Private delegated computations using strong isolation

Mathias Brossard[†], Guilhem Bryant[†], Basma El Gaabouri[†], Xinxin Fan[⋆], Alexandre Ferreira[†], Edmund Grimley-Evans[†], Christopher Haster[†], Evan Johnson[‡], Derek Miller[†], Fan Mo[¶], Dominic P. Mulligan[§], Nick Spinale[†], Eric van Hensbergen[†], Hugo J. M. Vincent[†], Shale Xiong[†]
[†]Systems Research Group, Arm Ltd., Cambridge UK & Austin, TX
[§]AWS [⋆]IoTeX.io [‡]University of California, San Diego [¶]Imperial College London

Abstract—Computations are now routinely *delegated* to third-parties. In response, *Confidential Computing* technologies are being added to microprocessors offering a *trusted execution environment* (*TEE*) that provides confidentiality and integrity guarantees to code and data hosted within—even in the face of a privileged attacker. TEEs, along with an attestation protocol, permit remote third-parties to establish a trusted "beachhead" containing known code and data on an otherwise untrusted machine. Yet, they introduce many new problems, including: how to ease provisioning of computations safely into TEEs; how to develop distributed systems spanning multiple classes of TEE; and what to do about the billions of "legacy" devices without support for Confidential Computing? Tackling these problems, we introduce *Veracruz*, a pragmatic framework that eases the design and implementation of complex privacy-preserving, collaborative, delegated computations among a group of mutually mistrusting principals. Veracruz supports multiple isolation technologies and provides a common programming model and attestation protocol across all of them, smoothing deployment of delegated computations over supported technologies. We demonstrate Veracruz in operation, on private in-cloud object detection on encrypted video streaming from a video camera. In addition to supporting hardware-backed TEEs—like AWS Nitro Enclaves and Arm® Confidential Computing Architecture Realms—Veracruz also provides pragmatic "software TEEs" on Armv8-A devices without hardware Confidential Computing capability, using the high-assurance *seL4* microkernel and our *IceCap* framework.

Index Terms—Confidential Computing, Trusted Execution Environments, Veracruz, IceCap, seL4, software enclaves, attestation

1 Introduction

Code and data are now routinely shared with a *delegate* who is better placed to host a computation. While Cloud computing is the obvious exemplar of this trend, other forms of distributed computing—including volunteer Grid Computing, wherein machines lend spare computational capacity to realize some large computation, and Ambient Computing, wherein computations are *mobile* and hop from device-to-device as computational contexts change—also see computations freely delegated to third parties.

At present, in the absence of the widespread deployment of Advanced Cryptography, delegating computations to a third party inexorably means entering into a trust relationship with the delegate, and for some especially sensitive computations this may be simply unacceptable. Yet, even for less sensitive delegated computations, there is still an interest in limiting the scope of this trust relationship. In the Cloud context, though hosts may be reputable, technical means may be desired to shield computations from prying or interference which may originate from many sources, not only from the hosting company: malefactors may exploit hypervisor bugs to spy on co-tenants, for example. Cloud hosts also see an interest in *deniable hosting*, wherein technical measures ensure that a customer's computations simply cannot be interfered with, or spied upon, by the hosts themselves—even in the face of legal compulsion. For Ambient and volunteer Grid Computing, nodes must be assumed hostile and assumed to be trying to *undermine* a computation, either through malice or as a consequence of bugs or transient glitches. As a result, volunteer Grid Computing deployments may schedule computations on multiple independent nodes and check for consistency.

In response, novel *Confidential Computing* technologies are being added to microprocessor architectures and Cloud infrastructure, providing protected computing environments—which we call trusted execution environments, or TEEs—that provide strong confidentiality and integrity guarantees to code and data hosted within, even in the face of a privileged attacker. TEEs are also typically paired with an *attestation* protocol, allowing a third-party to deduce, with high confidence, that a remote TEE is authentic and correctly configured. Taken together, one may establish a protected "beachhead" on an untrusted third-party's machine to protect delegated computations.

TEEs offer a range of benefits, namely allowing programmers to design arbitrarily complex privacy-preserving distributed systems using standard tools and programming idioms that run at close to native speed. Moreover, compared to cryptographic alternatives, Confidential Computing technology is available for use and deployment in real systems *today*. Yet, the emergence of Confidential Computing technology poses some interesting problems.

First, some TEE implementations have unfortunately fallen short of their promised guarantees. A substantial body of work, demonstrating that side-channel (see e.g., [1] amongst others) and fault injection attacks (see e.g., [2] amongst others) can be used to exfiltrate secrets from TEEs,

§. All work by Mulligan done whilst employed with Arm Research

now exists, and a perception—at least in the academic community and technical press—appears to be forming that TEEs are fundamentally broken and any research that builds upon them need necessarily justify that decision. We argue that this perception is an instance of *the perfect being the enemy of the good*, as many identified flaws will be gradually ironed out over time, either in point-fixes, iterated designs, or by the adoption of software models that avoid known vulnerabilities. Moreover, industrial adoption of TEEs will be widespread, arguably already in evidence with the formation of consortia such as the LF's *Confidential Computing Consortium*, and an ecosystem of industrial users who pragmatically evaluate TEEs in comparison with the *status quo*, where delegated computations are—by and large—left completely unprotected. It is *this* standard which should be applied when evaluating systems built around TEEs, not comparison with side-channel free Advanced Cryptography, still impractical in an industrial context.

Second, TEEs simply provide an empty, albeit secure, container. Associated questions like how computations are securely provisioned, how to make this process foolproof and straightforward, and how systems are designed and built around TEEs as a new primitive, are left unanswered. Moreover, for some distributed system such as Grid and Ambient computing systems it is feasible that *different* types of TEE will be used within a single larger system. Here, bridging differences in attestation protocol and programming model will be key, as will be easing deployment and scheduling of computations hosted within TEEs.

For this reason, we introduce our main research contribution: *Veracruz* (see §3), a framework that abstracts over TEEs and their attestation processes, supporting multiple different isolation technologies, including hardware-backed TEEs like AWS Nitro Enclaves and Arm Confidential Computing Architecture Realms (on a private branch). Adding support for more technologies is straightforward.

With Veracruz, our main contribution is a practical and pragmatic framework that allows non-experts to make use of TEE technologies, abstracting over their more complex aspects—attestation and secure provisioning of code and data, for example. In particular, Veracruz provides a uniform programming model across different TEEs—using WebAssembly (Wasm) [3]—and a generalized form of attestation, providing a "write once, TEE anywhere" style of development. As a result, programs can be protected using *any* supported isolation technology without recompilation.

Veracruz's portability and support for a wide range of TEE-like technologies, each with their own strengths and weaknesses, allows users to match their needs to a particular TEE technology, and even quickly swap technologies when these needs change. However, users must be aware of the inherent weaknesses of their chosen isolation technology when deploying computations with Veracruz, as there are some threats—for example, hardware side-channel attacks—that Veracruz simply cannot practically solve uniformly in software without becoming overly unwieldy or inefficient for real-world use-cases. With Veracruz, we make pragmatic decisions over what threats to try to counter, and how, and our threat model is discussed in §3.4.

Veracruz captures a general form of interaction between mistrusting parties, and is easily specialized to obtain an array of delegated, privacy-preserving computations of interest. To support this, we describe how Veracruz can be used for secure ML model aggregation, and an industrial case-study built around AWS Nitro Enclaves, demonstrating an end-to-end encrypted video decoding and object-detection flow, using deep learning to process video obtained from an IoT camera (see §4).

Lastly, *billions* of existing devices have been shipped without explicit support for Confidential Computing, and these will be used for years to come. Is there some *pragmatic* isolation mechanism that we could use on "legacy" devices which, while falling short of the confidentiality and integrity guarantees offered by hardware-backed TEEs, can yet provide believable isolation for workloads? In response, we introduce a second research contribution: *IceCap* (see §2), a pragmatic "software TEE" for Armv8-A devices without explicit support for Confidential Computing, using the high-assurance *seL4* microkernel to provide confidentiality and integrity for VMs, with little overhead. IceCap is supported by Veracruz, and taken together, one may design and deploy delegated computations across hardware- and software-TEEs on next-generation and legacy hardware, alike.

