Ryoan: A Distributed Sandbox for Untrusted Computation on Secret Data

TYLER HUNT and ZHITING ZHU, The University of Texas at Austin
YUANZHONG XU, Google
SIMON PETER and EMMETT WITCHEL, The University of Texas at Austin

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[1] 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) [40]) 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, health analysis, image processing and machine translation.

CCS Concepts: • **Security and privacy → Systems security**; **Operating systems security**;

Additional Key Words and Phrases: Intel SGX, enclaves, sandboxing, untrusted OS, private computation

ACM Reference format:
Tyler Hunt, Zhiting Zhu, Yuanzhong Xu, Simon Peter, and Emmett Witchel. 2018. Ryoan: A Distributed Sandbox for Untrusted Computation on Secret Data. *ACM Trans. Comput. Syst.* 35, 4, Article 13 (December 2018), 32 pages.
https://doi.org/10.1145/3231594

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

[1]This work was first published under the same title in the 12th USENIX Symposium on Operating System Design and Implementation (OSDI'16) [38]. It has been extended here with additional discussion and experiments.

We acknowledge funding from NSF grants CNS-1228843, CCF-1333594, and CNS-1618563 and a Google research award.
Authors' addresses: T. Hunt, Z. Zhu, S. Peter, and E. Witchel, The University of Texas at Austin, Computer Science Department, 2317 Speedway, Stop D9500 Austin, TX 78712; Y. Xu, Google, 1600 Amphitheatre Parkway Mountain View, CA 94043 United States.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM
0734-2071/2018/12-ART13 $15.00
https://doi.org/10.1145/3231594

services are often sensitive, such as tax documents and health data, which creates a dilemma for the user. To leverage the convenience and expertise of these services, he or she has to disclose sensitive data to them, potentially allowing them to disclose the data to further parties. If he or she wants to keep her data secret, then he or 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 the service providers can assemble 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 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,[2] 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 sandbox instances 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 the sandbox instances and protect their execution; third, Ryoan 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) [40]), 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 the secrecy of the data is protected, because privileged software sees only an encrypted version of the data and integrity is preserved by a collision-resistant hash. Haven [7] and SCONE [5] are examples of systems that use enclaves to protect a user's computation from potentially malicious system software. They include a library operating system to increase backward compatibility.

Ryoan faces issues beyond those faced by enclave-protected computation such as Haven [7]. Enclaves are intended to protect an application that is trusted by the user and does not collude with the infrastructure (though it may unintentionally leak data via side channels; see Section 2.3). In Ryoan's threat model, neither the application nor the infrastructure is under the user's control, and they may try to steal the user's secrets by colluding via *covert channels*—even if the application itself is isolated from 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.

Ryoan confines untrusted modules that make up an untrusted application. Confining untrusted code [51] is a longstanding problem that remains technically challenging. Ryoan meets the challenges of confinement by taking advantage of hardware-supported enclave protection and

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

assuming a request-oriented data model. 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 uses multiple instances of a trusted sandbox to confine an application. The trusted sandbox used in the Ryoan prototype is based on Native Client (NaCl) [78, 91], a state-of-the-art user-level sandbox (it can be built as a standalone binary, independent from the browser). NaCl uses compiler-based techniques to confine untrusted code rather than 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 sandbox instance that loads and executes untrusted modules. The sandbox 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 (similarly to secrecy labels from DIFC [68]) defined by users and service providers, which allow them to ensure that Ryoan confines any module that processes their secrets.

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 memory 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 instances. Each sandbox instance confines the code it loads ensuring that the code cannot leak secrets via storage, network or other channels provided by the underlying platform. Sandbox 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 service providers, then it becomes a user of those service providers and does not trust them to provide secrecy either. Each service provider can deploy its software on its 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 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, the untrusted code may use covert channels via *software interfaces*, such as system call 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; he or she could be a curious or even malicious administrator; he or she could be an invader who has taken control of the operating system and/or hypervisor; he or she might own a virtual machine physically co-located with the VM being attacked; he or she could even be a developer of the untrusted application or OS and write code to record user input directly.

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 Section 3.3 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 [14] (similarly to Haven [7], Inktag [37], Sego [50], and SCONE [5], and Graphene-SGX [85]) and encryption to protect data secrecy. Orthogonal techniques [13, 20, 45, 75, 95] 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 refuse to run, or the underlying untrusted operating system can refuse to schedule our code.

Although we consider covert channels based on software interfaces like system calls, in this article we do not consider side or covert channels based on *hardware limitations* (Section 2.3) or *execution time*. Untrusted enclaves can leak bits by modulating their cache accesses, page accesses, execution time, and so on. Such channels are themselves technically difficult and often require dedicated systems to address adequately [22, 28, 47, 58, 97]. Many well-regarded secure system designs factor out side/covert channels based on hardware limitations or execution time, at least to some degree [7, 61, 73, 86, 93], 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 (Section 3.1.1).

2.2 Intel Software Guard Extensions

SGX, which is available in new Intel processors, allows processes to shield part of their address space from privileged software. Processes on an SGX-capable machine may construct an *enclave* that 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. The hardware provides these guarantees [40].

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 sandbox instances are not impersonated, since SGX guarantees the integrity of enclave memory once execution begins. Enclave memory (including the executable instructions) can not be changed by the platform. SGX also enforces well-defined execution entry points to preventing the return-to-libc [12, 25] style control flow manipulation by the platform (in-enclave return-to-libc attacks are still possible).

Enclave code may access any part of the address space that 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

There are some known security limitations in modern Intel processors. We believe these limitations (and any additional limitations discovered in the future) must be addressed independently from Ryoan, and we hope they will be. Each of these limitations can compromise Ryoan's security goals. Part of the purpose of building prototype systems like Ryoan is to determine how its security guarantees depend on the security guarantees of the hardware they rely on, thereby providing motivation for fixing hardware-based limitations.

2.3.1 SGX Page Faults. On current hardware, 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 [89]. There is active research on detecting or preventing these attacks using other processor features, e.g., transactional memory [16, 80]. If SGX enclaves serviced their own page faults, then this leakage channel would disappear.

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

2.3.3 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, then the operating system could use the information as a covert channel to learn secrets via untrusted code that could modulate its behavior, e.g., to 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), however the uncore counters (e.g., last-level cache misses) are enabled [21]. Effective attacks based on branch history have been demonstrated [52]. It is unknown at this time how effective other attacks based on processor monitoring will be.

2.3.4 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 platform) the extraction of ElGamal keys from a non-colluding VM [99]. The problem is worse when processes can collude; others have demonstrated high-bandwidth covert channels using cache behavior [87, 88]. There are hardware proposals to address cache timing attacks [60].

2.3.5 Transient Instruction-based Attacks. Meltdown [57] and Spectre [48] are attacks that exploit out of order execution and speculative execution, respectively, to execute transient instructions (instructions that are executed but not retired because of an exception or incorrect branch prediction). These transient instructions influence the micro-architectural state-of-the-art processor creating covert channels; both attacks were demonstrated using the processor cache. Meltdown is specific to Intel processors and allows user-level programs to read arbitrary kernel memory. Spectre applies to a broader range of processors (including Intel, AMD, and ARM) but is more difficult to exploit. It allows the attacker to read memory belonging to a victim running in a separate address space.

Ryoan uses Native Client [78] to sandbox untrusted code. Native Client's sandboxing mechanisms prevent sandboxed code from mounting the Meltdown attack. Meltdown requires the attacker to issue memory operations for kernel addresses, which are then used in speculative execution; Native Client restricts the memory addresses that untrusted code can reference to a 4GB range that never overlaps with kernel addresses.

Native Client's sandboxing mechanisms also make it more difficult to mount Spectre attacks. Spectre attacks rely on the attacker's ability to train the hardware branch predictor. The attacker trains the branch predictor to make wrong predictions during execution of the victim. Native Client forces confined code to be position independent and restricts the targets of indirect branches to aligned blocks within a 4GB range. These measures reduce the attacker's freedom in manipulating the branch predictor, thereby shrinking the attack surface.

There have been successful Spectre attacks that violate SGX isolation, allowing non-enclave code to read enclave memory [15, 71]. Ryoan does not mitigate such violations, assuming that these security vulnerabilities will be fixed in future hardware iterations.

2.4 SGX-specific Limitations

These limitations exist in the threat model implied by SGX, in which privileged software is not trusted but controls resources like persistent storage and the network. Ryoan obviates these limitations by design (Section 3.6).

2.4.1 Rollback. A platform can roll back any persistent state. SGX can hash and encrypt data for storage using a hardware-generated secret key. However, it provides no guarantees about freshness. This drawback forces enclave designers to rely on hardware counters [82], which have limitations and performance issues, or software-based strategies that leverage other machines [63].

2.4.2 Enclave Indistinguishability. While SGX enables enclaves to attest their integrity to outside parties, nothing prevents the platform from instantiating multiple copies of enclaves. Without a mechanism to uniquely identify different instances of the same enclave, a malicious platform could confuse remote users about the state of a particular enclave by maliciously redirecting communication between a set of enclaves that remote users perceive as a single enclave. Mechanisms like ephemeral session keys (employed by TLS) allow the victim to detect such attacks.

2.5 Application Limitations

Ryoan forces applications to adopt a request-oriented data model. This data model is sufficient for batch processing of mostly unique inputs. There are application behaviors that do not map

cleanly—or at all—onto Ryoan's data model. Below are classes of application behavior that Ryoan does not support.

2.5.1 Storage. Ryoan is not suited for storage; it is intended to safeguard computation on sensitive inputs. Once a Ryoan module has seen user data, Ryoan prevents the module from writing to persistent storage.

2.5.2 Network Metadata. Ryoan takes no steps to protect network connection metadata like the user's IP address or the length of packets. Ryoan protects user data, but not connection metadata (though systems exist that do protect connection metadata, e.g., Tor [26] hides a client's network address from the server).

