Veracruz FAQ

Introduction

Veracruz is a research project exploring how secure distributed systems can be built using **strong isolation technology** and **remote attestation** protocols. Here, "strong isolation" is an idiosyncratic term that we use for any technology that exists on a continuum between pure hardware **secure enclaves** to high-assurance **hypervisor**-based isolation. The common factor in all "strong isolation" technologies is that they successfully isolate code within a container — providing strong confidentiality and integrity guarantees, even against privileged software co-tenanted on the same machine — whilst attempting to significantly minimise the trusted compute base.

Depending on the biases and interests of a particular audience, Veracruz can be explained in a number of ways:

Veracruz as a platform for privacy-preserving computation: under this view, Veracruz is an exploration of how to build flexible and efficient collaborative **privacy-preserving computations** between a group of mutually-distrusting collaborators, or principals. In this light, Veracruz aims to support similar use-cases to Advanced Cryptographic techniques, like *homomorphic encryption*, *functional encryption*, and *secure multi-party computations*, but uses a mixture of strong isolation technology, remote attestation, and transport layer security protocols — instead of pure cryptography — to affect these computations.

Veracruz as a demonstrator for strong isolation technology: under this view, Veracruz is an exploration of how strong isolation and remote attestation facilitate the design of data-intensive distributed systems. Our working thesis is that these are technologies that will allow engineers to build systems capable of exhibiting fine-grained control over data. Veracruz is an exploration, or **demonstrator**, of what can be done in this space — preparation for a future where large pools of data are a liability, rather than an asset.

Veracruz as an abstraction layer for strong isolation technology: under this view, Veracruz is an **abstraction layer** for strong isolation mechanisms, abstracting away low-level details of remote attestation protocols, secret provisioning, and container life management. Moreover, Veracruz provides users with a uniform — if restrictive — programming model irrespective of the particular isolation technology that a computation is being run on top of. The same program, compiled for the Veracruz platform, can be executed on any of the isolation technologies capable of hosting Veracruz.

Map of this document

The following sections explain Veracruz in more detail, and also compares Veracruz to other similar projects. We provide a summary of the top-level questions asked and answered in this document:

- See §What are Veracruz computations? to understand the principles behind a collaborative Veracruz computation, and understand the major conceptual components of the Veracruz framework.
- See §What is the Veracruz proxied attestation service? to understand some of the problems with remote attestation in Veracruz, and why we introduced a proxied attestation service.
- See §What are some Veracruz use-cases? for details of potential use-cases, of both commercial and more academic interest, of the Veracruz framework. These are just a small taste of the many potential use-cases we have in mind for Veracruz.
- See §What is the Veracruz threat model? for details of the Veracruz threat model: what we aim to protect against, what we do not protect against, what must be trusted by principals in a computation, and what need not be trusted. The precise threat model that we assume in Veracruz depends on the strong isolation technology in use for a particular computation.

- See §What is the Veracruz programming model? to discover how programs are actually written for the Veracruz framework, and learn about the Veracruz SDK.
- See §How does Veracruz compare to… for a comparison between Veracruz and two other *Confidential Compute Consortium* projects: **OpenEnclave** and **Enarx**.
- See §What's next for Veracruz? for a summary of potential future work. Veracruz is still evolving, and we have big ideas for where the framework can go, and what it can be used for.
- See §Who wrote Veracruz? for details of who wrote Veracruz, along with contact details for the authors.

What are Veracruz computations?

At a high-level of abstraction, every Veracruz computation involves some number of data-owners trying to collaborate with a program owner. Call the data sets of the respective data owners D_i for $1 \leq i \leq N$, and the program of the program owner, π. Collectively, the group wishes to compute the value $\pi(D_1, \ldots, D_N)$, perhaps making use of a third party, or delegate's, machine to power the computation.

Importantly, the data and program owners do not want to reveal anything about their respective data sets, or program, unless they explicitly choose to **declassify** (i.e. reveal to a third-party) that information themselves, or agree to a potential declassification when entering a collaborative Veracruz computation.

Moreover, the delegate does not want their machine to be damaged by a program that they cannot see or monitor. The trusted Veracruz runtime therefore provides strong sand-boxing guarantees to the delegate, ensuring a level of protection no matter what program, π, is running on their hardware.

Is Veracruz always privacy-preserving?

Absolutely not! Veracruz will always provide a base-level of security, for example by protecting against snooping by the delegate through the use of strong isolation, and the use of TLS-encrypted links also stops third parties from intercepting traffic between the program and data owners with the delegate's container, but these fall far short of guaranteeing a privacy-preserving computation *per se*.

What's missing is a consideration of where the results of a Veracruz computation are able to flow to, and what these results might say about the secret inputs to the computation.

To plug this gap, every Veracruz computation is configurable with a **global policy** that describes the identities of the participants to the computation, and their roles, amongst other things. Participants to a computation are expected to vet this global policy before collaborating in a computation, and understand the consequences of collaborating.

In certain circumstances, participants may find it useful to intentionally declassify information about their respective secrets **out-of-band** — that is, externally to the Veracruz computation, for example before the computation starts — before everybody enrolls in a computation, as a way of furthering a collaboration. Some situations where this makes sense will be discussed in §What are some Veracruz use-cases?. This may seem like a strange choice, but it is intentional: Veracruz is capable of operating around existing trust relationships, and likewise protecting against the lack of a trust relationship, as the situation requires. Not every collaborative computation requires heavy-weight privacy protections, with all the inconvenience (and often, inefficiencies) that this entails. Yet, some do. Veracruz allows you to pick and choose, as the situation demands.

This aspect of Veracruz is important to understand, so we'll reiterate: *Veracruz as a platform is not necessarily privacy-preserving. Rather, individual computations built on top of Veracruz may be privacy-preserving, whilst others may not.* Computations have to be reviewed on a case-by-case basis, by the principals party to those computations, taking into account existing trust relationships, to understand what protections are being provided.

The next few subsections walk through the various steps that a group of collaborators follow, when working with Veracruz.

What is in the global policy?

Henceforth, we'll call the collaborators in a Veracruz computation **principals**. We need a global policy to describe the **roles** and **identities** of every principal, to ensure everybody is aware of who will supply inputs, who will receive outputs, and the identities of all the other principals in the computations.

One role of the global policy is describing the "topology" of a Veracruz computation, describing who data will flow from, who will process it, and who will receive an output. It's therefore important that everybody understands the global policy, and has a chance to vet it before collaborating, as it plays an outsized role in a given principal understanding whether a Veracruz computation is sufficiently "safe" for them to take part in. As a result, we assume that the global policy is public, and vettable by each participant before the computation starts.

The global policy assigns to each principal party to a prospective computation a mixture of roles, as mentioned previously:

- **Data Provider**: provides an input data set to the computation. Whilst Veracruz supports computations with no Data Providers, in a typical Veracruz computation there may be N separate principals with this role, each providing a private input to the computation, D_i for $1 \leq i \leq N$ Moreover, one principal may provide multiple input data sets to the computation (i.e. effectively, one principal has this role twice).
- **Program Provider**: contributes an input program π. In the current design of Veracruz, there is exactly one principal with this role.
- **Computation Host**, or **Delegate**: provides the computational power necessary to compute the result of the computation, $\pi(D_1, \ldots, D_N)$. In the current version of Veracruz there is exactly one principal with this role.
- **Result Receiver**: may retrieve the result of the computation, as computed by the Delegate, once the computation has finished. Like the case with the Data Provider role, in a Veracruz computation there may be many principals with this particular role.

