

[44] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. STEALTHMEM: system-level protection against cache-based side channel attacks in the cloud. In *USENIX Security*, 2012.

[45] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans, Kaashoek Eddie, and Kohler Robert Morris. Information flow control for standard OS abstractions. In *SOSP*, 2007.

[46] Youngjin Kwon, Alan Dunn, Michael Lee, Owen Hofmann, Yuanzhong Xu, and Emmett Witchel. Sego: Pervasive trusted metadata for efficiently verified untrusted system services. In *ASPLOS*, 2016.

[47] Butler W. Lampson. A note on the confinement problem. *CACM*, 16(10), October 1973.

[48] Yanlin Li, Jonathan McCune, James Newsome, Adrian Perrig, Brandon Baker, and Will Drewry. MiniBox: A two-way sandbox for x86 native code. In *USENIX ATC*, 2014.

[49] Anyi Liu, Jim Chen, and Harry Wechsler. Real-time covert timing channel detection in networked virtual environments. In *International Conference on Digital Forensics*, 2013.

[50] Chang Liu, Michael Hicks, Austin Harris, Mohit Tiwari, Martin Maas, and Elaine Shi. GhostRider: A hardware-software system for memory timerace oblivious computation. In *ASPLOS*, 2015.

[51] Fangfei Liu, Hao Wu, and Ruby B. Lee. Can randomized mapping secure instruction caches from side-channel attacks? In *HASP*, 2015.

[52] Jed Liu, Michael D George, Krishnaprasad Vikram, Xin Qi, Lucas Waye, and Andrew C Myers. Fabric: A platform for secure distributed computation and storage. In *SOSP*, 2009.

[53] Jay Mahadeokar and Gerry Pesavento. Open sourcing a deep learning solution for detecting NSFW images. `https://yahooeng.tumblr.com/post/151148689421/open-sourcing-a-deep-learning-solution-for`. (Accessed: September 2016).

[54] Jonathan M. McCune, Bryan Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. Flicker: An execution infrastructure for TCB minimization. In *EuroSys*, April 2008.

[55] David A. McGrew and John Viega. The Galois/Counter mode of operation (GCM). `http://csrc.nist.gov/groups/ST/toolkit/BCM/documents/proposedmodes/gcm/gcm-revised-spec.pdf`, 2005. (Accessed: September 2016).

[56] Frank McKeen, Ilya Alexandrovich, Ittai Anati, Dror Caspi, Simon Johnson, Rebekah Leslie-Hurd, and Carlos Rozas. Intel software guard extensions (intel sgx) support for dynamic memory management inside an enclave. In *HASP*, 2016.

[57] Andrew C. Myers and Barbara Liskov. A decentralized model for information flow control. In *SOSP*, 1997.

[58] Olga Ohrimenko, Felix Schuster, Cdric Fournet, Sebastian Nowozin Aastha Mehta, Kapil Vaswani, and Manuel Costa. Oblivious multi-party machine learning on trusted processors. In *USENIX Security*, 2016.

[59] Raluca Ada Popa, Catherine Redfield, Nickolai Zeldovich, and Hari Balakrishnan. CryptDB: protecting confidentiality with encrypted query processing. In *SOSP*, 2011.

[60] Donald E. Porter, Michael D. Bond, Indrajit Roy, Kathryn S. McKinley, and Emmett Witchel. Practical fine-grained information flow control using laminar. *TOPLAS*, 37(1), 2014.

[61] Ashay Rane, Calvin Lin, and Mohit Tiwari. Raccoon: Closing digital side-channels through obfuscated execution. In *USENIX Security*, 2015.

[62] Ling Ren, Christopher W. Fletcher, Albert Kwon, Emil Stefanov, Elaine Shi, Marten van Dijk, and Srinivas De-

vadas. Constants Count: Practical improvements to oblivious RAM. In *USENIX Security*, 2015.

[63] Felix Schuster, Manuel Costa, Cedric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. VC3: Trustworthy data analytics in the cloud using SGX. In *IEEE S&P*, 2015.

[64] David Sehr, Robert Muth, Cliff Biffle, Victor Khimenko, Egor Pasko, Karl Schimpf, Bennet Yee, and Brad Chen. Adapting software fault isolation to contemporary cpu architectures. In *USENIX Security*, 2010.

[65] Justine Sherry, Chang Lan, Raluca Ada Popa, and Sylvia Ratnasamy. Blindbox: Deep packet inspection over encrypted traffic. In *SIGCOMM*, 2015.

[66] Deian Stefan, Edward Z Yang, Petr Marchenko, Alejandro Russo, Dave Herman, Brad Karp, and David Mazieres. Protecting users by confining JavaScript with COWL. In *OSDI*, 2014.

[67] Mohit Tiwari, Xun Li, Hassan M. G. Wassel, Frederic T. Chong, and Timothy Sherwood. Execution leases: A hardware-supported mechanism for enforcing strong noninterference. In *MICRO*, 2009.

[68] Mohit Tiwari, Jason K. Oberg, Xun Li, Jonathan Valamehr, Timothy Levin, Ben Hardekopf, Ryan Kastner, Frederic T. Chong, and Timothy Sherwood. Crafting a usable microkernel, processor, and i/o system with strict and provable information flow security. In *ISCA*, 2011.

[69] Steve Vandebogart, Petros Efstathopoulos, Eddie Kohler, Maxwell Krohn, Cliff Frey, David Ziegler, Frans Kaashoek, Robert Morris, and David Mazières. Labels and event processes in the asbestos operating system. *TOCS*, 25(4), December 2007.

[70] Zhenyu Wu and Zhang Xu. Whispers in the hyper-space: High-bandwidth and reliable covert channel attacks inside the cloud. *TON*, 23(2), April 2015.

[71] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *IEEE S&P*, 2015.

[72] Yuanzhong Xu and Emmett Witchel. Maxoid: Transparently confining mobile applications with custom views of state. In *EuroSys*, 2015.

[73] Yunjing Xu, Michael Bailey, Farnam Jahanian, Kaustubh Joshi, Matti Hiltunen, and Richard Schlilchting. An exploration of l2 cache covert channels in virtualized environments. In *CCSW*, 2011.

[74] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native Client: A sandbox for portable, untrusted x86 native code. In *IEEE S&P*, 2009.

[75] Tatu Ylonen and Chris Lonvick. RFC 5246: The Transport Layer Security (TLS) Protocol: Version 1.2. `https://tools.ietf.org/html/rfc5246`, August 2008. (Accessed: September 2016).

[76] Nickolai Zeldovich, Silas Boyd-wickizer, Eddie Kohler, and David Mazires. Making information flow explicit in histar. In *OSDI*, pages 263–278. USENIX Association, 2006.

[77] Nickolai Zeldovich, Silas Boyd-Wickizer, and David Mazieres. Securing distributed systems with information flow control. In *NSDI*, 2008.

[78] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. Practical control flow integrity and randomization for binary executables. In *IEEE S&P*, 2013.

[79] Danfeng Zhang, Aslan Askarov, and Andrew C. Myers. Predictive mitigation of timing channels in interactive systems. In *CCS*, 2011.

[80] Danfeng Zhang, Aslan Askarov, and Andrew C. Myers. Language-based control and mitigation of timing channels. In *PLDI*, 2012.

[81] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. Cross-VM side channels and their use to extract private keys. In *CCS*, 2012.