2.5.3 Repeated Computations on the Same/Similar Input Data. Ryoan cannot eliminate all timing channels, and instead mitigates their effect with its request-oriented data model. For services that repeatedly process the same or very similar inputs, Ryoan might leak too much confidential information. For example, some online photo services intend for users to repeatedly read and edit photos. Ryoan is not well suited for these services, because with enough repeated input, untrusted modules will be able to exfiltrate the input data.

2.5.4 Multi-user Computation. If a single request contains secrets from multiple, mutually distrusting users, then Ryoan cannot isolate these secrets. Ryoan tracks dataflows at a request granularity and applications are free to mix data within a single request, even if that data comes from different, mutually distrusting users.

2.6 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. The verifier disassembles the binary and validates the disassembled instructions as being safe to execute, to guarantee that the untrusted module cannot break out of NaCl's SFI sandbox.

NaCl executes system calls on behalf of the loaded application. System calls in the application transfer control to the NaCl runtime that 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 3.5.

3 DESIGN

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

A module consists of code, initialized data, and the maximum size of dynamically allocated memory. For backward compatibility, Ryoan modules support programs written for `libc`, which could include fully compiled languages and runtimes built on top of `libc`. A Ryoan module can be a Linux program, or it could contain a library operating system [7]. SGX disallows ring 0 execution in enclaves so Ryoan cannot directly support an operating system or hypervisor.

Confining modules without trusting privileged software (i.e., the operating system and hypervisor) is Ryoan's chief technical challenge. In the worst case, the modules and privileged software can collude to steal secrets. The possibility of collusion forces Ryoan to consider any behavior

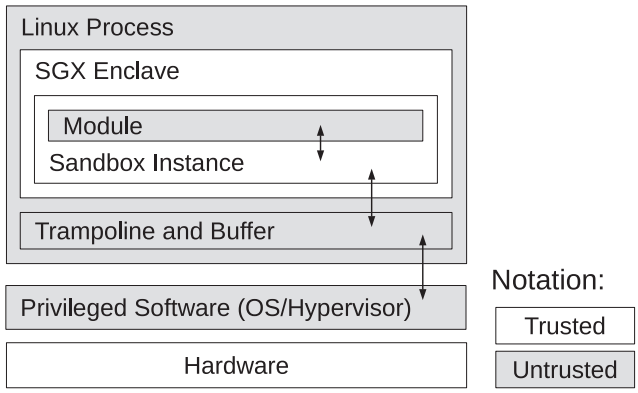

Fig. 1. One of several sandbox instances that make up a Ryoan deployment. The privileged software includes an operating system and an optional hypervisor.

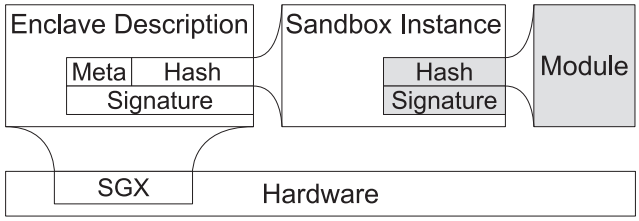

Fig. 2. The Ryoan chain of trust. SGX hardware attests that a valid sandbox instance is executing (Hash) with an intended SGX configuration (Meta). The sandbox instance ensures that the expected binary is loaded with a signed hash from the software provider (gray).

visible to privileged software (henceforth *externally visible behavior*) to be a potential channel for leaking secrets.

Figure 1 shows a single instance of the distributed sandbox. A principal (e.g., a company providing software as a service) can contribute a module that Ryoan loads and confines, enabling the module to operate on secret data safely. We will refer to any principle that provides a module as a *module provider*. The NaCl sandbox uses a load-time code validator to ensure that the module cannot violate the sandbox by accessing memory outside of its address range or making syscalls without Ryoan intervention.

Ryoan assures its secrecy and integrity by executing in hardware-protected enclaves provided by SGX. Hardware attests to Ryoan's initial state, and in doing so, hardware becomes the anchor for Ryoan's chain of trust (Figure 2). SGX generates an unforgeable remote attestation for the user that a sandbox instance is executing in an enclave on the platform. The user can establish an encrypted channel that he or she knows terminates within that sandbox instance. SGX guarantees the enclave cryptographic secrecy and integrity against manipulation by privileged software.

A master enclave creates all sandbox 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 his or her specification. Then the user inputs his or her secret data. Ryoan provides simple labels to protect secret data added by modules in the DAG (Section 3.3). All of Ryoan's sandbox instances together form a distributed sandbox that protects secret input data from being leaked by the untrusted modules that operate on it.

Table 1. Properties Ryoan Imposes on Untrusted Modules, The Technology That Enforces Them, and the Reason Ryoan Imposes Them

Module property	Enforced by	Reason
OS cannot access module memory (Section 2.2).	SGX	Security
Initial module code and data verified (Section 2.2).	SGX	Security
Can only address module memory (Section 2.6).	NaCl	Security
Ryoan intercepts syscalls (Section 2.6, Section 3.1).	NaCl	Security
Cannot modify SGX state (Section 3.2.3).	NaCl	Security
User defines topology (Section 3.2).	Ryoan	Security
Data flow tracked by labels (Section 3.3).	Ryoan	Security
Memory cleaned between requests (Section 3.1).	Ryoan	Security
Module defines initialized state (Section 3.4).	Ryoan	Performance
Unconfined initialization (Section 3.2.3).	Ryoan	Compatibility
In-memory POSIX API (Section 3.5).	Ryoan	Compatibility

Ryoan prevents modules from leaking sensitive data by decoupling externally visible behaviors from the content of secret data. Anything the module does in response to input data is in danger of being a side channel that communicates that data. Ryoan, therefore, makes the module's externally visible behavior independent of the input data. SGX hardware limits externally visible behaviors to explicit stores to unprotected memory and use of system services (syscalls). The NaCl toolchain and runtime eliminate unprotected stores.

Ryoan eliminates most 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 I/O 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 modules to achieve secure decoupling of observable behavior from secret input data.

Figure 4 shows an example of Ryoan processing input from user Alice whose sensitive data is processed by both 23andMe and Amazon. Each sandbox 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.

3.0.1 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 that is known to be faulty. Whether a user wants the audit trail and for what purpose is dependent on the application and the user.

3.1 Restricted I/O Model

In most cases, Ryoan disallows access to or replaces system services in the interest of eliminating module-controlled externally visible behaviors. However, I/O must be allowed, and cannot be replaced by Ryoan, which does not control devices directly. Instead, Ryoan enforces a restricted I/O model on modules. The I/O model ensures that dataflow is always independent of 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 [70].

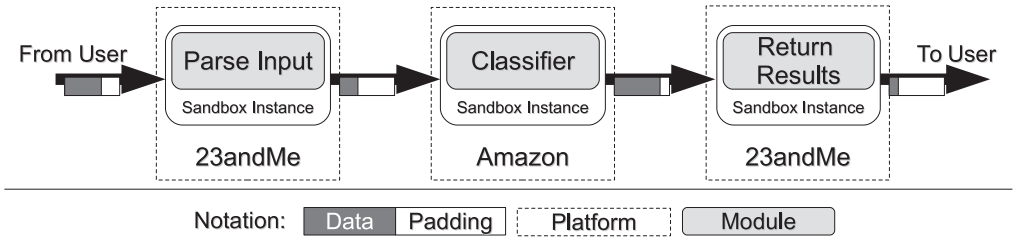

Fig. 3. Ryoan's distributed sandbox. Modules contributed by principals—in this case, the platform providers, 23andMe, and Amazon—are confined to process users' data safely.

Ryoan requires modules to be 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 a single unit of work. After generating output, the module must be destroyed (or reset, see Section 3.4) to prevent it from sending the secrets of one user to another, or using the processing time of future requests to leak information about past requests (see Section 3.1.1 for a full discussion).

Units of work can be any size, but Ryoan ensures that dataflow 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 sandbox instance reads its entire input from every input-connected sandbox instance before the module starts processing. (2) The size of the output is a fixed function of the input size, specified as part of the DAG. Sandbox instances pad or truncate all outputs to the exact length determined by the function and the size of the input. (3) Each sandbox instance is notified by its module when its output is complete, and it writes the module's output to all output-connected sandbox instances. Sandbox instances encapsulate module output in a message that contains metadata that describes what part of the message is module output and what part is padding (if any). The metadata are interpreted, and any padding is stripped away by the next sandbox instance before exposing the data to its module. These rules are sufficient, because they ensure that output traffic is independent of 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 he or she can hide the size by padding the input. The description of the application specifies that (1) the output of 23andMe's first module is padded to a fixed size defined by 23andMe that 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 sandbox 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 input-request size. 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. For many tasks, it is easy to bound the size of the output based on the input. For example, it is straightforward to bound the size of a machine-learning model with a known topology for a known task with known training data or to bound the size of a translation from one human language to another.

3.1.1 Processing-time Channels. While Ryoan carefully controls I/O, the amount of time it takes to carry out computation is controlled by the module. A module could use this fact to construct a timing channel by varying the length of time it takes to generate an output based on secrets in the input data. Ryoan takes the following steps to limit leaks through processing time channels.

- *One shot at input data.* Ryoan allows 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. Ryoan enforces the one-shot policy by (1) requiring that the data processing topology be a DAG to avoid cycles, (2) disallowing access to any state modified by processing a different unit of work, and (3) preventing input replay attacks by reinitializing all secure connections if any connection is ever broken. Secure communication protocols contain protection against replay attacks [92], so reinitializing broken links prevents input replay. Note that the OS can pause or stop the execution of an SGX enclave, but it cannot roll back its state [40], which means the state of a secure connection cannot be rolled back. Ryoan itself uses high-quality randomness available via the processor's RDRAND instruction to establish secure connections, which does not rely on the OS.
- *Randomness.* Users can specify whether confined modules need access to randomness. If the user allows, then 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, then 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.