Veracruz supports principals taking on more than one role. For example, a particular Veracruz computation may be configured so that a principal takes on the role of Program Provider and Data Provider, whilst another computation may be configured so that all of the principals with the Data Provider role also have the Result Receiver role, too.

How do you vet the global policy?

As mentioned above, Veracruz can support privacy-preserving collaborative computations. As a result, each principal in a Veracruz computation has a set of concerns when enrolling in a computation that is a function of the role, or combined set of roles, in the computation that they take on:

- Unless they actively choose to declassify their data sets, D_i, each principal with the Data Provider role wishes to keep their data sets secret from the other principals in the computation.
- Unless they actively choose to declassify their program, π, the principal with the Program Provider role wishes to keep their program a secret from the other principals in the computation.
- The Delegate wishes to prevent their machine from being damaged, subject to malware attack, or similar, by the program running on their machine. Note that this desire is in tension with the Data and Program Providers' desire to keep their inputs to the computation a secret: this desire implies that the Delegate cannot audit the program before, or monitor it during, the execution of the program on the input data sets, $\pi(D_1, \ldots, D_N)$.
- All principals party to the computation wish to ensure that the result of the computation is only revealed to principals with the Result Receiver role.

In short, in the *most general case*, principals party to a Veracruz computation are all **mutually distrusting**, and further each individual principal may assume that an arbitrary collection of other principals party to the computation are *conspiring* against them to try to steal their secrets, or damage their machine, depending on the principal. For example, a principal with the Data Provider role can assume that the Delegate and Program

Provider are working in cahoots, trying to subvert the Veracruz computation and steal the input data set.

The above is a description of the most general case: in various use-cases for Veracruz, we can imagine there being existing trust relationships between various principals and therefore this level of paranoia may be unwarranted, or even inconvenient. In other cases, there really is mutual distruct amongst all parties, in which case the data owners may demand to see the program that will be applied to their input data sets, for additional vetting, before enrolling in a computation. Again, some of these will be discussed in §What are some Veracruz use-cases?

Principals in a prospective Veracruz computation therefore need to carefully read the global policy file before committing to enrolling in a computation. In particular, they must properly understand who the results of a computation are shared with, and the consequences of this sharing needs to be evaluated with respect to any existing trust relationships that they may have with the Results Receiver. If they do not trust the Results Receiver then they may need to seek additional assurances from other principals in the computation, as a result. A data provider may demand access to the source code of the program, to be run on their data set, to make sure that the program does not declassify secrets as a side-effect of the computation.

We will discuss the Veracruz threat model in more detail, in §What is the Veracruz threat model?

The policy is acceptable. Now what?

As a first step, the Program Provider uses a **remote attestation** protocol to ensure that the the delegate has loaded the trusted Veracruz runtime correctly. The delegate's machine is assumed to be capable of spawning some instance of a *strong isolation* technology, which Veracruz uses as a "venue" within which a computation takes place. At the time of writing this document, Veracruz supports three different isolation mechanisms, covering different points on a continuum of paranoia. These are:

- **Arm TrustZone Trusted Applications**, deployed under the open-source **OPTEE** trusted execution environment.
- **Intel's Software Guard Extensions (SGX)** Secure Enclaves.
- Hypervisor-based isolation, using **seL4**, Data61's high-assurance, formally-verified capability-based microkernel, running in the `EL2` exception level (i.e. a privileged mode within which a hypervisor executes) on Arm `AArch64`.

The trusted Veracruz runtime's job is to receive, load, manage, and provide services to the program owner's program, π, as it executes on the data sets provided by the data set owners. As a result, this trusted runtime also needs to be vetted, or implicitly trusted, by everybody party to the computation, as it again plays an outsized role in the security of any collaborative computation. We therefore assume that the code of the trusted Veracruz runtime is publicly available, and auditable by everybody prior to enrolling in a computation. The trusted Veracruz runtime is **not** a secret.

The remote attestation protocol, mentioned above, therefore serves to convince the program owner that their audit of the trusted Veracruz runtime took place on the *same* code that is now loaded into the delegate's container, and that no backdoor has been surreptitiously inserted into the runtime by the delegate. Moreover, the attestation protocol will also convince the program owner that the trusted Veracruz runtime is executing within a genuine container, e.g. a genuine Intel SGX enclave, and not being emulate, in some way by a clever but malicious delegate.

Once the program owner has verified the authenticity of the runtime, using remote attestation, they **provision** their program into the container on the delegate's machine. To do this, they use an encrypted TLS link that terminates directly inside the container. The delegate — untrusted, in the general case, by the program owner — simply sees opaque, encrypted bytes being transferred into the container, and cannot interpret these without breaking TLS encryption.

Is that use of remote attestation really sufficient to trust the delegate's container?

No, the above description of Veracruz's remote attestation protocol was simplified quite substantially. When we designed the various protocols used in Veracruz we needed to be mindful of two problems:

1. **Identity**: principals should never be confused about which container they are communicating with. We therefore need some means of identifying containers.
2. **Freshness**: a malicious container should not be able to "recycle" attestation tokens from previous rounds, fooling a principal into thinking that they are unmodified Veracruz runtimes, executing inside an authentic container, when they are not.

The second problem is easiest to solve, and requires that principals generate fresh random numbers, or **nonces**, and bundle these with any communication with the container. These nonces are sent back to the principal in any response that the container makes. A principal receiving a message, with an unrecognized nonce, knows something has either gone awry or a malicious container is trying to recycle messages from a previous round of communication with another principal. In either case, the computation should be aborted. Note that this use of nonces is standard in cryptographic engineering for ensuring freshness.

For identity, we have a more significant problem: the usual mechanism for identifying an agent does not work, or at the very least is not very convenient when working with containers!

The issue with identity stems from the fact that Veracruz containers are *ephemeral* in the sense that they can come and go rapidly. Usually, to identify an agent, online say, we'd rely on some wider infrastructure based on **cryptographic certificates** issued by a **certificate authority**. This wider infrastructure only really makes sense when the certificates issued by the authority refer to the public keys of long-lasting entities, like a server hosting a website, for instance.

This is a real problem, as without some reliable way of identifying a container a principal cannot really be sure that they are establishing a TLS connection with the same container that they have authenticated using remote attestation. To solve this problem, Veracruz containers self-generate an asymmetric key-pair and then **self-sign** a certificate using the private key of this key-pair. The public key and self-signed certificate are then "published", or made available to every principal in the prospective computation. The hash of this self-signed certificate is included in any attestation token issued by the container. Now, attestation tokens are explicitly tied to a particular container, with a known identity and known public key, which can be used by principals establishing a TLS connection to ensure that they are communicating with the right container.

Can't you just look inside the container?

This depends on the particular container, and the method of looking! For Arm's TrustZone and Intel's SGX, the answer in general is "no", at least not without using sophisticated side-channel attacks against the container, or subverting the container's security guarantees, somehow. At present, we defend against neither of these — see §What is the Veracruz threat model? for more details on the precise Veracruz threat model.

One of our supported isolation technologies, Intel SGX, provides additional protection against a range of physical attacks against the container, which *architecturally* Arm's TrustZone does not do, though *implementations* may choose to do so. Hypervisors, such as seL4, are also incapable of defending against physical attacks without additional hardware support.

