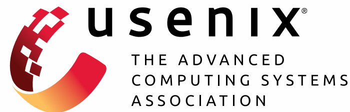

Secured Routines: Language-based Construction of Trusted Execution Environments

Adrien Ghosn, James R. Larus, and Edouard Bugnion, *EPFL*

https://www.usenix.org/conference/atc19/presentation/ghosn

This paper is included in the Proceedings of the 2019 USENIX Annual Technical Conference.

July 10–12, 2019 • Renton, WA, USA

ISBN 978-1-939133-03-8

Open access to the Proceedings of the 2019 USENIX Annual Technical Conference is sponsored by USENIX.

Secured Routines: Language-based Construction of Trusted Execution Environments

Adrien Ghosn James R. Larus Edouard Bugnion
EPFL, Switzerland

Abstract

Trusted Execution Environments (TEEs), such as Intel SGX enclaves, use hardware to ensure the confidentiality and integrity of operations on sensitive data. While the technology is available on many processors, the complexity of its programming model and its performance overhead have limited adoption. TEEs provide a new and valuable hardware functionality that has no obvious analogue in programming languages, which means that developers must manually partition their application into trusted and untrusted components.

This paper describes an approach that fully integrates trusted execution into a language. We extend the Go language to allow a programmer to execute a `goroutine` within an enclave, to use low-overhead channels to communicate between the trusted and untrusted environments, and to rely on a compiler to automatically extract the secure code and data. Our prototype compiler and runtime, GOTEE, is a backward-compatible fork of the Go compiler.

The evaluation shows that our compiler-driven code and data partitioning efficiently executes both microbenchmarks and applications. On the former, GOTEE achieves a 5.2× throughput and a 2.3× latency improvement over the Intel SGX SDK. Our case studies, a Go `ssh` server, the Go `tls` package, and a secured keystore inspired by the go-ethereum project, demonstrate that minor source-code modifications suffice to provide confidentiality and integrity guarantees with only moderate performance overheads.

1 Introduction

Our era is defined by the emergence of a digital society in which established notions of privacy, confidentiality, and trust are undercut by the shortcomings of today's technology, which is increasingly reliant on cloud computing. In the cloud, developers and users implicitly trust the cloud provider but are still susceptible to: (1) hardware and firmware flaws, such as the recent Meltdown [41] and Spectre [36] attacks, (2) vulnerabilities within the hypervisor [3], (3) exploits in libraries and Software as a Service (SaaS) infrastructures [1,2,4,5], (4) malicious employees with physical and administrative access to both computer and storage resources, and (5) intrusive or extra-territorial government surveillance [10, 12, 18].

To address these concerns, processor vendors, following ARM's lead [9], introduced Trusted Execution Environments (TEEs), a hardware mechanism based on memory encryption and attestation that isolates program execution and state from the underlying operating system, hypervisor, firmware, I/O devices, and even people with physical access to a machine. TEEs have been portrayed as *the* solution to the problem of trust in the cloud [16, 17, 39, 51]. In particular, Intel SGX [6] partitions hardware and software into two mutually distrustful domains: a CPU, trusted user code, and a specified region of memory form the *trusted* domain, while the remainder of the hardware and software form the *untrusted* domain. SGX *enclaves* execute trusted user code in a trusted domain. Entering an enclave guarantees, through hardware, the confidentiality and integrity of the enclave's code, data, and execution.

Despite SGX's availability on current-generation processors, uptake has been slow, probably due to the absence of support on server-grade CPUs, the difficulty of programming enclaves, their performance overhead, and the need to refactor applications. The private messaging application Signal [43] is one of the few applications that appears to use enclaves, and Microsoft Azure only recently offered the first cloud solution to expose SGX features [42,45]. A major challenge is that this new technology lacks a clear programming model. Previous solutions fall into two broad categories: (1) run complete user applications in the trusted domain [16,17] and (2) separate the portions of a program that require trusted execution [7, 8, 40]. Solutions in the first category provide an abstraction, such as an operating system [17] or a container [16], to execute unmodified applications in an enclave. The other alternative requires a developer to identify and partition [7, 8, 46], or provide annotations that a program analysis tool can use to partition [40], an application into trusted and untrusted components. None of these prior approaches integrates the TEE into language-specific abstractions and semantics.

This paper describes an approach that fully integrates trusted execution into a modern programming language in an appropriate manner. We extend the Go language to allow a programmer to execute a `goroutine` within an enclave, to use low-overhead channels to communicate between the trusted and untrusted environments, and to rely on the compiler to automatically extract the code and data necessary to run the enclave. Our solution provides language support for trusted execution that is idiomatically compatible with the Go programming language.

We introduce *secured routines*, a new language-based feature that hides the hardware intricacies with little overhead. A secured routine is a user-level thread that executes a closure, *i.e.,* a function call, in the enclave at the request of untrusted code. The secured routine abstraction cleanly distinguishes trusted and untrusted code. Communications between the two domains are possible solely via *cross-domain channels*, an extension to native Go channels that deep-copies values to prevent cross-domain pointer references.

GOTEE extends the Go programming language with a single keyword, `gosecure`, to identify secured routines. GOTEE is an open-source fork of `golang/go` [15]. Starting from `gosecure` calls, the compiler identifies the minimal code required within the enclave and extracts it into a statically-linked trusted binary. Trusted and untrusted domains have their own runtime, memory management, and scheduler. GOTEE coordinates interactions between trusted and untrusted code, replaces control transfers between these domains with inexpensive synchronized data transfers using strongly-typed cross-domain channels.

Our contributions include:

- A language-based, expressive, strongly-typed, high-performance, remote-execution model for TEEs that strengthens isolation between trusted and untrusted code.
- A practical implementation of these ideas using the Go programming language and runtime. Our evaluation using microbenchmarks demonstrates that an enclave core serving secured routines can achieve 5.2× the throughput of domain-crossing control transfers.
- A demonstration that secured routines provide an expressive model to implement secured applications: we partitioned the `tls` module (a built-in Go library), protected a full ssh server, and extended the go-ethereum keystore (a popular cryptocurrency client) to isolate all operations that access private keys and certificates without a significant loss of performance.

We describe the necessary background (§2), the secured routine abstraction (§3), the implementation of GOTEE (§4), and evaluate it using both microbenchmarks and three security-sensitive applications (§5). Finally, we discuss possible architectural improvements in §6, related work in §7, and conclude in §8.