  Some natural types of input data can function as a source of randomness. Ryoan assumes that repeated inputs are bit-for-bit identical. If a computation's input contains ever-changing metadata (e.g., an embedded timestamp of the request), then a confined module can use these changing bits to seed a pseudo-random number generator to leak multiple bits from the semantically identical input. Just like users must take care to prevent leaking the size of their input data, they must also take care to avoid semantically identical inputs encoded into different bit representations.

Below are other design choices that would provide stronger leak mitigation. They are not implemented in the prototype.

- *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; and thus 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 runtimes, because all runtimes would be padded to the worst case. However, fixed processing time can be quite modest for computations with highly predictable runtimes (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 sandbox instance that supports it. Execution time could also be a fixed-function of input length, to add flexibility with no loss of security.
- *Quantized processing time.* Processing time channels are mitigated by reducing the granularity of potential processing times. Execution is padded to a fixed number of quantized, pre-defined values [6, 83, 96, 97]. 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, then it leaks three bits).

3.2 Secure Initialization

Ryoan's secure initialization ensures that modules are loaded correctly by genuine sandbox instances in the specified topology for a particular application. A Ryoan application is described by a *DAG specification*, which specifies how modules should be connected (always a DAG for safety, see Section 3.1.1). The user either defines the DAG specification or explicitly approves it.

3.2.1 Initializing the Application. The DAG specification is first passed to a bootstrap enclave, which we call the *master*, that contains standard, trusted initialization code provided by Ryoan. The master requests that the platform instantiate enclaves that contain sandbox instances for modules listed in the specification. These enclaves can be hosted on different machines. The master uses SGX local or remote attestation to verify the validity of each sandbox instance, then informs the sandbox instances of the location of their neighbors in the DAG specification. Sandbox instances establish cryptographically protected communication channels via key exchange with their neighbors using the appropriate untrusted communication medium (e.g., the network or local interprocess communication) as transport.

The user can verify the validity of the master via attestation, and ask it whether the desired topology has been initialized. If this is true, then the user establishes secure channels with the entry and exit sandbox instances of the DAG, and data processing begins.

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

3.2.2 Ryoan Identity and Module Identity. SGX attests to the sandbox instances using processor hardware, and the sandbox instances in turn attest to the modules' initial state using software cryptography (Figure 2). SGX supports two forms of identity, one based on a hash of the enclave'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.

3.2.3 Module Initialization. A sandbox instance begins by verifying the module it is given to load matches the DAG specification. On successful verification, the sandbox instance continues by loading and validating its module. Successfully validated modules are allowed to initialize. While initializing, the module is not confined and 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 accumulation of secret data.

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 execution fetches. The detailed guarantees of NaCl are available as prior work [78, 91] 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.

3.2.4 Sandbox instance Migration. To balance server utilization, Ryoan might periodically reconfigure the deployment of the data processing DAG. Because Ryoan is intended to process secret data once, it does not maintain or migrate persistent state. However, modules might maintain persistent data, for example databases for initialization. Ryoan makes no guarantees about module persistent state; module providers should consider their trust relationship with the data provider and delivery system when deciding to depend on persistent data (e.g., using encryption and MACs to provide rollback protection). If a module stores persistent data, then the service provider is expected to make that data available to the module when it is re-initialized after migration, e.g., by storing it in a distributed data store accessible on the new node.

The Ryoan prototype only supports the most coarse-grained migration achieved by shutting down the processing DAG and recreating it on a new set of nodes. Should migration become a more frequent operation, it can be optimized by coordinating with the master enclave, e.g., to migrate only certain nodes in the DAG.

3.3 Protecting Module Provider Secrets

Ryoan uses security labels to prevent module provider secrets from flowing back to the user. Conceptually, *a label is a set of tags*, where each tag is an opaque identifier drawn from a vast universe that identifies a principal, indicating secrets from this principal. Ryoan uses public keys as tags. Ryoan assigns the user's tag to any data provided by the user. Module binaries are signed; a loaded module's tag is the public key that correctly verifies the signature on its binary. A module provider could choose to use different key pairs to sign its module binaries, enabling privilege separation.

Ryoan adapts previous label-based systems to enable multiple mutually distrustful modules to process sensitive data cooperatively. Ryoan labels are similar to labels in DIFC systems [49, 61, 68, 73, 73, 86, 93] but are far simpler. Ryoan labels are only used to reason about data secrecy (not integrity) and are coarse grained; they are applied to entire modules and the data they generate. Ryoan's use of labels could also be thought of as taint tracking [18] at enclave-level granularity, with per-principal classes of taint. Taint is attached to data at unit of work granularity (where the units of work are application defined).

3.3.1 Label Manipulation Rules. Each module is created with an empty label and 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), the module's label is replaced with the union of the data's label and the module's old label. Ryoan marks a module's output data with the module's label.

In Figure 4, 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 handing them off to Amazon's machine-learning module. This control is essential, 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.

3.3.2 Non-confining Labels. If a module's label does not contain tags from other principals, then the module is not confined. Such labels are called *non-confining labels.* A module with a non-confining label may perform any file system operation, network communication, or address space

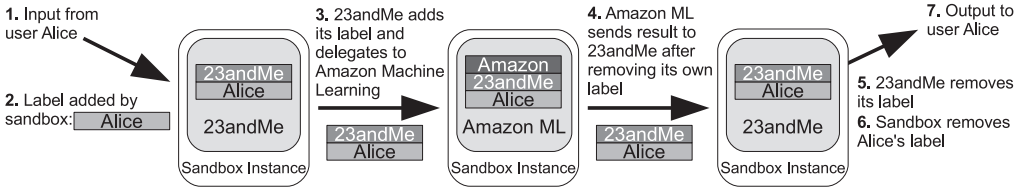

Fig. 4. Sandbox instances manage labels on data and modules. The user's tag is propagated to all modules, making them confined after receiving input; for example, 23andMe's tag is kept when it outsources to Amazon Machine Learning to prevent leaking secrets from 23andMe.

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, each principal trusts their own module not to leak their secrets (Section 2.1) and to validate any data it receives from an untrustworthy source.

In many DIFC systems [49, 61, 73, 86, 93], principals are independent of 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 (module providers) to be principals who trust their own code, which is different from the standard DIFC model. Although a module 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 (Section 2.1). Ryoan protects a principal's data when that data are processed by modules that are not under the principal's control.

A module 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 contain sandbox 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.

3.3.3 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 is confined by Ryoan. Such labels are called *confining labels*. A confining label indicates the module may have seen the secrets of other principals; Ryoan must prevent the module from leaking those secrets.

Ryoan prevents modules with confining labels from persisting data. As a result, Ryoan's label system is far simpler than DIFC systems [49, 68, 73, 86, 93]. 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 once and only once.

3.4 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.

3.4.1 Checkpoint-based Enclave Reset. Creating and initializing modules often requires far more CPU time than processing a single request (see Section 6 for measurements). For instance,

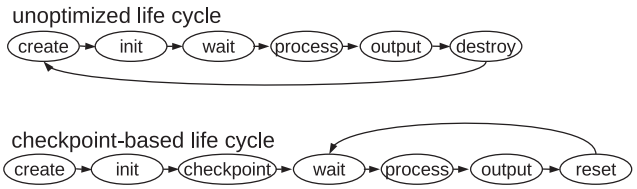

Fig. 5. Sandbox instance lifecycle: unoptimized vs. checkpoint based.

loading the data necessary for virus scanning takes 24s, orders of magnitude greater than the ≈0.124s it takes to process a single email. Ryoan manages the module lifecycle efficiently using checkpoint-based enclave reset.

Creating and initializing a hardware protected enclave is slow (e.g., we measured 30ms 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. However, Ryoan does not allow any data from one input unit of work to be carried forward to the next, so each input requires that the 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 5). 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 cannot depend on data from 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 unit of work.

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 that pages get modified (avoiding a memory copy during processing). Only the contents of pages that were modified during input processing are restored (Section 4.6). 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.

3.4.2 Batch Requests before a 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. However, if a module can process more than one unit for the same data source, it can accumulate secrets across multiple wait-process-output cycles. Having access to more secret data for longer time periods 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 multiple bits of a password contained in one email by using the processing-time channel across multiple wait-process-output cycles.

3.5 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. To allow code developed for general purpose computing environments to be used within Ryoan, the trusted Ryoan runtime can provide services for backward compatibility. 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 provides a system API sufficient for most legacy code to perform their function without having to be modified. Ryoan provides these services:

- The most important service is an *in-memory virtual file system.* First, Ryoan allows users to preload files into module 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 mappings 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 [59, 76]) can hide data access patterns, but at a performance and resource cost we deem too high for Ryoan.

3.6 Protecting Ryoan from Privileged Software

A sandbox instance requires services provided by the untrusted operating system and possibly the hypervisor. The sandbox instance must check the results coming from the untrusted operating system to make sure it is not misled. 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`). SCONE [5] and Graphene-SGX [85], also provide protection by modifying `libc`.

3.6.1 Iago Attacks. Ryoan-libc guards against all known Iago attacks [14] 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 [30] memory inside and outside the enclave. Ryoan shares the need for this checking with all systems distrustful of the operating system [17, 37, 50], though some check at a lower level than system calls [7].

3.6.2 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 OS 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 `EACCEPT` 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, then SGX provides the `EMODPE` instruction to make them more permissive and the `EMODPR` instruction that 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, then it must signal privileged software to request the restriction but can validate that it was done correctly through another use of the `EACCEPT` instruction.

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., he or she privately mapped a file) and to be read-only, then Ryoan-libc copies the requested portion of the file into enclave memory and ensures those pages have read-only permissions before returning.