Moreover, strong isolation technology usually provides an **integrity** guarantee, in addition to privacy guarantees, which means that third parties cannot influence the course of the computation inside the container, from outside. In many Veracruz use-cases, this integrity guarantee is as interesting to users as the guarantee of privacy as it means that, via remote attestation, they're able to know precisely what program is loaded into a container, and know that the delegate cannot influence the behaviour of that code when it executes.

Principals are expected to evaluate the container technology in use by the delegate before enrolling in a computation. For some collaborative computations, especially where an existing trust relationship exists, hypervisor-based approaches may be perfectly adequate. In other circumstances, the additional protections offered by hardware-based approaches, like SGX and TrustZone, may be necessary.

Why does the delegate trust π?

They don't, as they never see it, and cannot monitor its runtime behaviour due to the strong isolation technology in use on their machine!

However, they do trust the Veracruz runtime, which is auditable by them. A key property of the trusted Veracruz runtime is that it correctly **sandboxes** the program, π, which is a **WebAssembly** (or WASM, henceforth) program compiled specifically for the Veracruz platform. The WASM program cannot escape its sandbox, at least not without exploiting some bug in the execution engine that the Veracruz runtime uses.

Aside from sandboxing the program, we also tightly control its capabilities. Specifically, the only capabilities that the WASM program can exploit are those explicitly granted to it, by the Veracruz runtime, and the program is given very few by Veracruz. More will be said on this later in this section, and in §What is the Veracruz programming mode?

Why is π provisioned first?

Suppose, as a precondition to enrolling in a computation, that the data owners demand that the program owner declassifies their program, π, before the computation starts. What then stops the program owner from showing the data owners one program, and then provisioning another program into the trusted Veracruz runtime?

To prevent this, we require that the program owner provisions their secret first. Then, the trusted Veracruz runtime **measures** the provisioned program by computing a hash of its binary. This hash can then be requested, by the data owners, before they choose to continue to provision their secrets into the container, and compared against a reference hash derived from the program, π.

Note, even in cases where the program, π, is not declassified by the program owner this measurement of the loaded program may be useful. For example, the data owners may have an interest in ensuring that a known-good program, even if it cannot be audited directly, has been loaded into the container perhaps as part of a series of collaborative computations on different data sets. The data owners (and result receivers) have a natural interest in ensuring that the *same* program is used to transform input data, even if they do not know what it is.

Isn't running a secret program on data strange?

No, not at all — it's very common, in fact. Consider just one example: ancestry testing services which process cheek-swab data and extract genetic information, deriving ancestry data. There are many of these companies, and at heart they merely extract data and run a proprietary program on it, which the customer never sees. What does the program do? Does it have bugs? Is it anything other than a random ancestry generator? Customers in general do not know as they cannot audit the code. Yet they still use it, on particularly sensitive data too!

More commonly, any closed-source code is — for all practical reasons — completely inscrutable to users.

After the program, the data is provisioned, right?

Yes. The data providers use the same remote attestation steps that the program owner took, to check the authenticity of the trusted Veracruz runtime. Moreover, this time they can also make reference to the measurement, as described above, of the provisioned program, π, before they choose to commit to fully enrolling in the computation.

Note that, per the global policy file, the trusted Veracruz runtime knows exactly how many input data sets are due to be provisioned, and from whom. Once all of the data has been loaded, the container's **lifecycle state** switches, and the program becomes ready to execute.

Why should the data owners trust the program, π?

In the general case, they should not! As a result, the data owners need to carefully vet the global policy file to ensure that they understand who gains access to the result of the computation once it completes, especially when the program, π, is not declassified. For example, a malicious data owner, conspiring with a results receiver, may load an *identity program* which produces as output one of the secret inputs passed to it (to properly understand how you program against Veracruz, see §What is the Veracruz programming model?) thereby granting the results receiver access to the input secret. Moreover, even if the program, π, isn't so brazenly insecure as this, the output of many computations still gives away a lot of information about that computation's inputs. In many cases, especially when the program, π, is not declassified, the only sensible option may be for the data owners to also be the result receivers.

Moreover, what stops the program, π, from simply dumping all of its secret inputs to `stdout` on the delegate's machine? This is a legitimate question, given the fact that — generally — a data owner is allowed to assume that the program owner and the delegate may be working in cahoots to steal a secret! To stop this, Veracruz puts strict limits on the **expressivity** of the program that is loaded into the container. To a first approximation, a Veracruz program may only compute a **pure** function of its inputs, modulo sampling from a source of random data. As a result, the only **side-effect** of a program compiled for the Veracruz platform is its result: it cannot open files on the delegate's machine; it cannot directly control hardware on the delegate's machine; it cannot sample from any data source other than the inputs passed to it by the data owners, other than a random data source. Note that, aside from the data owners, this limitation is also necessary to protect the delegate's machine, too.

In future versions of Veracruz, the trusted runtime may supply additional services aside from a random number source to the program, π. For instance, the runtime providing a cryptographic API to the program may make some use-cases easier to develop. For more, see §What's next for Veracruz? for more details of our aspirations for future Veracruz features.

Why does the Veracruz runtime have a lifecycle state?

The trusted Veracruz runtime is **stateful**, in the sense that there's an explicit lifecycle state, with a defined set of **state transitions** between lifecycle states that trigger once e.g. the program is loaded. The runtime's current lifecycle state can be queried by anybody party to the computation, at any point.

There's a few reasons why the trusted Veracruz runtime is stateful. First, as discussed above, it's important for a Veracruz computation that the program, π, is loaded before the data sets are loaded. An explicit state machine enforces this property, and the defined state transitions ensure that once a program is loaded, and data therefore becomes loadable into the container, the program owner cannot load another program thereby tricking the data owners into thinking the computation will use one program instead of another. Similarly, the Veracruz runtime also needs to make sure that the program, π, only becomes executable when all expected data sets have been loaded by the data owners. In short: the defined lifecycle states in the trusted Veracruz runtime allow us to maintain a number of important **system invariants** that help ensure the correctness of Veracruz.

In addition to this, the runtime's state machine prevents some **probing attacks**, wherein a malicious data set owner repeatedly modifies their data set and runs a computation over-and-over again, observing differences in the output, to deduce information about the program, π, or similar. To stop this, we adopt an important invariant in Veracruz: a computation is only ever executed when the owners of the program and the input data sets explicitly **consent** to this. Here, we take a program or data owner's provisioning of their secret as their consent to the computation. As a result, every computation, even ones using the same program or data sets as a previous one on a given platform, requires an explicit provisioning act by the program and data owners. No computation can be endlessly re-run, over and over, perhaps with minor modification of data or program, without the owners being made aware of this and actively consenting. Under this scheme, probing attacks are still potentially possible, but they require the victim to be actively complicit in the attack.

Lastly, the lifecycle state is sometimes useful when designing collaborative computations, as we can use details of the state transition diagram to definitively know that a party to a computation has committed to some decision, safe in the knowledge that Veracruz will prevent that commitment from being reneged on. An example will be discussed in §What are some Veracruz use-cases?

Everything's provisioned. Now what?

The trusted Veracruz runtime is now in a "ready to execute" lifecycle state. As a result, one of the principals with the Results Receiver role can go ahead and ask the runtime to execute the program, and in response the trusted Veracruz runtime starts executing the WASM program, π.

The program, π, will either diverge, or finish executing and therefore producing a result, $R = \pi(D_1, \ldots, D_N)$. Once this happens, all of the principals who are authorised to receive R, per the global policy file, are permitted to request it, and the runtime will forward it to them.