2 IceCap

IceCap is a hypervisor with a minimal trusted computing base (TCB) built around the formally verified *seL4* microkernel. Admittedly, IceCap cannot provide as strong a security promise as hardware TEEs, such as Arm Confidential Computing Architecture (Arm CCA), but it does provides a *pragmatic*, *flexible* and *better-than-nothing software TEE* for many existing Armv8-A devices. The IceCap hypervisor relegates the untrusted operator to a domain of limited privilege called the host. This domain consists of a distinguished virtual machine—housing a rich operating system such as Linux—and a minimal accompanying virtual machine monitor. The host domain manages the device's CPU and memory resources, and drives device peripherals which the TCB does not depend on. This includes opaque memory and CPU resources for confidential virtual machines—or TEEs. However, the host does not have the right to access the resources of TEEs—while scheduling and memory management *policy* is controlled by the host, *mechanism* is the responsibility of more trustworthy components.

IceCap's TCB includes the seL4 microkernel and compartmentalized, privileged seL4-native services running in `EL0`—the least privilege level for AArch64. These co-operate defensively with the host to expose the TEE lifecycle, scheduling, and memory management mechanisms.

At initialization, the hypervisor extends from the device's root of trust via a device-specific measured boot process and then passes control to the untrusted host domain. A remote party coordinates with the host to spawn a new TEE by sending a declarative specification of the TEE's initial state to IceCap's *trusted spawning service*, via the host, which then carves-out the requested memory and CPU resources from resources which are inaccessible to the host. A process on the host, called the *shadow virtual machine monitor*, provides untrusted paravirtualized device backends to TEEs, and also acts as a *token* representing the TEE in the host's

scheduler allowing the host operating system to manage TEE scheduling policy with minimal modification.

To support attestation of TEEs, IceCap would use a platform-specific measured boot to prove its own identity and then attest that of an TEE to a remote challenger. This is not yet implemented, with IceCap attestation stubbed to support Veracruz. It is straightforward to implement.

seL4 is accompanied by security and functional correctness proofs, checked in *Isabelle/HOL* [4], [5], [6], providing assurance that IceCap correctly protects TEEs from software attacks. By using seL4, IceCap will also benefit from ongoing research into the elimination of certain classes of timing channels [7]. The trusted seL4 userspace components of IceCap are not yet verified, though they are compartmentalized and initialized using *CapDL* [8], which has a formal semantics known to be amenable to verification from previous work. Using the high-level seL4 API, these components are also implemented at a high level of abstraction in Rust, making auditing easier and eliminating the need to subvert the Rust compiler's memory safety checks—even for components which interact with hardware address translation structures. The IceCap TCB is small and limited in scope—about 40,000 lines of code. Virtual machine monitors are moved to the trust domains of the virtual machines they supervise, thereby eliminating emulation code from the TCB. Towards that end, cross-domain fault handling is replaced with higher-level message passing via seL4 IPC.

TEEs are also protected with the System MMU (SMMU) from attacks originating from peripherals under the host's control. IceCap is designed to seamlessly take advantage of hardware security features based on address translation-based access controls—Arm TrustZone, for example. TrustZone firmware typically uses the `NS` state bit to implement a coarse context switch, logically partitioning execution on the application processor into two *worlds*. IceCap could use this to run TEEs out of secure-world memory resources, protected by platform-specific mechanisms which may mitigate certain classes of physical attack.

Under IceCap, TEE and host incur a minimal performance overhead compared to host and guests under *KVM* [9]. We use *Firecracker* [10]—an open-source VMM for KVM from AWS—as a point of comparison, due to its minimalism for the sake of performance, and preference for paravirtualization over emulation. Compute-bound workloads in IceCap TEEs incur a $\sim$2.2% overhead compared to native Linux processes and a $\sim$1.8% overhead compared to Firecracker guests due to context switches through the TCB on timer ticks (see Table 1). The virtual network bandwidth between host and TEE represents how data flows through IceCap in bulk. However, at the time of writing, untrusted network device emulation differs from Firecracker's trusted network device emulation in ways that hinder a satisfying comparison, and with this in mind, we note guest-to-host incurs a $\sim$9.9% bandwidth overhead, whereas host-to-guest outperforms Firecracker by a small margin. As IceCap's implementation matures, we expect virtual network bandwidth overhead to settle between these two points.

The great performance of seL4 IPC helps reduce IceCap's performance overhead, further helped by minimizing VM exits using aggressive paravirtualization: VMMs for both host and guest do not even map any of their VMs' memory into their address spaces—their only runtime responsibility is emulating the interrupt controller, with their VMs employing interrupt mitigation to even avoid that.

	Events per second (via `sysbench`)	
	Host	Guest
Firecracker	586.18	582.65 (-0.60%)
IceCap	583.68 (-0.43%)	572.28 (-2.18%)
	Bandwidth (Gbits/sec)	
	Guest $\rightarrow$ Host	Host $\rightarrow$ Guest
Firecracker	3.42	3.14
IceCap	3.08 (-9.9%)	3.18 (+1.3%)

Table 1: Overheads for IceCap compute-bound workloads (top) and virtual network performance (bottom)

Next, we introduce a framework for designing and deploying privacy-preserving delegated computations across various different isolation technologies—IceCap included.

3 Veracruz

Throughout this section we make reference to the system components presented in the schematic in Fig. 1. In particular, we make reference to components of the global policy, presented on the left of the diagram, in the text below.

Veracruz is a *framework* which may be specialized to obtain a particular privacy-preserving, collaborative computation of interest. A Veracruz computation involves an arbitrary number of **data owners**, trying to collaborate with multiple **program owners**. The framework places no limits on the number of **principals** collaborating, but a particular computation obtained by specializing Veracruz will always spell out a precise number, Ⓟ. We use π_m to denote the program of one of the P program owners, and use D_i for $1 \leq i \leq N$ to denote the data sets of the various data owners in an arbitrary Veracruz computation.

Collectively, the goal of the various principals, Ⓟ, is straightforward: they wish to compute the value $\pi_m(D_1, \ldots, D_N)$, that is, the value of the program π_m applied to the N inputs of the various data owners. To do this, they may choose to make use of a third party machine to power the computation, Ⓓ. We refer to the owner of this machine as the **delegate**, assumed capable of launching an TEE of a type that Veracruz supports, loaded with the Veracruz **trusted runtime**, Ⓥ. This runtime acts as a "neutral ground" within which a computation takes place, and provides strong **sandboxing** guarantees to the delegate, who is loading untrusted code in the form of π_m, onto their machine. The runtime is open-source, auditable by principals assuming bit-for-bit reproducible builds.

Each principal in a Veracruz computation has a mixture of **roles**, consisting of some combination of **data provider**, **program provider**, **delegate**, and **result receiver**. While the first three have been implicitly introduced, the latter role refers to principals who will receive the result of the computation. The identification details of each principal, in the form of cryptographic certificates (or an IP address for the delegate), and their mixture of roles, is captured in a public **global policy** configuration file, which parameterizes

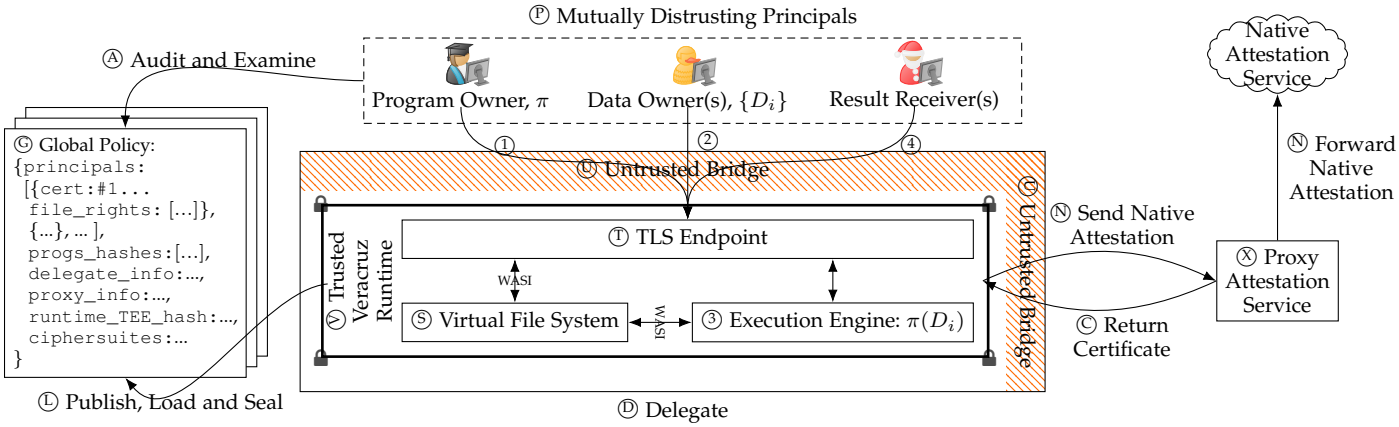

Figure 1: An overview of an abstract Veracruz computation, showing principals and their roles, major system components, and a suggestive depiction of data-flow. TEEs, that host the Veracruz runtime, are marked with boxes with padlocks.

each computation, and which also contains other important bits of metadata. Only one principal may be delegate.