3.6.3 Rollback. Privileged software can roll back any persistent state. Ryoan does not depend on persistent state, preventing rollback attacks by design. Ryoan also provides mechanisms that allow module providers to avoid dependence on persistent state. Initialization of Ryoan depends only on its initial in-memory state that is protected and attested by hardware. All other state is derived from hardware randomness or provided securely at runtime by the sandbox provider.

A module might use persistent state, for example during initialization before seeing any user-supplied secrets. In Figure 3, Amazon's machine-learning classifier might load pre-computed parameters stored in a Ryoan-managed per-application directory in the local file system. Module providers should employ encryption, hashing, and rollback protection as appropriate to their trust relationship with the platform provider (and with any provider of information).

Persistent state protection for modules is the module provider's responsibility, just as module functionality/correctness is the module provider's responsibility. Ryoan guarantees that once a module sees user data, it cannot leak that data, but it does not guarantee that modules act according to a specification, e.g., that a module correctly identifies spam.

3.6.4 Enclave Indistinguishability. While SGX enables enclaves to attest their integrity to outside parties, nothing prevents the platform from instantiating multiple copies of enclaves. Ryoan prevents the platform from exploiting this fact by establishing secure channels between different enclaves and between enclaves and the user with keys that are never persisted and therefore must be renegotiated with each new enclave.

4 IMPLEMENTATION

The sandbox instance prototype is based on NaCl version 2d5bba1 with the last upstream commit on January 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 sandbox 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 that is compatible with our version of NaCl.

4.1 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. Ryoan relies on changing memory protections to implement efficient checkpoint recovery. Furthermore, our first generation SGX-capable machine makes only a limited amount of physical memory available to SGX (128MB on our machine).

4.2 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 [17, 37, 50], while Haven protects a smaller system interface [7].

4.2.1 Fault Handling. Signals allow user-level code to be interrupted by the system. The sources of most signals are 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 sandbox instances through 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 that will re-execute the instruction that faulted. If the enclave is resumed without calling the enclave fault handler, then the instruction will simply re-fault.

4.3 Module Address Space

x86-64 NaCl allocates an 84GB region for a NaCl module with 4GB of module address space flanked above and below by 40GB of inaccessible guard pages, but current SGX hardware only allows enclaves with 64GB of virtual address space. Fortunately, the original x86-64 NaCl design [78] overestimated the number of guard pages needed to allow for future changes in the architecture. A detailed analysis [32] indicates we can remain safe by keeping the upper guard region unchanged but reducing the lower region from 40GB to 4GB. A sandbox instance, therefore, requires 48GB of virtual address space that fits into current SGX hardware.

4.4 I/O Control

A sandbox 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.

4.4.1 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 4KB 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 1GB. 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 sandbox instance keeps the original file blocks. When the module is reset preloaded files are restored to their original versions, and temporary files are deleted.

4.4.2 Input/output Buffers. For each unit of work, a module calls `wait_for_work` (a system service implemented by Ryoan), and the sandbox 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 sandbox instance flushes the buffer to output channels after padding or truncating the output to a size calculated using a fixed 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.

4.5 Key Establishment between Enclaves

Sandbox instances implement protected channels using an authenticated encryption algorithm (AES-GCM [65]) provided by the libsodium [55] library. Encryption keys are agreed on at runtime using Diffie-Hellman key exchange. SGX allows enclave code to embed the key parameters in attestations, accelerating a Diffie-Hellman key exchange between enclaves [41]. On our hardware (Section 6), 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.

4.6 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 (Section 4.2). The entire module is write protected by the OS when it is confined. Ryoan verifies that the protection was done using `EACCEPT`. As the module writes, the sandbox instance catches permission faults and records the offending page's address before changing the permissions to allow writes and resumes the module. However, updating the permissions in the page table requires ring-0 privilege. The sandbox instance's signal handler first executes outside the enclave and makes an `mprotect` system call to change the page permissions, to avoid an extra enclave exit. Once that process is complete it enters enclave mode to update SGX page permissions with `EMODPE` (and performs the aforementioned bookkeeping). With that done, the handler returns and normal execution resumes.

All written pages are restored to their initial value and made unwritable again to reset the enclave. In our prototype, before an untrusted module is confined for the first time, the sandbox instance creates a checkpoint by copying the module's complete writable memory state. This copy-on-initialize strategy optimizes the case where sandbox instances are created once and then used and reset for many requests. If the copy-on-initialize cost is too high, then 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

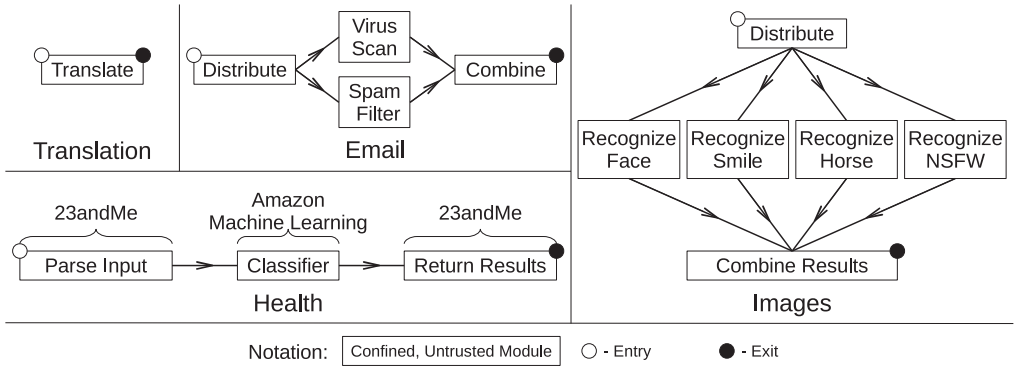

Fig. 6. Topologies of Ryoan example applications. Nodes in the graph are sandbox instances, though we identify them by their untrusted module. Users establish secure channels with trusted Ryoan code for the source and sink nodes to provide input and get output respectively.

semantics about what state will not persist across invocations, and allows the sandbox instance to purge any secrets kept in registers.

Restoring a checkpoint does incur additional page faults that 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 [89]. Page faults will continue to leak information about enclave execution until future generations of hardware enclaves can service their own page faults (Section 2.3), or SGX provides another hardware fix. 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.

5 USE CASES

This section explains four scenarios where Ryoan provides a previously unattainable level of security for processing sensitive data. For all examples, the sandbox 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.

5.1 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 sandbox instances, each confining a data processing module (see Figure 6). An email arrives at the entry enclave over a secure channel, which 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 that constructs a response to the user over a secure channel.

5.2 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 the 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 machine-learning service in the cloud (e.g., Amazon Machine Learning [3]) to train its model and generate predictions. 23andMe's trade secret is 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) that queries that model. After training a model this way, 23andMe wants to keep the input to the classifier a secret from parties who have the means to map the inputs back to secret health 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 4 and 6. 23andMe compiles a training dataset that it transmits to Amazon to construct a model. Amazon provides the classifier that queries that model as a Ryoan module. Users provide their genetic information, medical history, and activity log in a request. On 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 are 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 health features to other parties (in particular the user), since they cannot remove 23andMe's label to release data out of Ryoan's confinement. Amazon's module passes the results of classification to another module owned by 23andMe that verifies its proprietary transformations are not being leaked before removing the 23andMe label and allowing results to be returned to the user.

Real genetic prediction models are proprietary, unknown to us, and out of scope for this article; our workload uses general knowledge and best practices. We train a support vector machine (SVM) and choose 20 well-studied diseases and their top 500 correlated genes, according to a database provided by DisGeNet [33]. The SVM models are trained using synthetic data based on that database. Our prototype uses stochastic gradient descent as the training algorithm [10] that allows incremental updates to existing models.

5.3 Image Processing

Image classification as a service is an emerging area that could benefit from Ryoan's security guarantees (e.g., Clarifai [19] or IBM's Visual Recognition service [39]). 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 [62] while another might do an excellent job of recognizing and segmenting horses. The image processing DAG in Figure 6 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.

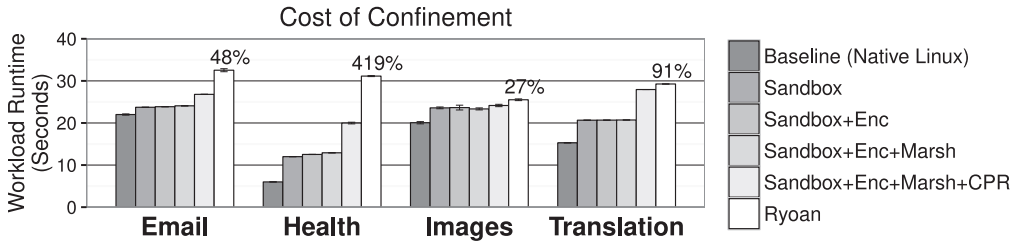

Fig. 7. Runtimes of applications with Ryoan overheads enumerated. Each bar represents the mean of 5 trials annotated with the 95% confidence interval. Ryoan bars show percent slowdown over native. (Enc: encryption; Marsh: syscall marshaling; CPR: checkpoint restore; Ryoan: Sandbox+Enc+Marsh+CPR+SGX).

5.4 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 [67], a statistical machine translation system. We train a phrase-based French to English model using the News Commentary dataset released for the 2013 workshop in machine translation [2].

6 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 an Intel Core i5-6200U 2.3GHz processor (with Skylake microarchitecture and SGX version 1) and 4GB RAM. We use a laptop, because it contains the first SGX-enabled processor we could purchase; however, we validate our measurements using a more recent Intel E3-1270 (see the analysis for SGX Overhead below). We use Intel's SGX Linux Driver [43] and SDK [42] to measure the costs of SGX instructions.