At the time of writing, Veracruz offers two **execution strategies** when executing the WASM program, π: **interpretation** and **just-in-time compilation** (or JITing, henceforth). Naturally — at least in the limit — JITing exhibits a large performance boost when compared to interpretation, so why bother supporting both execution strategies? There's two reasons for this:

1. First, for computations on very small datasets, interpretation may actually be faster than JITing, due to the overhead of the JIT compiler walking the program's AST and translating it into machine code before execution can begin.
2. JIT compilers perform all sorts of tricks with memory, notably writing machine code into memory pages and thereafter making them executable. As a result, JITs could make an attractive target for malicious users to try to exploit. Depending on the

participant's level of paranoia, they may therefore opt to interpret the program, π, rather than JIT it, for security reasons.

As a result of supporting multiple execution strategies, the particular strategy that Veracruz will use is specified in the global policy file, and therefore everybody party to a computation is aware, before enrolling in the computation, how the program will be executed. At the moment, not every isolation technology supports both execution strategies: only the seL4 hypervisor is capable of supporting both interpretation and JITing, but we are actively working on extending this support to our other platforms. Note that, with seL4, we are able to ameliorate potential security problems with JITs, as discussed in point (2) above, somewhat, by isolating the JIT from the rest of the trusted Veracruz runtime, using process-level isolation offered by seL4. Now, an attacker trying to exploit a security problem in a JIT not only has to break the JIT, but they also have to break seL4's isolation mechanisms, too.

Is that the computation finished?

Not quite: we need to consider teardown of the container. Veracruz prevents a second computation reusing a container that a previous computation used as its "venue". Rather, the container, an SGX Secure Enclave for instance, must be explicitly destroyed: this is another reason why the trusted Veracruz runtime is stateful, and has an explicit state transition system.

The reason for this is to try to mask **microarchitectural effects**, such as a computation's effect on the cache or various microarchitectural buffers, left over from a previous computation that could be observed by a subsequent one.

We assume that the delegate destroys the container once the computation is finished.

In future versions of Veracruz, this restriction may be relaxed, in which case this fact will be reflected in the global policy file. See §What's next for Veracruz? for more details of our aspirations for future Veracruz features.

What is the Veracruz proxied attestation service?

As mentioned, Veracruz supports a number of different isolation technologies, and in some cases a container technology and a remote attestation protocol are intimately linked. This is the case with Intel SGX, for instance, which is deeply tied to Intel's attestation protocol, EPID, and uses the Intel Attestation Service — a web-service for authenticating EPID attestation tokens. Arm, too, has its own attestation protocol in PSA Attestation, which we use for attestation on Arm TrustZone (though this is not yet fully implemented). More generally, if Veracruz adds another supported isolation technology the chances of having to support another attestation protocol are quite high. This poses a few problems for Veracruz:

1. Client code interacting with Veracruz containers becomes more complicated, as it needs to support multiple attestation protocols. Aside from bloating the client, this also breaks the abstraction over isolation technology that Veracruz aims to provide.
2. For Intel SGX Enclaves, for example, multiple calls to Intel's Attestation Service are needed when working with multiple enclaves. Whilst the EPID protocol is designed with anonymity in mind, there are still several reasons why this may be undesirable.
3. Changes in **policy** require an update of all client code, at present. Here we have in mind changes in trust relationships between developers of client code and particular hardware manufacturers, or models of hardware, either driven by security disclosures, geopolitical trends, or shifting business attitudes/contexts.

Our solution to these problems is to introduce a **proxy attestation service**. This attestation service presents a uniform protocol interface to client code, no matter which underlying isolation technology is in use on the delegate's machine, through the use of Arm's PSA attestation protocol.

The introduction of this proxied service solves all three problems identified above.

First, note that this attestation service must still use the native attestation protocol — for example, EPID, in the case of Intel SGX — at some point, but client code never sees this, instead only ever interacting with the proxy attestation service using PSA Attestation. Note that, as our attestation service essentially "translates" one attestation protocol into another, or mediates between protocols, we sometimes refer to it as a **hybrid proxy**. This hybrid proxying significantly simplifies client code interacting with Veracruz: now, it need only support Arm's PSA Attestation protocol, rather than a panoply of differing protocols, depending on the particular isolation technology in use.

Second, the mechanism by which the proxied attestation process works reduces the number of calls to any external attestation service to one when authenticating containers on a delegate's machine. A single distinguished container, the **root container**, is authenticated using the native attestation protocol associated with that container technology — EPID, in the case of Intel's SGX — by the proxied attestation service itself. This root container is **enrolled** by the service by the establishment of a shared secret between the two, the equivalent of a **root attestation key** in more standard attestation schemes. This root container is then used to authenticate any other container, started on the same machine, by the proxied attestation service. Client code of Veracruz never observes the enrollment process of the root container: it's assumed to happen before any container is launched on the delegate's machine.

Without the proxied attestation service, a delegate launching E enclaves on their machine, each started for the purposes of hosting a single Veracruz computation involving D_i data providers for $1 \leq i \leq E$, required $\sum_{1 \leq i \leq E} D_i + 1$ calls to the native external attestation service (the 1 in the expression corresponds to the program provider in each Veracruz computation). Now, this is reduced to a single call: the authentication of the root enclave.

Lastly, observe that the introduction of this proxied service now allows client code using it to specify precisely what they are willing to trust, through the use of an **attestation pol-**

icy. In standard attestation schemes, this decision must be made in the client, and if, for example, a particular bit of hardware is revealed to be fundamentally broken, especially prone to leaking secrets from its strong container implementation due to microarchitectural weaknesses, after client code is deployed, then working around this requires an update to all client code deployed in the field. Instead, the introduction of this proxy allows clients, or a third party, to inform the proxy of which containers they are willing to trust, which can be taken into account by the proxy itself when authenticating or rejecting attestation tokens on the client's behalf. (Note, we have not yet got around to implementing this feature in Veracruz.)

What's the downside?

The major downside of the introduction of our hybrid proxied attestation service is that it increases the trusted compute base of the Veracruz framework. Principals in a Veracruz computation now need to explicitly trust the proxied attestation service, and also need to carefully vet the source code (assumed open, and auditable) of the software running inside the root enclave, launched on the delegate's machine. This is reflected in the Veracruz threat model — further discussed in §What is the Veracruz threat model? — which explicitly states that all principals must trust this service.

What are some Veracruz use-cases?

We'll consider three potential Veracruz use-cases, here. Some of these are of potential commercial interest, some are more of academic interest intended to demonstrate why some of Veracruz's design decisions were made. The chosen use-cases are intended to capture Veracruz's breadth of applicability.

Privacy-preserving machine learning

We consider "privacy-preserving machine learning" from two different angles: the protection of datasets, and the protection of machine learning algorithms.

Protection of datasets Two small online retailers — Acorp and Bmart — have a problem: they cannot adequately compete with their larger competitors in providing useful product recommendations to viewers of their respective websites based on past product purchases. The real problem for Acorp and Bmart is that their larger competitors have much larger datasets to play with, and therefore can train more accurate machine learning models, than either of them.

However, the situation becomes more interesting if Acorp and Bmart choose to pool their respective data sets, and learn a joint model over this data set. Now, the two are able to learn a more accurate model in concert than either would be able to do individually. Unfortunately, both Acorp and Bmart are in fierce competition, not just with their larger competitors, but also with each other, and therefore the idea of sharing data with each other is unconscionable!

