CC-BY 4.0. This is the author's version of the work. The definitive version will be published
in the proceedings of the 2019 20th Annual Middleware Conference (MIDDLEWARE'19).

AccTEE: A WebAssembly-based Two-way Sandbox for Trusted Resource Accounting

David Goltzsche
TU Braunschweig, Germany
goltzsche@ibr.cs.tu-bs.de

Manuel Nieke
TU Braunschweig, Germany
nieke@ibr.cs.tu-bs.de

Thomas Knauth
Intel, United States
thomas.knauth@intel.com

Rüdiger Kapitza
TU Braunschweig, Germany
rrkapitz@ibr.cs.tu-bs.de

Abstract

Remote computation has numerous use cases such as cloud computing, client-side web applications or volunteer computing. Typically, these computations are executed inside a sandboxed environment for two reasons: first, to isolate the execution in order to protect the host environment from unauthorised access, and second to control and restrict resource usage. Often, there is mutual distrust between entities providing the code and the ones executing it, owing to concerns over three potential problems: (i) loss of control over code and data by the providing entity, (ii) uncertainty of the integrity of the execution environment for customers, and (iii) a missing mutually trusted accounting of resource usage.

In this paper we present AccTEE, a two-way sandbox that offers remote computation with resource accounting trusted by consumers and providers. AccTEE leverages two recent technologies: hardware-protected trusted execution environments, and WebAssembly, a novel platform independent byte-code format. We show how AccTEE uses automated code instrumentation for fine-grained resource accounting while maintaining confidentiality and integrity of code and data. Our evaluation of AccTEE in three scenarios – volunteer computing, serverless computing, and pay-by-computation for the web – shows a maximum accounting overhead of 10%.

CCS Concepts • **Security and privacy → Trusted computing**; **Distributed systems security**; • **Software and its engineering → Middleware**;

Keywords Trusted Computing, Sandbox, Resource Accounting, WebAssembly, Intel SGX

ACM Reference Format:
David Goltzsche, Manuel Nieke, Thomas Knauth, and Rüdiger Kapitza. 2019. AccTEE: A WebAssembly-based Two-way Sandbox for Trusted Resource Accounting. In *20th International Middleware Conference (Middleware '19), December 8–13, 2019, Davis, CA, USA*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3361525.3361541

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
Middleware '19, December 8–13, 2019, Davis, CA, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-7009-7/19/12…$15.00
https://doi.org/10.1145/3361525.3361541

1 Introduction

Offloading computation to remote infrastructure has many use cases with cloud computing, client-side web applications and volunteer computing systems being the most prominent examples. Utilising resources that are not available locally and bringing the computation closer to customers are the main drivers for this everlasting trend. Since the beginning of remote execution of code, *infrastructure providers* distrust *workload providers* and guard their host environments from unauthorised access while also implementing measures to control and log resource usage by utilising sandbox mechanisms. So far, workload providers have to trust infrastructure providers to not take advantage of the provided code and data, to not tamper with the performed computation, and to actually provide and accurately account the resources that were utilised. In the context of cloud computing, trust is established through the reputation of well-known providers; however the absence of trust was often seen as a major obstacle, preventing more wide-spread use of cloud computing. For web-based applications the issue of missing trust is usually overcome by simply avoiding offloading sensitive code and data to the client or by re-validating important user inputs at server side. To summarise, the entities involved in remote computation (*workload provider* and *infrastructure provider*) generally mistrust each other, which inhibits broader application.

With the advent of novel trusted execution technologies, this situation changig. Taking ARM® TrustZone® [1] as the first widely-available implementation and more recently Intel® Software Guard Extensions (Intel® SGX) [2], it has become possible to guard computations on a remote machine from unauthorised access and tampering. This has been explored for cloud computing by systems such as Haven [3], SCONE [4] and Ryoan [5] but also in the context of web applications by securing computations inside a browser [6]. However, none of these efforts offer mechanisms for resource accounting trusted by both customers as well as infrastructure providers. Also, CPU time or runtime is usually used as a metric for resource usage; an approach which is not compatible with technologies such as Intel® SGX (see § 2.2). Research has also shown performance in cloud architectures to have a high time-dependent variability, i.e. resources costing customers the same have different performance [7]. Additionally, cloud providers implement different pricing models, which makes offerings hard to compare for customers. Therefore, from the workload providers' perspective, we see the need for a more comparable and accurate accounting of resources. Providing such a functionality would improve the trustworthiness of cloud computing but more importantly strengthen existing and enable

new remote computation use cases, where fine-grained computations are outsourced to a dynamic set of infrastructure providers.

In cloud computing we see a trend towards "serverless computing", with Function-as-a-Service (FaaS) [8] as the main driver, where customers provide functions that can be executed highly scalable. Customers are usually billed by the runtime of the deployed functions, while the price per time increases, if customers choose a higher upper limit of usable memory for the function. The runtime of a function can vary due to the underlying software and hardware stack and other effects [7]. In the extreme, a FaaS provider that has a cheaper price per runtime unit could be actually more expensive because of a slower infrastructure. All these aspects are more or less invisible to the user and hardly comprehensible. The demand for a trustworthy execution and accounting will even rise once serverless computing is used in the context of fog and edge cloud computing where highly distributed and possibly diverse infrastructure providers are utilised.

Trusted execution and accounting might be even more important in the context of volunteer computing systems such as BOINC [9] and Folding@home [10]. These projects assemble large amounts of donated computational power to solve problems like genome analysis. However, they face the issue of untrustworthy infrastructure providers: Participants may resubmit existing or outright bogus results either to sabotage the project or to improve their leader board ranking. Here, trusted execution combined with trusted accounting could substantially improve the situation as workloads do not have to be executed multiple times to verify results which is currently best practice [9].

As mentioned before, the use of trusted execution has also been proposed in web-applications [6]. If additionally equipped with trusted accounting, web-application providers and users could implement a new form of micro-payments where donated computing resources can be traded for ad-free web pages or access to digital assets such as news articles or other digital goods. However, if resource accounting is not protected, users will cheat to gain advantages without investing resources.

In this paper we present **AccTEE**, a two-way sandbox that offers accounting of resource usage trusted by workload and infrastructure providers in remote computation scenarios. To help protect the execution and data from unauthorised access by the resource provider, AccTEE utilises trusted execution as offered by Intel® SGX. To help prevent unauthorised access to the infrastructure providers machine, AccTEE takes WebAssembly [11] as a sandboxing mechanism. Thereby, the WebAssembly execution environment is operated under the protection of trusted execution. This way, infrastructure providers and workload providers can validate the integrity of the sandbox that cannot be altered by either side without being detected. AccTEE tracks resource usage inside this execution environment by counting WebAssembly instructions, memory allocation and I/O operations. Taking this two-way sandbox as a basis, AccTEE's core contribution is a fine-grained, platform independent and trusted accounting infrastructure supported by an automated instrumentation of WebAssembly. To our knowledge, AccTEE is the first system that realises a two-way sandbox based on the two novel technologies SGX and WebAssembly. We illustrate both the need and practicality of trusted resource accounting across client and server workloads. Finally, we evaluate the performance impact of trusted resource accounting using representative microbenchmarks and real-world applications from three scenarios: volunteer computing, serverless computing, and pay-by-computation for the web.

2 Background

This section further motivates the need for trustworthy resource accounting by describing four use case scenarios (§ 2.1). We give details on Intel® SGX, which AccTEE uses to help protect the confidentiality and integrity of code and data (§ 2.2). Also, we describe WebAssembly, which AccTEE employs to sandbox executable code (§ 2.3) and which is instrumented for resource accounting. The background section concludes by detailing our assumed threat model (§ 2.4).

2.1 Use Case Scenarios

We describe four use case scenarios that would benefit from trusted resource accounting as offered by AccTEE. They consist of client and server workloads, illustrating AccTEE's applicability in a range of computing environments.

Volunteer Computing. Mainstream volunteer computing systems like BOINC [9] and Folding@home [10] attract millions of participants to donate TeraFLOPS of computing power to a wide range of research projects [12, 13]. However, today's volunteer computing systems suffer from the following shortcomings: First, accounting is done by logging donated CPU time, e.g. the result is not necessarily trustworthy and does not include other resources such as memory or I/O. Additionally, as participants are expected to own vastly different CPU generations, it is impossible to fairly compare the CPU times donated. Second, these systems waste resources by executing each task multiple times to ensure result integrity in case a particular client misbehaves (either unintentionally due to a bug or intentionally to cheat). Finally, volunteers have access to the code and data which restricts the eligible workloads to domains where this is acceptable.

AccTEE addresses the points raised above. It provides a platform independent and therefore comparable resource accounting and helps protect integrity of results by executing the workload inside a trusted execution environment (TEE), thereby saving resources otherwise wasted on multiple workload executions. At the same time AccTEE also tracks resource usage through its accountable sandbox to prevent cheaters from advancing on the score board without actually expending the resources. AccTEE's TEE also helps keep the data (and optionally code) confidential to enable a wider range of use cases. In summary, while no computing system can be absolutely secure, AccTEE can serve as a platform for *trusted volunteer computing*.