The global policy, Ⓖ, captures the **topology** of a computation, specifying where information may flow from, and to whom, in a computation, while varying the program π_m varies precisely **what** is being computed. Veracruz stores data and programs as files in a **virtual**, **in-memory filesystem** (VFS) maintained by the Veracruz runtime, Ⓢ, and whose content never leaves the TEE, destroyed when the TEE is torn down. With VFS, the constraint on information flow is achieved by assigning different file *capabilities* (`file_rights`, a mapping from paths to capabilities) to different principals (`principals`). Program behaviours are indirectly constrained by agreed programs hashes (`progs_hashes`) in the policy; programs with incorrect or unexpected hashes cannot be executed by the Veracruz runtime. By varying topology and policy, Veracruz can capture a general pattern of interaction shared by many delegated computations, and one could, for example, effect a varied palette of computations of interest:

- *Moving heavy computations safely off a computationally-weak device to an untrusted edge device or server.* The computationally-weak device is both data provider and result receiver, the untrusted edge device or server is delegate, and the computationally-weak device or its owner is the program provider, providing the task to be performed.
- *Privacy-preserving machine learning between a pair of mutually distrusting principals with private datasets, but where learnt models are made available to both principals.* Both principals are data providers, contributing their datasets provided in some common format, and also act as result receivers for the learnt model. Arbitrarily one acts as the program provider, providing the machine learning algorithm of interest, and the delegate, e.g., a Cloud host.
- *A DRM mechanism wherein novel IP (e.g., computer vision algorithms) are licensed out on a "per use" basis, and where the IP is never exposed to customers.* The IP owner is program provider, and the licensee is both data provider and result receiver, providing the inputs to, and receiving the output from, the private IP. The IP owner themselves may act as delegate, or this can be contracted out to a third-party. With this, the IP owner never observes the input or output of the computation, and the licensee never observes the IP.

In addition, it is easy to see how more complex distributed systems can be built *around* Veracruz. For example, a volunteer Grid computing framework where confidentiality is not paramount, but computational integrity is; an Ambient computing runtime for mobile computations across a range of devices; a privacy-preserving MapReduce [11] or Function-as-a-Service (FaaS, henceforth) style framework. Here, computational nodes act as an independent delegate for some aspect of the wider computation, and *different* isolation technologies may also be used in a single computation, either due to availability for Grid or Ambient computing, or due to scheduling of sensitive sub-computations onto stronger isolation mechanisms for MapReduce.

In the most general case, each principal in a Veracruz computation is mutually mistrusting, and does not wish to **declassify** their data: data providers do not wish to divulge their input datasets and the program provider does not wish to divulge their program. We are not awared of any practical and general solution to directly bounding the program behaviours, therefore, in this case, principals need to blindly trust the program hashes in the global policy. Nevertheless, as the examples enumerated above indicate, for some computations declassification *can* be useful, for example as inducement to other principals to enroll in the computation, a public "nothing up my sleeve" demonstration. Referring back to the privacy-preserving machine learning use-case, above, a program provider may *intentionally* declassify their program for auditing—before other principals agree to participate—as a demonstration that the program implements the correct algorithm, and will not (un)intentionally leak secrets. Similarly, for a Grid computing project, revealing details of the computation, as an enticement to users to donate their spare computational capacity, may be beneficial. Declassification can also occur as a side effect of the computation, for example when the result—which can reveal significant amounts of information about its inputs, depending on π_m—is shared with an untrusted principal. Principals must evaluate the global policy carefully, before enrolling, to understand where results will flow to, and what they may say about any secrets. Though Veracruz can be used to design privacy-preserving computations, not *every* computation is necessarily privacy-preserving.

Once a TEE is spawned, with the Veracruz runtime loaded, the program and data owners establish a TLS connection, using a modified TLS handshake (see §3.1), with the TEE Ⓣ. This handshake assures the principals that the TEE is, in fact, executing the Veracruz runtime specified in the global policy, and that the TEE is the other end of their TLS connection. Once this TLS channel is established, the program and data providers use it to **provision** their secrets directly into the TEE, ① and ②, making use of an untrusted bridge, Ⓤ, on the delegate's machine but outside of the TEE, to forward encrypted TLS data into the TEE itself. To the delegate, communication via this bridge is immutable and opaque—except for sizing and timing information that TLS leaks—unless they can subvert TLS. TLS configuration options, including permitted TLS ciphersuites, are also publicly detailed in the global policy.

Provisioned secrets are stored as files in the VFS, Ⓢ. The paths of data inputs, D_i, are specified in the global policy, Ⓖ, as the program π_m needs to know where its inputs are stored for processing when the computation starts executing. Similarly, each program is also stored as a file, read from the filesystem during loading.

A result receiver may now request the result of the computation, triggering the Veracruz runtime to load the provisioned program, π, into the execution engine, Ⓢ, and either compute the result $\pi_m(D_1, \ldots, D_N)$, terminate with an error code, or diverge. Assuming a result is computed, it is stored by the program as a file in the filesystem at a path specified by the global policy. The runtime reads this path, or fails with an error if the program did not write a result there, and makes the result retrievable securely, via TLS, to all result receivers, ④. The computation is now complete.

3.1 Attestation

Given Veracruz supports multiple isolation technologies, this poses a series of attestation-related problems:
- *Complex client code*: Client software used to delegate a computation to Veracruz must support multiple attestation protocols, complicating it. As support for more TEEs are added—with new attestation protocols—this client code must be updated to interact with the new class of TEE.
- *Leaky abstraction*: Veracruz abstracts over TEEs, allowing principals to delegate computations without worrying about the programming or attestation model associated with any one TEE. Forcing clients to switch attestation protocols, depending on the TEE, breaks this uniformity.
- *Potential side-channel*: For some attestation protocols, each principal in a Veracruz computation must refer attestation evidence to an external attestation service.
- *Attestation policy*: Principals may wish to disallow computations on delegates with particular isolation technologies. These policies may stem from security disclosures—vulnerabilities in particular firmware versions, for example—changes in business relationships, or geopolitical trends. Given our support for heterogeneous isolation technologies, being able to declaratively specify who or what can be trusted becomes desirable. Without the attestation service taking policy into account this role is inevitably pushed onto client code—problematic if policy changes, as inevitably more software needs updating than the centralized attestation service.

In response, we introduce a **proxy attestation service** (PAS) for Veracruz, Ⓧ, open-source and auditable by anyone, with associated server and management software. Though PAS *must* be trusted by all principals together with the existing attestation infrastructure for a particular TEE in deployment, it is a practical solution to the listed problems. Whilst not protected by a TEE, in principle PAS could be, and doing so would allow principals to check its authenticity before trusting it. Implementing this would be straightforward; for now we assume that the attestation service is trusted, implicitly. The PAS first uses **onboarding** to enroll a TEE hosting Veracruz, after which the TEE can act as a TLS server for principals participating in a computation. We describe these steps, referring to Fig. 2.

Onboarding a TEE: The PAS maintains a root CA key (public/private key pair) and a Root CA certificate containing the root CA public key, signed by the root CA private key. This root CA certificate is included in the global policy file of any computation using that PAS. Then:

1) Upon initialization inside the TEE, the Veracruz runtime Ⓥ generates an asymmetric key pair, along with a *Certificate Signing Request* (CSR) [12] for that key pair.
2) The Veracruz runtime performs the platform's **native attestation request** Ⓞ① with the PAS acting as challenger Ⓧ. These native attestation flows provide fields for user-defined data, which we fill with a cryptographic hash (`SHA-256`) of the CSR, which cryptographically binds the CSR to the attestation data, ensuring that they both come from the same TEE. The Veracruz runtime sends the CSR to the PAS along with the attestation evidence.
3) The PAS **authenticates** the attestation evidence received via the native attestation flow, Ⓞ②. Depending on the protocol, this could be as simple as verifying signatures via a known-trusted certificate, or by authenticating the received evidence using an external attestation service.
4) The PAS computes the hash of the received CSR and compares it against the contents of the user-defined field of the attestation evidence. If it matches, it confirms that the CSR is from the same TEE as the evidence.
5) The PAS converts the CSR to an `X.509` Certificate containing a custom extension capturing details about the TEE derived from the attestation process, including a hash of the Veracruz runtime executing inside the TEE (and optionally other information about the platform on which the TEE is executing). The certificate is signed by the private component of the PAS's Root CA key.
6) The PAS returns the generated certificate to the Veracruz runtime inside the TEE.