Ordinarily, this would be the end of the matter, if not for Veracruz…

Specifically, Acorp and Bmart are going to collaborate in a limited sense, using Veracruz to learn a machine learning model over their joint data set. Importantly, neither retailer will learn anything about the other retailer's data set, nor will any other third party. Moreover, the only principals that will gain access to the machine learning model are the two retailers.

Out-of-band, Acorp and Bmart first negotiate the following aspects of their collaboration:

- The encoding, if any, that will be used when communicating the data sets over TLS.
- The columns that the joint data set will contain, and their interpretation.
- The machine learning algorithm that will be applied to the data sets.
- The encoding, if any, that will be used when communicating the result of the machine learning model, once learning has finished.
- Any error codes that will be returned by the machine learning algorithm, and their interpretation.

After this negotiation, one of the two corporations — let's assume it's Acorp — prepares a Veracruz program realizing the negotiated machine learning algorithm, as above, and gives access to Bmart for auditing. Note here that Acorp is specifically declassifying its program as a means of inducing Bmart to enroll in the computation: a "nothing up my sleeve" step. Moreover, Acorp is now taking on two roles in the computation: that of Data Provider and Program Provider.

Separately, the two prepare a Veracruz global policy specifying their respective roles — Data Provider, Program Provider, Results Receiver in the case of Acorp, and Data Provider, Results Receiver in the case of Bmart — and select a host on which to execute the computation. If the host is an established Cloud host, with a good reputation, the two may opt to use Veracruz's hypervisor-based isolation, for instance. In other contexts, for example the host being one of Acorp or Bmart themselves, they may choose to use a hardware-based isolation technology, either TrustZone or SGX.

Once setup, the two provision their program and then their respective data sets, as described in §What are Veracruz computations?, and obtain the machine learning model extracted from their joint data set. Neither learns anything, other than this machine learning model, from the other participant's data set, nor does anybody else, as desired.

Protection of algorithms Suppose Alice, a machine learning researcher, develops an innovative machine learning algorithm and wishes to license this algorithm to third parties, charging a small for every invocation of their algorithm to a data set. Suppose also that Bob wants to license this algorithm from Alice, and apply it to a data set that he owns. Bob would also like to keep his data set private, if possible.

Out-of-band, Alice tells Bob the binary format that he should store his data set in, and the binary format of the output answer that the algorithm will produce, and how to interpret it, along with any error codes that the machine learning algorithm may produce. Alice then informs Bob of the hash of her algorithm's implementation, for his record keeping.

Alice writes a global policy for a Veracruz computation: Alice will take the role of Program Provider, Bob the role of Data Provider and Results Receiver. The two agree to use a neutral third party as the delegate, for example a commercial Cloud host. The two follow the protocol outlined in §What are Veracruz computations?, and provision their respective secrets into Veracruz, before it computes the final result, which Bob can access.

Note in this case, Alice does *not* declassify her program. However, from Bob's perspective, this is "safe", as Alice will not be able to learn anything about Bob's data set from the invocation of her program, as she is not a Results Receiver in the computation. Moreover, Bob will not learn anything about Alice's program — as he never sees it — other than what can be deduced from the program's output.

Delegated computation

Suppose a company deploys a fleet of computationally-weak Internet of Things (IoT, henceforth) devices. The fleet of devices can be augmented with functionality that ordinarily requires a more computationally sophisticated device (e.g. natural language speech processing, or similar) by **delegating** the computation to a more sophisticated device. Unfortunately, this delegation poses two risks:

1. A loss of privacy. This aspect becomes especially serious when the delegated task is handed off to untrusted, multi-tenanted Edge devices, rather than centralized data centres, under the control of the owner of the IoT device fleet.
2. Problems with integrity, wherein the delegate computes a different function, or provides a different capability, than the delegating device is expecting, either due to bad programming, misconfiguration, or due to nefarious interference by e.g. a co-tenant, as in point (1) above.

Instead, Veracruz can be used as a means of delegating computations from one device to another, solving problems with both privacy and integrity of the computation.

Briefly, an untrusted server will act as the delegate, and is assumed capable of spawning a strong container capable of running the trusted Veracruz runtime. Depending on the particular model of delegation, either the device itself, or a third-party managing the device, takes on the role of Program Provider, with the delegating device acting as both Data Provider and Results Receiver. The device and program provider — if this role is not taken on by the device itself — follow the protocol outlined in §What are Veracruz computations? to provision their secrets into the delegate's container, after which the result is computed, and made thereafter made available to the delegating device.

Commitments

This use-case is more of academic interest, but demonstrates that the trusted Veracruz runtime's stateful nature is sometimes useful when designing a collaborative computation to achieve a desired effect.

Alice and Bob are playing a distributed coin-tossing game, wherein Bob "calls" a coin toss, guessing either *heads* or *tails*, before Alice tosses a fair coin. If Bob guessed correctly, prior to Alice tossing the coin, then Bob wins a prize, otherwise he loses and wins nothing. Alice does not trust Bob: she worries that he will change his guess after the coin is tossed in order to try to obtain a prize he is not entitled to. (Alice on the other hand is known to be unscrupulously fair by everybody, including Bob, who has no reason not to trust her.) Cryptographic approaches to solving this problem exist, using **cryptographic commitments**, but we sketch an approach base on Veracruz, here.

Alice prepares a program, π, which accepts a single input: Bob's guess. She presents this program to Bob (i.e. intentionally declassifies it) and Bob audits it. The program has a very simple behaviour: it receives Bob's guess as input, and returns it as its output — that is, it's a glorified identity function. Bob audits the program and accepts it.

Alice and Bob then agree a global policy for a Veracruz computation. Alice is listed as the Program Provider and Results Receiver, and Bob is listed as the Data Provider. They arbitrarily choose some delegate to host the computation, who loads the trusted Veracruz runtime into a strong container.

Following the protocol outlined in §What are Veracruz computations?, Alice provisions her program into the delegate's container, and waits for Bob to provision his guess into the container as an input. Note that Alice need not trust that Bob has guessed here, at this point, as she can instead query the state of the Veracruz runtime: if Bob has **committed** to a guess, then the Veracruz runtime should be in a "ready to execute" lifecycle state. Alice, trusting that the Veracruz runtime is correct, knows that at this point Bob cannot renege on his guess and guess again, as the runtime's state machine ensures that secrets, once provisioned, cannot be changed.

Once Alice knows that Bob has committed to a guess, she makes her coin toss. Now, to obtain Bob's guess, she requests the result of the computation from the Veracruz runtime, which executes the program, returning Bob's guess to Alice as its result. (See §What is the Veracruz programming model? for a full explanation of this point.) Alice then compares the guess to the coin toss, granting or denying Bob his prize as appropriate.

What is the Veracruz threat model?

We now specify precisely what a principal taking part in a Veracruz computation needs to trust, and what is left untrusted. Some aspects of the Veracruz threat model are common to all of Veracruz's supported strong isolation technologies, so we handle these cases first.