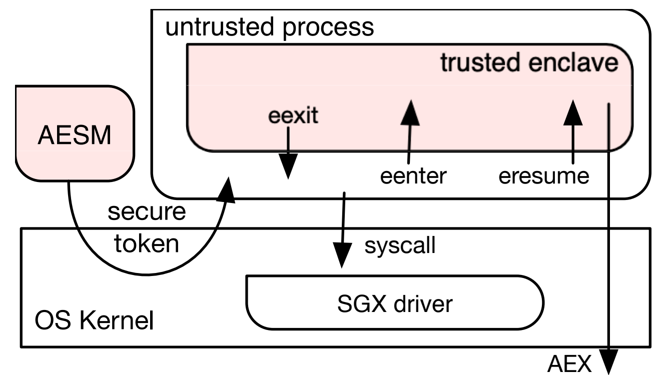

Figure 1: Trusted Execution Environments with Intel SGX; the enclaves and trusted parts are colored.

2 Background

2.1 Intel Software Guard Extension

Intel Software Guard Extension (SGX) [6], introduced in 2015 with Intel's sixth generation Skylake processor, allows user-level creation of *enclaves*. These are contiguous regions of virtual memory, protected against outside access and modification, even by software running at high privilege levels or by I/O devices.

Figure 1 illustrates the partition of a process between secure, trusted code and data and non-secure, untrusted code and data. SGX enforces an asymmetric trust model: the enclave has access to the entire memory, while untrusted code is unable to access or modify enclave memory. SGX further ensures that control from the untrusted domain enters the enclave only at pre-approved entry points. In SGX, Intel provides the root of trust through the `aesm` module that cryptographically ensures the validity of the initial state of the enclave.

SGX reserves, at boot time, a contiguous portion of physical memory, called the *Processor Reserved Memory* (*PRM*), with a maximal size of 128 MB. A 94 MB subset of this region, called the *Enclave Page Cache* (*EPC*), is used to allocate enclave memory pages. The integrity of the EPC is ensured through Merkle trees implemented in hardware [11]. The EPC size is a hard limitation on the amount of code and data that can be loaded into an enclave without incurring expensive page evictions to regular DRAM [17, 53]. The CPU's Memory Management Engine (MME) ensures confidentiality by encrypting cache lines evicted to memory and by decrypting them as they are brought into the CPU running in enclave mode; this reduces the available memory bandwidth by 4× [53].

Creation of an enclave requires the execution of a complex instruction sequence using new instructions such as `ecreate`, `eadd`, `eextend`, and `einit` that respectively create the enclave; define its resources together with their initial state and

access rights; and finally initialize the enclave. The number of concurrent threads allowed inside the enclave corresponds to the number of `eadd`ed Thread Control Structures (*TCS*) and is fixed at enclave initialization.

After enclave initialization, user-level software uses the `eenter` instruction to perform an `ecall`, a control transfer to a pre-defined location within the enclave (Figure 1). The `eexit` instruction allows to perform `ocalls`, *i.e.,* a voluntary control transfer to untrusted code. SGX also supports asynchronous enclave exits (AEX) to service interrupts and exceptions, which is necessary since the enclave forbids privileged instructions. An AEX saves the current state of the enclave within the EPC, restores the untrusted context, and transfers control to the operating system handler. The untrusted code resumes enclave execution by performing an `eresume`.

Finally, SGX provides a remote attestation mechanism that allows developers to verify the integrity of the software in the enclave. As part of enclave creation, developers need to provide a *measurement* of the enclave, *i.e.,* a signed hash of the SGX instructions and arguments used to instantiate the enclave, as well as of selected portions of the enclave's code and data. A remote party can compare this measurement with its expected, precomputed value and proceed with the enclave's execution only if the two values match.

2.2 Building Secured Systems

One approach to utilizing SGX is to run all of an application in the enclave. The literature contains examples of complex abstractions—including an entire operating system, Haven [17]; a library operating system, Graphene [23]; and a container platform, SCONE [16]—running in SGX. While convenient for developers and effective at reducing expensive enclave crossings [53], this approach has significant drawbacks: (1) it greatly expands the amount of code running inside the enclave, which puts pressure on system resources and incurs pervasive memory decryption overheads and (2) it brings into the enclave code and third-party libraries—not necessarily used, understood, or validated—which can facilitate attacks on the enclave (*e.g.,* ROP [47]).

Another approach necessitates a deeper understanding of an application, as it requires splitting the application's code and data into trusted and untrusted portions, following the *Intel SGX Software Development Kit* (*SDK*) [7] model. This SDK is a set of C/C++ libraries and tools that enable programmers to create and deploy enclaves. The Intel SGX SDK exposes an API similar to the SGX instructions. Trusted and untrusted code and data reside within distinct source files. A configuration file describes the required *ecall*s and *ocall*s functions. The compilation process first invokes an IDL compiler to generate boilerplate code and then compiles the enclave code as a position-independent binary with all dependencies statically linked, invokes a signing tool on this `.so` to meter the enclave's code, and finally compiles the untrusted application code as a regular executable.

SCONE [16] and Eleos [44] both rely on message passing to implement asynchronous system calls and avoid expensive enclave exits. Following the same approach, Intel recently published `switchless` [14], a (under-development) mini-framework that provides a simple C++ messaging mechanism on top of the SDK.

Asylo [31] is a C++ framework, compatible with gRPC [32], that abstracts TEE technologies behind a concise API and a set of C++ classes. From a practical point-of-view, Asylo is an improved version of Intel's SDK that exposes a smaller API, requires less boilerplate code, supports different TEE implementations, and provides transparent support to perform system calls from the enclave.

Glamdring [40] automates code partitioning, as it only requires a developer to mark data that needs to be protected. It then relies on static analysis to determine the portion of code that accesses this data and needs to run within the enclave. As an optimization, Glamdring uses heuristics to enlarge the trusted code base and limit the number of expensive enclave crossings. This can help balance a trade-off between EPC memory consumption and the number of domain crossings. Glamdring provides less fine-grained control over code partitioning than the Intel SGX SDK, but hides the technology's intricacies and exposes a very simple programming model.

2.3 SGX Limitations

The SGX technology, and its implementation on current Skylake processors, presents major performance challenges: while the magnitude of these overheads may change in the future with refinement of the processor's micro-architecture, or by adding dedicated silicon, these overheads are, to some extent, tied to the mechanisms providing confidentiality and integrity in the SGX design.

