Ryoan: A Distributed Sandbox for Untrusted Computation on Secret Data

TYLER HUNT, ZHITING ZHU, YUANZHONG XU, SIMON PETER, EMMETT WITCHEL

Tyler Hunt is a PhD student at the University of Texas at Austin, working with Emmett Witchel. His research interests involve designing and building systems with interesting security properties. thunt@cs.utexas.edu

Zhiting Zhu has been a PhD student at the University of Texas at Austin since 2014, where he works with Emmett Witchel. He is interested in operating systems. zhitingz@cs.utexas.edu

Yuanzhong Xu received his PhD in computer science from the University of Texas at Austin in 2016. He is generally interested in systems and security. He currently works for Facebook as a research scientist. yxu@utexas.edu

Simon Peter is an Assistant Professor at the University of Texas at Austin, where he conducts research in operating systems and networks. He received a PhD in computer science from ETH Zurich in 2012 and an MSc in computer science from the Carl von Ossietzky University of Oldenburg, Germany, in 2006. Before joining UT Austin in 2016, he was a Research Associate at the University of Washington from 2012 to 2016. simon@cs.utexas.edu

Emmett Witchel is a Professor in Computer Science at the University of Texas at Austin. He received his doctorate from MIT in 2004. He and his group are interested in operating systems, security, and concurrency. witchel@cs.utexas.edu

Ryoan provides a distributed sandbox, leveraging hardware enclaves (e.g., Intel's software guard extensions (SGX)) 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.

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

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

We propose Ryoan [1], a distributed sandbox that forces data-processing services to keep user data secret, without trusting the service's software stack, developers, or administrators. Ryoan's name is inspired by a famous dry landscape Zen garden that stimulates contemplation (Ryōan-ji). First, Ryoan provides a sandbox to confine individual data-processing modules and prevent them from leaking data; second, it uses trusted hardware to allow a remote user to verify the integrity of individual sandbox instances and protect their execution; third, the sandbox can be configured to allow confined code modules to communicate in controlled ways, enabling flexible delegation among mutually distrustful parties. Ryoan gives a user confidence that a service has protected her secrets.

A key enabling technology for Ryoan is hardware enclave-protected execution (e.g., Intel's Software Guard Extensions (SGX) [2]), a new hardware primitive that uses trusted hardware to protect a user-level computation from potentially malicious privileged software. The processor hardware keeps unencrypted data on chip but encrypts data when it moves into RAM. The hypervisor and operating system retain their ability to manage memory (e.g., move memory pages onto secondary storage), but privileged software sees only an encrypted version of the data that is protected from tampering by a cryptographic hash. Haven [3] and SCONE [4] are examples of systems that use enclaves to protect a user's computation from potentially malicious system software, including a library operating system to increase backward compatibility.

Ryoan faces issues beyond those faced by enclave-protected computation systems such as Haven. Enclaves are intended to protect an application that is trusted by the user, which does not collude with the infrastructure, though it may unintentionally leak data via side channels. In Ryoan's model the application and the infrastructure are under the control of an adversary and may collude to steal the user's secrets. Even if the application itself is isolated from the infrastructure using enclave protection, the adversary could exercise its control to construct covert channels between the application and the platform. Ryoan's goal is to prevent such covert channels and stop an untrusted application from intentionally and covertly using users' data to modulate events like system call arguments or I/O traffic statistics, which are visible to the infrastructure.

An untrusted application in Ryoan is confined by a trusted sandbox. For the Ryoan prototype we use Native Client (NaCl) [5, 6], which is a state-of-the-art user-level sandbox. NaCl 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 NaCl sandbox instance that loads and executes untrusted modules. The NaCl instances communicate with each other to form a distributed sandbox that enforces strong privacy guarantees for all participating parties—the users and different service providers. Confining untrusted code [7] is a longstanding problem that remains technically challenging, but Ryoan benefits from hardware-supported enclave protection. Ryoan also assumes a request-oriented data model, where confined modules only process input once and cannot read or write persistent storage after they receive their input. This model makes Ryoan applicable only to request-oriented server applications—but such servers are the most common way to bring scalable services to large numbers of users.

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

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

Background and Threat Model

Ryoan assumes a processor with hardware-protected enclaves, e.g., Intel's SGX-enabled Skylake (or later) architecture. The address space of a protected enclave has its privacy and integrity guaranteed by hardware. Hardware encrypts and hashes memory contents when it moves off chip, protecting the contents from other users and also from the platform's privileged software (operating system and hypervisor). Code within an enclave can manipulate user secrets without fear of divulging them to the underlying execution platform. Code within an enclave cannot have its code or control manipulated by the platform's privileged software.