In every Veracruz computation principals must carefully check the global policy, and evaluate this with respect to any pre-existing trust relationships that they may have with other principals, as discussed in §What are Veracruz computations?. Moreover, in every Veracruz computation principals must trust:

- The trusted Veracruz runtime. NThis is assumed to be open, and auditable by everybody prior to enrollment in the computation. Note that principals must convince themselves that the trusted Veracruz runtime:
  - Only loads and executes whatever WASM program is provisioned into it by the Program Provider, and nothing else.
  - Only provides data sets as input to the WASM program that are explicitly provisioned into it by the Data Providers, and nothing else.
  - Correctly enforces the global policy describing the computation, namely that it enforces the described roles for the named principals, including the role of Results Receiver.
  - Correctly sandboxes the execution of the WASM program, providing only the host services detailed in §What is the Veracruz programming model? and nothing more.
  - Correctly implements the Veracruz host-call API.
  - Correctly implements the Veracruz state transition system so that, e.g. two different programs cannot be provisioned into the trusted Veracruz runtime, or that a second Veracruz computation cannot be scheduled by the Veracruz runtime without the explicit consent of all parties to the computation.
  - Only returns the result registered by the WASM program, when run on the data sets provisioned by the Data Owners, and nothing else.
  - Faithfully executes the WASM program, on its inputs, according to the formal operational semantics of WebAssembly, and does not diverge from these published semantics.
  - Is free of memory corruption errors, that could be exploited by an attacker to gain control of a container using e.g. **Return Oriented Programming** techniques. To mitigate this problem, the trusted Veracruz runtime is written in the Rust programming language, which aims to provide enhanced levels of memory-safety for systems code. Notably, all "unsafe" memory accesses are explicitly marked using an `unsafe` keyword, making them easier to audit.
- The attestation protocol. Principals must be able to trust that if the attestation protocol certifies an attestation token as being issued from an authentic strong container C and corresponds to code with a given **measurement** (a cryptographic hash of the binary loaded into C), M, then a binary with measurement M is indeed loaded inside container C.
- The Veracruz proxied attestation service.
- The security and integrity guarantees of the TLS protocol.

In every Veracruz computation principals need not trust: - Any other principal to the computation. - The untrusted Operating System, Hypervisor, or other system software running on the delegate's machine which is not explicitly marked as part of the trusted compute base in the subsections below. - Any networking stack: for every supported strong isolation technology supported by Veracruz, the networking stack is handled in untrusted code.

Side-channel attacks against the trusted Veracruz runtime are explicitly outside of the Veracruz threat model, at the time of writing. Various **controlled side-channel attacks** have been used by a number of academic research labs to break the security of Intel SGX and Arm TrustZone isolation. We do not explicitly defend against these, at present.

Note also that the trusted Veracruz runtime does not provide **liveness guarantees**. Namely, a principal to the computation could try to perform a denial of service attack against an ongoing Veracruz computation. This is especially easy for the delegate, who is capable of denying scheduled execution time to computations running inside an e.g. Intel SGX enclave. Defending against this is, again, outside of the Veracruz threat model. In situations where this matters, e.g. in collaborative computations wherein mutually distrusting parties are both Results Receivers, principals should ensure that the computation is structured in such a way that one party cannot receive the result of the computation whilst denying the result to the other party.

...for Intel SGX?

For Veracruz computations using Intel SGX as the isolation technology principals must trust:

- The security guarantees of the Intel SGX architecture/implementation on a given Intel microprocessor model. Namely, the inability of code executing outside of the enclave to break the confidentiality and integrity of the code executing inside the enclave boundary, without explicit declassification on the part of the enclaved code.
- The Apache Teaclave Rust-SGX SDK (formerly: Baidu Rust SGX SDK) which is used to develop Veracruz, and forms part of the trusted base of the trusted Veracruz runtime.
- The Intel Attestation Service, a web-service operated by Intel and used to authenticate attestation tokens issued by Intel SGX Secure Enclaves.

For computations using Intel SGX, principals need not trust any other software other than the trusted Veracruz runtime and the attestation service, mentioned above.

...for Arm TrustZone?

For Veracruz computations using Arm TrustZone as the isolation technology principals must trust: - The OP-TEE operating system, and its implementation of relevant GlobalPlatform APIs. Specifically, principals should trust that the OP-TEE operating system correctly isolates trusted applications from each other. - The security guarantees of the Arm TrustZone architecture. Namely, the inability of code executing in the non-secure world to break the confidentiality and integrity of the code executing in the secure world, without explicit declassification on the part of secure code. - The implementation of Arm TrustZone on the Delegate's machine is a faithful implementation of the Arm TrustZone architecture. - The Baidu Rust TrustZone/OP-TEE SDK which is used to develop Veracruz, and forms part of the trusted base of the trusted Veracruz runtime.

Note that, at present, the remote attestation flow for Arm TrustZone is not fully implemented in Veracruz. To do so, requires implementing a trusted boot mechanism for OP-TEE, which we leave for future work as it seems fairly straightforward to implement. When attestation for TrustZone is implemented, principals must additionally trust:

- The native attestation service used to authenticate attestation tokens issued by a TrustZone trusted application under OP-TEE.

...for hypervisor-based isolation with seL4?

For Veracruz computations using seL4 as the isolation technology principals must trust: - The security guarantees of the seL4 micro-kernel, acting as a hypervisor, on Arm AArch64.

Note, again, that at present the remote attestation flow for seL4 is not fully implemented in Veracruz. To do so, requires implementing a trusted boot mechanism, which we leave for future work as it seems fairly straightforward to implement along with an identity key provisioning service for seL4 protected virtual machines. When attestation for seL4 is implemented, principals must additionally trust:

- The native attestation service used to authenticate attestation tokens issued by an seL4-protected virtual machine.
- The identity key provisioning service, used to provision globally unique identity keys into an seL4-protected virtual machine for attestation purposes.

What is the Veracruz programming model?

In this section, we walk through a simple Veracruz program, and discuss the wider programming model.

Host calls, and `libveracruz`

The Veracruz runtime exposes a set of **host functions** (or **H-calls**) to the WASM program running on top of the runtime. These H-calls serve a similar purpose to **syscalls** in operating systems, namely provide a programming interface for relatively-unprivileged programs to access services managed by the privileged runtime/operating system. Whilst a typical desktop operating system may provide hundreds of different syscalls, the Veracruz runtime provides a very limited interface to programs. The following table describes this interface in its entirety:

H-call name	*Description*
`getrandom(buf, sz)`	Writes `sz` bytes of random data, taken from a trusted entropy source, into the buffer `buf`. The precise source used differs depending on the particular isolation technology being used by the Veracruz computation. If no suitable entropy source is available, or if random number generation fails for some reason, the H-call returns an appropriate error.
`input_count()`	Returns the number of secret inputs, provisioned by the data providers, that can be accessed by the program. This number will always match the number of data sources specified by the global policy, and is used to programmatically discover that number, so that programs parametric in the number of inputs can be written.
`input_size(idx)`	Returns the size, in bytes, of input `idx`. Note that data providers can provision their data sets in an arbitrary order, as part of the Veracruz provisioning process. However, once all data sets are provisioned, the runtime sorts the data sets to match the order specified in the global policy. As a consequence, the result of this H-call is always well-defined. Returns an error if `idx` is greater than the number of data sets provisioned by the data providers.
`get_input(buf, sz, idx)`	Reads input `idx` into the buffer `buf` with size `sz`. Returns an error if `idx` is greater than the number of data sets provisioned by the data providers, or if the size of input `idx` is grater than `sz`.
`write_result(buf, sz)`	Writes the result of a computation by copying the content of buffer `buf` with size `sz` into a buffer in the trusted Veracruz runtime. Returns an error if a result has already previously been written.