The limited EPC working set and the reduced memory bandwidth are inherent in the design. Similarly, the control-flow transitions between the trusted and untrusted execution (*i.e.,* ecalls, ocalls, and AEX) are expensive because of TLB shootdowns, CPU state changes, and cache flushes needed to mitigate foreshadow attacks [22]. These domain crossings are an order of magnitude more expensive than a system call [53], between $\sim$2µs [53] and $\sim$3.5µs on our hardware. This corresponds to a throughput of less than 1M enclave entries per second with four cores performing `ecalls` in parallel (see §5.2). Keep in mind that system calls within an enclave require a domain crossing, as SGX is limited to user-level execution, and as a consequence these calls also become an order of magnitude more expensive.

Put together, these limitations require a programmer to worry about the size of the trusted code base and the trusted working set, to reduce the exposed attack surface as well as the frequency of EPC page evictions; to optimize the application for cache locality; to understand precisely the threading model

of the application; and to minimize domain crossings, system calls, interrupts, and signals.

3 Design

The *secured routine* extension to Go enables GOTEE to partition an application's code, data, and execution between trusted and untrusted domains, while *cross-domain channels* reinforce memory isolation and enable cross-domain communication and cooperation. This section presents a high-level description of the design and semantics of GOTEE's extensions.

3.1 Threat Model

We follow the threat model of other work in SGX [16, 17, 23, 40] in which an adversary tries to access confidential data or to damage the SGX enclave's integrity. The attacker has administrative access to the machine and control over both hardware and software, and may modify any code or data in untrusted memory, including the operating system and the hypervisor. We consider Iago attacks [24] for GOTEE's runtime and system call interposition mechanism in Section 4.3.

Denial-of-service attacks, a known limitation of SGX [26], and hardware side channels (*e.g.,* based on caches, page faults, or branch shadowing) are out of scope. We assume a correct underlying implementation of SGX that provides confidentiality and integrity for enclave code and data.

3.2 Quick Overview of `Golang`

The Go programming language (`golang`) is a modern, memory-safe, garbage-collected, structurally-typed, compiled, systems programming language. Go supports concurrency based on the Communicating Sequential Processes (CSP) model [33]. The unit of execution within a Go program is called a *goroutine*, a user-level thread executing a closure that is created by prefixing a function call with the `go` keyword. Goroutines are multiplexed and scheduled on a pool of operating system threads, using a cooperative scheduling model implemented by the Go runtime. Goroutines communicate and synchronize using *channels*, which are synchronized, typed message queues with copy and blocking read/write semantics.

3.3 Secured Routines & Cross-domain Channels

From a programming point of view, a secured routine provides a simple and familiar abstraction that allows a programmer to execute a goroutine within an enclave and to use cross-domain channels to communicate between the trusted and untrusted environments.

```
1  var secretKey *Key
2  func generateSymKey(*io.Reader) *Key {...}
3
4  func InitSymKey(done chan bool){
5      fmt.Println(``Creating a new secret key``)
6      secretKey = generateSymKey(rand.Reader)
7      done <- true
8  }
9
10 func EncryptServer(request, reply chan []byte){
11     for {
12       msg := <- request
13       reply <- secretKey.Encrypt(msg)
14     }
15 }
16
17 func TrustedEncryption(msg []byte) []byte{
18     done := make(chan bool)
19     gosecure InitSymKey(done)
20     _ = <- done
21     request := make(chan []byte)
22     reply := make(chan []byte)
23     msg := []byte(``The quick brown fox...``)
24     gosecure EncryptServer(request,reply)
25     request <- msg
26     res := <- reply
27     fmt.Println(``Encryption done``)
28     return res
29 }
```

Listing 1: Using secured routines to isolate a secret key within the TEE.

Listing 1 presents a sample program that secures a secret encryption key, `secretKey`, within the enclave. The `TrustedEncryption` function uses the `gosecure` keyword to spawn a secured routine that creates the key within the enclave. A subsequent `gosecure` call spawns an encryption server, `EncryptServer`, within the enclave. The untrusted code sends the message to the server (line 25) and gets back the encrypted result (line 26).

The programmer relies on `gosecure` to inform the compiler how to partition the code between trusted and untrusted domains. The compiler determines the functions that can be reached by the execution within the enclave, in this example `InitSymKey`, `EncryptServer` and their dependencies `fmt.Println`, `generateSymKey`, `*Key.Encrypt`, *etc.* GOTEE compiles these functions into a statically-linked executable.

Unlike prior work [7, 8, 40], secured routine's code partitioning does not require disjoint trusted and untrusted code. Functions can exist in both environments, *e.g.,* the function `fmt.Println` in Listing 1.

GOTEE hardens memory isolation between trusted and untrusted domains, as compared with the SGX hardware model, in three ways. First, each domain manages its own set of symbols, data, and global variables independently, allowing them to have distinct copies of data and globals. This also differs from Glamdring [40], where the trusted and untrusted

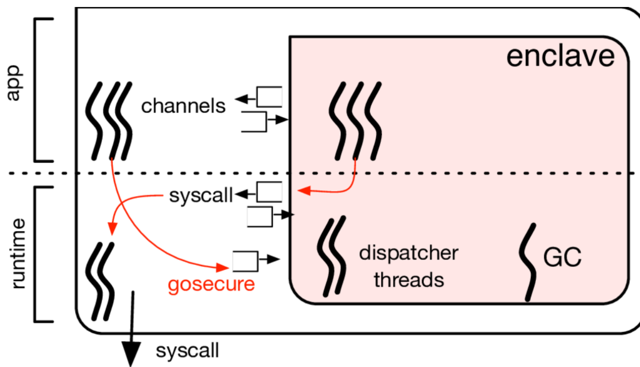

Figure 2: Channel-based cooperation between runtimes.

namespaces cannot overlap.

Second, GOTEE allows only *cross-domain channel*s across the trusted boundary. Cross-domain channels are an extension to the native Go channels allowing secured communications across domains with *deep-copy* semantics. Cross-domain channels are declared and used like regular go channels. However, they provide deep-copy semantics to prevent pointers from crossing a domain boundary. For example, in Listing 1, the `msg` byte slice received at line 12 is an in-enclave copy of the untrusted one sent at line 25.

Third, function arguments passed to secured routines, with the exception of cross-domain channels, are deep-copied inside the enclave by GOTEE's runtime. The deep-copy mechanism can be seen as a marshalling step similar to the one needed to send complex objects or structures over a network. GOTEE emits compilation warnings if a deep-copy, due to a secured routine or a cross-domain channel, requires to dereference a pointer.