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

Threat Model

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

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

A user of a service does not trust the software at any privilege level in the computational platform. For example, the attacker could be the machine's owner and operator, a curious or even malicious administrator, an invader who has taken control of the operating system and/or hypervisor, the owner of a virtual machine physically co-located with the VM being attacked, or even a developer of the untrusted application or OS writing code to directly record user input.

Although we consider covert channels based on software interfaces like system calls, we do not consider side or covert channels based on *hardware limitations* or *execution time*. Untrusted enclaves can leak bits by modulating their cache accesses, page accesses, execution time, etc. While we do not claim to prevent the execution-time channel, Ryoan does limit the use of this channel to once per request.

Intel Software Guard Extensions

Software Guard Extensions (SGX), available in new Intel processors, allow processes to shield part of their address space from privileged software. Processes on an SGX-capable machine may construct an *enclave,* which is an address region whose contents are protected from all software outside of the enclave via encryption and hashing. Code and data loaded into enclaves, therefore, can operate on secret data without fear of unintentional disclosure to the platform. These guarantees are provided by the hardware [2].

SGX provides attestation of enclave identity, which for Ryoan is a hash of the enclave's initial state, that is, memory contents and permissions offset from the enclave base address. Ryoan assumes 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. Ryoan loads service provider code after it initializes. Before the service code is loaded and before passing sensitive data to Ryoan, a user will request an attestation from SGX and verify the identity of the enclave.

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

Hardware security limitations

Enclaves on modern Intel processors have security limitations including page faults [8], cache timing, address bus monitoring, and the information exposed by processor monitoring units. We believe these limitations must be addressed independently from Ryoan, and we hope they will be. Each of these limitations compromise Ryoan's security goals. If there are other hardware limitations, they also must be addressed independently from Ryoan. Part of the purpose in constructing the Ryoan prototype is to demonstrate the importance of addressing these hardware-based information leaks.

Native Client

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

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

Design

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

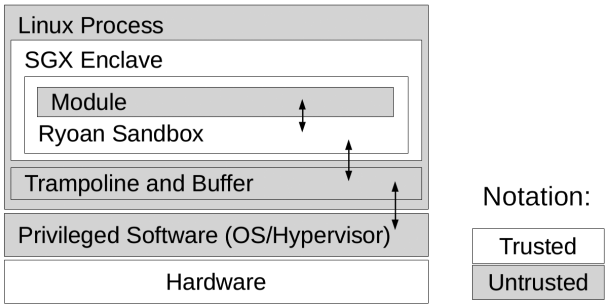

Figure 1: A single instance of Ryoan's distributed sandbox. The privileged software includes an operating system and an optional hypervisor.

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

Unprotected stores are eliminated by the NaCl tool chain and runtime. Ryoan mostly eliminates system calls by providing their functionality from within NaCl. For example, Ryoan provides `mmap` functionality by managing a fixed-sized memory pool within the SGX enclave. However, untrusted modules must read input and write output, so Ryoan provides a restricted I/O model that prevents data leaks: for example, the output size is a fixed function of input size. A module cannot communicate the contents of the input data by changing the size of the output.

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

Ryoan executes inside of hardware-protected enclaves and does not trust the operating system nor the hypervisor. SGX generates an unforgeable remote attestation for the user that a Ryoan instance is executing in an enclave on the platform. The user can establish an encrypted channel that she knows terminates within that Ryoan instance. SGX guarantees the enclave cryptographic secrecy and integrity against manipulation by privileged software.

Enforcing Topology

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

Figure 2 shows an example of Ryoan processing input from user Alice whose sensitive data is processed by both 23andMe and Amazon. Each Ryoan instance executes in an enclave on the same or different machines. The host machine(s) might be provided by 23andMe, Amazon, or a third party. In all cases, Ryoan ensures no leakage of the user's secrets and also prevents leakage of any trade secrets used by 23andMe and Amazon.

Data-Oblivious Communication

One of the primary safety functions of Ryoan is to prevent the computational platform from inferring secrets about the input data by observing data flow among modules. Therefore, data flow must be independent from the contents of the input data: Ryoan never moves data in response to activity under the control of the untrusted module once the module has read its input data. This safety property is sometimes called being data oblivious [9].