The table above is provided for illustrative purposes, only: the Veracruz H-call layer is explicitly not stable, and may change over time. As is the case with a typical operating system, programmers are not expected to invoke the trusted Veracruz runtime's H-calls directly from application code, but are instead expected to make use of an abstraction layer. With Unix-family operating systems, this abstraction layer is `libc`; for Veracruz, it is `libveracruz` which provides a stable layer over the Veracruz H-call API, and also provides higher-level functions, more amenable for use in typical application code, that build on top of the raw H-call layer.

At the time of writing, a version of `libveracruz` is only provided for the Rust programming language, and therefore all Veracruz programs must be written in Rust — or rather, it's only convenient to write Veracruz programs in Rust without manually invoking the unstable H-call API. Support for other languages, including C and C++, is planned but not a priority at present.

In the next subsection, we walk through a simple Veracruz program, written using `libveracruz`, and make reference to various notable aspects of the Veracruz programming model.

Case-study: linear regression

We will now walk through a simple Veracruz program, built using `libveracruz`, which expects a single data source of binary-encoded 64-bit floating point values as input. The program will then perform a linear regression on the input data set, before returning a binary-encoded answer as its result. Note that this example is both complete and executable on Veracruz, and can be found in the `$VERACRUZ/sdk/examples/linear-regression` directory.

```
#![no_std]
extern crate alloc;
extern crate veracruz_rt;
use alloc::vec::Vec;
use libveracruz::{generate_write_result, host, return_code};
use serde::Serialize;
```

For this example, we do not need to link against the full Rust standard library, `std`, but rather a subset of this library called `alloc` which assumes the presence of a global allocator, and provides a host of useful data structures that need to allocate memory. As a result, we mark the code as `#![no_std]` using the attribute on the first line, above. Veracruz supports programming with or without the Rust standard library, as is appropriate. In `!#[no_std]` contexts, we also provide a support library, `veracruz_rt`, which can be linked against the program, and sets up a global allocator and other low-level tasks, for convenience.

```
fn read_input() -> Result<Vec<(f64, f64)>, i32> {
    if host::input_count() != 1 {
         return return_code::fail_invariant_failed();
    } else {
        let dataset: Vec<u8> = host::read_input(0).unwrap();
        match pinecone::from_bytes(&dataset) {
            Err(_err) => return_code::fail_bad_input(),
            Ok(dataset) => Ok(dataset)
        }
    }
}
```

Above, the program uses functions exported from the `host` module of `libveracruz` to first query the number of inputs that have been provisioned by the data providers, and are therefore available to it. In this instance, the program is hardwired to expect a single input — other programs can work parametrically in the number of inputs, as appropriate — and returns a suitable error code if this is not the case.

Once this test is completed, the program grabs the raw bytes of the first input (at index 0) from the Veracruz runtime. This step should never fail, as we've already checked that an input is available, so the call to `read_input(0).unwrap()` is safe. In our examples and test code, we've fixed on using the Rust `pinecone` library as a way of serializing and deserializing Rust data structures to-and-from raw bytes. We therefore try to deserialize the collection of pairs of 64-bit floats, returning an error code if this fails.

Generally speaking, the Veracruz runtime doesn't care how inputs and outputs are serialized, as all it sees is unstructured byte data and any structured imposed on these bytes is something computation participants need to negotiate out-of-band, before the computation begins.

Below, we introduce a simple data structure that will capture the result of our linear regression algorithm, if it is successful:

```
#[derive(Serialize)]
struct LinearRegression {
    gradient: f64,
    intercept: f64,
}

generate_write_result!(LinearRegression);
```

Above, the `generate_write_result!` macro automatically generates serialization code, and handles the H-call that writes a result to the Veracruz runtime. We pass it the type of the return result, as parameter, and in response it automatically synthesizes a function `write_result()` accepting a `LinearRegression` struct as its argument.

Note that, above, we use the `Serialize` trait with the `LinearRegression` struct. This is a standard trait from the Rust Serde library, a common framework for serializing and deserializing data structures. Moreover, earlier, we used the `pinecone` library, which again is another "off-the-shelf" Rust library. Any Rust library that can be compiled to WASM can be used when programming Veracruz computations — to a first approximation, this means any Rust library written in pure Rust, and not depending on external C libraries, or any library not depending on operating system-specific functionality, such as devices or filesystems, can be used with Veracruz.

The code below is the meat of the linear regression implementation. This is straightforward, not tied to Veracruz, and is left here for completeness. We offer no comment on this code.

```
fn means(dataset: &[(f64, f64)]) -> (f64, f64) {
    let mut xsum: f64 = 0.0;
    let mut ysum: f64 = 0.0;
    let length = dataset.len();
    for (x, y) in dataset.iter() {
        xsum += *x; ysum += *y;
    }
    (xsum / length as f64, ysum / length as f64)
}
fn linear_regression(data: &[(f64, f64)]) -> LinearRegression {
    let (xmean, ymean) = means(&data);
    let mut n: f64 = 0.0;
    let mut d: f64 = 0.0;
    for datum in data {
        n += (datum.0 - xmean) * (datum.1 - ymean);
        d += (datum.0 - xmean) * (datum.0 - xmean);
    }
    LinearRegression {
      gradient: n / d,
      intercept: ymean - (n / d) * xmean,
    }
}
```

We now come to the entry point of the Veracruz computation. Like all standard Rust and C programs, a Veracruz program uses `main()` as its entry:

```
fn main() -> return_code::Veracruz {
    let data = read_input()?;
    let result = linear_regression(&data);
    write_result(result)
}
```

Our `libveracruz` offers a range of pre-baked error codes that can be used for signalling failures, of different kinds, of a Veracruz computation. We've already seen some above in `read_inputs()`. These errors are "floated up" to `main()`, and used as the return code of the program. The Veracruz runtime can recognize these error codes, and forward them as appropriate to participants of the computation.

The `main` function above captures the typical form of a Veracruz computation:

1. Inputs are checked, grabbed from the runtime, and deserialized.
2. The main algorithm is run on the deserialized inputs, with the result serialized into the correct, negotiated form.
3. The result is written back to the Veracruz runtime. Here, this "write back" step is achieved by calling the synthesized `write_result()` function, discussed above, with the result.

Veracruz programs are compiled for a custom 32-bit WASM compile target, called `wasm32-arm-veracruz`. A target specification file is provided as part of the Veracruz SDK.

Testing with our freestanding execution engine

It would be awkward to have to develop and debug a program, like the one above, in a distributed setting. As a result the Veracruz SDK provides a freestanding version of the execution engine used inside the Veracruz runtime for offline testing before deployment. This can be found in `$VERACRUZ/sdk/freestanding-chihuahua`. We can execute our linear regression example, above, using this as follows:

```
$ RUST_LOG=info
  ./target/release/freestanding-chihuahua
      --program  ../examples/linear-regression/target/wasm32-arm-
veracruz/release/linear-regression.wasm
    --data ../datasets/melbourne-houses-distance-price.dat
    --execution-strategy jit
```

The program successfully executes, returning a success error code, and produces 16 bytes of `pinecone`-encoded data corresponding to a `LinearRegression` structure.

Random number generation

Many prospective Veracruz programs depend on the ability to generate random numbers, and Rust provides a series of programmer-friendly libraries for working with random number sources, generating collections of random numbers, and working with different distributions of random numbers.

As part of the Veracruz SDK, we have ported the standard Rust `getrandom` and `rand` crates to the `wasm32-arm-veracruz` target, making these libraries available to programmers wishing to target Veracruz. Note that our fork of `getrandom` ultimately calls out to the `getrandom()` H-call provided by the Veracruz runtime described above.