Reimbursed Computing. Reimbursed computing [14–16] can be seen as the commercialisation of volunteer computing. Instead of donating computational resources for a worthwhile cause, participants offer spare resources of their private machines and are reimbursed for it in, for example, cryptocurrency tokens. This approach solves an additional drawback of today's volunteer computing systems: low incentive. Typically, credit systems with leader boards are implemented, which help to keep participants engaged through gamification. However, many participants abandon the volunteer

computing platform quickly, which results in surprisingly low numbers of active users (BOINC: 3.1 %[12], Folding@home: 1.3 %[13]). A way to reimburse participants for the work would of course shift the paradigm of volunteer computing systems, but open up new fields of application. This would enable anyone with spare resources to sell them whenever their hardware is underutilised. While anyone can participate and become an infrastructure provider, this model suffers from a lack of trust, as infrastructure providers are unknown to workload providers. Also, a model like this would certainly attract malicious infrastructure providers who will try to cheat and wrongfully collect reimbursements.

With AccTEE and its trusted execution environment, the workload provider has the ability to protect their workload independent of the underlying infrastructure and infrastructure provider. The mutually trusted accounting within AccTEE correctly determines a workload's resource usage. This prevents attempts of infrastructure providers to receive reimbursements for unassigned resources and attempts of workload providers to underpay their counterparts.

Serverless Computing. A recent trend in cloud computing is *serverless computing*, with Function-as-a-Service (FaaS) being the main driver. Instead of managing servers, developers write and submit code the size of a single function. The cloud provider sets up the function's execution context, scales the number of parallel function instances and connects the function's in- and outputs. Out of necessity, the workload provider trusts the infrastructure provider with its code and data in traditional FaaS implementations. In addition, the infrastructure provider is also trusted to honestly account the resources consumed by the workload provider. When a TEE protects the workload provider's code and data, the infrastructure provider can only track resources used within the TEE from the outside. In a model where the workload provider only trusts the TEE, the metrics collected outside of it are by definition untrusted. Similarly, any resource tracking happening inside the TEE is invisible to the infrastructure provider and hence untrusted. Additionally, performance in cloud architectures varies [7], which makes it near impossible for workload providers to compare different providers.

AccTEE solves these issues by executing code and data inside a TEE. In addition, AccTEE's resource accounting is comparable across providers and is trusted by both the workload provider and infrastructure provider. While the infrastructure provider cannot track resources within the TEE, it trusts the TEE to correctly do so for accounting purposes, and to attest to this fact. Within the TEE, language-based software fault isolation prevents the workload provider from interfering with the accounting. Interestingly, large companies recently started to investigate in this direction as well. Like AccTEE, Cloudflare Workers [17] use the same software fault isolation based on WebAssembly, illustrating the growing maturity of the technology.

Pay-by-Computation. Today, web content providers rely primarily on online advertising to finance their operations. On the other hand, content consumers go to great lengths to remove advertisements from websites as they are perceived as annoying, intrusive, detrimental to the user experience, and even compromise security [18]. Browser vendors even started to build the blocking technology right into their browsers [19]. Considering the overall negative sentiments towards online advertisement, the need of an alternative, less intrusive mechanism to compensate the content provider is obvious.

An accountable sandbox such as AccTEE can help address this problem. Instead of displaying advertisements, the user implicitly or explicitly agrees to run short-lived tasks in exchange for accessing the web page. The tasks execute in the background, utilizing otherwise idle resources and without disturbing the visual user experience. AccTEE's two-way sandbox protects the task's confidentiality and integrity. The browser's integrity and security is provided by the language-based isolation of AccTEE. The two-way sandbox limits the overall resource consumption and provides periodic feedback to the content provider on the task's progress.

2.2 Intel® Software Guard Extensions

Recent 6th generation Intel® CPUs support a versatile and performant TEE in the form of Intel® Software Guard Extensions (Intel® SGX). It helps protect the integrity and confidentiality of code and data through compartments called *enclaves*. Computations performed inside an enclave are isolated from potentially malicious privileged software, including the operating system, hypervisor and system management code. This is achieved by adding new instructions to create and manage enclaves. Code executing in the enclave is integrity-protected, i.e. it cannot be altered without detection; an important feature for AccTEE. Code and data occupy an isolated logical memory range inside the address space of a process. Off-chip enclave memory is stored in a system-reserved memory range called enclave page cache (EPC). Data moved between the CPU caches and the EPC is transparently en-/decrypted with a negligible performance overhead [20]. Intel® SGX helps protect the integrity, confidentiality and freshness of this range with checksums, memory encryption and versioning.

In addition to providing those protection mechanisms for code and data, Intel® SGX can authenticate enclaves through local and remote *attestation*: With local attestation, two enclaves on the same platform can authenticate each other. The authentication is based on an enclave identity that includes the enclave's code, creator and other supplemental attributes. Optionally, the attestation report may contain arbitrary user-defined data, e.g., to cryptographically bind a public key to a specific enclave instance. *Remote attestation* allows a challenger to gain trust in a remote enclave [21]. The attestation report is signed by the platform's quoting enclave. The signed report (*quote*) in combination with either the Intel® Attestation Service (IAS) [22] or alternate attestation infrastructure [23] allows a remote challenger to assess the trustworthiness of an enclave, i.e. whether the enclave runs on a genuine Intel® SGX-capable CPU and whether the platform software is up-to-date.

While Intel® SGX offers a performant and versatile TEE, some restrictions apply. Since Intel® SGX isolates the enclave from the untrusted environment, operations that could compromise this isolation are disallowed. For example, to issue a system call the processors needs to switch into supervisor mode, but mode switches are disallowed while in the enclave. Besides certain instructions being unavailable inside the enclave, a second limitation is the restricted EPC size of 128 MB per machine, with only 93 MB being usable. Larger enclaves are possible, but securely swapping EPC pages to and from unprotected memory incurs a potentially hefty performance penalty which highly depends on the application's memory access pattern [4, 24].

Also, the trusted time source of Intel® SGX has limitations: It can only be used as a *lower bound*, as calls traverse untrusted code and

thus can be withheld by an attacker. Some systems have proposed to approximate a trusted timer using a dedicated thread in a busy-loop [25–27]. However, this workaround needs fine-tuning for each hardware-platform and wastes an entire core to count CPU cycles.

AccTEE builds on Intel® SGX to help protect the confidentiality and integrity of programs. The ability to gain confidence in an enclave's identity through remote attestation enables a remote party to trust the resource accounting, making Intel® SGX a good choice for implementing the our AccTEE prototype.

2.3 WebAssembly

WebAssembly [11, 28] (abbreviated WASM) is a novel, platform independent binary instruction format. Its major goal is a safe, fast and portable low-level code format. While WebAssembly's first main use case was for client-side execution of web applications within browsers, its design is independent of this initial use case: stand-alone execution of backend services is already possible with Node.js [29] and other lightweight JavaScript-independent WebAssembly runtimes [30–33] are under active development.

Attempting to create a safe and performant execution environment in browsers for untrusted code has a long history. Predecessor technologies include Microsoft's ActiveX [34] on Windows, Native Client (NaCl) [35] and Portable Native Client (PNaCl) [36] in Google's Chrome browser, and Mozilla's asm.js [37]. As opposed to these previous efforts, WebAssembly is developed by a consortium of companies. As such, support to execute WebAssembly is already present on all major platforms and browsers.

WebAssembly itself is platform independent and serves as a portable compilation target for higher-level languages. A growing list of toolchains already support WebAssembly as a compilation target for different source languages, including mature support for C/C++ and Rust based on Emscripten [38]. Support for other programming languages is actively being developed, including C#, Java, Go, Python, TypeScript and many more [39]. The front-end language is compiled to WebAssembly which is translated to machine code for the target platform before execution. At a technical level, WebAssembly represents a one-way sandbox based on software fault isolation. WebAssembly *modules* are isolated from each other by disjoint memory spaces: the memory space for code, execution stack, execution environment data structures, and heap are separated, thereby preventing arbitrary code execution as well as data corruption outside the WebAssembly module's own data. To provide isolation while preserving close to native performance, all memory accesses to the contiguous *linear memory* are protected by simple bounds checks which can typically be performed directly in hardware [11]. Besides memory isolation, WebAssembly provides a protected call stack, which only contains variables of fixed size. More complex structures are moved to a stack in the linear memory. This protects the control-flow by eliminating the possibility of buffer overflow exploits. WebAssembly modules are usually shipped as binaries, but WebAssembly offers the human-readable *WebAssembly text format* (WAT), which is equivalent to the binary format. AccTEE builds on WebAssembly's built-in software fault isolation to prevent the program from interfering with the accounting system itself.

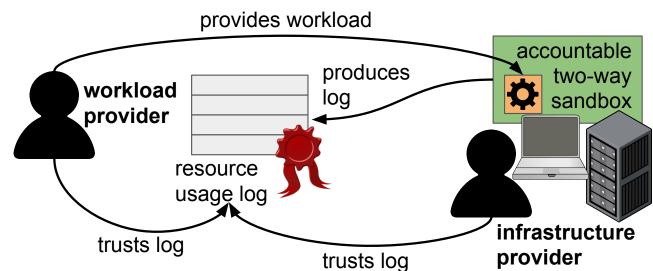

Figure 1. The accountable two-way sandbox of AccTEE generates a resource log trusted by the two mutually distrusting parties: infrastructure provider and workload provider.

2.4 Threat Model

Generally, all resources provided by the infrastructure provider are not trustworthy, as they elude the control of the workload provider. In contrast, from the infrastructure provider's point of view, all input provided by the workload provider is untrusted, as it might contain malicious code. Therefore, we assume both the infrastructure provider and workload provider as powerful attackers, that can behave arbitrarily and all participating execution platforms as untrusted. However, we do trust the Intel® SGX implementation and associated services, e.g., the platform services (PSW), architectural enclaves and attestation service. We are aware of reports of side-channel attacks [40–43] that may affect Intel® SGX. In addition to workarounds like disabling hyper-threading, researchers [25, 26, 44] and hardware companies [45] are working on mitigations, which we assume will eventually succeed. Therefore, we acknowledge potential side-channel attacks as a concern, but ultimately we consider them outside the scope of this work. Finally, we do also not consider denial-of-service attacks.