In typical CA infrastructure, a delegated certificate is revoked by adding it to a *Certificate Revocation List*, checked by clients before completing a TLS handshake. This scheme is possible with our system, but we elected to set the expiry in the TEE's certificate to a relatively short time in the future, so that the PAS can limit the amount of time a compromised TEE can be used in computations.

Augmented TLS handshake: After a TEE is onboarded, Ⓞ①, a principal can attempt to connect to it, using an augmented TLS handshake, Ⓡ. In response to the "Client Hello" message sent by the principal, the TEE responds with a "Server Hello" message containing the certificate that the TEE re-

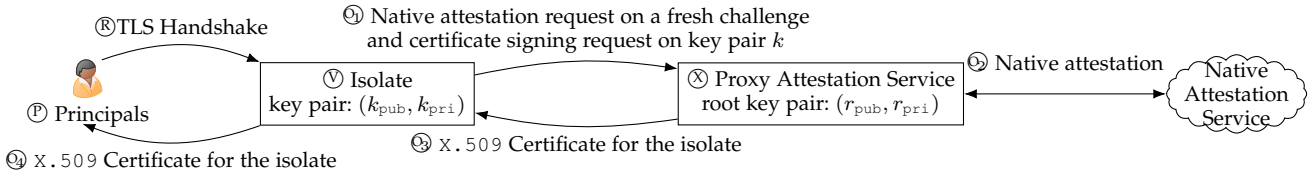

Figure 2: A schematic diagram of the Veracruz attestation service onboarding and challenge protocols

ceived from the PAS, described above. The principal then verifies if the certificate matches the PAS root CA certificate contained within the global policy. Assuming successful verification, the principal then checks the data contained in the custom extension, the Veracruz runtime hash, against the expected values in the global policy. If they match, the principal continues the TLS handshake, confident that it is talking to a Veracruz runtime inside of a supported TEE.

The PAS solves all problems described above. Client code is provided with a uniform attestation interface—the *CA-based attestation flow* described above —independent of the underlying isolation technology in use, with no principals in a computation communicating with a native attestation service. Thus, the native attestation service knows that software was started in a supported TEE, but it has no knowledge of the identities or number of principals. Finally, the global policy represents the only source of policy enforcement, with the authors able to declaratively describe who and what they are willing to trust, with a principal's client software taking this information into account when authenticating or rejecting an attestation token. Our attestation process is also designed to accommodate client code running on embedded microcontrollers—e.g., Arm Cortex®-M3 devices—with limited computational capacity, constrained memory and storage (often measured in tens of kilobytes), and which tend to be battery-powered with limited network capacity. Communication with an attestation service is therefore cost- and power-prohibitive, and using a certificate-based scheme allows constrained devices to authenticate an TEE running Veracruz efficiently. To validate this, we developed Veracruz client code for microcontrollers, using the Zephyr embedded OS. Our client code is 9 kB on top of the `mbedtls` stack, generally required for secure communication anyway. Using this, small devices can offload large computations safely to an attested Veracruz instance.

3.2 Programming model

Wasm [3] is designed as a *sandboxing* mechanism for use in security-critical contexts, designed to be embeddable, is widely supported as a target by a number of high-level programming languages, and has high-quality interpreters and JITs available. We use Wasm as our executable format, supporting both interpretation and JIT execution, with the strategy specified in the global policy. Veracruz uses Wasm to protect the delegate's machine from the executing program, to provide a *uniform* programming model, to constrain the behavior of the program, and to act as a *portable* executable format for programs, abstracting away the underlying instruction set architecture.

A program needs a way of reading inputs from data providers and writing outputs for the result receivers. However, we would like to constrain the behavior of the program: a program dumping one of its secret inputs to `stdout` on the host's machine would break the privacy guarantees that Veracruz aims to provide, for example. Partly for this reason, we have adopted the WebAssembly System Interface (Wasi) as our programming model, which can be thought of as "Posix for Wasm", providing a system interface for querying Veracruz's in-memory filesystem, generating random bytes, and similar. (In this light, the Veracruz runtime can be seen as a simple operating system for Wasm.) By adopting Wasi, one may also use existing libraries and standard programming idioms when targeting Veracruz. Wasi uses **capabilities** and a program may only use functionality which it has been explicitly authorized to use. The program, π_m's, capabilities are specified in the global policy, and typically extend to reading and writing inputs and outputs, and generating random bytes.

3.3 Ad hoc acceleration

Many Veracruz applications make use of common, computationally intense, or security-sensitive routines: e.g., cryptography or (de)serialization. It is beneficial to provide a single, efficient, and correct implementation for common use. We introduced "native modules" to accelerate common tasks, and which are linked into the Veracruz runtime and invoked from Wasm programs, although they increase the size of runtime. In benchmarking a module accelerating (de)serialization of `Json` documents from the `pinecone` binary format we observe a 35% speed-up when (de)serializing a vector of 10,000 random elements (238s native vs. 375s Wasm). More optimization will further boost performance. Given the *ad hoc* nature of these accelerators, we opt for an interface using **special files** in the Veracruz filesystem, with modules invoked by Wasm programs writing-to and reading-from these files, reusing existing programming idioms and filesystem support in Wasi.

3.4 Trust and threat model

Trusted Computing Base: The Veracruz TCB includes the TEE, the Veracruz runtime, and the implementation of the Veracruz PAS. The host of the PAS must also be trusted by all parties, as must the native attestation services or keys in use. The correctness of the various protocols in use—TLS, platform-specific native attestation, and PSA attestation [13]—must also be trusted. Purely cryptographic techniques merely rely on a trustworthy implementation, and the correctness of the primitive itself. As demonstrated in §4, Veracruz provides a degree of efficiency and practicality currently out of reach for purely cryptographic techniques, at the cost of this larger TCB.

The Wasm execution engine must also be trusted to correctly execute and sandbox a binary. Recent techniques have been developed that use post-compilation verification to establish this trust [14]—we discuss ongoing experiments in this area in §5. Memory issues have been implicated in attacks against TEEs in the past [15]—we write Veracruz in Rust in an attempt to avoid this, with the compiler therefore also trusted. Veracruz does not defend against denial-of-service attacks: the delegate is in charge of scheduling execution, and liveness guarantees are impossible to uphold. A malicious principal can therefore deny others access to a computation's result, or block a computation from starting.

Different isolation technologies defend against different classes of attacker, and as Veracruz supports multiple technologies we must highlight these differences explicitly.

AWS Nitro Enclaves protect computations from the AWS customer running the EC2 instance associated with the TEE. While AWS assures users that TEEs are protected from employees, this is difficult to validate (and, as silicon manufacturer, AWS and its employees must be trusted). Our TCB therefore contains the Nitro hardware, Linux host used inside the TEE, the attestation infrastructure for Nitro Enclaves, and AWS employees with potential access.

For Arm CCA Realms only the *Realm Management Monitor* (RMM), a separation kernel isolating Realms from each other, has access to the memory of a Realm, other than the software executing in the Realm itself. Realms are protected from the non-secure hypervisor, and any other software running on the system other than the RMM, and will be protected against a class of physical attacks using memory encryption. Our TCB therefore contains the RMM, the system hardware, Linux host inside the Realm, along with the attestation infrastructure for Arm CCA.

For IceCap our TCB includes seL4, bolstered by a body of machine-checked security and functional correctness proofs (at present these do not cover the `EL2` configuration for AArch64). For a typical hypervisor deployment of seL4, the SMMU is the only defence against physical attacks.

Threats and mitigations: Communication between principals and the Veracruz runtime is protected by a modified TLS protocol, preventing data from being viewed or modified in transit to the runtime. Additionally, the participants must authenticate themselves to the Veracruz runtime with self-signed client certificates that are included in the global policy file. This prevents unauthorized parties from impersonating principals. The server certificate presented to the principals in the TLS handshake contains a custom extension that contains the SHA256 hash of the Veracruz runtime loaded in the TEE. The client check this hash value against the value embedded in the global policy file to ensure they are communicating with a valid instance of the Veracruz runtime.

Prior to provisioning data into the runtime, principals request a hash of the loaded Wasm program from the runtime, and compare it against a value embedded in the global policy file. This ensures that only Veracruz runtimes executing the chosen program may receive the principal's confidential data. TEEs protect the data and computation once it is provisioned into the runtime.