To test our implementation and overcome the limitations of our hardware, we built an SGX emulator based on QEMU [74] (full emulation mode), augmented with SGX version 2 instructions. To emulate the performance of SGX V2, we insert delays based on our measurements of current SGX hardware, flush the TLB according to Intel's SGX specification, and estimate overhead for V2 instructions based on the performance of V1 instructions. We could not use OpenSGX [44], because it lacked 64-bit signals. Our emulator can run a complete software stack including an SGX-aware Linux kernel.

6.1 Understanding Workload Performance

Figure 7 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 5).

6.1.1 Inputs. Workload inputs are designed to be realistic. Email bodies are taken from a spam training set [24]. Email attachments are a set of PDFs randomly attached to 30% of emails (and that figure is taken from a study of corporate email characteristics [34]). Images are a mix of photographs, computer-generated patterns, and logos. Gene data were synthesized based on DisGeNet [33]. Translation text comes from the News Commentary dataset [2].

Table 2. Inputs for Each Application

Application	Input	
Email	250	emails, 30% with 103KB–12MB attachment
Health	20,000	1.4KB Boolean vectors from different users
Images	12	images, sizes 17KB–613KB
Translate	30	short paragraphs, sizes 25–300B, 4.1KB total

Table 3. For Each Workload, Report Counts of Significant Events During One Execution of Each Module

		Load Size (MB)	Inited Size (MB)	CPR Size	Init Time (s)	CPU Time (s)	Sys. Calls Count	PF Count	Int Count
Email	Distribute	18.0	18.1	11.6MB	0.59	1.32	47k	60k	473
	DSPAM	19.6	273.5	45.3MB	11.15	22.10	1.29m	1.81m	6k
	ClamAV	21.1	403.9	83.3MB	24.96	29.17	247k	423k	7k
	Combine	18.0	18.1	16KB	0.59	0.11	12k	2k	77
Health	LoadModel	19.3	19.4	28KB	0.58	12.52	82k	280k	56k
	Classifier	19.3	19.4	36KB	0.58	18.23	1.84m	359k	151k
	Return	18.0	18.1	16KB	0.59	6.77	668K	162k	3k
Images	Distribute	18.0	18.1	632KB	0.59	0.42	2k	2k	36
	Recognize	26.6	27.1	83.2MB	0.63	24.79	88k	174k	6k
	Combine	18.0	18.1	2.5MB	0.59	0.36	14k	3k	129
	Translation	25.3	386.9	29.1MB	2.34	26.65	303k	248k	8k

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, Int: interrupts. "Images: Recognize" reports the maximum of all four image recognition enclaves.

6.1.2 Confinement Overhead. In Figure 7, 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%.

6.1.3 Checkpoint Restore Overhead. The CPR Size column in Table 3 shows the amount of memory copied/zeroed on checkpoint restore. Figure 7 (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 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).

6.1.4 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 flush the TLB on all enclave exits (we could not directly measure execution on our hardware, because it lacks SGX version 2 features (Section 4.1)). Besides explicit `EEXIT` instructions, we also model enclave exits due to events like exceptions and interrupts (Table 3). We measure a hardware delay of 3.9μs for each `EENTER`/`EEXIT` pair, and 3.14μs for each `ERESUME`/Async-Exit pair.

We also measured SGX instruction costs on the more recent and powerful Intel Xeon E3-1270 v6 3.80GHz processor. On the Xeon processor, `EENTER`/`EEXIT` pairs cost 2.34μs, and `ERESUME`/

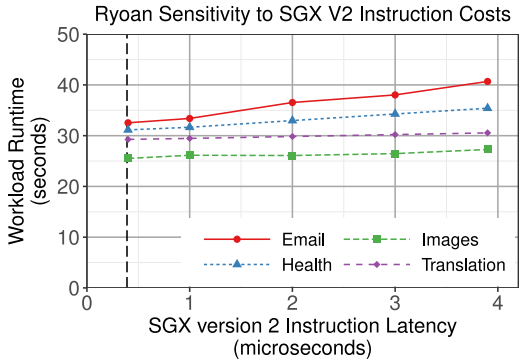

Fig. 8. Ryoan application workloads' sensitivity to emulated instruction cost. The dashed vertical line denotes the delay (0.39μs) used to compute the Ryoan bars in Figure 7.

Async-Exit pairs cost 1.85μs. This processor's clock rate is about 65% faster than the laptop, and the SGX costs have been reduced by about that factor.

Version 2 instructions EACCEPT, EMODPE, EMODPR are simpler than EENTER and EEXIT, so we model their cost at one-10th of one EENTER/EEXIT pair. Figure 8 explores the effect of varying this cost on the runtime of our workloads. If the version 2 instructions turn out to be as costly as an EENTER/EEXIT pair (3.9μs), for instance, then 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 additional synchronization (via extended behaviors of ETRACK) when modifying enclave page state [66]. We believe the performance effect on these workloads will be negligible, given that our applications only have one thread per enclave. SGX execution also requires syscall marshaling to copy system call arguments and results to and from untrusted memory, but we measure the overhead of marshalling as negligible. All results are shown in Figure 7.

6.1.5 Checkpoint Restore vs. Initialization. Creating an enclave and loading a module takes less than 0.5s for all our cases, but Table 3 shows application-level initialization times are over 20s 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 30ms. In comparison, Ryoan's checkpoint-based reset is much more efficient, and the per-unit cost is under 10ms.

6.2 SGX Encryption Overheads

Enclave memory is encrypted whenever it leaves the processor. This means additional operations are required when the processor reads memory from RAM: Encrypt on write, decrypt on read. These additional operations add latency to last level cache (LLC) misses. Encryption related performance penalties are absent from our performance model; here we explore their cost.

6.2.1 LLC Miss Microbenchmark. To measure the memory controller overheads of SGX, we use a microbenchmark that executes a fixed number of instructions per cache miss (read or write). We execute the same microbenchmark as part of a normal process and compare with execution in an enclave protected by SGX. The slowdown incurred by running the microbenchmark in an SGX enclave for a varying number of retired instructions (and for read or write LLC misses) is

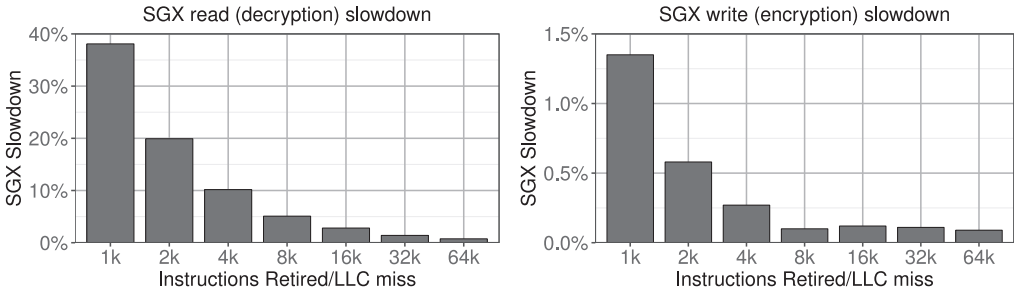

Fig. 9. Slowdown observed running our cache-miss microbenchmark inside an SGX enclave versus running the same code without SGX.

Table 4. Instructions per LLC Miss on Ryoan Benchmarks

Benchmark	Module	Instructions Retired/LLC miss Read Miss	Instructions Retired/LLC miss Write Miss	Memory controller SGX slowdown
Email	ClamAV	1,260	5,090	32.0%
Health	Classifier	14,310	24,650	3.2%
Images	Recognize	32,760	9,000	1.4%
Translation	(one module)	12,560	34,510	3.5%

Memory controller SGX slowdown is the slowdown measured for microbenchmarks of equivalent miss patterns on SGX hardware.

shown in Figure 9. When computation does not access memory (and we have a large number of instructions per cache miss), the performance of enclave code is very similar to unshielded execution. The microbenchmark makes no system calls, and page faults are eliminated by ensuring all enclave memory is touched before measurement begins. Therefore the only enclave exits are due to interrupts and their effect on the total time is insignificant.

Write misses are cheap, because the processor does not wait for the memory controller to encrypt data. A last-level write cache miss every 1,000 instructions incurs about 1.4% execution time overhead. Read misses can cause significant delays and programs that have high read miss rates will run slowly within an enclave. A read miss every 1,000 instructions causes a 38.1% performance overhead, which falls to 10.1% once the read misses happen every 4,000 instructions. The processor often needs the data from a read before it can do any useful work and therefore will stall waiting for the data to be decrypted.

To understand the slowdown that would be incurred for Ryoan's benchmarks due to SGX encryption overheads, we measure the number of instructions retired per LLC miss, reported in Table 4. We focus on the enclaves that dominate the performance of the benchmarks.

The "Memory controller SGX slowdown" column reports our projected enclave slowdown based on the workloads LLC rate and measurements of our LLC miss microbenchmark. Email shows the largest effect at a projected 32% slowdown, much higher than the other benchmarks. The other applications execute large numbers of instructions for every last level miss, putting our estimate of SGX encryption overheads at less than 5%.

7 RELATED WORK

Ryoan is most closely related to shielding computation on secret data from an untrusted platform (Section 7.1), preventing disclosure to unauthorized parties (Section 7.2), and controlling leaks through timing and termination channels (Section 7.3).

7.1 Shielding Systems.

Shielding systems are designed to protect secret data while it is being processed in an untrusted environment. Ryoan defends against the untrusted environment but also confines the application so that it need not be trusted to maintain data secrecy.

7.1.1 Software Shielding. Software shielding uses a hypervisor or compiler to preserve the privacy and integrity of computations executing on an untrusted platform. Overshadow [17], InkTag [37], and Sego [50] use a trusted hypervisor to protect trusted applications from an untrusted operating system. InkTag and Sego allow a trusted application to verify untrusted operating system services (e.g., a file system) with help from the hypervisor. Virtual Ghost [23] uses a trusted compiler rather than a hypervisor for protection.