3 Design

According to the threat model (see § 2.4), we define two parties that interact with each other in a deployment of AccTEE, as shown in Fig. 1: the *infrastructure provider* performs executions on his platform for the *workload provider* inside an *accountable two-way sandbox*. While the workload provider distrusts the software on the machine of the infrastructure provider, he trusts the hardware-protected sandbox running on it. Both parties mutually trust the *resource usage log* produced by the trusted sandbox. We start this section by outlining the desired qualities of a trusted resource accounting mechanism.

3.1 System Requirements

A trusted accountable execution system should fulfil the following six requirements:

R1: *Polyglot input.* The system has to support a variety of high-level programming languages. Workload providers should not be restricted in their choice of programming language and be able to use a language they are comfortable with. Also, supporting a range of languages allows for running a variety of existing workloads without much effort.

R2: *Platform independence.* To enable a diverse class of platforms, the results of the accounting should be independent of the platform.

Besides translating the workload code into executable instructions for the target platform, there should be no other target-specific adaptation required.

R3: *Two-way isolation.* The execution must be isolated in two directions, as the infrastructure provider and workload provider do not trust each other: on the one hand, the workload must be isolated from the infrastructure provider's environment, on the other hand the infrastructure provider should not be able to interfere with the workload execution.

R4: *Trusted Resource Accounting.* The system should ensure that resources consumed by a workload are accounted in a trustworthy manner: the accounting mechanism must be trusted by the infrastructure provider *and* the workload provider. This requires that neither party is able to interfere with the accounting mechanism, e.g. manipulate the accounting to their own advantage.

R5: *Workload integrity and confidentiality.* The system must prevent the infrastructure provider from violating the workload's integrity. This allows the workload provider to rely on results obtained without trusting the underlying infrastructure. Optionally, the system should be able to keep the workload confidential, which includes code and data.

R6: *Low performance overhead.* To be practical, the system should induce a relatively low performance overhead compared to the unaccounted execution.

3.2 Accounting on Different Abstraction Levels

In general, resource accounting can be performed at different *abstraction levels*: (i) at the task level, where completed results of certain computations get accounted; (ii) at the hardware level, where resources consumed by the actual machine are accounted (e.g. CPU usage); and, in between those (iii) at the level of executed code, e.g. by instrumenting the code for accounting. Although easy to implement, accounting at the task level is only useful if tasks have homogeneous resource usage profiles. Otherwise, accounting tasks is too coarse-grained to track the resource expenditure because each task's complexity depends on the task itself as well as the actual inputs – both of which can vary widely. Unless only similar tasks with similar inputs are executed, task-level accounting is simply not a useful metric to track resource usage in the general case. Accounting based on hardware utilisation is the predominant way, with recording CPU usage being a common practice. While this seems fair and a pragmatic choice, the user is confronted with the issue that different CPUs provide different performance. Thus, the same task executed on an older or cheaper CPU likely takes longer than on the flagship CPU of the latest generation. Therefore, cloud providers account usage of virtual CPUs (vCPUs) – where vCPU is a normalised metric. While at the first glance appealing, the conversion factor is under the control of the provider and hidden from the user. Additionally, each cloud provider has its own definition of vCPU and associated conversion factor. Thus, cost per CPU usage is only a rough estimator to compare cloud providers. Also, trustworthy measurement of CPU usage from within an Intel® SGX enclave incurs a substantial overhead. Since the counters reporting CPU usage may be manipulated by the untrusted environment, a trustworthy measurement has to track CPU usage through other means (see §6). To summarise, accounting raw hardware utilisation is not sensible for and cannot be applied in AccTEE.

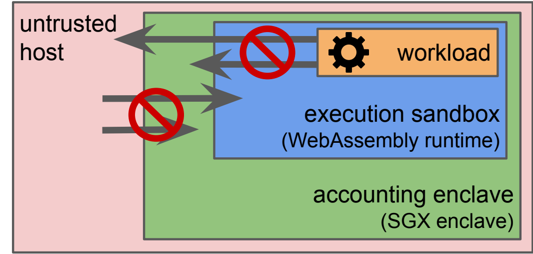

Figure 2. Architecture of AccTEE: combining the accounting enclave with an execution sandbox creates a two-way sandbox isolating the workload from the host and vice versa.

This leaves the instrumentation of code to enable a fine-grained accounting. In principle this can be introduced at intermediate or native code level. AccTEE instruments intermediate code to gain platform independence, specifically, we use WebAssembly to support a growing number of languages. For the same (deterministic) task and input parameters, the number of executed WebAssembly instructions will be the same across different hardware platforms and even across different WebAssembly runtimes. Thus, a per instruction pricing model enables users the fair comparison of offerings. Nevertheless, the infrastructure provider has still the opportunity to implement an internal pricing model that takes all relevant cost factors (e.g. costs for management, energy, hardware) into account. The following chapter explains how AccTEE's code instrumentation is done on a technical level.

3.3 Resource Accounting in AccTEE

Here, we describe how resource accounting works in AccTEE and how this satisfies the requirements laid out above. AccTEE combines two sandboxing mechanisms to achieve trusted resource accounting. As shown in Fig. 2, an *execution sandbox* helps prevent the untrusted workload from interfering with the resource accounting and with the host while an *accounting enclave* helps prevent untrusted host software from spying on the workload or manipulating the accounting. To realise the accounting enclave, our AccTEE prototype uses language-based isolation provided by WebAssembly (*R1*, *R2*).

Additionally, AccTEE leverages recently introduced TEE technology to implement the execution sandbox. In particular, our prototype uses Intel® SGX enclaves (see § 2.2) to help protect the workload's confidentiality and integrity from an otherwise untrusted system (*R5*). The hardware-enforced isolation helps protect the enclave even from privileged execution modes such as the operating system or system management mode. Although other security coprocessors, such as Trusted Platform Modules (TPMs), also allow to verify the software stack running on a machine, they typically do not allow to run arbitrary code on them. While TPMs can measure the system state and provide a standard API to securely carry out common cryptographic operations, workloads are still unprotected and susceptible to all the usual attacks hardware TEEs are designed to protect against.

The workload provider and infrastructure provider have the common goal of ensuring that the workload is correctly instrumented. AccTEE achieves this by enabling both parties to trust two enclaves: the instrumentation enclave (IE) and the accounting enclave (AE).

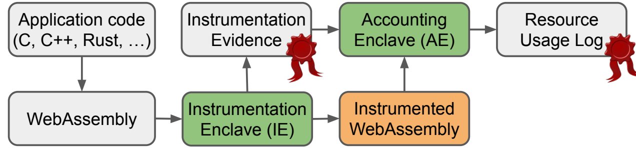

Figure 3. AccTEE's conceptual workflow from original application to obtaining a trusted resource accounting log.

The code of both enclaves needs to be publicly available and can therefore be audited, i.e. each party can verify the correctnes of the enclaves and then calculate the enclave measurement, a hash identifying the enclave code, independently. Through the attestation process, the workload provider and infrastructure provider both gain confidence that the intended enclave is running and has not been tampered with (*R4*). The combination of Intel® SGX and WebAssembly, AccTEE creates the desired two-way isolation (*R3*). Here, we call this combination, a *two-way sandbox*.

Even though our initial prototype focuses on Intel® SGX, AccTEE's principles are platform independent and therefore portable to other architectures. While the WebAssembly-based accounting enclave is platform-agnostic by design, the choice of TEE varies between target platforms. For example in case of ARM-based systems one would use TrustZone® [1], while in case of AMD the SEV [46] extension is an option.

AccTEE tracks three resource types: CPU usage, memory as well as I/O usage. Thereby, we use the number of executed WebAssembly instructions as a metric for CPU usage. To this end, the WebAssembly code is instrumented prior to execution. Similarly, AccTEE tracks I/O operations by accumulating the number of bytes sent and received via various I/O channels. Our prototype primarily tracks resources consumed within one WebAssembly module. Resources consumed outside of a WebAssembly module, e.g. in the WebAssembly runtime or operating system, are currently not tracked; an exception are I/O operations. We assume that the majority of resources is consumed within WebAssembly modules, which is certainly true for self contained and compute intensive workloads. In fact this applies to all our targeted usage scenarios.

The overall workflow is illustrated in Fig. 3. The application is first compiled to WebAssembly. The next step is to instrument the WebAssembly code for accounting purposes. This functionality is encapsulated in a separate enclave independent of the accounting enclave. The instrumentation enclave analyses the input WebAssembly and instruments it to perform resource accounting. It outputs the instrumented WebAssembly together with a cryptographic evidence (e.g. a signed statement) attesting that the instrumentation enclave produced this output. The accounting enclave is instantiated at the infrastructure provider and receives as input the instrumentation evidence and the WebAssembly code. After verifying that the WebAssembly code was indeed produced by the instrumentation enclave, it commences execution of the workload. During execution, the accounting enclave either periodically or upon request produces a resource accounting log detailing the workload's resource utilisation.

Disaggregating the instrumentation from the execution is beneficial in many respects. The code only needs to be instrumented once. A cached copy of the instrumented code can be re-used across many invocations. As a further optimisation, the instrumented code may even be translated to native code ahead of time. This saves the accounting enclave from translating the WebAssembly code into native code, also reducing its complexity in terms of code running inside the sandbox.