For Arm CCA, software outwith the root of trust has no mapping from virtual address to hardware address to the TEE's memory, and therefore no ability to modify or read the TEE's data. The TEE's DRAM is also encrypted, so snooping the system buses by the operator will not leak secrets. Some CCA platforms may also provide authenticated encryption, which would prevent modification by the system operator. For Icecap, software outside of the root of trust has no ability to access TEE memory. However, there is no DRAM encryption, so the system operator could probe the system buses to read or modify the TEE memory. As discussed, AWS asserts that policies preventing access to customer data are enforced, though this is impossible to verify.

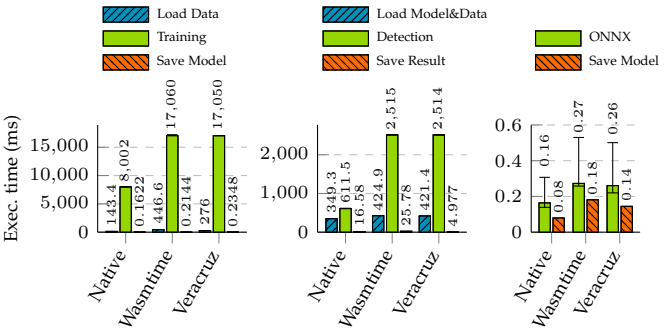

Figure 3: Execution time of the DL examples, classifier training (L), inference (M), *ONNX* model aggregation (R)

4 Evaluation

This section uses the following test platforms: Intel Core i7-8700, 16 GiB RAM, 1 TB SSD (*Core i7*, henceforth); `c5.xlarge` AWS VM, 8 GiB RAM, EBS (*EC2*, henceforth); Raspberry Pi 4, 4 GiB RAM, 32 GB μSD (*RPi4*, henceforth). We use GCC 9.30 for x86-64, GCC 7.5.0 for AArch64, and Wasi SDK-14.0 with LLVM 13.0 for Wasm. Our evaluation shows that Veracruz can be deployed to tackle real-world use-cases (§4.1, §4.2), and has a good performance (§4.3).

4.1 Case-study: deep learning

Training datasets, algorithms, and learnt models may be sensitive IP and the learning and inference processes are vulnerable to malicious changes in model parameters that can cause a negative influence on a model's behaviors that is hard to detect (see e.g., [16]). We present two case-studies in protecting deep learning (DL) applications: privacy-preserving training and inference, and privacy-preserving model aggregation service, a step toward *federated* DL. We use *Darknet* in both cases, and the *Open Neural Network eXchange* (*ONNX*) as the aggregation format. We focus on the *execution time* of training, inference, and model aggregation on the *Core i7* test platform.

In the training and inference case-study, the program receives datasets from the respective data providers and a prelearnt model from a model provider, and thereafter starts training or inference, protected inside Veracruz, with the results—the trained model or prediction—made available to a result receiver. In the model aggregation case-study, clients conduct *local training* with their favorite DL frameworks, convert the models to *ONNX*, and provision these derived models into Veracruz. The program then aggregates

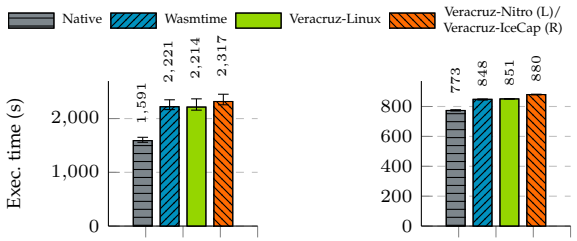

Figure 4: VOD execution time on *EC2* (L) and *RPi4* (R)

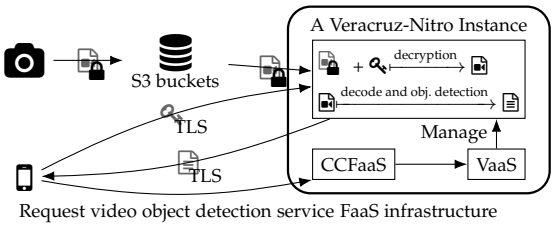

Figure 5: Video object detection case-study

the models, making the result available to all clients. By converting to *ONNX*, we support a range of local training frameworks including *PyTorch*, *Tensorflow*, and *Darknet*.

We trained a LeNet [17] on *MNIST* [17], a dataset of 60,000 training and 10,000 validation images of handwritten digits. Each image is 28×28 pixels and less than 1 KiB; we used a batch size of 100 in training, obtaining a trained model of 186 KiB. We take the average of 20 trials for training on 100 *batches* (hence, 10,000 images) and ran inference on one image. For aggregation, we use three copies of this *Darknet* model (186 KiB), obtaining three *ONNX* models (26 KiB), performing 200 trials for aggregation, as aggregation time is significantly less. Fig. 3 presents results.

For all DL tasks we observe the same execution time between Wasmtime and Veracruz, as expected, with both around 2.1–4.1× slower than native CPU execution, likely due to more aggressive code optimization available in native compilers. Recall that Veracruz use Wasmtime as its JIT engine. However, the similarity between Wasmtime and Veracruz diverges for file operations such as loading and saving of model data. Loading data from disk is 1.2–3.1× slower when using Wasmtime compared to executing natively. However, I/O in Veracruz is usually *faster* than Wasmtime, and sometimes faster than native execution, e.g., when saving images in inference. This is likely due to Veracruz's in-memory filesystem exhibiting a faster read and write speed transferring data, compared to the SSD of the test machine, while Wasmtime uses the filesystem provided by the operating system (Linux in our experiment) by default, which accesses both memory and disk.

4.2 Case-study: video object detection

We have used Veracruz to prototype a *Confidential FaaS*, running on AWS Nitro Enclaves and using *Kubernetes*. In this model, a cloud infrastructure or other delegate initializes an TEE that contains only the Veracruz runtime and provides an appropriate global policy file. Confidential functions are registered in a *Confidential Computing Function-as-a-Service* (CCFaaS) component, which acts as a registry for clients wishing to use the service and which collaborates, on behalf of clients, with a *Veracruz as a Service* (VaaS) component which manages the lifetime of any spawned Veracruz instances. Together, the CCFaaS and VaaS components draft policies and initialize Veracruz instances, while attestation is handled by clients, using the PAS.

Building atop this infrastructure, we applied Veracruz in a full end-to-end encrypted video object detection flow (VOD, see Fig. 5), demonstrating that Veracruz can be applied to industrially-relevant use-cases: here, a video camera manufacturer wishes to offer an object detection service to customers while providing believable guarantees that they cannot access customer video. Encrypted video clips, originating from an IoTeX Ucam video camera, are stored in an *AWS S3* bucket. The encryption key is owned by the camera operator and perhaps generated by client software on their mobile phone or tablet. Independently, a video processing and object detection function, compiled to Wasm, is registered with the CCFaaS component which takes on the role of program provider in the Veracruz computation. This function makes use of the Cisco `openh264` library as well as the Darknet neural network framework and a prebuilt YOLOv3 model, discussed in §4.1, for object detection.

Upon the camera owner's request, the CCFaaS and VaaS infrastructure spawn a new AWS Nitro Enclave loaded with the Veracruz runtime, configured using an appropriate global policy that lists the camera owner as having the data provider and result receiver roles. The confidential FaaS infrastructure forwards the global policy to the camera owner, where it is analyzed by their client software, with the camera owner thereafter attesting the AWS Nitro Enclave instance. If the policy is acceptable, and attestation succeeds, the camera owner securely connects to the spawned TEE, containing the Veracruz runtime, and securely provisions their decryption key using TLS, as data provider. The encrypted video clip is also provisioned into the TEE, by an AWS S3 application also listed in the global policy as a data provider, and the computation can start. Once complete, metadata containing bounding boxes of objects detected in the video can be securely retrieved by the camera owner via TLS, as result receiver, for interpretation by their client.

This FaaS infrastructure preserves desirable cloud application characteristics: the computation is on-demand and scalable, and allows multiple instances of Veracruz, running different functions, to be executed concurrently. Only the AWS S3 application, the camera owner's client application, and the video decoding and object detection function are specific to this use-case. All other modules are generic, allowing other applications to be implemented. Moreover, no user credentials or passwords are shared with the FaaS infrastructure in realizing this flow, beyond the name of the video clip to retrieve from the AWS S3 bucket and a one-time access credential for the AWS S3 application—keys are only shared with Veracruz inside an attested TEE.