7.1.2 Hardware Shielding. Hardware shielding uses hardware primitives (such as SGX) to protect computations from platform software. Haven [7], Scone [5], and Graphene-SGX [85] allow a trusted program and its library operating system to execute in an SGX enclave that protects them from attack by host software. VC3 [77] secures trusted MapReduce using SGX. Ryoan defends against platform software environment, but also confines the application so that it need not be trusted to maintain data secrecy.

ARM TrustZone [56] is another commercially available hardware primitive that protects computations from platform software. TrustZone provides a single "secure world" that allows code to execute in multiple privilege levels; in contrast, SGX provides an unlimited number of enclaves, all of which execute at the user level. TrustZone does not currently encrypt memory, so it is less resistant to physical attacks, but TrustZone can deliver page faults to privileged code in the secure world, eliminating controlled channel attacks [89]. Komodo [27] uses formally verified software to provide an enclave abstraction on top of TrustZone. To replace SGX with TrustZone, Ryoan would require a management layer like Komodo.

Attempts to use late launch and TPMs for user assurance (e.g., Flicker [64]) suffer from poor usability due to the restricted execution environment required by the TPM. Late launch code has no access to the operating system and must manage the bare machine. Code executing in an enclave can be more complex than what is practical to execute in late launch.

Ironclad [36] addresses the limitations of the late launch environment with a (small) verified system stack that must be included with each trusted binary. Ironclad is not backward compatible and requires users to write verified code, placing a burden on the programmer.

MiniBox [54] uses a TPM and Native Client to protect an application and the OS from each other. Unlike Ryoan, MiniBox uses Native Client strictly to protect the OS and their secure hypervisor, not to prevent applications from leaking sensitive data.

For all TPM-based systems, a computation's data is visible on the memory bus, where an unscrupulous administrator of the host platform can steal it. SGX enclave data is encrypted before it travels across the memory bus, preserving an enclave's secrecy.

7.1.3 Cryptographic Shielding. Homomorphic encryption [11, 31] allows untrusted code to compute directly on encrypted data with strong security guarantees. Unfortunately, practical implementations of general-purpose homomorphic encryption are not available, and current overheads are prohibitive.

Property-preserving encryption (e.g., order-preserving encryption [8]) can protect the secrecy of some computations [69], and some systems use these primitives [9, 72, 79]. However, these systems have weaker security guarantees [35], apply to limited scenarios, or have a significant performance overhead. In comparison, Ryoan's confinement does not require domain-specific knowledge about the applications.

7.2 Decentralized Information Flow Control

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 [4, 49, 61, 73, 86, 93]; similar enforcements have also been realized in a browser (COWL [81]) and a mobile device (Maxoid [90]). Two exceptions are DStar [94] and Fabric [61], which do not have a centralized reference monitor. However, although a DStar or Fabric user does not need to trust all machines involved in the system, he or she must trust the machine on which he or she processes his or her data, which means a correct reference monitor (the OS or runtime that supports DIFC) must be 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 [53, 83, 84, 98] form a basis for strong information flow guarantees, and close timing and cache channels ignored by Ryoan. However, such hardware is not available and as designed does not include the privacy and integrity guarantees provided by SGX.

7.3 Timing and Termination

Timing and termination channels are studied in previous work [29, 46] 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.

8 CONCLUSION

Ryoan allows users to process data with software they do not trust, executing on a platform they do not control safely, thereby benefiting users, data processing services, and computational platforms. Ryoan does this by confining untrusted application code via a trusted sandbox (provided by Google's NaCl) that is itself made tamperproof via hardware enclave-protected execution (provided by Intel's SGX). Ryoan also defines and enforces an execution model that allows mutually distrustful software nodes to exchange data without disclosing secrets to each other or the platform provider. We implement and evaluate a Ryoan prototype over various case studies of real-world applications. Our evaluation, based on real SGX hardware and simulation, shows that Ryoan overhead is workload dependent, 27% in the best case and up to 419% in the worst case.

ACKNOWLEDGMENTS

We thank Mark Silberstein, Christopher J. Rossbach, Bennet Yee, and Petros Maniatis, the anonymous reviewers and Jeff Chase for comments on early revisions of this work. We also thank Shane Williams for background on mail servers and Robert McInvale for his work exploring SGX encryption overheads.

REFERENCES

[1] 23andMe. 2016. 23andMe Compares Family History and Genetic Tests for Predicting Complex Disease Risk. Retrieved September 2016 from http://mediacenter.23andme.com/blog/23andme-compares-family-history-and-genetic-tests-for-predicting-complex-disease-risk/.

[2] ACL. 2013. Shared Task: Machine Translation. Retrieved August 23, 2018 from http://www.statmt.org/wmt13/translation-task.html.

[3] Amazon. 2016. Amazon Machine Learning. Retrieved August 23, 2018 from https://aws.amazon.com/machine-learning/.

[4] Owen Arden, Michael D. George, Jed Liu, K. Vikram, Aslan Askarov, and Andrew C. Myers. 2012. Sharing mobile code securely with information flow control. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy (SP'12)*. IEEE Computer Society, Washington, DC, 191–205. DOI: https://doi.org/10.1109/SP.2012.22

[5] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, David Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. 2016. SCONE: Secure linux containers with intel SGX. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, Berkeley, CA, 689–703. http://dl.acm.org/citation.cfm?id=3026877.3026930

[6] Aslan Askarov, Danfeng Zhang, and Andrew C. Myers. 2010. Predictive black-box mitigation of timing channels. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS'10)*. ACM, New York, NY, 297–307. DOI: https://doi.org/10.1145/1866307.1866341

[7] Andrew Baumann, Marcus Peinado, and Galen Hunt. 2015. Shielding applications from an untrusted cloud with haven. *ACM Trans. Comput. Syst.* 33, 3, Article 8 (Aug. 2015), 26 pages. DOI: https://doi.org/10.1145/2799647

[8] Alexandra Boldyreva, Nathan Chenette, Younho Lee, and Adam O'Neill. 2009. Order-preserving symmetric encryption. In *Advances in Cryptology: EUROCRYPT 2009*, Antoine Joux (Ed.). Springer, Berlin, 224–241. DOI: https://doi.org/10.1007/978-3-642-22792-9_29

[9] Raphael Bost, Raluca Ada Popa, Stephen Tu, and Shafi Goldwasser. 2015. Machine learning classification over encrypted data. In *Proceedings of the 2015 Network and Distributed System Security Symposium (NDSS'15)*. Internet Society.

[10] Lèon Bottou. 2016. Stochastic Gradient SVM. Retrieved August 23, 2018 from http://leon.bottou.org/projects/sgd#stochastic_gradient_svm.

[11] Zvika Brakerski and Vinod Vaikuntanathan. 2011. Fully homomorphic encryption from ring-LWE and security for key dependent messages. In *Advances in Cryptology: CRYPTO 2011*, Phillip Rogaway (Ed.). Springer, Berlin, 505–524. DOI: https://doi.org/10.1007/978-3-642-01001-9_13

[12] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. 2008. When good instructions go bad: Generalizing return-oriented programming to RISC. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS'08)*. ACM, New York, NY, 27–38. DOI: https://doi.org/10.1145/1455770.1455776

[13] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI'08)*. USENIX Association, Berkeley, CA, 209–224.

[14] Stephen Checkoway and Hovav Shacham. 2013. Iago attacks: Why the system call API is a bad untrusted RPC interface. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'13)*. ACM, New York, NY, 253–264. DOI: https://doi.org/10.1145/2451116.2451145

[15] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H. Lai. 2018. SgxPectre attacks: Leaking enclave secrets via speculative execution. arxiv:1802.09085. Retrieved from https://arxiv.org/abs/1802.09085

[16] Sanchuan Chen, Xiaokuan Zhang, Michael K. Reiter, and Yinqian Zhang. 2017. Detecting privileged side-channel attacks in shielded execution with DéJà Vu. In *Proceedings of the 2017 ACM Asia Conference on Computer and Communications Security (ASIA CCS'17)*. ACM, New York, NY, 7–18. DOI: https://doi.org/10.1145/3052973.3053007

[17] Xiaoxin Chen, Tal Garfinkel, E. Christopher Lewis, Pratap Subrahmanyam, Carl A. Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan R.K. Ports. 2008. Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'08)*. ACM, New York, NY, 2–13. DOI: https://doi.org/10.1145/1346281.1346284

[18] Jim Chow, Ben Pfaff, Tal Garfinkel, Kevin Christopher, and Mendel Rosenblum. 2004. Understanding data lifetime via whole system simulation. In *Proceedings of the 13th USENIX Security Symposium (USENIX Security'04)*. USENIX Association, Berkeley, CA, 321–336.

[19] Clarifai. 2016. Clarifai. Retrieved August 23, 2018 from https://www.clarifai.com

[20] Bart Coppens, Ingrid Verbauwhede, Koen De Bosschere, and Bjorn De Sutter. 2009. Practical mitigations for timing-based side-channel attacks on modern x86 processors. In *Proceedings of the 2009 IEEE Symposium on Security and Privacy (SP'09)*. IEEE Computer Society, Washington, DC, 45–60. DOI: https://doi.org/10.1109/SP.2009.19

[21] Victor Costan and Srinivas Devadas. 2016. *Intel SGX Explained*. https://eprint.iacr.org/2016/086.pdf

[22] Victor Costan, Ilia Lebedev, and Srinivas Devadas. 2016. Sanctum: Minimal hardware extensions for strong software isolation. In *Proceedings of the 25th USENIX Security Symposium (USENIX Security'16)*. USENIX Association, Berkeley, CA, 857–874.