3.4 I/O Operations in a Two-way Sandbox

The two technologies combined by AccTEE– WebAssembly and Intel® SGX – both impose restrictions on I/O operations. WebAssembly does not specify an interface for I/O operations, therefore WebAssembly code is expected to be embedded within a runtime system that exposes the necessary primitives as functions that can be invoked from WebAssembly code. WebAssembly code executed within a web browser or Node.js utilises the existing JavaScript runtime to interface with the outside world, i.e. WebAssembly can perform all I/O operations that are offered by the browser or Node.js API, such as issuing HTTP requests.

In SGX enclaves, system calls are disallowed (see § 2.2); the enclave has to be explicitly exited to perform untrusted I/O operations. However, AccTEE relies on SGX-LKL [47] to execute legacy binaries. SGX-LKL utilizes the Linux Kernel Library (LKL) to provide user-level threading, signal handling, and paging within the enclave. Wherever possible, SGX-LKL handles system calls inside the enclave, however, system calls requiring direct access to external resources (e.g. I/O operations) are handled by the host operating system and are therefore untrusted. To help maintain integrity and confidentiality of transmitted data, AccTEE's workloads can use encryption layers either on system or platform level: for example, LKL provides block device encryption, while Node.js offers support for HTTPS. AccTEE is also able to account these I/O operations, as described in the following section.

3.5 Accounting enclave

AccTEE's accounting enclave tracks three principal resources: CPU, memory and I/O. Next, we detail how AccTEE tracks each of these resources to produce a trustworthy resource accounting log.

Processor. Today, CPU time or runtime is usually used as a metric for CPU usage. Beside the outlined issue that this metric is hard to compare between different platforms or cloud providers, for AccTEE, this requires a special trusted time source. In the case of Intel® SGX, this source does not have the desired properties for resource usage tracking: (i) trusted time can only be measured in seconds, which is insufficient for accounting shorter workloads; and (ii) determining the time spent in an enclave is impossible, as the measured time can be inflated by an attacker (see § 2.2).

To solve these issues, AccTEE tracks CPU usage by maintaining a counter (referred-to as *instruction counter*) and incrementing it for every executed WebAssembly instruction. We support different optimisations to reduce the actual number of counter increments (see § 3.6). Also, AccTEE supports weights for WebAssembly instructions to account for different complexities (see § 3.7). Therefore, we use the term *weighted instruction counter*. Counting WebAssembly instructions makes the accounting platform independent since the same WebAssembly instructions are executed on every platform – after translation to native code. Additionally, the approach is independent of the WebAssembly runtime, as it does not need to be modified.

Before workload execution, AccTEE instruments the original WebAssembly code to count the number of executed WebAssembly instructions. AccTEE adds a global counter variable to the original code. At runtime, the counter is initialised to zero and represents the weighted number of executed WebAssembly instructions so far. AccTEE also adds WebAssembly instructions to increment the counter at appropriate points. Since WebAssembly code is structured into basic blocks, AccTEE adds a sequence of instructions to increment the instruction counter at the end of each basic block (see Fig. 4 on the left). The counter is incremented based on the number of WebAssembly instructions contained in the basic block (excluding the counting instruction). By definition it is only possible to enter a basic block at the first instruction and leave it through the last instruction. Hence, incrementing the counter at the end of the basic block ensures the correctness of our instruction counting.

To prevent the untrusted WebAssembly code from modifying the instruction counter, AccTEE scans the code and chooses a previously unused variable name to refer to the counter. The instruction counter is stored in a module global variable. Since operations on global variables must identify the operand at compile time, it is impossible to modify the counter other than with the injected code.

Memory. Each WebAssembly module has access to a contiguous block of memory called *linear memory* (see § 2.3). Linear memory is similar to traditional heap memory but with a few important constraints. While linear memory is initialised with a specific size defined by the module, it can grow dynamically up to a certain maximum. By design, linear memory can only grow: WebAssembly does not provide instructions to reduce the size of a module's linear memory. Thus, AccTEE can use the linear memory size for the accounting of memory consumed by the workload.

Based on this, two policies can be implemented: either AccTEE reports the peak memory usage by adding the size of all linear memories at the end of a workload. Or, as an alternative, a fine-grained memory accounting is possible by combining AccTEE's executed instruction counter with the linear memory size. This allows AccTEE to calculate the integral of the linear memory usage over the execution time, which is approximated by the instruction counter. AccTEE leaves it as a policy decision to the workload and infrastructure providers to agree on which policy to use to track memory usage. If shrinking of linear memory becomes possible in the future, AccTEE can be adapted to accommodate for this as well. Using the first policy, we evaluated the cost of WebAssembly instructions in dependence of memory access ranges (see § 5.2).

Accounting of I/O Operations. As described in § 3.4, WebAssembly runtimes expose primitives to WebAssembly code for performing I/O operations. In the context of AccTEE, the WebAssembly runtime is part of the trusted execution sandbox. Thus, it is possible to instrument the runtime's existing I/O functions for accounting purposes. AccTEE has an additional counter to accumulate how many bytes flow in and out of the WebAssembly module through I/O functions. Today's cloud providers do similar tracking to bill users for network or disk I/O. Our evaluation in § 5.3 shows this accounting mechanism to have a negligible overhead.

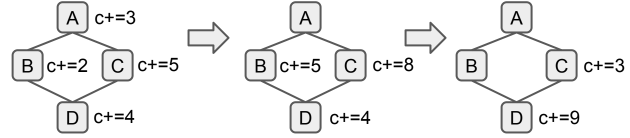

Figure 4. Example of `flow-based` optimisation. In this case, 2 out of 4 increments can be eliminated.

3.6 Reducing Counter Increments

The naive accounting of executed WebAssembly described in the previous section is straightforward. However, if the program's control flow is amenable to it, fewer accounting instructions may be required while maintaining correctness. We present two optimisations that reduce the runtime accounting overhead by performing static program analysis when instrumenting the code (*R6*). Essentially, through control flow analysis we can elide weighted instruction counter increments on certain code paths.

The key insight is that instead of incrementing the counter in each basic block, for some control flows it is sufficient to increment the counter at the end of a common path. More formally, if a basic block B dominates an immediate successor block S, it is sufficient to only update the weighted instruction counter at the end of block S. In this case, basic block S will increment the counter by the number of instructions contained in B and S. Fig. 4 gives an example. Here, node A's update is combined with that of nodes B and C because A dominates B and C, i.e., each control flow reaching either B or C must pass through A.

Another observation concerns nodes with multiple predecessors. Consider a node N with predecessors $P_1, P_2, \ldots, P_N$. The following transformation is possible. Let I_{Pmin} be the minimum number of instructions in $P_1, P_2, \ldots, P_N$ and I_N is the number of WebAssembly instructions in block N. At the end of block N update the counter by $I_N + I_{Pmin}$ while incrementing the counter in each predecessor by $I_{P_n} - I_{Pmin}$. The predecessor with the minimum number of instructions increments the counter by zero, i.e. the pass can remove the additional incrementation code. Fig. 4 gives an example. Node D has two predecessors B and C. Traversing either B or C increases the counter by at least 5. Hence, the transformation adds this minimum increment to node D's update, pushing it to 9. At the same time, the update for node C is adjusted to $3 = 8 - 5$. Node B no longer requires an extra update. Only a single pass through the control flow graph is required to perform the transformations. Even though they are two distinct transformations, we always perform them together. In the evaluation we refer to this group of transformations as `flow-based` optimisation.

Our second optimisation, referred to as `loop-based`, moves instruction counter increments out of loop bodies. This optimisation only applies to control-flow independent instructions inside the loop body. Instead of incrementing the instruction counter on each iteration, the counter is incremented only once after exiting the loop. The optimisation pass uses a heuristic to identify a loop variable that is incremented or decremented by a constant value on each iteration. If a loop variable is identified, the optimiser injects code to calculate the number of iterations based on the variable's value before and after the loop. The instruction counter is then incremented by the number of instructions in the loop's body times the number of iterations. Attackers could try to exploit this optimisation by

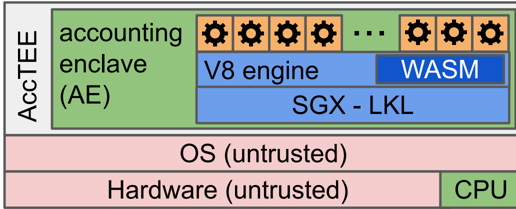

Figure 5. Components of AccTEE: inside the accounting enclave, we utilise SGX-LKL to run an instance of the V8 JavaScript/WebAssembly engine to execute workloads.

decreasing the loop variable in the last loop operation. AccTEE helps prevent this by only allowing one single write access to the loop variable which has to be executed in every loop iteration. If the loop variable is written more than once per iteration, this optimisation is not applied.

3.7 WebAssembly Instruction Weights

WebAssembly instructions have different complexities, as they perform various calculations ranging from cheap arithmetic floor operations to more expensive divisions (see Fig. 7). Additionally, the cost of instructions performing memory accesses depend on the memory access pattern and memory range because of caching: while linear accesses are cheap, random accesses are expensive (see Fig. 8). To account for that, AccTEE introduces weights for every WebAssembly instruction. These weights are used to increment the weighted instruction counter accordingly, leading to a more accurate accounting result. In practice, we expect the weights to have minor differences for different processors. In AccTEE, runtime adjustments are possible, allowing weight adjustment without requiring the release of new enclaves. In § 5.2, we show measurements that obtain the weights for most WebAssembly instructions. However, the cost of instructions that access memory (`load` and `store`) cannot be predicted. This because it is impossible to foresee the memory access pattern of non-trivial workloads: the resource demand of these instructions not only depends on their input values, but mostly on the memory access pattern, which is a unique characteristic of every workload (and other system activity). Therefore, we resort to the common de facto standard of using the peak memory usage for estimating the cost for memory accesses using the results shown in § 5.2.