While more restrictive than the original SGX model, GOTEE's design ensures that enclave code cannot be subverted or leak secrets by inadvertently dereferencing or writing to an unsafe memory location. All data that leaves the enclave does so by being explicitly sent over a cross-domain channel, while all data referenced by the application's trusted code resides in the enclave.

3.4 Runtime Cooperation

The secured routine abstraction requires mutually distrustful domains to cooperate. More specifically, it allows the untrusted domain to trigger execution of a closure within the trusted one. For example, when the untrusted execution reaches line 19 in Listing 1, the trusted runtime spawns a new routine that invokes the `InitSymKey(done)` closure.

Figure 2 presents the general overview of runtime cooperation. Both domains have their own code and data, their own thread pools to multiplex execution, and their own managed memory regions that are separately garbage collected. Between the two domains, dedicated cross-domain channels are used by the runtimes to trigger the execution of secured routines and to enable enclave system calls. Specifically, and unlike a normal `go` closure, a secured routine is implemented by passing its arguments on a dedicated channel not visible to `golang` programmers. The trusted runtime verifies the validity of the closure's entry point before scheduling it within the enclave. System call interposition operates in a similar manner: the trusted runtime copies the system call's arguments into a dedicated, hidden channel; the untrusted runtime then reissues the system call asynchronously and returns the result over a private channel.

Since full copy semantics are enforced between the two domains, each garbage collector can safely manage its own memory space without synchronizing with the other one.

3.5 Compatibility With SGX

The secured routine abstraction and its design are compatible with the SGX technology and its performance model:

Minimum trusted code: The code loaded into the trusted domain is automatically extracted by the compiler and is minimal. This is both security- and resource-efficient as it reduces the number of EPC pages consumed by the enclave as well as its attack surface.

Control transfers: Control transfers between the two domains are replaced with inexpensive, synchronized, and typed data transfers via cross-domain channels for both application-level communication as well as runtime synchronization. The expensive SGX domain crossings are only necessary in the initialization phase, to block threads when they are idle or in the stop-the-world GC phase, and to service an EPC miss.

Defensive programming: Cross-domain channels, used to launch closures and to invoke system calls, perform memory copies and sanitize arguments. Moreover, they are the single point of interaction between mutually distrustful domains and are therefore easy to augment with defensive programming techniques.

Thread multiplexing: The SGX environment chooses, at enclave creation time, the number of threads that can execute simultaneously within the trusted domain. The Go thread pool size can be fixed at the beginning of the execution to match the number of TCS in the enclave. This, however, does not impose any limitation on the number of concurrently executing secured routines, which means that concurrency is not bounded by this SGX limitation.

System call interposition: The use of channels to communicate and synchronize between the two runtimes simplifies system call interposition. The runtime detects system calls from trusted code, performs argument sanitization, copies arguments to untrusted memory buffers, and sends the system call to the untrusted runtime. Once the system call is serviced, the enclave runtime can perform additional checks to validate the result before delivering it to the application.

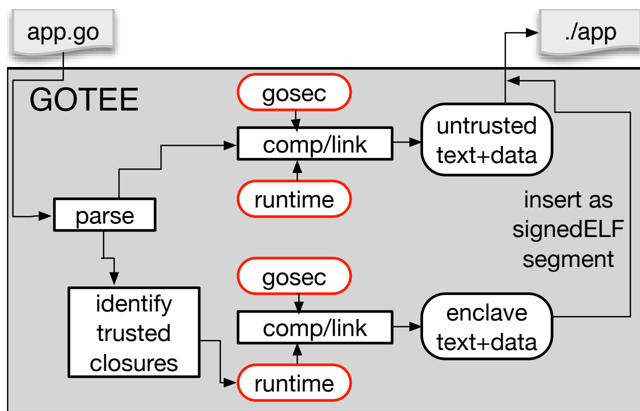

Figure 3: The GOTEE compilation pipeline.

No global variables or cross-domain references: secured routines reinforce the isolation between the two domains by prohibiting shared global variables and cross domain memory references. This forces data sharing to be explicit and passed through either typed communication channels or typed function arguments, with deep-copy semantics. This design eliminates implicit sharing and cross-domain references, which pose the risk of mistakenly leaking data and violating confidentiality.

Secured routines do not provide any guarantee or protection with regards to denial of service attacks. As with previous work [16,23,40], we consider the challenge of bringing secrets into the enclave to be out-of-scope for this paper. These are known, fundamental limitations of the SGX technology that GOTEE does not ameliorate.

Compatibility with other TEE designs: The secured routine abstraction is not tied to the SGX model. From a high-level point-of-view, secured routines and cross-domain channels allow cooperation between two (memory-isolated) peer environments that communicate solely via specific channels. The GOTEE compiler can be extended to support other TEE implementations without requiring application code modifications.

4 Implementation

The GOTEE compiler and runtime extend the Go system. This section describes the changes to the compiler, a new library written in Go that provides SGX support, and the changes to the runtime environment.

4.1 Compiler Support for `gosecure`

The GOTEE compiler is responsible for partitioning code and data according to the design of §3.3. GOTEE is a backward-compatible extension of the standard Go compiler with a new keyword `gosecure`, and an extension to Go channels, *cross-domain channels*. The changes are small, consisting of ∼400 modified lines and ∼2000 lines of new code written in pure Go.

Figure 3 illustrates the process: GOTEE compiles each instance of `gosecure` by type-checking and validating the closures at compile time. The generated code differs slightly from the standard goroutine support. On the caller side, the closure arguments are sent over a cross-domain channel. On the callee side, within the enclave, the runtime library pulls the in-enclave copy of the closure arguments and a function identifier from the channel, validates the target function, spawns the corresponding routine with the arguments, and then schedules it. Compared to a standard goroutine, GOTEE adds a level of indirection, with a write to and read from a cross-domain channel, and the deep-copy of each argument.

GOTEE records functions with the `gosecure` keyword as valid targets for the secured routine abstraction within the enclave. GOTEE then initiates a full compilation for enclave code, using the Go compiler's analysis to determine the minimum transitive closure of code reachable from these functions, as well as the global variables used by this code. The compiler also creates a `main` function for the enclave that serves as the `eenter` entry point and that initializes the runtime servers for cross-domain cooperation. The result of this compilation step is a statically-linked, non-relocatable binary to be loaded into the enclave as the trusted code.