Units of work can be any size, but Ryoan ensures that data flow patterns do not leak secrets from input data by making module output size a fixed, application-defined function of the input size. Ryoan protects communication with the following rules: (1) each Ryoan instance reads its entire input from every input-connected Ryoan instance before the module starts processing; (2) the size of the output is a polynomial function of the input size, specified as part of the DAG, and Ryoan pads/truncates all outputs to the exact length determined by the polynomial and the size of the input; (3) Ryoan is notified by the module when its output is complete, and it writes the module's output to all output-connected Ryoan instances. Ryoan encapsulates module output in a message that contains metadata which describes what is module output and what is padding (if any). The metadata is interpreted, and any padding is stripped away by the next Ryoan instance before exposing the data to its module. Each Ryoan instance must receive the complete input of a work unit before executing its module. These rules are sufficient because they ensure that output traffic is independent from input data

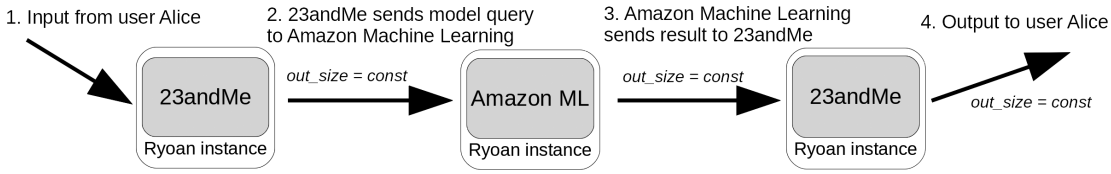

Figure 2: Ryoan's distributed sandbox. In this example, the application spans the administrative domains of 23andMe and Amazon.

(though there are possible alternatives—for example, each request could specify its output size).

Consider the scenario in Figure 2. Each input comes from a user. The user can choose to leak the size of the input, or he can hide the size by padding the input. The description of the DAG specifies that (1) the output of 23andMe's first module is padded to a fixed size, defined by 23andMe, which can hold the largest possible model query; (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.

Ryoan's Confined Environment

Any module with access to user data is executed in Ryoan's confined environment, which prevents information leakage while reducing porting effort. When a module receives the secret data contained within a request, it enters the confined environment and loses the ability to communicate with the untrusted OS via any system call. Therefore, Ryoan must provide a system API sufficient for most legacy code to function properly. To reduce porting effort, Ryoan provides an *in-memory virtual file system* and supports anonymous memory mappings from a pre-allocated memory region to support module dynamic memory.

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, ClamAV creates temporary files to store objects extracted from the PDF. Ryoan's in-memory file system satisfies these requirements.

Module Life Cycle

A Ryoan instance enforces the following life cycle on its module: creation, initialization, wait, process, output, destruction/reset. The sandbox begins by validating its module and verifying that its identity matches the DAG specification. The instance allows the module to initialize with full access to the system services exposed by vanilla NaCl. Nonconfined initialization makes module creation more efficient and 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, the module processes a request, generates its output, and then is destroyed or reset to prevent accumulating secret data. Ryoan instances are request oriented: input can be any size, but each input is an application-defined "unit of work." For example, a unit of work can be an email when classifying spam or a complete file when scanning for viruses. Each module gets a single opportunity to process its input data.

Checkpoint-Based Enclave Reset

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

Ryoan provides a checkpoint service that allows the application to be rolled back to an untainted, but initialized, memory state (Figure 3). In our prototype this state is at the first invocation of `wait_for_work`. Ryoan does not allow an enclave that has seen secret input to be checkpointed, because its data model is request-oriented: modules should not depend on past requests to operate. Checkpointing a module that has seen secret data would (potentially) give that module multiple execution opportunities on a single request's data.

Checkpoint restore allows Ryoan to save the cost of tearing down and rebuilding the SGX enclave, and it saves the cost of executing the application's initialization code. Ryoan checkpoints are taken once but restored after each request is processed. Therefore, Ryoan makes a full copy of the module's writable state and simply tracks which pages get modified, avoiding a memory copy during processing. Only the contents of pages that were modified during input processing are restored. 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.

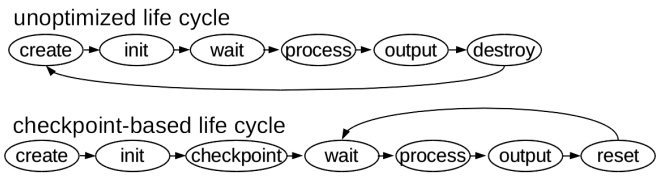

Figure 3: Instance life cycle: unoptimized vs. checkpoint based

Use Cases

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

Email Processing

A company can use Ryoan to outsource email filtering and scanning while keeping email text secret. We consider spam filtering and virus scanning, using popular legacy applications—DSPAM 3.10.2 and ClamAV 0.98.7. The computation DAG for this service contains four Ryoan instances, each confining a data processing module (see Figure 4). An email arrives at the entry enclave over a secure channel. The entry enclave simply distributes the email text and attachments to the enclaves containing DSPAM and ClamAV, respectively. The results of virus scanning and spam filtering are sent to a final post-processing enclave, which constructs a response to the user over a secure channel.

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.