Infrastructure providers have to find a middle ground between linear and random accesses when designing their cost model for billing memory usage. To summarise, both workload provider and infrastructure provider have to accept the instruction weights provided by AccTEE; they are part of the mutually trusted, attested execution environment.

4 Implementation

A concrete realisation of AccTEE with its major components is depicted in Fig. 5. At the bottom AccTEE relies on an SGX-capable CPU. Several layers of software are running on top of trusted hardware to provide AccTEE's trusted accounting enclave. In our prototype, AccTEE uses a single Intel® SGX enclave to help protect the workload and its execution from the untrusted infrastructure provider. We refrained from implementing the instrumentation in an enclave, as the instrumentation is only executed once and the process itself is not evaluated. AccTEE uses SGX-LKL [47] to run unmodified Linux binaries on Intel® SGX. We chose SGX-LKL over alternatives such as Haven [3], SCONE [4], Graphene [48], and Panoply [49] because we are most familiar with it. In principle, AccTEE is compatible with any of these systems and could be ported with low effort. In contrast, adapting AccTEE to non-x86 platforms requires the use of a different TEE technology and an associated, non-trivial engineering effort.

Since the WebAssembly text format (see § 2.3) is easier to parse, analyze and manipulate, we refrain from resorting to more complex tool chains like LLVM or similar. The current prototype implements the instrumentation pass in 605 lines of JavaScript code. We rely on JavaScript, as it integrates easily with the existing WebAssembly workflow, as instantiation and runtime support for WebAssembly all happens within the existing JavaScript runtime.

4.1 WebAssembly Execution

Since WebAssembly only specifies a binary instruction format, it requires an execution context into which it is embedded. Because the initial deployment of WebAssembly targets the web, currently available execution contexts are more or less tied to web browsers. Currently, alternative standalone execution environments like WAVM [30] or wasmi [31] are being developed. Because of their early stage, our AccTEE prototype uses the mature *Node.js* [29] JavaScript runtime. Underneath, it uses Chrome's JavaScript engine V8 to execute JavaScript and WebAssembly.

To execute WebAssembly modules, V8, like other major engines, requires "glue code" to bridge between the JavaScript engine and the WebAssembly execution context. In V8, the glue code compiles and instantiates a WebAssembly module within the JavaScript engine. Depending on the application, the glue code also needs to expose an interface for functionalities which are not directly accessible to the WebAssembly code, such as I/O. The glue code is, therefore, usually specifically tailored for and generated together with the corresponding WebAssembly code.

However, accepting and executing such JavaScript code together with its WebAssembly module poses a security risk, as it can directly interfere with the operation of AccTEE, like analysing other WebAssembly modules or falsifying the accounting. In order to mitigate this risk, we chose a different approach which is based on the separation of main- and side-modules, provided by Emscripten [38]. The main module provides an interface comprising all standard library functions it imports and produces all necessary JavaScript glue code. Each additional module, also called side module, only imports the required standard library functions from the main module. No additional glue code is required when loading a side module. In our framework we simply statically include a main module which provides all standard library functions together with its glue code, while each dynamically loaded module is a side module which imports required functionality from the main module. The code for compiling and instantiating each side module is integrated directly into the framework.

5 Evaluation

We begin the evaluation with a set of micro-benchmarks to assess the overhead of WebAssembly over native execution. While WebAssembly is designed to be lightweight and compiles to native

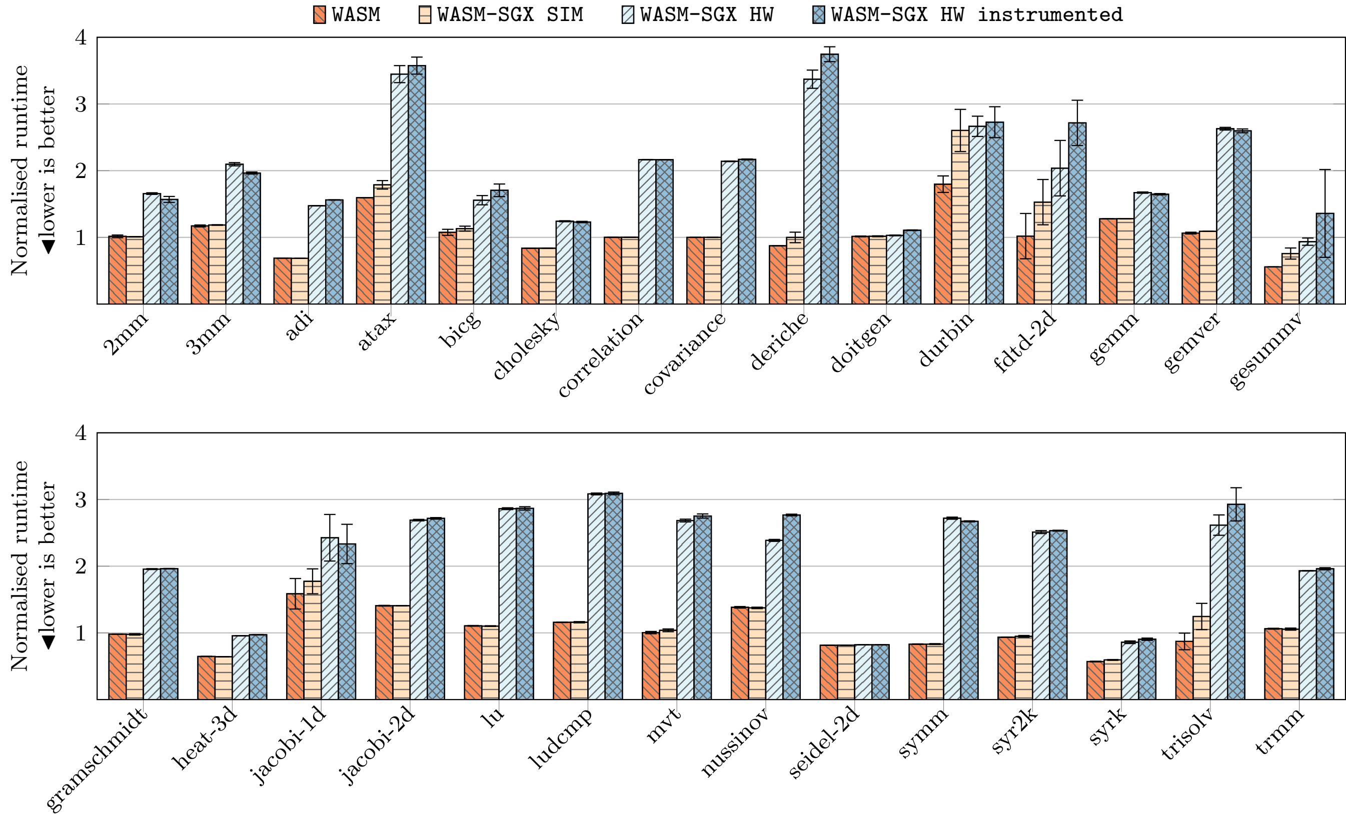

Figure 6. Mean overhead of runtimes of WebAssembly execution in Node.js without SGX, with simulated SGX, SGX in hardware mode and additional instrumentation normalised to native execution time for the PolyBench/C benchmark suite.

code, we still expect overheads due to the isolation enforcement and the embedding context. Next, we evaluate the performance impact of AccTEE on programs from domains we believe can benefit from a trusted resource accounting solution. The programs stem from the domains of volunteer/reimbursed computing, serverless/FaaS, and pay-by-computation. We separate the overheads due to the accounting instrumentation from the WebAssembly and TEE-related overheads (see § 3.3). Throughout this section, each data point is the average of 10 runs, unless noted otherwise. We report the standard deviation whenever it exceeds 5%.

Our evaluation setup consists of two identical machines, used as client and server. Each machine has an SGX-capable Xeon E3-1230 v5 CPU and 32 GB of RAM and is connected to a switched 10 Gbps network. We use clang v3.8.0, rustc v1.30.0 and go v1.11.1 to compile C/C++, Rust and Go code, respectively. To translate C to WebAssembly, we use Emscripten [38] v1.38.16. WebAssembly compilation from Rust and Go is provided the official compilers. We use Node.js v10.11 on top SGX-LKL (commit `5fb6d120`) to instantiate and run WebAssembly code. Exclusively for cycle measurements (see § 5.2), we additionally use the WAVM runtime [30] (commit `6c5d2465`).

5.1 Sandboxing Overhead

To assess the impact of the various sandboxing techniques as well as the instrumentation across a range of programs we use the PolyBench/C [50] benchmark suite (version 4.2.1). The suite consists of 29 programs performing various compute-intensive tasks like matrix multiplication, Gaussian filters and image processing. An earlier version of the same benchmark suite was used in the original WebAssembly paper [11] to demonstrate the competitiveness of WebAssembly over native code. We compile the code with the highest optimisations level (flag `-O3`). The times reported only include the actual program runtime excluding VM startup and compilation of WebAssembly to native code.