[23] John Criswell, Nathan Dautenhahn, and Vikram Adve. 2014. Virtual ghost: Protecting applications from hostile operating systems. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'14)*. ACM, New York, NY, 81–96. DOI: https://doi.org/10.1145/2541940.2541986

[24] CSMINING Group. 2016. CSMINING Group: Spam Email Datasets. Retrieved April 2016 from https://csmining.org/index.php/spam-email-datasets-.html.
[25] Solar Designer. 1997. "return-to-libc" attack. *Bugtraq*.
[26] Roger Dingledine, Nick Mathewson, and Paul Syverson. 2004. Tor: The second-generation onion router. In *Proceedings of the 13th USENIX Security Symposium (USENIX Security'04)*. USENIX Association, Berkeley, CA, 21–21.
[27] Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. 2017. Komodo: Using verification to disentangle secure-enclave hardware from software. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP'17)*. ACM, New York, NY, 287–305. DOI: https://doi.org/10.1145/3132747.3132782
[28] Andrew Ferraiuolo, Yao Wang, Danfeng Zhang, Andrew C. Myers, and G. Edward Suh. 2016. Lattice priority scheduling: Low-overhead timing-channel protection for a shared memory controller. In *Proceedings of the 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA'16)*. 382–393. DOI: https://doi.org/10.1109/HPCA.2016.7446080
[29] Bryan Ford. 2012. Plugging side-channel leaks with timing information flow control. In *Proceedings of the 4th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud'12)*. USENIX Association, Berkeley, CA, 24–24.
[30] Hubertus Franke, Rusty Russell, and Matthew Kirkwood. 2002. Fuss, futexes and furwocks: Fast userlevel locking in linux. In *Proceedings of the 2002 Ottawa Linux Symposium*. 479–495.
[31] Craig Gentry. 2009. *A Fully Homomorphic Encryption Scheme*. Ph.D. Dissertation. Stanford University, Stanford, CA.
[32] Google. 2016. Implementation and safety of NaCl SFI for x86-64. Retrieved September 2016 from https://groups.google.com/forum/#!topic/native-client-discuss/C-wXFdR2lf8.
[33] GRIB/IMIM/UPF Integrative Biomedical Informatics Group. 2016. The DisGeNET Database. Retrieved February, 2016 from http://www.disgenet.org/ds/DisGeNET/files/current/DisGeNET_2016.db.gz.
[34] The Radicati Group. 2009. The Radicati Group, Inc: Email Statistics Report 2009–2013 (summary). Retrieved August 23, 2018 from http://www.radicati.com/wp/wp-content/uploads/2009/05/email-stats-report-exec-summary.pdf.
[35] Paul Grubbs, Thomas Ristenpart, and Vitaly Shmatikov. 2017. Why your encrypted database is not secure. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems (HotOS'17)*. ACM, New York, NY, 162–168. DOI: https://doi.org/10.1145/3102980.3103007
[36] Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Arjun Narayan, Bryan Parno, Danfeng Zhang, and Brian Zill. 2014. Ironclad apps: End-to-end security via automated full-system verification. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*. USENIX Association, Berkeley, CA, 165–181.
[37] Owen S. Hofmann, Sangman Kim, Alan M. Dunn, Michael Z. Lee, and Emmett Witchel. 2013. InkTag: Secure applications on an untrusted operating system. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'13)*. ACM, New York, NY, 265–278. DOI: https://doi.org/10.1145/2451116.2451146
[38] Tyler Hunt, Zhiting Zhu, Yuanzhong Xu, Simon Peter, and Emmett Witchel. 2016. Ryoan: A distributed sandbox for untrusted computation on secret data. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, Berkeley, CA, 533–549.
[39] IBM. 2016. IBM Visual Recognition service. Retrieved August 23, 2018 from http://www.ibm.com/smarterplanet/us/en/ibmwatson/developercloud/visual-recognition.html.
[40] Intel. 2014. Intel Software Guard Extensions Programming Reference. Retrieved August, 23 2018 from https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf.
[41] Intel. 2015. Intel Software Guard Extensions Evaluation SDK Users Guide: Diffie-Hellman Key Exchange. Retrieved August 23, 2018 from https://software.intel.com/sites/products/sgx-sdk-users-guide-windows/Default.htm.
[42] Intel. 2016. Intel Software Guard Extensions for Linux OS: linux-sgx. https://github.com/01org/linux-sgx. (commit:d686fb0).
[43] Intel. 2016. Intel Software Guard Extensions for Linux OS: linux-sgx-driver. https://github.com/01org/linux-sgx-driver. (commit:0fb8995).
[44] Prerit Jain, Soham Desai, Seongmin Kim, Ming-Wei Shih, JaeHyuk Lee, Changho Choi, Youjung Shin, Taesoo Kim, Brent Byunghoon Kang, and Dongsu Han. 2016. OpenSGX: An open platform for SGX research. In *Proceedings of the 2016 Network and Distributed System Security Symposium (NDSS'16)*. Internet Society.
[45] Min Gyung Kang, Stephen McCamant, Pongsin Poosankam, and Dawn Song. 2011. DTA++: Dynamic taint analysis with targeted control-flow propagation. In *Proceedings of the 2011 Network and Distributed System Security Symposium (NDSS'11)*. Internet Society.
[46] Vineeth Kashyap, Ben Wiedermann, and Ben Hardekopf. 2011. Timing- and termination-sensitive secure information flow: Exploring a new approach. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy (SP'11)*. IEEE Computer Society, Washington, DC, 413–428. DOI: https://doi.org/10.1109/SP.2011.19
[47] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. 2012. STEALTHMEM: System-level protection against cache-based side channel attacks in the cloud. In *Proceedings of the 21st USENIX Security Symposium (USENIX Security'12)*. USENIX Association, Berkeley, CA.

[48] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2018. Spectre attacks: Exploiting speculative execution. arxiv:1801.01203. Retrieved from https://arxiv.org/abs/1801.01203

[49] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. 2007. Information flow control for standard OS abstractions. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles (SOSP'07)*. ACM, New York, NY, 321–334. DOI: https://doi.org/10.1145/1294261.1294293

[50] Youngjin Kwon, Alan M. Dunn, Michael Z. Lee, Owen S. Hofmann, Yuanzhong Xu, and Emmett Witchel. 2016. Sego: Pervasive trusted metadata for efficiently verified untrusted system services. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'16)*. ACM, New York, NY, 277–290. DOI: https://doi.org/10.1145/2872362.2872372

[51] Butler W. Lampson. 1973. A note on the confinement problem. *Commun. ACM* 16, 10 (Oct. 1973), 613–615. DOI: https://doi.org/10.1145/362375.362389

[52] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. 2017. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *Proceedings of the 26th USENIX Security Symposium (USENIX Security'17)*. USENIX Association, Berkeley, CA, 557–574.

[53] Xun Li, Vineeth Kashyap, Jason K. Oberg, Mohit Tiwari, Vasanth Ram Rajarathinam, Ryan Kastner, Timothy Sherwood, Ben Hardekopf, and Frederic T. Chong. 2014. Sapper: A language for hardware-level security policy enforcement. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'14)*. ACM, New York, NY, 97–112. DOI: https://doi.org/10.1145/2541940.2541947

[54] Yanlin Li, Jonathan McCune, James Newsome, Adrian Perrig, Brandon Baker, and Will Drewry. 2014. MiniBox: A two-way sandbox for x86 native code. In *Proceedings of the 2014 USENIX Annual Technical Conference (USENIX ATC'14)*. USENIX Association, Berkeley, CA, 409–420. http://dl.acm.org/citation.cfm?id=2643634.2643676

[55] libsodium. 2016. libsodium: A modern and easy-to-use crypto library. Retrieved September 2016 from https://github.com/jedisct1/libsodium.

[56] ARM Limited. 2009. Building a Secure System using TrustZone Technology. Ref. PRD29-GENC-009492C.

[57] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Dkaniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown. arxiv:1801.01207. Retrieved from https://arxiv.org/abs/1801.01207

[58] Anyi Liu, Jim Chen, and Harry Wechsler. 2013. Real-time covert timing channel detection in networked virtual environments. In *Advances in Digital Forensics IX*, Gilbert Peterson and Sujeet Shenoi (Eds.). Springer, Berlin, 273–288. DOI: https://doi.org/10.1007/978-3-642-41148-9_19

[59] Chang Liu, Austin Harris, Martin Maas, Michael Hicks, Mohit Tiwari, and Elaine Shi. 2015. GhostRider: A hardware-software system for memory trace oblivious computation. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'15)*. ACM, New York, NY, 87–101. DOI: https://doi.org/10.1145/2694344.2694385

[60] Fangfei Liu, Hao Wu, and Ruby B. Lee. 2015. Can randomized mapping secure instruction caches from side-channel attacks?. In *Proceedings of the 4th Workshop on Hardware and Architectural Support for Security and Privacy (HASP'15)*. ACM, New York, NY, Article 4, 8 pages. DOI: https://doi.org/10.1145/2768566.2768570

[61] Jed Liu, Michael D. George, K. Vikram, Xin Qi, Lucas Waye, and Andrew C. Myers. 2009. Fabric: A platform for secure distributed computation and storage. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP'09)*. ACM, New York, NY, 321–334. DOI: https://doi.org/10.1145/1629575.1629606

[62] Jay Mahadeokar and Gerry Pesavento. 2016. Open Sourcing a Deep Learning Solution for Detecting NSFW Images. Retrieved August 24, 2018 from https://yahooeng.tumblr.com/post/151148689421/open-sourcing-a-deep-learning-solution-for.

[63] Sinisa Matetic, Mansoor Ahmed, Kari Kostiainen, Aritra Dhar, David Sommer, Arthur Gervais, Ari Juels, and Srdjan Capkun. 2017. ROTE: Rollback protection for trusted execution. In *Proceedings of the 26th USENIX Security Symposium (USENIX Security'17)*. USENIX Association, Berkeley, CA, 1289–1306.