We benchmark by passing a 1920×1080 video to the VOD program, which decodes frame by frame, converts, downscales, and passes frames to the ML model. We compare four configurations on two different platforms:

- On *EC2*, a native `x86-64` binary on Amazon Linux; a Wasm binary under Wasmtime-0.27; a Wasm binary

Description	Time (ms)
PAS start	7
Onboard new Veracruz TEE	3122
Request attestation message	54
Initialization of Veracruz TEE	1
Check hashes (including TLS handshake)	184
Provision object detection program	798
Provision data (model, video)	282323

Table 2: Overheads for VOD on AWS Nitro Enclaves

inside Veracruz as a Linux process; a Wasm binary inside Veracruz on AWS Nitro Enclaves. The video is 240 frames long and fed to the YOLOv3-608 model.

- On *RPi4*: a native AArch64 binary on Ubuntu 18.04 Linux; a Wasm binary under Wasmtime-0.27; a Wasm binary inside Veracruz as a Linux process; a Wasm binary inside Veracruz on IceCap. Due to memory limits the video is 240 frames long and fed to the YOLOv3-tiny model.

We take the native `x86-64` configuration as our baseline, and present average runtimes for each configuration, along with observed extremes, in Fig. 4.

EC2 *results:* Wasm (with experimental SIMD support in Wasmtime) has an overhead of ∼39% over native; most cycles are spent in matrix multiplication, which the native compiler can better vectorize than Wasmtime. The majority of execution time is spent in neural network inference, rather than video decode or downscaling. Since execution time is dominated by the Wasm execution, Veracruz overhead is negligible. A ∼5% performance discrepancy exists between Nitro and Wasmtime, which could originate from our observation that Nitro is slower at loading data into an enclave, but faster at writing, though Nitro runs a different kernel with a different configuration, on a separate CPU, making this hard to pinpoint. Deployment overheads for Nitro are presented in Table 2, showing a breakdown of overheads for provisioning a new Veracruz instance.

RPi4 *results:* The smaller ML model significantly improves inference performance at the expense of accuracy. Wasm has an overhead of ∼10% over native, smaller than the gap on *EC2*, potentially due to reduced vectorization in GCC for AArch64. Veracruz overhead is again negligible, though IceCap induces an overhead of ∼3% over Veracruz-Linux, approximately matching the overhead of ∼2% for CPU-bound workloads measured in Fig. 1, and explained by extra context switching through trusted resource management services during scheduling operations.

Using "native modules", as in §3.3, explicit support for neural network inference could be added to Veracruz, though our results suggest a max ∼38% performance boost by pursuing this, likely less due to the costs of marshalling data between the native module and Veracruz file system. For larger performance boosts, dedicated ML acceleration could be used, requiring support from the Veracruz runtime, though establishing trust in accelerators outside the TEE is hard, with PCIe attestation still a work-in-progress.

4.3 Further performance comparisons

PolyBench/C microbenchmarks: We further evaluate the performance of Veracruz on compute-bound programs using the PolyBench/C suite (version 4.2.1-beta), a suite of small,

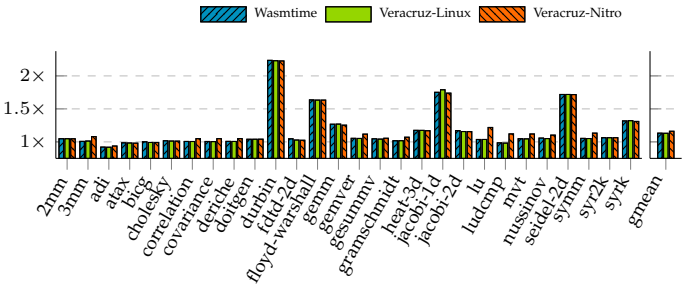

Figure 6: Relative execution time (vs. native) of PolyBench/C (large) on *EC2*. `gmean` is geo. mean of results

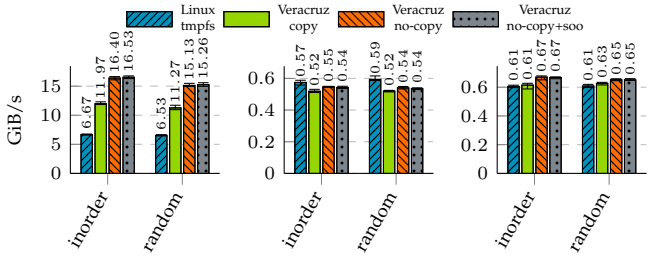

Figure 7: VFS bandwidth: read (L), write (M) and update (R)

simple computationally-intensive kernels. We compare execution time of four different configurations on the *EC2* instance running *Amazon Linux 2*: a native `x86-64` binary; a Wasm binary under Wasmtime-0.27; a Wasm binary under Veracruz as a Linux process; and a Wasm binary executing under Veracruz in an AWS Nitro Enclave. We take `x86-64` as our baseline, and present results in Fig. 6. Wasmtime's overhead against native CPU execution is relatively small with a geometric mean of ∼13%, though we observe that some test programs execute even faster under Wasmtime than when natively compiled. Again, we compile our test programs with Wasmtime's experimental support for SIMD proposal, though this boosts performance for only a few programs. Veracruz-Linux doesn't exhibit a visible overhead compared to Wasmtime, which is expected as most execution time is spent in Wasmtime, and the presence of the Veracruz VFS is largely irrelevant for CPU-bound programs. Veracruz-Nitro exhibits a small but noticeable overhead (∼3%) compared to Veracruz-Linux, likely due to the reasons mentioned in §4.2.

VFS performance: We evaluate Veracruz VFS I/O performance, as discussed in §3.2. Performance is measured by timing common granular file-system operations and dividing by input size, to find the expected bandwidth.

Results were gathered on *Core i7* test platform with zero swap size so that measurements would not be invalidated by physical disk access (see Fig. 7). Here, **read** denotes bandwidth of file read operations, **write** denotes bandwidth of file write operations with no initial file, and **update** denotes bandwidth of file write operations with an existing file. We use two access patterns, in-order and random, to avoid measuring only file-system-friendly access patterns. All random inputs, for both data and access patterns, used reproducible, pseudorandom data generated by `xorshift64` to ensure consistency between runs. All operations manipulate a 64 MiB file with 16 KiB buffer size—in practice, we expect

most files will be within an order of magnitude of this size.

We compare variations of our VFS against Linux's `tmpfs`, the standard in-memory filesystem for Linux. **Veracruz copy** moves data between the Wasm's sandboxed memory and the VFS through two copies, one at the Wasi API layer, and one at the internal VFS API layer. **Veracruz no-copy** improved on this by performing a single copy directly from the Wasm's sandboxed memory into the destination in the VFS. This was made possible thanks to Rust's borrow checker, which is able to express the temporarily shared ownership of the Wasm's sandboxed memory without sacrificing memory or lifetime safety. In theory this overhead can be reduced to zero copies through `memmap`, however this API is not available in standard Wasi. **Veracruz no-copy+soo** is our latest design, extending the no-copy implementation with a small-object optimization (SOO) `iovec` implementation—a Wasi structure describing a set of buffers containing data to be operated on, which for the majority of operations contain a reference to a single buffer. Through this, we inline two or fewer buffers into the `iovec` structure itself, completely removing memory allocations from the read and write path for all programs we tested with. Performance impact is negligible, however.

Being in-memory filesystem, the internal representation is relatively simple: directories and a global `inode` table are implemented using hash tables, with each file a vector of bytes. While naïve, these data-structures have seen decades of optimization for in-memory performance, and even sparse files perform efficiently due to RAM overcommitment by the runtimes. However, we were still surprised to see very close performance between Veracruz and `tmpfs`, with Veracruz nearly doubling the `tmpfs` performance for reads, likely due to the overhead of syscalls necessary to communicate with `tmpfs` in Linux. (Unfortunately `tmpfs` is deeply integrated into the Linux VFS layer, so it is not possible to compare with `tmpfs` in isolation.)

Both Veracruz and `tmpfs` use hash tables to store directory information, with the file data-structure and memory allocator representing significant differences. In Veracruz we use byte vectors backed by the runtime's general purpose allocator, whereas `tmpfs` uses a tree of pages backed by the Linux VFS's page cache, acting as a cache-aware fixed-size allocator. We expect this page cache to have much cheaper allocation cost, at the disadvantage of storing file data in non-linear blocks of memory—observable in the difference between the **write** and **update** measurements. For **write**, `tmpfs` outperforms Veracruz due to faster memory allocations and no unnecessary copies, while **update** requires no memory allocation, and has comparable performance.