GOTEE implements restrictions on the enclave code. First, the compiler detects channels passed via arguments to secured routines and ensures that these are declared as cross-domain channels. Second, the compiler inspects secured routine's target signatures as well as cross-domain channel types and emits warnings if their deep-copying requires dereferencing pointers. Third, GOTEE does not allow function pointers as arguments to secured routines or cross-domain channels. Finally, GOTEE only allows pure Go code within the enclave and rejects dependencies on C code and shared libraries.

GOTEE also compiles the untrusted code using the standard Go compiler, without these restrictions. As a final step, GOTEE packages the statically linked trusted executable into an ELF segment of the untrusted binary.

GOTEE can optionally generate a signed measurement of the enclave at compile time and store it within a dedicated ELF section of the untrusted binary, so as to perform remote attestation upon deployment. If not done by the compiler, the measurement and signature of the trusted code can be performed at run time.

4.2 `gosec` – an SGX Library in Go

The GOTEE compiler includes an SGX library, completely implemented in Go, as a standard Go package called `gosec`. It contains ∼1000 lines of code.

Loading an enclave: `gosec` mirrors the Intel SGX API in that it provides functions to (1) create an enclave, (2) load

a static binary into the enclave, (3) take a measurement of the enclave, and (4) perform eenter and eresume to the enclave. The gosec package communicates with the Intel SGX Linux kernel driver via ioctl to execute the privileged SGX instructions, *i.e.,* ecreate, eadd, and einit. It also communicates with the Intel aes module [26] that delivers the token required to perform the initialization (einit, see §2.1). The gosec package implements step (2) by parsing the ELF binary and extracting the enclave code. At run time, the package spawns a new, untrusted, operating system thread to execute an eenter instruction that starts the enclave. The number of concurrent threads allowed inside the enclave can be selected by setting an environment variable. By default, the loader adds only two TCS to the enclave: one to execute the user code, the other to support garbage collection.

Measuring an enclave: Measuring an enclave is a series of distinct steps that involve the SGX driver (to execute privileged instructions), the SGX daemon (to retrieve a cryptographic token), the measurement byte array generated by the gosec library while creating the enclave [26] (§4.1), and the enclave binary itself. First, the enclave's memory boundaries are determined by reading the ELF sections of the trusted binary. This information is used to perform the ecreate call. Then, individual page contents are registered via the driver, which performs the eadd and eextend accordingly. At the same time, gosec builds the corresponding measurement byte array, which is then used to retrieve a token from the SGX aes module daemon. Finally, gosec issues the einit driver call, using the token, to finalize the enclave.

AEX handler: Asynchronous exits from the enclave, *e.g.,* faults and exceptions, are first passed to the operating system. Then, a user-space AEX handler, implemented in gosec, is called. The handler runs outside of the enclave and plays a fundamental role in the debugging process. The gosec AEX handler reads a shared region of memory where the GOTEE runtime dumps information before performing a panic or throwing an exception. This, of course, is reliable only for debugging purposes. If no GOTEE runtime cause for the AEX is found, the gosec AEX handler performs an eresume to return in the enclave.

4.3 GOTEE Runtime

The third component of GOTEE is the runtime library that is statically linked to the enclave code. It consists of the Go runtime modified to run in an enclave, including its cooperative user-level thread scheduler and garbage collector, and extensions to allow trusted and untrusted code to cooperate. It supports cross-domain channels as the *sole* means of communications with untrusted code. The code patch consists of ∼760 lines of new code and ∼300 modified ones.

Enclave runtime initialization: GOTEE replaces most of the Go runtime initialization steps. The gosec package preallocates all trusted heap, thread local storage, and memory pools during the enclave creation as part of the load and initialization sequence. This is necessary because of the SGX metering requirements. As a result, the entry point of the enclave simply switches execution onto a protected stack that is part of the enclave and skips over most of the Go runtime memory allocation steps. After this, the enclave runtime shares most of the Go runtime, with minor changes to avoid enclave-disallowed instructions such as cpuid or rdtsc.

Allowing multiple trusted threads: GOTEE lazily spawns enclave threads. During the execution, when a new thread is required, the current enclave thread first atomically acquires a TCS from the pool. It then performs an enclave exit and a clone system call before resuming its enclave execution. While exits and entries are expensive, these are bounded by the maximal number of TCS allocated for the enclave. The newly created thread performs an eenter and jumps to the pre-defined enclave entry point to initialize its state before serving secured routines.

Securing untrusted channels: The channel implementation, as well as the goroutine structure, were extended to support a secured communication mechanism between the trusted and untrusted environments. To pass copies of values to and from secured routines, GOTEE uses buffers allocated within the unprotected memory region. Upon performing a blocking operation, the trusted runtime allocates an unprotected buffer that will either hold the value that it writes, thereby allowing an untrusted routine to access it, or be used to receive the value produced by an untrusted routine's write to the channel. When unblocked, the secured routine copies the content of the buffer to the appropriate memory location within the enclave. For complex types, the enclave performs a deep-copy. This adds an extra step for secured routines compared to standard Go, which allows direct read/write to the blocked goroutine's enqueued address, *e.g.,* a stack, a heap, or a data variable. GOTEE automatically identifies and instruments cross-domain channels at runtime, hence limiting the effort required to port existing applications. Communications within the same domain are unaffected.

Cross-domain synchronization: The two runtimes, and in particular their schedulers, must cooperate to synchronize access to channels across domains to ensure the timely delivery of messages. In Go, a blocking operation on a channel deschedules the routine and wraps it within a special data-structure along with a pointer to the read (write) memory location. In the case of a cross-domain channel, the wrapper must be accessible from both runtimes. GOTEE's enclave runtime manages a private untrusted memory area from which such wrappers are allocated. A secured routine that needs to enqueue itself will therefore allocate a wrapper, along with an untrusted memory buffer, and then enqueue itself in the untrusted cross-domain channel.

The unblocking operations on cross-domain channels

also required changes. An untrusted routine cannot directly reschedule a trusted routine, and vice versa. Instead, unblocking a routine enqueues it in a ready queue that belongs to the appropriate domain. These queues are polled by the corresponding runtime's scheduler. The scheduler ensures that the address of the goroutine is valid, *i.e.,* that it was registered at creation and is still live, before executing it. Note that this extra step only applies to cross-domain communications.