Why not WASI?

The **WebAssembly System Interface** (WASI, henceforth) aims to provide a POSIX-style interface to system resources for WASM programs. We do not use this interface, instead preferring our own, as described above as for our purposes WASI is overkill. A Veracruz program is relatively constrained, reading inputs, processing them, and writing outputs. It cannot open files on a delegate's machine, by design, so large chunks of the WASI API are completely irrelevant, for Veracruz's purposes.

Whilst we could stub out the aspects of WASI that are irrelevant for our purposes, we instead take an alternative approach, beginning with a minimalist host API, and slowly adding features when there is a demonstrated need for them.

How does Veracruz compare to...

Several other commercial and academic projects exist in the broad area of privacy-preserving compute, using hardware-enclaves and remote attestation. In this section, we draw comparisons between these other projects and Veracruz.

Microsoft's OpenEnclave

Veracruz and OpenEnclave differ quite markedly, both in their design goals and in their implementation.

OpenEnclave is intended as a thin layer over different low-level APIs, such as Intel's SGX SDK, which allows programmers to develop enclave-oriented applications, across multiple enclave implementations, using a consistent and uniform API and a single set of associated tools. Programmers using OpenEnclave still need to design communication, provisioning, and attestation protocols on top of OpenEnclave, as these are application-specific.

In contrast, Veracruz is a platform for facilitating secure multi-party computations amongst a range of mutually distrusting principals. To support this, Veracruz fixes a particular communication and attestation protocol that is made to support the use-cases that we care about. Moreover, whilst Veracruz also provides a uniform programming model across multiple strong container technologies, this programming model is far more restrictive than that offered by OpenEnclave, as this is again optimized for the particular use-cases that we have in mind.

Lastly, we observe that Veracruz could be built on top of OpenEnclave. At present, we use a range of different low-level APIs when developing Veracruz — including Baidu SDKs for SGX and TrustZone. Instead, Veracruz could make use of OpenEnclave as a hardware-abstraction layer.

Enarx

Enarx is perhaps the closest project to Veracruz in terms of both design goals and implementation. Both projects make use of WASM code running inside a strong container. However, there are still differences both in the design and implementations of the two projects.

First, the Enarx project's focus is on providing confidentiality for existing workloads on third-party computing services. In contrast, the Veracruz project's focus is providing a framework for developing secure, collaborative computations between mutually untrusting principals. These two differing visions have an effect on design decisions taken in the two projects: for instance, Veracruz requires a notion of global policy, and a defined provisioning protocol built around the idea that multiple parties will be provisioning code and data into the trusted Veracruz runtime, that is unneccessary in Enarx.

Moreover, note that Veracruz has a relative paucity of services provided to the WASM program by the trusted Veracruz runtime. This is a key design decision in enabling secure multi-party computations wherein principals may be actively trying to steal the secrets of others. In contrast, Enarx's plan of record is to support the WASI host interface, which is implemented on top of a syscall-proxying mechanism wherein syscalls are handled by the host operating system. This latter observation also highlights another point of departure between the two projects: Enarx is built around the idea of protecting existing code, and therefore needs to provide a comprehensive syscall interface to support the full breadth of deployed programs, whereas Veracruz aims to protect a single WASM application that for the most part will be written specifically for use with Veracruz.

Aside from broader design differences between the two projects, there also exist implementation differences. First, Enarx at the time of writing supports deployment on top of **AMD Secure Encrypted Virtualization** (or SEV, henceforth) protected virtual machines as its isolation technology. In contrast, Veracruz uses Intel SGX, Arm TrustZone, and hypervisor-based isolation through seL4. In theory, Veracruz on AMD SEV could be supported, though at the time of writing we have no concrete plans to do so. Note that these implementation differences also have an effect on the trusted computing base of the two projects: Enarx under AMD SEV needs a trusted micro-kernel hosting the WASM/WASI runtime inside the SEV isolate. For some backends, e.g. Intel SGX, Veracruz does not need this trusted micro-kernel at all, and simply relies on the hardware's security guarantees to protect the Veracruz runtime. In contrast, Veracruz on Arm TrustZone is indeed running under a trusted operating system, in OP-TEE.

What's next for Veracruz?

Veracruz is an active research project, within Arm Research, and we have several research strands that we wish to pursue over the near term. Here, we highlight just three of our thoughts for future work. We warn that these ideas enumerated below are not firm commitments to new features, nor are they exhaustive, but reflect some of the current thoughts of the Veracruz team.

Streaming data

At the moment, the design of Veracruz requires that all input data be provisioned up-front by the Data Owners into the isolate on the delegate's machine. Moreover, the Veracruz programming model sees the WASM program, running on top of the trusted Veracruz runtime, copy all of an input into a WASM buffer before it can be used.

Several weaknesses with this approach are apparent, not least the fact that some strong isolation technologies, notably Intel's SGX Secure Enclaves, put an (architectural) upper limit on the memory footprint of the enclave, which in practice means that the footprint of any enclaved application cannot exceed around 96 megabytes. Other isolate technologies, like Arm TrustZone, have no architectural upper limit but in practice deployments of these technologies tend to have tight upper limits. Whilst paging and other techniques can ameliorate some of these problems, they tend to incur a large performance penalty.

An alternative model would be to offer a **streaming** model of computation for the Veracruz platform — in addition to the current **batch** model — wherein data is fed in chunks into the delegate's isolate, processed, and then fed back out in a chunk as output.

Whilst not every algorithm can easily be converted into a streaming version, for algorithms that can this approach would allow a Veracruz computation to handle much larget input data sets than it can, currently.

Larger, more complex use-cases

All of the Veracruz use-cases outlined in §What are some Veracruz use-cases? use a single isolate running a Veracruz runtime. Yet, we can imagine many other use-cases wherein one Veracruz instance feeds its output into other instances as an input, with isolates organized into a network topology.

One example of this kind of distributed Veracruz computation is **map-reduce**, wherein large monolithic computations on data sets are split into two steps: a "map" step, wherein a fixed function f is applied (in parallel) to chunks of the data set, before the outputs of these parallel transformations-under-f are fed into a "reducer" step, which applies a final transformation, r, to produce a single output. Following this pattern, a "privacy-preserving map-reduce" is easily imagined, wherein Veracruz "mapper nodes" feed their outputs into a Veracruz "reducer node", with the transformations f and r being applied to the data, and the data set itself, kept secret by Veracruz.

Provisioning and managing Veracruz instances

With larger, more complex Veracruz use-cases, as in the privacy-preserving map- reduce example mentioned above, we face a potential problem: how to adequately orchestrate a fleet of Veracruz isolates? Once the number of isolates becomes significant, it becomes infeasible to do this by hand, and specialized orchestration mechanisms need to be developed. Specifically, mapping computations onto idle Veracruz isolates, monitoring when a Veracruz isolate die, and bringing it back up, handling remote attestation and the rest of the Veracruz protocols, are all Veracruz-specific challenges that an orchestration mechanism will have to handle.

Who wrote Veracruz?

Veracruz originated in the Security Research Group of Arm Research. The main authors of Veracruz are:

- Derek Miller, Arm Research, Austin, Texas — derek.miller@arm.com,
- Dominic Mulligan, Arm Research, Cambridge, UK — dominic.mulligan@arm.com,
- Nick Spinale, Arm Research, Cambridge, UK — nick.spinale@arm.com,
- Shale Xiong, Arm Research, Cambridge, UK — shale.xiong@arm.com.