We compare four setups: execution of WebAssembly in Node.js (`WASM`) to measure WebAssembly-related overhead. Next, we run Node.js on SGX-LKL in simulation (`WASM-SGX SIM`) and hardware mode (`WASM-SGX HW`). This allows us to evaluate the impact of SGX and SGX-LKL separately. Finally, we measure the instrumented WebAssembly code (using `loop-based`) executing in Node.js on top of SGX-LKL in hardware mode (`WASM-SGX HW instrumented`). Fig. 6 shows runtime of these setups normalised to native code. The overhead of `WASM` is between −45% and 80% which aligns with the numbers reported in the original WebAssembly paper [11]. For `WASM-SGX SIM` the overheads range from −41% to 60%, i.e. SGX and SGX-LKL do not add overhead by themselves. Only when running in SGX hardware mode do the overheads change to between −18% and 244%. For programs with a large increase in overhead when moving from simulated to hardware SGX mode we identified EPC paging as the main contributor. If future versions of SGX come with a significantly increased EPC, it will mitigate the source of this overhead. AccTEE's instrumentation (`loop-based`) adds no overhead in the best case and 9% in the

worst case over `WASM-SGX HW`. Averaged over all benchmarks, the execution times for `WASM` and `WASM-SGX HW` are 1.1× and 2.1× higher, respectively. The instrumentations, on average, add 4% over `WASM-SGX HW`.

To summarize, WebAssembly inside Intel® SGX shows competitive performance compared to the native execution. Only when the workload exceeds the available EPC memory (currently 93 MB), does the performance degrade noticeably. However, the overhead of the accounting instrumentation is small compared to other sources of overhead.

WebAssembly Overhead. The overhead of WebAssembly itself has already been evaluated in several publications. The paper which proposed WebAssembly in 2017 by Haas et al. [11] also uses the PolyBench/C benchmark suite and reports an execution overhead below 10% for 7 of all benchmarks and below 2× for nearly all of them. We were able to reproduce these results in our own measurements (see § 5.1). A more recent work by Jangda et al. [51] proposes to use a more established benchmark suite of larger and more diverse programs, namely the SPEC benchmark suite. However, the execution of these benchmark programs requires development of system support, as the programs rely on system calls. For this benchmark, the authors report average execution overheads between 45% for Firefox and 55% for Chrome. According to the authors, these higher overheads originate partly from design constraints of WebAssembly. However, other parts of the overheads stem from missing optimizations or flaws in code generation and can therefore be mitigated by using different compilers.

5.2 WebAssembly Instruction Weights

As described in § 3.7, AccTEE's weighted instruction counter relies on a list of weights for all WebAssembly instructions. These weights are provided by AccTEE as part of the trusted execution environment. Although the weights depend on CPU architecture, CPU generation and the WebAssembly runtime, we expect them to be comparable across different combinations thereof. To measure the cost of WebAssembly instructions, we extended the WebAssembly runtime by a function for reading the time stamp counter (TSC) register to measure the CPU cycles to execute each WebAssembly instruction. For the following two measurements, we use WAVM [30] as the WebAssembly runtime instead of Node.js. WAVM is less complex and thus easier to extend for these specific micro benchmarks, but does not yet offer the same functionality.

Specific Instruction Costs. For every WebAssembly instruction, we generate WebAssembly code that executes the instruction n times after pushing the correct number of necessary operands to the stack. The number of cycles per instruction is then calculated by dividing the total number of cycles by n. These values are then used as weights for AccTEE's resource accounting for the given CPU/runtime combination. The results for $n = 10.000$ are depicted in Fig. 7. It shows the cycles for 127 WebAssembly instructions, which excludes `load` and `store` instructions. We see that the majority of WebAssembly instructions (74%) are executed in less than 10 cycles. Fewer instructions (e.g. `f32.floor` and `f64.ceil`) need up to 32 instructions. Only a few exceptions (e.g. `i64.div_s` and `f32.sqrt`) are especially expensive, needing more than 50 cycles. Please note that these numbers include a low benchmarking overhead, but still give a good overview of the distribution of costs.

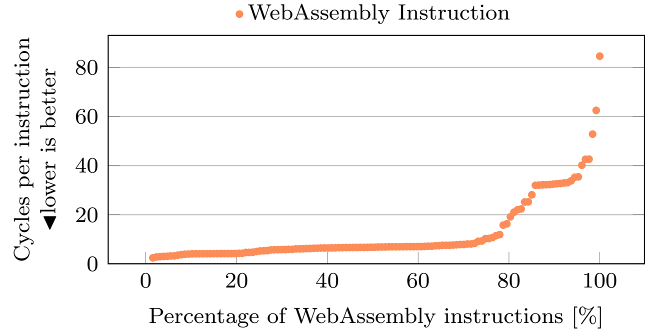

Figure 7. Distribution of cycles needed for execution of WebAssembly instructions.

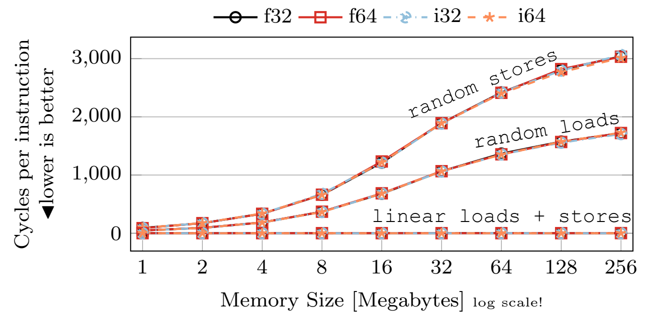

Figure 8. Average cycles for memory accesses from WebAssembly in dependence of memory size, comparing load and store operations with linear and random access patterns.

Costs for Memory Accesses. We evaluate memory accesses separately, as their costs are dependent on the memory access pattern and the size of the linear memory (see § 3.5). In WebAssembly, memory accesses are performed using special `load` and `store` instructions, which read or write variables from or to a given offset in the linear memory. These instructions exist for all available data types in WebAssembly, e.g. `f32`, `f64`, `i32`, `f64`. To explore the costs of these instructions, we perform the following benchmark: For memory sizes between 1 MB and 256 MB and for all WebAssembly types, we execute 10.000 `load` and `store` instructions with different access patterns: linear and random access within the given range. Fig. 8 shows the average cycles per load/store of this benchmark. First, we see very similar results for the WebAssembly types. The next insight is that random `store` instructions are up to 1.8× more costly than random `load` instructions (at 256 MB), which in turn are much more expensive (up to 1700×) than linear `load/store` instructions due to the increasing probability of cache misses. These measurements show that the cost of instructions accessing memory does in fact depend on the size of the linear memory and the memory access pattern as described in § 3.7.

5.3 Use Case Scenarios

Function-as-a-Service. To evaluate AccTEE in the FaaS domain, we measure the request throughput of two FaaS functions: The

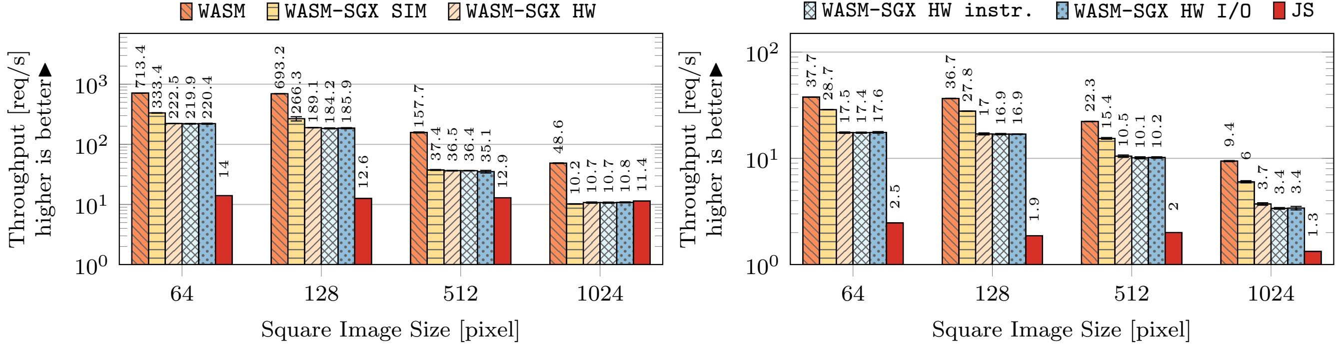

Figure 9. Comparison of throughput of echo function (left) and resize function (right) deployed using OpenFaas or AccTEE.

echo function replies with its input. The resize function returns the input JPG image scaled to 64 × 64 pixels. We generate random input images with sizes between 64 and 1024 pixels (4 KB to 1 MB). We choose these two as representative functions: resize is an example for a compute-heavy function, whereas echo does not perform any computation itself. We induce load with h2load [52] using 10 concurrent clients and compare six setups: AccTEE behind a Node.js HTTP server, without SGX, with SGX in simulation and hardware mode, instrumented WebAssembly code (using loop-based) executed in an SGX enclave with and without I/O accounting, and a pure JavaScript implementation in a Docker container deployed on OpenFaaS [53]. These are referred to as WASM, WASM-SGX SIM, WASM-SGX HW, WASM-SGX HW instr., WASM-SGX HW I/O and JS. To maintain isolation between the functions, the HTTP Server instantiates a new WebAssembly module for every incoming request. The I/O accounting is implemented in JavaScript and counts the number of bytes written to and from the WebAssembly module. The implementations of the image resize functions are based on JIMP [54] for JavaScript and the zero-dependency library zupply [55] for WebAssembly. Fig. 9 shows the mean throughput of 10 runs targeting either the echo or resize function for the aforementioned setups. We observe a reduction in throughput of 2.1× up to 4.8× for echo when moving the workload to SGX-LKL. The additional performance drop when using an actual SGX enclave is negligible for larger inputs and up to 50% for small ones. The high overheads are due to the echo function performing no computation. This benchmark is a worst case scenario exposing the inefficiencies in the intermediate software layers when moving WebAssembly to SGX-LKL. Because the resize function is compute-heavy, the relative overheads of moving the computation into SGX-LKL are less pronounced, they are between 31% and 56%. Here, using an actual SGX enclave has an additional performance penalty between 47% and 64%. Compared to JavaScript, the throughput of the two functions deployed on AccTEE is up to 16× higher. For both functions, the overheads of the accounting instrumentation and the I/O accounting are nonexistent or negligible. For all these measurements, there is no noteworthy error to report.