Fully-homomorphic encryption: An oft-suggested use-case for fully-homomorphic encryption (FHE) is protecting delegated computations. We compare Veracruz against SEAL, an FHE library, in computing a range of matrix multiplications over square matrices of various dimensions. Both algorithms are written in `C`, with floating point arithmetic replaced by the SEAL multiplication function for use with FHE. Our results (see Fig. 8) demonstrate that FHE remains impractical, even for simple computations.

Teaclave: Apache Teaclave [18] is a privacy-preserving FaaS infrastructure using Intel SGX, supporting Python and interpreted Wasm with a custom programming model. We

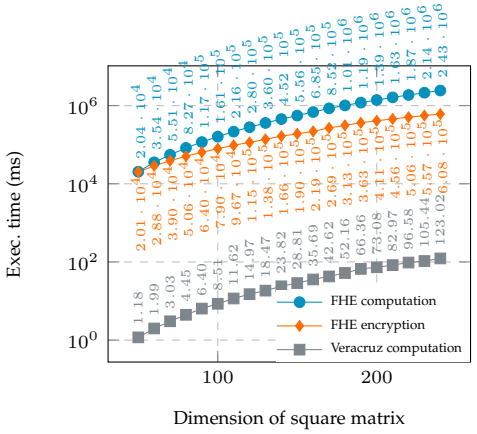

Figure 8: SEAL and Veracruz computation performance

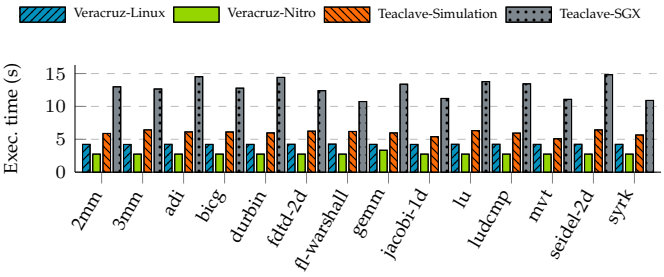

Figure 9: Execution times of Veracruz and Apache Teaclave

compare the performance of Teaclave running under Intel SGXv1 against Veracruz running as a Linux process, both on *Core i7*, and Veracruz on AWS Nitro enclaves on *EC2*. This is admittedly an imperfect comparison, due to significant differences in design, isolation technology, Wasm runtime, and hardware between the two. We run the PolyBench/C suite with its mini dataset—Teaclave's default configuration errors for larger datasets—and measure end-to-end execution time, which includes initialization, provisioning, execution and fetching the results, which we present in Fig. 9. While Veracruz has better performance than Teaclave when executing Wasm—with Veracruz under AWS Nitro exhibiting a mean 2.11× speed-up compared to Teaclave in simulation mode, and faster still than Teaclave in SGXv1—the fixed initial overhead of Veracruz, ∼4s in Linux and ∼2.7s in AWS Nitro, dominates the overall overhead in either case. At the time of writing, Intel SGXv2 hardware is available on Microsoft Azure though non-trvial to deploy applications on top, and it is possible that Teaclave's performance would benefit from SGXv2.

5 Closing Remarks

Veracruz is a framework for designing and deploying privacy-preserving delegated computations among a group of mutually mistrusting principals, using TEEs as a "neutral ground" to protect computations from prying or interference. Veracruz supports a mix of hardware-backed Confidential Computing technologies—such as AWS Nitro Enclaves and Arm Confidential Computing Architecture Realms—and pragmatic "software TEEs" via *IceCap*, using the high-assurance seL4 microkernel, on Armv8-A platforms without any other explicit support for Confidential

Computing, to provide strong isolation guarantees. Veracruz and IceCap provide a uniform programming and attestation model across emerging and "legacy" hardware platforms, easing the deployment of delegated computations. Both are open-source[1], with Veracruz adopted by the *Confidential Computing Consortium*.

Related work TEEs have been used to protect a zoo of computations of interest, e.g., ML [19] and genomic computations [20], and have been used to emulate or speed up cryptographic techniques such as secure multi-party computations [21]. These can be seen as use-cases of Veracruz. Cloud providers, e.g., AWS and Microsoft Azure, now provide TEEs solutions. Veracruz can simplify the deployment model to TEEs in Cloud Computing. A body of work related to Grid Computing (see e.g., [22], [23], [24]) has also explored similar themes to Veracruz and other related Confidential Computing projects.

Previous work [25] suggested a framework similar to Veracruz. This was never implemented. *Google Oak*, *Profian Enarx*, *Apache Teaclave*, *Inclavare Containers* [26], *Fortanix Confidential Computing Manager* and *SCONE* [27] are similar to Veracruz—all provide user-friendly interfaces for TEEs—though significant differences exist. Oak's emphasis is information flow control, while Enarx, Fortanix, Inclavare, and SCONE protect the integrity of legacy computations, either requiring recompilation of source code, or using containerized workloads, respectively. Moreover, SCONE, Fortanix, and Inclavare's main focus is deployment on SGX, while Veracruz is designed to achieve portability across multiple TEE-like technologies. Users of Inclavare, however, can choose between multiple deployment strategies on SGX, including the small WAMR Wasm runtime, or the Occlum library OS. The combination of a proxy attestation service and certificate-based attestation protocol in Veracruz, especially suitable for clients on resource-constrained devices, is also unique, though Inclavare does also offer a uniform attestation process. Lastly, Apache Teaclave is the most similar project to Veracruz, and also uses Wasm for portability reasons. However, as discussed in §4, we perform better.

Protected KVM (pKVM) is an attempt to minimize the TCB of KVM, enabling virtualization-based confidential computing on mobile platform, similar to IceCap. pKVM, with an EL2 kernel specifically designed for the task, may have higher performance than IceCap, but will not benefit from the formal verification effort invested in seL4.

OPERA [28] places a proxy between client code and the Intel Attestation Service, exposing the same EPID protocol to clients as the web-service exposes. The PAS exposes a potentially different protocol to client code, compared to the native protocol, due to the variety of TEEs Veracruz supports. Intel's *Data Center Attestation Primitives (DCAP)* is also similar though specific to Intel SGX.

Ongoing and future work The PAS, currently signs each generated certificate with the same key, though could sign certificates for different isolation technologies with different keys, each associated with a different root CA certificate. A global policy could then choose which technology to support based on the selection of root CA certificate embedded in the policy, and if multiple isolation technologies were to be supported, more than one root CA certificate could be embedded. The PAS could also maintain multiple Root CA certificates, arranged into a "decision tree of certificates", with the server choosing a CA certificate to use when signing the TEE's certificate from the tree, following a path from the root described by characteristics of the TEE itself (e.g., name of the manufacturer, whether memory encryption is supported, and so on). Again, the certificate associated with the security profile of the desired isolation technology can be embedded in the policy.

We intend to pragmatically bound some runtime properties of Veracruz programs. Cryptography is perhaps most sensitive to timing attacks, and we aim to provide a limited defense by supplying a constant-time cryptography implementation via the native module facility (see §3.3). We also aim to explore the use of a statically verified, constant-time virtual machine to gives users the option to statically verify timing properties of their programs—an area of significant recent academic interest—though likely at the cost of limiting their program to constant-time constructs, which is intractable for general-purpose programming. With FaCT [29] we could provide flexible, verifiably constant-time components such as virtual machines or domain specific functions, while the not-yet-standardised CT-Wasm [30] extension for Wasm also provides verifiable, constant-time guarantees as a set of secrecy-aware types and instructions.

We are also continuing work on statically verifying the *Software Fault Isolation* (SFI) safety of sandboxed applications. SFI systems, such as Wasm, add runtime checks to loads, stores, and control flow transfers to ensure sandboxed code cannot escape its memory region, though bugs in SFI compilers can (and do) incorrectly remove these checks and introduce bugs that let code escape its sandbox. To address this—following other SFI systems [31]—we have built a static verifier for binary code executed by Veracruz, an extension of *VeriWasm* [32], an open-source SFI verifier for compiled Wasm code. To adapt *VeriWasm* to Veracruz, we added support for AArch64, and ported VeriWasm from the *Lucet* toolchain to Wasmtime, as used by Veracruz. We plan to further extend VeriWasm to check other properties besides software fault isolation, e.g., Spectre [33] resistance.

Finally, a provisioned program, π_m, is either kept classified by its owner, or declassified to other principals in the computation (maybe all). In the former case, other principals must either implicitly trust that π_m behaves in a particular way, or establish some other mechanism bounding the behavior of the program, out-of-band of Veracruz. We aim for a middle ground, allowing a program owner to declassify runtime *properties* of the program, enforced by Veracruz, while retaining secrecy of the binary (using e.g., [34]).

5.1 Author biographies

All authors have agreed not to provide biographies.