Memory management: The GOTEE runtime restricts the amount of available heap memory because of SGX memory-size limitations. The standard Go runtime assumes a 64-bit address space with gigabytes of memory and places its runtime heap, spans, and bitmap for memory management accordingly. During runtime initialization, and throughout code execution, the Go runtime `mmap`s portions of the address space corresponding to these regions and frequently extends them. An enclave's maximum memory live working-set is 94 MB, and even less if we want to avoid page evictions. As a result, GOTEE uses a fixed-size heap whose address and size are computed as a fixed offset from the code and data. The heap size can be set either at compile time if a measurement is generated or at run time before loading the enclave.

Thread Local Storage: Go relies on thread local storage (*TLS*) to quickly access runtime values such as the current routine (*G*) or the current machine abstraction (*M*). Go normally allocates *M* in the heap and sets it as the TLS base. SGX, on the other hand, requires a TCS to declare its TLS at creation time. GOTEE circumvents the SGX limitations by preallocating *M*s into the enclave's `.bss` segment. As all `.bss` data structures are part of the garbage collector's root set (unlike an arbitrary location in memory), this approach allows the enclave to use the unmodified Go garbage collector.

Garbage collection and Stack shrinking: Go performs mark and sweep concurrent garbage collection. The GC requires a short pause time with all threads blocked at a safe point for mark and sweep terminations. As a result, secured routines need a way to exit the enclave and perform a blocking `futex` sleep. Other than that, the original Go GC is unmodified, and it executes independently from the untrusted domain's runtime. Untrusted memory buffers are allocated and managed by an in-enclave allocation library and are not traced by either GCs. The trusted runtime keeps references to secured routines blocked on cross-domain channels, which both allows a safety check when they are rescheduled and keeps them in the live-set of objects during garbage collection.

Goroutine stacks can shrink and stack frames can be relocated in memory when the goroutine is blocked on a channel. In standard Go, the destination location of channel data may be on the stack, and therefore handled as part of stack relocation. In GOTEE, when a secured routine is blocked on a cross-domain channel, the destination address points to a location in untrusted memory, *i.e.,* not on the stack, while the stack pointer used as the final recipient of the deep-copy is the one updated during stack shrinking.

Mitigating SGX limitations: The current version of SGX disallows several instructions in the enclave, such as `syscall`, `cpuid`, and `rdtsc`. While these have to be completely avoided during the runtime initialization, due to the limited environment at that time (no heap or channels during the early init phases), they can later be emulated. The system call interposition mechanism allows the enclave to forward system calls to the untrusted runtime. The same mechanism can be used to execute a `rdtsc`, with the communication overhead reducing its accuracy. For the `cpuid` call, most of the information provided by the instruction is fixed at enclave creation, which simplifies its emulation.

Go relies on `futex` calls to implement locking within the runtime. These are optimistic locks, performing a limited amount of spinning before sleeping. In an enclave, a `futex sleep` would require to exit the enclave and re-enter upon a `futex wake up`, with high overheads. Instead, in GOTEE, a secured routine that needs to obtain a cross-domain channel lock will spin until it acquires the lock. Upon an unlock, GOTEE checks if any unsafe thread is sleeping on the `futex`. If so, it spawns a dedicated routine to use the system call interposition to perform the unblocking `futex wake up` system call. This approach is similar to the one used by standard Go for blocking system calls, except that GOTEE relies on routines rather than operating system threads.

Network support: The Go runtime relies on `epoll` calls, as part of the scheduler's logic, for network events. GOTEE extends the scheduler's implementation to ensure that a single idle thread at a time is allowed to exit the enclave and perform the `epoll` call.

Iago Attacks: GOTEE's runtime is hardened against Iago attacks and only relies on 4 syscalls: `mmap` to allocate unsafe memory, checked against enclave boundaries and known unsafe areas; `futex` calls for idle threads, which are used to reduce CPU utilization, not mutual exclusion; and `epoll` calls performed by idle threads as described above.

On the application side, GOTEE provides a single point of system call interposition which relies on channels with deep-copy semantics for memory isolation. This currently performs system call filtering and safety checks on both arguments and results, and could be extended, in the future, to allow user-defined filtering policies.

Debugging: Debugging code within an enclave is challenging as the AEX user-space exception handler provides little information to identify the cause of an asynchronous exit from the enclave. GOTEE has an optional flag that allows a program to run in a simulation environment with identical memory layout and run time behavior as the SGX program, but without the SGX protection mechanisms.

	Workload	Text	Data	RO-data	Total	Main Package Dependencies	Application LOC
GOTEE	runtime only	493	25	273	793	-	-
	hello world	+72	+1	+16	+91	++ fmt, syscall, strconv, os, io, reflect, runtime, unicode	13
	enclave-cert	+174	+1	+45	+221	++ crypto/rsa, math, bytes, hash, unicode	75
	ssh	+1036	+4	+291	+1332	++ golang.org/x/crypto/*, crypto, gnet, encoding	71
	keystore	+1165	+ 4	+329	+1499	++ crypto/ecdsa, crypto/elliptic, crypto/aes	474
SDK	runtime only	67	2	4	75	-	-
	hello world	+49	+0	+1	+51	-	355

Table 1: Per case-study enclave TCB breakdown in KB, package dependencies, and application lines of code (LOC). + and ++ are, respectively, an increase over the baseline `runtime only`, and over all previous table entries.

5 Evaluation

Our experiments were performed on a Microsoft Azure Cloud Confidential Computing server, with an Intel(R) Xeon(R) E-2176G CPU @ 3.70GHz with 4 physical cores, configured with Ubuntu 18.04 LTS running Linux kernel 4.15.0-1036-azure. GOTEE operates with the standard Intel SGX 2.0 Linux kernel driver (`sgx2`) and attestation daemon (`aesm`). All GOTEE experiments were run with garbage collection enabled and a single thread servicing secured routines in the enclave.

The purpose of our evaluation is to validate: (1) the effectiveness of secured routines as a way to partition code (§5.1); (2) the performance, latency, and throughput of secured routines and their cost in comparison to the crossing-oriented approach of the Intel SDK (§5.2); (3) with three case studies, GOTEE's usability and ability to hide critical secrets within the enclave by executing a full application in the enclave(§5.3), by performing a fine-grained partitioning of a standard Go package(§5.4), and by extending a real-world application with a TEE-specific implementation(§5.5).

5.1 Code Size

We first evaluate the impact of secured routines on the enclave code size. To this end, we add a baseline `hello world` benchmark that invokes `fmt.Println` in the enclave, and compare it to the Intel SDK C++ `hello world` code sample.