Volunteer Computing. From the volunteer computing domain (see § 2.1), we evaluate three BOINC programs. We adapted the *NFS@Home*'s [56] *MSieve* program [57] where participants calculated (the project is dormant since 2016) integer factorisations

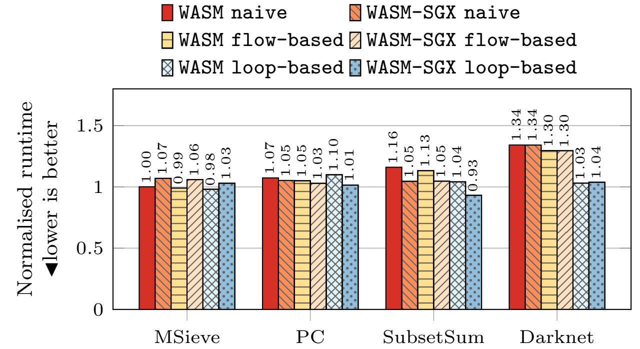

Figure 10. Overhead of instrumentation optimisations on volunteer computing and pay-by-computation use cases. Runtimes are normalised to no instrumentation on WASM and WASM-SGX.

of large random numbers. The second use case is an implementation [58] of the PC algorithm (named after its inventors Peter and Clark) [59], the core component of the still active *gene@Home* project [60]. Our last use case is the ongoing *SubsetSum@Home* project [61, 62], which examines the computation of subset sums and aims to prove a hypothesis regarding the complexity class of the problem. Fig. 10 shows the runtime overhead of these programs; it ranges between −7% and 10%.

Pay-by-Computation. As an example for this domain, we use a machine learning workload that classifies images. The motivation is to replace advertisements when browsing the web with contributions of compute resources. For evaluation, we compiled the machine learning system Darknet [63] to WebAssembly and use its pretrained reference model for image classification. Fig. 10 highlights the significance of our optimisations for this particular use case. The unoptimised instrumentation (naive) increases the execution time by 34%. At the highest optimisation level (loop-based) the instrumentation's impact is reduced to only 3% compared to uninstrumented WebAssembly and to only 4% for WebAssembly on SGX.

5.4 Binary Size Overhead

This experiment covers the impact of our code instrumentation on the size of the WebAssembly binaries. We compare all 136 WebAssembly binaries used in the evaluation. Their non-instrumented

code size ranges from 0.5 KB to 901 KB. The instrumented binaries are between 4% and 39% larger without optimisations and between 4% and 27% after enabling all optimisations.

6 Related Work

Volunteer Computing and Remote Computation. In the area of volunteer computing, several works have been published: Sarmenta [64] and Kopal et al. [65] propose mechanisms to oppose malicious computation nodes in volunteer computing systems.

Airtnt [66] outlines a system for reimbursing users for provided computational resources based on a remote attestation-capable TEE and smart contracts.

Ryoan [5] implements a distributed sandbox using WebAssembly's predecessor NaCl [35] and SGX enclaves, but does not address accounting of resources.

Instead of producing a resource usage log like AccTEE, *LibSEAL* [67] creates an audit log of service operations to verify the integrity of Internet services. This audit log is trusted, as it is also created within an SGX enclave.

Airbox [68] is similar to AccTEE, as it considers trusted execution of functions offloaded to remote, untrusted devices. Airbox targets *edge functions*, that are moved closer to end users for reduction of latency and bandwidth usage. By utilising Docker containers containing SGX enclaves, Airbox protects the execution of edge functions, which may process sensitive user data or business logic. In contrast to AccTEE, Airbox does neither consider the case of malicious functions, nor the accounting of resources consumed during execution.

Brenner and Kapitza [69] present a generic and secure serverless computing infrastructure based on the V8 engine an SGX enclaves. In contrast to AccTEE, they focus on FaaS only and do not consider resource accounting.

Similar to AccTEE, another approach Vrancken et al.[70] utilise SGX enclaves to perform distributed computations on untrusted workers. However, as incentive systems are explicitly out of scope, the system does not take resource accounting into account, as AccTEE does. In general, AccTEE's contributions could be applied to that work.

Resource Accounting. *VeriCount* [71] proposes a verifiable accounting system for resources consumed inside SGX enclaves in a cloud computing scenario for billing purposes, benefiting both client and cloud provider. They add accounting functions using a special compiler and implement checks during execution. However, the authors of VeriCount incorrectly use the trusted time functionality of SGX: as described in § 2.2, trusted time can only be used as a lower bound. This can be exploited by a malicious cloud provider to bill clients for artificially inflated CPU usage.

S-FaaS [27] offers accountability for trusted cloud functions. The system is based on SGX for trusted execution and TSX for transactional memory as well as hyper-threading. It utilises two dedicated threads inside one SGX enclave: a timer and a worker thread. The former executes a busy loop to measure how many CPU cycles the worker thread executes. Enclave exits are synchronised across both threads using TSX transactions. This approach performs correct accounting on a fine-grained level, as actual CPU cycles are measured. However, it consumes 2× CPU resources, as the timer thread expends the same number of cycles as the worker. Additionally, the approach conceptually requires single-threaded workloads. Finally, unlike AccTEE, the approach relies on multiple hardware features and their interaction specific to Intel® CPUs: SGX, TSX and hyper-threading.

Instrumentation. Resource accounting as a means for limiting resource usage by instrumenting Java code [72, 73] has been proposed before and could be adapted for AccTEE. However, none of the these approaches considered trusted accounting based on a two-way sandbox.

REM [74] proposes a less wasteful mining process for blockchains based on code instrumentation and execution inside SGX enclaves. Similar to our least optimised instrumentation approach, REM increments an instruction counter for each executed basic block. In contrast to AccTEE, REM does not guard itself against malicious workloads and leaves the instruction counter vulnerable to manipulation by the workload. While REM's approach could in principle be ported to other architectures, AccTEE is inherently platform independent because it builds on WebAssembly. Finally, REM conceptually requires a single thread inside the enclave, while multi-threaded workloads are possible with WebAssembly but at this point are not supported in AccTEE.

Varys [25] attempts to detect side-channel attacks by determining the ratio of executed instructions to enclave exits. To this end, it instruments code on a basic block level of the LLVM intermediate representation, similar to our unoptimised instrumentation. The resulting instruction count is then compared to a time calculated using a timer thread, similar to S-FaaS, to detect unusually frequent enclave exits. As the workload is assumed to be benign, however, this approach would not work for our threat model, as a workload provider could attempt to influence the accounting.

7 Conclusion

In this paper, we presented AccTEE, a system that offers a two-way sandbox enabling mutually trusted accounting of resource usage for workload and infrastructure providers. Utilised core technologies are hardware-protected trusted execution and WebAssembly. While the former helps protect the integrity and confidentiality of consumer-owned workloads, the latter helps protect the provider from malicious code and enables fine-grained and low-overhead resource accounting based on automated code instrumentation and a weighted instruction counter.

Acknowledgments

This work has received funding from Intel® Corporation in the scope of the *TFaaS* research project and from the German Research Foundation (DFG) under grant no. KA 3171/9-1.

References

[1] ARM Ltd., "Introducing ARM TrustZone." https://developer.arm.com/ip-products/security-ip/trustzone, 2019.
[2] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, "Innovative Instructions and Software Model for Isolated Execution," HASP, 2013.
[3] A. Baumann, M. Peinado, and G. Hunt, "Shielding Applications from an Untrusted Cloud with Haven," OSDI, 2014.
[4] S. Arnautov, B. Trach, F. Gregor, T. Knauth, *et al.*, "SCONE: Secure linux containers with Intel SGX," OSDI, 2016.
[5] T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel, "Ryoan: a distributed sandbox for untrusted computation on secret data," OSDI, 2016.
[6] D. Goltzsche, C. Wulf, D. Muthukumaran, K. Rieck, P. Pietzuch, and R. Kapitza, "TrustJS: Trusted Client-side Execution of JavaScript," EuroSec, 2017.