1. Veracruz: https://github.com/veracruz-project/veracruz, and IceCap: https://gitlab.com/arm-research/security/icecap/icecap

References

[1] J. V. Bulck, N. Weichbrodt, R. Kapitza, F. Piessens, and R. Strackx, "Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution," in *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, Aug. 2017, pp. 1041–1056.

[2] K. Murdock, D. Oswald, F. D. Garcia, J. Van Bulck, D. Gruss, and F. Piessens, "Plundervolt: Software-based fault injection attacks against Intel SGX," in *Proceedings of the 41st IEEE Symposium on Security and Privacy (S&P'20)*, 2020.

[3] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien, "Bringing the web up to speed with WebAssembly," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 185–200.

[4] T. Sewell, S. Winwood, P. Gammie, T. C. Murray, J. Andronick, and G. Klein, "sel4 enforces integrity," in *Interactive Theorem Proving - Second International Conference, ITP 2011, Berg en Dal, The Netherlands, August 22-25, 2011. Proceedings*, 2011, pp. 325–340.

[5] T. C. Murray, D. Matichuk, M. Brassil, P. Gammie, and G. Klein, "Noninterference for operating system kernels," in *Certified Programs and Proofs - Second International Conference, CPP 2012, Kyoto, Japan, December 13-15, 2012. Proceedings*, 2012, pp. 126–142.

[6] T. C. Murray, D. Matichuk, M. Brassil, P. Gammie, T. Bourke, S. Seefried, C. Lewis, X. Gao, and G. Klein, "sel4: From general purpose to a proof of information flow enforcement," in *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*, 2013, pp. 415–429.

[7] Q. Ge, Y. Yarom, T. Chothia, and G. Heiser, "Time protection: The missing os abstraction," in *Proceedings of the Fourteenth EuroSys Conference 2019*, ser. EuroSys '19. New York, NY, USA: Association for Computing Machinery, 2019.

[8] I. Kuz, G. Klein, C. Lewis, and A. Walker, "capDL: A language for describing capability-based systems," in *Proceedings of the 1st ACM Asia-Pacific Workshop on Systems (APSys)*, 06 2010, pp. 31–36.

[9] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "KVM: the Linux virtual machine monitor," in *In Proceedings of the 2007 Ottawa Linux Symposium (OLS'-07*, 2007.

[10] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D.-M. Popa, "Firecracker: Lightweight virtualization for serverless applications," in *NSDI*, 2020.

[11] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *Sixth Symposium on Operating System Design and Implementation (OSDI)*, San Francisco, CA, 2004, pp. 137–150.

[12] M. Nystrom and B. Kaliski, "PKCS #10: Certification request syntax specification version 1.7," Internet Requests for Comments, RFC Editor, RFC 2986, November 2000.

[13] H. Tschofenig, S. Frost, M. Brossard, A. Shaw, and T. Fossati, "Arm's Platform Security Architecture (PSA) attestation token," Nov 2019, accessed 2020-04-15. [Online]. Available: https://tools.ietf.org/id/draft-tschofenig-rats-psa-token-05.html

[14] E. Johnson, D. Thien, Y. Alhessi, S. Narayan, F. Brown, S. Lerner, T. McMullen, S. Savage, and D. Stefan, "Доверя́й, но проверя́й: SFI safety for native-compiled Wasm," in *NDSS*. Internet Society, 2021.

[15] J. Lee, J. Jang, Y. Jang, N. Kwak, Y. Choi, C. Choi, T. Kim, M. Peinado, and B. B. Kang, "Hacking in darkness: Return-oriented programming against secure enclaves," in *Proceedings of the 26th USENIX Conference on Security Symposium*, ser. SEC'17. USA: USENIX Association, 2017, p. 523–539.

[16] F. Mo, H. Haddadi, K. Katevas, E. Marin, D. Perino, and N. Kourtellis, "PPFL: privacy-preserving federated learning with trusted execution environments," in *Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services*, 2021, pp. 94–108.

[17] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[18] R. Duan, L. Li, C. Zhao, S. Jia, Y. Ding, Y. Zhang, H. Wang, Y. Cheng, L. Wei, and T. Chen, "Rust SGX SDK," https://github.com/apache/incubator-teaclave-sgx-sdk, Jun 2020, accessed 2020-04-15.

[19] O. Ohrimenko, F. Schuster, C. Fournet, A. Mehta, S. Nowozin, K. Vaswani, and M. Costa, "Oblivious multi-party machine learning on trusted processors," in *Proceedings of the 25th USENIX Conference on Security Symposium*, ser. SEC'16. USA: USENIX Association, 2016, p. 619–636.

[20] A. Mandal, J. C. Mitchell, H. Montgomery, and A. Roy, "Data oblivious genome variants search on Intel SGX," in *Data Privacy Management, Cryptocurrencies and Blockchain Technology*, J. Garcia-Alfaro, J. Herrera-Joancomartí, G. Livraga, and R. Rios, Eds. Cham: Springer International Publishing, 2018, pp. 296–310.

[21] D. Gupta, B. Mood, J. Feigenbaum, K. Butler, and P. Traynor, "Using Intel Software Guard Extensions for efficient two-party secure function evaluation," in *Financial Cryptography and Data Security*, J. Clark, S. Meiklejohn, P. Y. Ryan, D. Wallach, M. Brenner, and K. Rohloff, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 302–318.

[22] I. T. Foster, Y. Zhao, I. Raicu, and S. Lu, "Cloud computing and grid computing 360-degree compared," *CoRR*, vol. abs/0901.0131, 2009.

[23] X. Zhao, K. Borders, and A. Prakash, "Svgrid: a secure virtual environment for untrusted grid applications," in *Proceedings of the 3rd international workshop on Middleware for grid computing, MGC 2005*, R. Nandkumar, B. Schulze, and P. Henderson, Eds., 2005, pp. 2:1–2:6.

[24] M. Bazm, M. Lacoste, M. Südholt, and J. Menaud, "Secure distributed computing on untrusted fog infrastructures using trusted linux containers," in *2018 IEEE International Conference on Cloud Computing Technology and Science, CloudCom 2018*. IEEE Computer Society, 2018, pp. 239–242.

[25] P. Koeberl, V. Phegade, A. Rajan, T. Schneider, S. Schulz, and M. Zhdanova, "Time to rethink: Trust brokerage using Trusted Execution Environments," in *Trust and Trustworthy Computing - 8th International Conference, TRUST 2015*, ser. Lecture Notes in Computer Science, M. Conti, M. Schunter, and I. G. Askoxylakis, Eds., vol. 9229. Springer, 2015, pp. 181–190.

[26] "Inclavare Containers," https://github.com/inclavare-containers/inclavare-containers, 2022, accessed 2022-11-06.

[27] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. Stillwell, D. Goltzsche, D. M. Eyers, R. Kapitza, P. R. Pietzuch, and C. Fetzer, "SCONE: Secure Linux containers with Intel SGX," in *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, 2016, pp. 689–703.

[28] G. Chen, Y. Zhang, and T.-H. Lai, "OPERA: Open remote attestation for Intel's secure enclaves," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 2317–2331.

[29] S. Cauligi, G. Soeller, B. Johannesmeyer, F. Brown, R. S. Wahby, J. Renner, B. Grégoire, G. Barthe, R. Jhala, and D. Stefan, "FaCT: A DSL for timing-sensitive computation," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 174–189.

[30] C. Watt, J. Renner, N. Popescu, S. Cauligi, and D. Stefan, "CT-Wasm: Type-driven secure cryptography for the Web ecosystem," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, jan 2019.

[31] 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 *2009 30th IEEE Symposium on Security and Privacy*. IEEE, 2009, pp. 79–93.

[32] E. Johnson, D. Thien, Y. Alhessi, S. Narayan, F. Brown, S. Lerner, T. McMullen, S. Savage, and D. Stefan, "Trust, but verify: SFI safety for native-compiled Wasm," in *Network and Distributed System Security Symposium (NDSS)*. Internet Society, 2021.

[33] S. Narayan, C. Disselkoen, D. Moghimi, S. Cauligi, E. Johnson, Z. Gang, A. Vahldiek-Oberwagner, R. Sahita, H. Shacham, D. Tullsen, and D. Stefan, "Swivel: Hardening WebAssembly against Spectre," in *USENIX Security Symposium*. USENIX, August 2021.

[34] D. P. Mulligan and N. Spinale, "The Supervisionary proof-checking kernel, or: a work-in-progress towards proof-generating code (extended abstract)," https://dominicpm.github.io/publications/mulligan-supervisionary-2022.pdf, 2022.