Table 1 shows, for each case study, (1) the size of the enclave code measured as an increase on the baseline size of GOTEE runtime for the enclave, (2) the main Go package dependencies, and (3) the application lines of code. Entries in the table are sorted such that each case study only reports extra packages imported compared to previous lines.

First, we observe that both the Go runtime and the generated code are larger than the C++. Second, the `ssh-server` is responsible for the greatest increase, in TCB size, over the runtime baseline, due to its numerous dependencies. This result is expected as this particular case study does not leverage the fine-grain partitioning provided by GOTEE and simply puts the entire application code inside the enclave. The `keystore` prototype only adds a few crypto subpackages to the TCB dependencies.

On the other hand, Table 1 also shows the difference in source-code level complexity between GOTEE and the Intel SDK. In `hello world`, the lack of transparent forwarding of system calls in the SDK requires a programmer to forgo `printf` in the enclave and instead: (1) call `sprintf` to write to an intermediate buffer, (2) define and `ocall` with the IDL compiler, and (3) use it to issue a `write` system call. Additionally, programmers are still responsible for properly implementing all the boilerplate code required to define, create, and load the enclave. As a result, the C++/SDK `hello world` consists of 355 LOC, 13 files, requires 85 lines of configuration, and 161 lines of `Makefile`.

By comparison, the GOTEE 13 lines of code `hello world` compiles with the `gotee build` command.

5.2 Microbenchmarks

This evaluation uses the following microbenchmarks:

- `syscall`-lat: from within a trusted closure, execute a `getuid()` system call in a loop; report the mean latency.
- `gosecure+block`-lat: spawn a trusted closure and wait for a response over a private cross-domain channel; report end-to-end median latency.
- `gosecure-server`-lat: a single secured routine performs blocking writes to a cross-domain channel in a loop. An untrusted routine measures the latency of performing a read on the same channel. The difference between this measurement and `gosecure+block`-lat corresponds to the runtime overhead required to trigger a secured routine.
- `gosecure`-tput: multiple untrusted goroutines concurrently spawn a trusted closure and wait for a response over a private cross-domain channel.

bench-name	Go	GOTEE	SDK
syscall (getuid)	0.23	1.35	3.69
gosecure+block	0.30	1.5	3.50
gosecure-server	0.20	0.60	-

Table 2: Latency microbenchmarks in µs.

- gosecure-server-tput: a single trusted closure receives requests on a public cross-domain channel from multiple concurrent untrusted goroutines and replies individually on private channels, effectively bypassing the runtime cooperation required to spawn new secured routines.

GOTEE latencies: Table 2 compares the latencies of basic operations in Go, GOTEE, and, when applicable, the equivalent C++ implementation with the Intel SGX SDK. All experiments report the median (mean for syscall-lat) over 500K iterations.

The latency to spawn a secured routine and have it write to a private channel is 1.5µs. The equivalent standard Go program has a latency of 0.30µs, suggesting that GOTEE runtime cooperation and SGX memory overheads have an impact of ∼1.2µs (5.0×). We believe that the implementation can be optimized to reduce contention on cross domain events and runtime cooperation overheads. Still, GOTEE shows a 2.3× improvement over the Intel SDK latency, which requires a full crossing (eenter followed by eexit).

For a trivial system call, that requires going through the syscall interposition mechanism over channels, GOTEE is able to achieve a 2.7× improvement over the Intel SDK crossing-oriented approach.

GOTEE throughput: The throughput experiments consist of multiple concurrent requests to the enclave. For the Intel SDK, different threads perform *ecall*s in parallel, yielding a throughput of 281 Kops for one thread and 938 Kops for all four cores.

For GOTEE, Figure 4 presents two variants, running with a single thread inside the enclave: (1) gosecure-tput, the closest in behavior to the Intel SGX SDK, and (2) gosecure-server-tput. The former shows a throughput improvement of 5.2× (1.46 Mops) over the SDK for a single core running in the enclave. GOTEE can allow a single enclave thread to achieve 1.6× the throughput of four cores executing the Intel SDK. GOTEE's throughput depends on the number of concurrent untrusted goroutines (multiplexed on a single thread) performing gosecure calls. For fewer than three untrusted goroutines, the runtime cooperation requiring to reschedule the secured routines dispatcher is the main bottleneck. After that, there are enough concurrent goroutines to avoid blocking the dispatcher.

The second GOTEE experiment shows the benefit of avoiding the secured routine creation overheads. Its performance degrades, however, as contention on the cross-domain channel increases; both runtimes compete to obtain the lock and

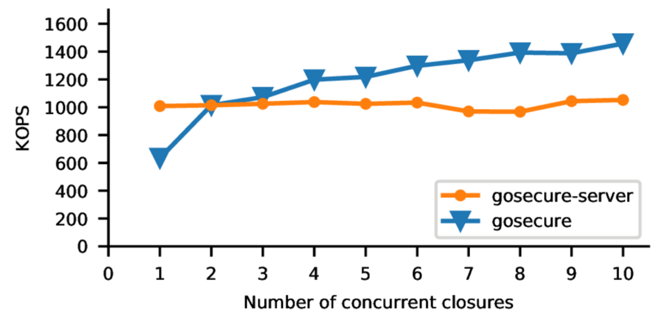

Figure 4: Synchronous closure execution rate for secured routine multiplexed on top of a single enclave thread.

must cooperate to reschedule unblocked routines. Vanilla Go, which is not subjected to our cooperation overhead or SGX performance costs, achieves over 4.1Mops.

The garbage collector's impact on these microbenchmarks is negligible. The Go memory statistics show that for the full throughput experiment, 21 GC cycles were completed inside the enclave, with a median pause time of 13µs. However, the total GC pause time only accounts for 0.033% of the application's available CPU time. In the latency microbenchmark, we measured a similar median for 2 completed cycles, which accounts for 0.015% of the benchmarks CPU time.

5.3 A full in-enclave ssh server

GOTEE can be used to port a full application to the enclave. The enclave size breakdown is reported in Table 1. The Go programming language provides, under golang.org/x/crypto/ssh, a fully functional implementation of an ssh-server. This implementation relies on the default net package. While *none* of the application logic code for the server was changed, this port required a few modifications to the net package, which relies on C bindings for socket structures in order to stay compatible with the Linux kernel headers. As GOTEE allows only pure Go code inside the enclave, we created gnet, a new package that redefines relevant C structures (*e.g.,* struct_sockaddr, struct_in_addr, struct_addrinfo) and constants in pure Go. This package adds 70 LOC to the native net package.