[64] Jonathan M. McCune, Bryan J. Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. 2008. Flicker: An execution infrastructure for Tcb minimization. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008 (Eurosys'08)*. ACM, New York, NY, 315–328. DOI: https://doi.org/10.1145/1352592.1352625

[65] David A. McGrew and John Viega. 2005. The Galois/Counter Mode of Operation (GCM). Retrieved September 2016 from http://csrc.nist.gov/groups/ST/toolkit/BCM/documents/proposedmodes/gcm/gcm-revised-spec.pdf.

[66] Frank McKeen, Ilya Alexandrovich, Ittai Anati, Dror Caspi, Simon Johnson, Rebekah Leslie-Hurd, and Carlos Rozas. 2016. Intel software guard extensions (Intel SGX) support for dynamic memory management inside an enclave. In *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016 (HASP'16)*. ACM, New York, NY, Article 10, 9 pages. DOI: https://doi.org/10.1145/2948618.2954331

[67] Moses. 2016. Moses Statistical Machine Learning Translation System. Retrieved August 23, 2018 from http://www.statmt.org/moses/.

[68] Andrew C. Myers and Barbara Liskov. 1997. A decentralized model for information flow control. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP'97)*. ACM, New York, NY, 129–142. DOI: https://doi.org/10.1145/268998.266669

[69] Muhammad Naveed, Seny Kamara, and Charles V. Wright. 2015. Inference attacks on property-preserving encrypted databases. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security (CCS'15)*. ACM, New York, NY, 644–655. DOI: https://doi.org/10.1145/2810103.2813651

[70] Olga Ohrimenko, Felix Schuster, Cedric Fournet, Aastha Mehta, Sebastian Nowozin, Kapil Vaswani, and Manuel Costa. 2016. Oblivious multi-party machine learning on trusted processors. In *Proceedings of the 25th USENIX Security Symposium (USENIX Security'16)*. USENIX Association, Berkeley, CA, 619–636.

[71] Dan O'Keeffe, Divya Muthukumaran, Pierre-Louis Aublin, Florian Kelbert, Christian Priebe, Josh Lind, Huanzhou Zhu, and Peter Pietzuch. 2018. spectre-attack-sgx. Retrieved from https://github.com/lsds/spectre-attack-sgx.

[72] Raluca Ada Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. 2011. CryptDB: Protecting confidentiality with encrypted query processing. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP'11)*. ACM, New York, NY, 85–100. DOI: https://doi.org/10.1145/2043556.2043566

[73] Donald E. Porter, Michael D. Bond, Indrajit Roy, Kathryn S. Mckinley, and Emmett Witchel. 2014. Practical fine-grained information flow control using laminar. *ACM Trans. Program. Lang. Syst.* 37, 1, Article 4 (Nov. 2014), 51 pages. DOI: https://doi.org/10.1145/2638548

[74] QEMU. 2016. QEMU: Open source processor emulator. Retrieved August 23, 2018 from http://wiki.qemu.org/Main_Page.

[75] Ashay Rane, Calvin Lin, and Mohit Tiwari. 2015. Raccoon: Closing digital side-channels through obfuscated execution. In *Proceedings of the 24th USENIX Security Symposium (USENIX Security'15)*. USENIX Association, Berkeley, CA, 431–446. http://dl.acm.org/citation.cfm?id=2831143.2831171

[76] Ling Ren, Christopher Fletcher, Albert Kwon, Emil Stefanov, Elaine Shi, Marten Van Dijk, and Srinivas Devadas. 2015. Constants count: Practical improvements to oblivious RAM. In *Proceedings of the 24th USENIX Security Symposium (USENIX Security'15)*. USENIX Association, Berkeley, CA, 415–430.

[77] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. 2015. VC3: Trustworthy data analytics in the cloud using SGX. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy (SP'15)*. IEEE Computer Society, Washington, DC, 38–54. DOI: https://doi.org/10.1109/SP.2015.10

[78] David Sehr, Robert Muth, Cliff Biffle, Victor Khimenko, Egor Pasko, Karl Schimpf, Bennet Yee, and Brad Chen. 2010. Adapting software fault isolation to contemporary CPU architectures. In *Proceedings of the 19th USENIX Security Symposium (USENIX Security'10)*. USENIX Association, Berkeley, CA, 1–1.

[79] Justine Sherry, Chang Lan, Raluca Ada Popa, and Sylvia Ratnasamy. 2015. BlindBox: Deep packet inspection over encrypted traffic. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication (SIGCOMM'15)*. ACM, New York, NY, 213–226. DOI: https://doi.org/10.1145/2785956.2787502

[80] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. 2017. T-SGX: Eradicating controlled-channel attacks against enclave programs. In *Proceedings of the 2017 Network and Distributed System Security Symposium (NDSS'17)*. Internet Society.

[81] Deian Stefan, Edward Z. Yang, Petr Marchenko, Alejandro Russo, Dave Herman, Brad Karp, and David Mazières. 2014. Protecting users by confining javascript with COWL. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*. USENIX Association, Berkeley, CA, 131–146.

[82] Raoul Strackx and Frank Piessens. 2016. Ariadne: A minimal approach to state continuity. In *Proceedings of the 25th USENIX Security Symposium (USENIX Security'16)*. USENIX Association, Berkeley, CA, 875–892.

[83] Mohit Tiwari, Xun Li, Hassan M. G. Wassel, Frederic T. Chong, and Timothy Sherwood. 2009. Execution leases: A hardware-supported mechanism for enforcing strong non-interference. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 42)*. ACM, New York, NY, 493–504. DOI: https://doi.org/10.1145/1669112.1669174

[84] Mohit Tiwari, Jason K. Oberg, Xun Li, Jonathan Valamehr, Timothy Levin, Ben Hardekopf, Ryan Kastner, Frederic T. Chong, and Timothy Sherwood. 2011. Crafting a usable microkernel, processor, and I/O system with strict and provable information flow security. In *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA'11)*. ACM, New York, NY, 189–200. DOI: https://doi.org/10.1145/2000064.2000087

[85] Chia-Che Tsai, Donald E. Porter, and Mona Vij. 2017. Graphene-SGX: A practical library OS for unmodified applications on SGX. In *Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC'17)*. USENIX Association, Berkeley, CA, 645–658.

[86] Steve Vandebogart, Petros Efstathopoulos, Eddie Kohler, Maxwell Krohn, Cliff Frey, David Ziegler, Frans Kaashoek, Robert Morris, and David Mazières. 2007. Labels and event processes in the asbestos operating system. *ACM Trans. Comput. Syst.* 25, 4, Article 11 (Dec. 2007). DOI: https://doi.org/10.1145/1314299.1314302

[87] Zhenyu Wu, Zhang Xu, and Haining Wang. 2015. Whispers in the hyper-space: High-bandwidth and reliable covert channel attacks inside the cloud. *IEEE/ACM Trans. Netw.* 23, 2 (Apr. 2015), 603–614. DOI: https://doi.org/10.1109/TNET.2014.2304439

[88] Yunjing Xu, Michael Bailey, Farnam Jahanian, Kaustubh Joshi, Matti Hiltunen, and Richard Schlichting. 2011. An exploration of L2 cache covert channels in virtualized environments. In *Proceedings of the 3rd ACM Workshop on Cloud Computing Security Workshop (CCSW'11)*. ACM, New York, NY, 29–40. DOI: https://doi.org/10.1145/2046660.2046670

[89] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. 2015. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy (SP'15)*. IEEE Computer Society, Washington, DC, 640–656. DOI: https://doi.org/10.1109/SP.2015.45

[90] Yuanzhong Xu and Emmett Witchel. 2015. Maxoid: Transparently confining mobile applications with custom views of state. In *Proceedings of the Tenth European Conference on Computer Systems (EuroSys'15)*. ACM, New York, NY, Article 26, 16 pages. DOI: https://doi.org/10.1145/2741948.2741966

[91] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. 2009. Native client: A sandbox for portable, untrusted x86 native code. In *Proceedings of the 2009 IEEE Symposium on Security and Privacy (SP'09)*. IEEE Computer Society, Washington, DC, 79–93. DOI: https://doi.org/10.1109/SP.2009.25

[92] Tatu Ylonen and Chris Lonvick. 2008. RFC 5246: The Transport Layer Security (TLS) Protocol: Version 1.2. Retrieved August 23, 2018 from https://tools.ietf.org/html/rfc5246.

[93] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. 2006. Making information flow explicit in histar. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI'06)*. USENIX Association, Berkeley, CA, 263–278.

[94] Nickolai Zeldovich, Silas Boyd-Wickizer, and David Mazières. 2008. Securing distributed systems with information flow control. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI'08)*. USENIX Association, Berkeley, CA, 293–308.

[95] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. 2013. Practical control flow integrity and randomization for binary executables. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy (SP'13)*. IEEE Computer Society, Washington, DC, 559–573. DOI: https://doi.org/10.1109/SP.2013.44

[96] Danfeng Zhang, Aslan Askarov, and Andrew C. Myers. 2011. Predictive mitigation of timing channels in interactive systems. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS'11)*. ACM, New York, NY, 563–574. DOI: https://doi.org/10.1145/2046707.2046772

[97] Danfeng Zhang, Aslan Askarov, and Andrew C. Myers. 2012. Language-based control and mitigation of timing channels. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'12)*. ACM, New York, NY, 99–110. DOI: https://doi.org/10.1145/2254064.2254078

[98] Danfeng Zhang, Yao Wang, G. Edward Suh, and Andrew C. Myers. 2015. A hardware design language for timing-sensitive information-flow security. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'15)*. ACM, New York, NY, 503–516. DOI: https://doi.org/10.1145/2694344.2694372

[99] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. 2012. Cross-VM side channels and their use to extract private keys. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS'12)*. ACM, New York, NY, 305–316. DOI: https://doi.org/10.1145/2382196.2382230

Received February 2018; accepted June 2018