Secrecy for both users and 23andMe is protected with a DAG (see Figures 2 and 4). Amazon provides the classifier, which queries a model as a Ryoan module. Users provide their genetic information, medical history, and activity log in a request. Upon receiving a user's request, 23andMe's first module constructs a Boolean vector of health features and forwards it to Amazon's module. Amazon's module generates predictions based on the model and forwards the result to 23andMe's second enclave, which then forwards the result back to the user.

Image Processing

Image classification as a service is an emerging area that could benefit from Ryoan's security guarantees. We envision a scenario where a user wants different image classification services based on her expertise. For example, one service might be known

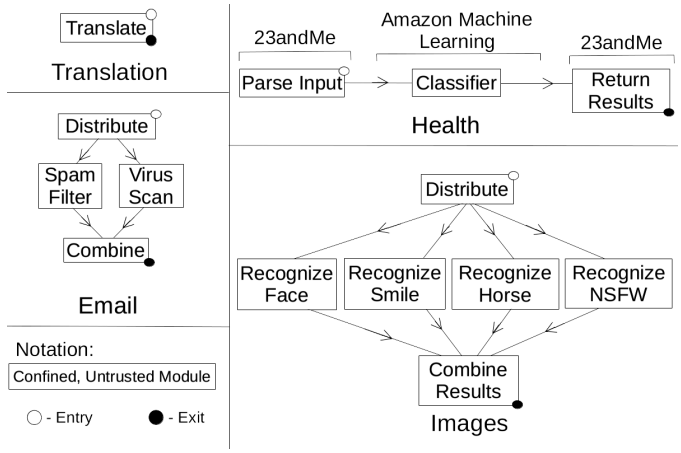

Figure 4: Topologies of Ryoan example applications. Nodes in the graph are Ryoan instances, though we identify them by their untrusted module. Users establish secure channels with trusted Ryoan code for the source and sink nodes to provide input and get output, respectively.

for accurate identification of adult content while another might do an excellent job recognizing and segmenting horses. The image processing DAG in Figure 4 shows an example where an image filtering service outsources different subtasks to different providers and then combines the results. 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.

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, a statistical machine translation system. We train a phrase-based French to English model using the News Commentary data set released for the 2013 workshop in machine translation [10].

Evaluation

We evaluated Ryoan's overhead on realistic workloads for each of these use cases. Slowdowns range from 27% to 419%. The Ryoan prototype relies on some unreleased SGX features. Therefore, our evaluation involves an SGX performance model where applicable. For evaluation details see the original publication [1].

Conclusion

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

References

[1] T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel, "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)*, pp. 533–549: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/hunt.

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

[3] A. Baumann, M. Peinado, and G. Hunt, "Shielding Applications from an Untrusted Cloud with Haven," in *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)*, pp. 267–283: https://www.usenix.org/system/files/conference/osdi14/osdi14-paper-baumann.pdf.

[4] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. L. Stillwell, D. Goltzsche, D. Eyers, R. Kapitza, P. Pietzuch, and C. Fetzer, "SCONE: Secure Linux Containers with Intel SGX," in *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*, pp. 689–703: https://www.usenix.org/system/files/conference/osdi16/osdi16-arnautov.pdf.

[5] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, "Native Client: A Sandbox for Portable, Untrusted X86 Native Code," in *Proceedings of the 30th IEEE Symposium on Security and Privacy*, 2009, pp. 79–93: http://regmedia.co.uk/2008/12/09/native_client_paper.pdf.

[6] D. Sehr, R. Muth, C. Biffle, V. Khimenko, E. Pasko, K. Schimpf, B. Yee, and B. Chen, "Adapting Software Fault Isolation to Contemporary CPU Architectures," in *Proceedings of the 19th USENIX Security Symposium (USENIX Security '10)*, pp. 1–11: https://www.usenix.org/legacy/event/sec10/tech/full_papers/Sehr.pdf.

[7] B. W. Lampson, "A Note on the Confinement Problem," *Communications of the ACM*, vol. 16, no. 10, October 1973.

[8] Y. Xu, W. Cui, and M. Peinado, "Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2015, pp. 640–656: http://www.ieee-security.org/TC/SP2015/papers-archived/6949a640.pdf.

[9] O. Ohrimenko, F. Schuster, C. Fournet, S. Nowozin, A. Mehta, K. Vaswani, and M. Costa, "Oblivious Multi-Party Machine Learning on Trusted Processors," in *Proceedings of the 25th USENIX Security Symposium (USENIX Security '16)*, pp. 619–636: https://www.usenix.org/system/files/conference/usenixsecurity16/sec16_paper_ohrimenko.pdf.

[10] Shared Task: Machine Translation: http://www.statmt.org/wmt13/translation-task.html.