[7] A. Iosup, N. Yigitbasi, and D. Epema, "On the performance variability of production cloud services," CCGrid, 2011.
[8] G. C. Fox, V. Ishakian, V. Muthusamy, and A. Slominski, "Status of Serverless Computing and Function-as-a-Service(FaaS) in Industry and Research," *arXiv preprint arXiv:1708.08028*, 2017.
[9] D. P. Anderson, "Boinc: A system for public-resource computing and storage," GridCom, 2004.
[10] S. M. Larson, C. D. Snow, M. Shirts, and V. S. Pande, "Folding@Home and Genome@Home: Using distributed computing to tackle previously intractable problems in computational biology," *arXiv preprint arXiv:0901.0866*, 2009.
[11] 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 *PLDI*, 2017.
[12] "Boinc statistics." https://boincstats.com/en/stats/-1/project/detail/overview, 2019.
[13] "Folding@home statistics." https://stats.foldingathome.org/, 2019.
[14] "iexec: Blockchain-based decentralized cloud computing." https://iex.ec, 2019.
[15] "Golem: Worldwide supercomputer." https://golem.network, 2019.
[16] "We're making decentralization work.." https://enigma.co/, 2019.
[17] Cloudflare Inc., "Cloudflare workers." https://www.cloudflare.com/products/cloudflare-workers.
[18] Symantec Inc., "Internet security threat report," 2013.
[19] Google Inc., "Building a better web for everyone." https://blog.google/technology/ads/building-better-web-everyone/, 2017.
[20] S. Gueron, "A Memory Encryption Engine Suitable for General Purpose Processors.," *IACR Cryptology ePrint Archive*, 2016.
[21] I. Anati, S. Gueron, S. Johnson, and V. Scarlata, "Innovative technology for CPU based attestation and sealing," HASP, 2013.
[22] S. Johnson, V. Scarlata, C. Rozas, E. Brickell, and F. Mckeen, "Intel® Software Guard Extensions: EPID Provisioning and Attestation Services," 2016.
[23] V. Scarlata, S. Johnson, J. Beaney, and P. Zmijewski, "Supporting Third Party Attestation for Intel® SGX with Intel® Data Center Attestation Primitives," tech. rep., Intel Corporation, 2018.
[24] S. Brenner, C. Wulf, D. Goltzsche, N. Weichbrodt, M. Lorenz, C. Fetzer, P. Pietzuch, and R. Kapitza, "SecureKeeper: Confidential ZooKeeper using Intel SGX," Middleware, 2016.
[25] O. Oleksenko, B. Trach, R. Krahn, M. Silberstein, and C. Fetzer, "Varys: Protecting SGX Enclaves from Practical Side-Channel Attacks," ATC, 2018.
[26] M.-W. Shih, S. Lee, T. Kim, and M. Peinado, "T-SGX: Eradicating controlled-channel attacks against enclave programs," NDSS, 2017.
[27] F. Alder, N. Asokan, A. Kurnikov, A. Paverd, and M. Steiner, "S-FaaS: Trustworthy and Accountable Function-as-a-Service using Intel SGX," *arXiv preprint arXiv:1810.06080*, 2018.
[28] World Wide Web Consortium (W3C), *WebAssembly Core Specification*, 2019.
[29] Node.js Foundation, "Node.js." https://nodejs.org, 2019.
[30] A. Scheidecker *et al.*, "Webassembly virtual machine." https://github.com/WAVM/WAVM, 2019.
[31] Parity Technologies, "Wasm interpreter in Rust." https://github.com/paritytech/wasmi, 2019.
[32] D. Gohman *et al.*, "wasmtime: Standalone JIT-style runtime for WebAsssembly, using Cranelift." https://github.com/CraneStation/wasmtime, 2019.
[33] S. Akbary, "wasmer: Universal Binaries Powered by WebAssembly." https://github.com/CraneStation/wasmtime, 2019.
[34] S. Garfinkel and G. Spafford, *Web Security, Privacy and Commerce.* O'Reilly Media, Inc., 2 ed., 2002.
[35] 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," S&P, 2009.
[36] A. Donovan, R. Muth, B. Chen, and D. Sehr, "PNaCl: Portable Native Client Executables," *Google White Paper*, 2010.
[37] A. Z. David Herman, Luke Wagner, "asm.js specification." http://asmjs.org/spec/latest, 2019.
[38] A. Zakai, "Emscripten: an LLVM-to-JavaScript compiler," OOPSLA, 2011.
[39] S. Akinyemi, "Awesome WebAssembly Languages." https://github.com/appcypher/awesome-wasm-langs, 2019.
[40] N. Weichbrodt, A. Kurmus, P. Pietzuch, and R. Kapitza, "AsyncShock: Exploiting Synchronisation Bugs in Intel SGX Enclaves," ESORICS, 2016.
[41] S. Lee, M.-W. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado, "Inferring fine-grained control flow inside SGX enclaves with branch shadowing," in *USENIX Security*, 2017.
[42] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "FORESHADOW: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution," USENIX Security, 2018.
[43] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss, "ZombieLoad: Cross-Privilege-Boundary Data Sampling," *arXiv preprint arXiv:1905.05726*, 2019.
[44] R. Strackx and F. Piessens, "The Heisenberg Defense: Proactively Defending SGX Enclaves against Page-Table-Based Side-Channel Attacks," *CoRR*, 2017.
[45] Intel Corporation, "Speculative execution side channel mitigations." https://software.intel.com/security-software-guidance/api-app/sites/default/files/336996-Speculative-Execution-Side-Channel-Mitigations.pdf, 2019.
[46] AMD Inc., "AMD Secure Encrypted Virtualization (SEV)." https://developer.amd.com/sev/, 2019.
[47] P.-L. Aublin *et al.*, "SGX-LKL." https://github.com/lsds/sgx-lkl, 2019.
[48] C.-C. Tsai, D. E. Porter, and M. Vij, "Graphene-sgx: A practical library os for unmodified applications on sgx," ATC, 2017.
[49] S. Shinde, D. Le Tien, S. Tople, and P. Saxena, "Panoply: Low-TCB Linux Applications With SGX Enclaves," NDSS, 2017.
[50] L.-N. Pouchet *et al.*, "PolyBench/C the Polyhedral Benchmark suite." https://sourceforge.net/projects/polybench/, 2018.
[51] A. Jangda, B. Powers, E. D. Berger, and A. Guha, "Not So Fast: Analyzing the Performance of WebAssembly vs. Native Code," in *ATC*, 2019.
[52] "h2load." https://nghttp2.org/documentation/h2load.1.html, 2019.
[53] A. Ellis *et al.*, "OpenFaaS: Serverless Functions Made Simple." https://github.com/openfaas/faas, 2019.
[54] O. Moran, "JIMP: JavaScript Image Manipulation Program." https://github.com/oliver-moran/jimp, 2019.
[55] J. Z. Zhang *et al.*, "Zupply - A light-weight portable C++ 11 library for Researches and Demos." https://github.com/zhreshold/zupply, 2019.
[56] G. Childers, "Factorization of a 1061-bit number by the special number field sieve.," *IACR Cryptology ePrint Archive*, 2012.
[57] jasonp_sf, "Msieve." https://sourceforge.net/projects/msieve, 2019.
[58] F. Asnicar, "PC++ (BOINC version)." https://bitbucket.org/francesco-asnicar/pc-boinc/overview, 2016.
[59] M. Kalisch and P. Bühlmann, "Estimating high-dimensional directed acyclic graphs with the pc-algorithm," *Journal of Machine Learning Research*, 2007.
[60] F. Asnicar *et al.*, "TN-Grid and gene@ home project: Volunteer Computing for Bioinformatics," in *BOINC-based High Performance Computing*, 2015.
[61] T. Desell, "Subsetsum@home." https://github.com/travisdesell/subset_sum_at_home, 2016.
[62] T. O'Neil and T. Desell, "Empirical support for the high-density subset sum decision threshold," CWIT, 2015.
[63] J. C. R. et al., "Darknet: Open Source Neural Networks in C." https://pjreddie.com/darknet/, 2019.
[64] L. F. Sarmenta, "Sabotage-tolerance mechanisms for volunteer computing systems," *Future Generation Computer Systems*, 2002.
[65] N. Kopal, M. Wander, C. Konze, and H. Heck, "Adaptive Cheat Detection in Decentralized Volunteer Computing with Untrusted Nodes," in *DAIS*, 2017.
[66] M. Al-Bassam, A. Sonnino, M. Król, and I. Psaras, "Airtnt: Fair Exchange Payment for Outsourced Secure Enclave Computations," *arXiv preprint arXiv:1805.06411*, 2018.
[67] P.-L. Aublin, F. Kelbert, D. O'Keeffe, D. Muthukumaran, C. Priebe, J. Lind, R. Krahn, C. Fetzer, D. Eyers, and P. Pietzuch, "Libseal: Revealing service integrity violations using trusted execution," EuroSys, 2018.
[68] K. Bhardwaj, M.-W. Shih, P. Agarwal, A. Gavrilovska, T. Kim, and K. Schwan, "Fast, scalable and secure onloading of edge functions using AirBox," in *SEC*, 2016.
[69] S. Brenner and R. Kapitza, "Trust more, serverless," SYSTOR, 2019.
[70] K. Vrancken, F. Piessens, and R. Strackx, "Securely deploying distributed computation systems on peer-to-peer networks," in *SAC*, 2019.
[71] S. Tople, S. Park, M. S. Kang, and P. Saxena, "VeriCount: Verifiable Resource Accounting Using Hardware and Software Isolation," in *ACNS*, 2018.
[72] G. Czajkowski and T. Von Eicken, "JRes: A resource accounting interface for Java," *ACM SIGPLAN Notices*, 1998.
[73] W. Binder, J. G. Hulaas, and A. Villazón, "Portable resource control in java," in *ACM SIGPLAN Notices*, vol. 36, pp. 139–155, ACM, 2001.
[74] F. Zhang, I. Eyal, R. Escriva, A. Juels, and R. Van Renesse, "REM: Resource-Efficient Mining for Blockchains," in *USENIX Security*, 2017.