5.4 Webserver with enclave-cert

The loss or leakage of an SSL private certificate can have serious reputational consequences. However, a private certificate must reside in the memory of the process that handles an SSL connection. Our case study designs and implements the enclave-cert package, which isolates within an enclave the two operations that require access to an SSL certificate's private key: signing the handshake hash and decrypting the client's symmetric session key.

We modified the native Go `tls` package to allocate the server's private certificate key within the enclave and to perform these operations in the trusted environment. The `enclave-cert` package uses channels to pass encryption and decryption requests to the enclave. A single secured routine is spawned by the user application when a certificate is loaded or created. The secured routine then waits on the request channel, performs the requested decryption, and notifies the untrusted requester.

The code patch consists of 9 additional LOC that add optional request channels to the TLS certificate structure. The enclave code is in `enclave-cert`, a new package of 35 LOC that defines the operations on the private key. The `http` package is unmodified. Any webserver application that uses `enclave-cert` operates like a corresponding Go webserver application.

The separation of functionality between the `tls` package (which does not depend on GOTEE or `gosec`) and `enclave-cert` eliminates circular dependencies and ensures backward compatibility when SGX is not available.

In this experiment, we have an `apache-bench` client connect repeatedly over `https://localhost` to a simple webserver and request a single page load per session. The workload is totally dominated by the TLS handshakes.

We compare the built-in Go `http` and `tls` packages with the modified `enclave-cert`. The built-in server achieves an average of 400 reqs/sec, while `enclave-cert` achieves 353 reqs/sec (*i.e.,* 88% of native). The `apache-bench` output shows they have the same mean for connection and processing time, but `enclave-cert` has a higher (6×) standard deviation. In fact, the run time cooperation between trusted and untrusted domains is a source of variability that impacts the system's stability and tail latency. A similar experiment with Glamdring [40] reported only 60% of native throughput due to the cost of frequent enclave crossings.

5.5 Keystore based on go-ethereum

The go-ethereum [29] project is the official implementation of the Ethereum protocol [13] in Go. A particular feature of the project is the ability to manage ethereum signature keys (ECDSA) as part of a `keystore`. The go-ethereum project allows safely encrypting keys with a passphrase before storing them on disk. The keystore is responsible for loading and decrypting the keys using the user-provided passphrase. To reduce the window of vulnerability, go-ethereum zeroes-out, in memory, decrypted keys after signing a hash or a transaction.

As a proof-of-concept, we implemented a simplified version of this keystore with GOTEE. The keystore executes in the enclave and enables: (1) loading an encrypted private key from the disk in the enclave, (2) decrypting the private key using a user-provided passphrase (e.g., via a secured ssh connection), and (3) signing a hash if the user validates it. Our keystore is 500 lines of Go code. The primary benefit of this approach is the elimination of the window of vulnerability. The keystore can safely keep private keys cached in secure memory. It took a single developer one day to implement this simplified secured keystore.

The enclave size break down is reported in Table 1. The amount of code loaded in the enclave, more than 1MB, is large compared to other experiments. This is mostly due the embedded ssh server, the cryptographic libraries, *e.g.,* elliptic curves and AES, and the encoding libraries, required to unmarshal decrypted private keys.

Along the TLS benchmark, this implementation validates that GOTEE can support popular Go cryptographic libraries (RSA, AES, and ECDSA) without modifying these packages.

6 Discussion

GOTEE demonstrates that language support for TEEs can alleviate SGX limitations and that the GOTEE programming model can be used to effectively increase the integrity and confidentiality of sensitive server-side computations. At the same time, the viability of SGX, beyond simple use cases in digital-rights management, as a foundational *trust* technology is doubtful given the large number of SGX vulnerabilities found to date and the complexity of the current architecture. SGX is a complex extension to a complex instruction set with an optimized implementation. Verifying the correctness of this extension and of its interactions with the large number of existing instructions is challenging [11, 26]. SGX has already been shown to be vulnerable to side-channel attacks based on caches [20], page faults [49], branch shadowing [38], and processor side-channel attacks ("Foreshadow" [22], a variant of Spectre [36] and Meltdown [41]).

GOTEE's increased isolation and decoupling between trusted and untrusted code, as well as the channel abstraction as the sole mean of communication, allows GOTEE applications to remain agnostic to the underlying technology's programming model. GOTEE seems ideally suited to provide a programming model for more radical TEE designs, that better protect trusted code in isolated environments comprising dedicated cores, TLBs, and (larger) dedicated, encrypted DRAM. One such TEE design could allocate processors and memory at kernel boot time. With a reserved co-processor, its TLB could be dedicated to an enclave and the responsibility of managing the virtual address space could shift from the operating system kernel to a kernel driver, with a small and verifiable implementation. A robust solution would also partition the cache hierarchy to avoid cache-based side-channel attacks.

7 Related Work

A GOTEE-compiled program results in side-by-side execution of two peer environments that communicate over type-

checked, message-passing channels. Using language-based message passing to isolate parts of a program is similar to the Singularity operating system [35], which used strongly typed channels as its only communication mechanism among processes and the kernel.

Program partitioning has been used to transform programs to run sensitive computations on isolated or secure processors. The Jif/split [50] system used security types and information-flow analysis to partition programs so that secure computations could be distributed and executed on trusted processors. Swift [25] partitioned a web app to run its trusted computation on a server. Wedge [19] was a Linux extension that supported least-privileged partitioning and execution of programs. The Crowbar tool used static program analysis to partition programs so that operations could be performed with least privilege. Privtrans [21] partitioned a program to enforce privilege separation. GOTEE, inspired by these systems, provides a language-base, compiler-driven code and data partitioning for TEEs that presents a simple programming model and which could be extended to support other TEE hardware, as well as secured co-processor or remote execution setups.

TrustScript [30] provides language support for running TypeScript (JavaScript) code in an enclave. Similar to GOTEE, it relies on keyword annotation of trusted code and uses asynchronous message passing between the trusted and untrusted runtimes. Unlike GOTEE's automated, fine-grain partitioning, TrustScript developers must implement all trusted code in an annotated namespace, and the TrustScript's security model is unclear.

Glamdring [40] uses data-driven code partitioning between an SGX enclave and an untrusted environment. The compiler and toolchain try to reduce the number of enclave crossings by bringing more code into the enclave. GOTEE takes a different approach, as it provides programmers with fine-grained control over the TCB, a stricter memory isolation between the two domains, and replaces enclave crossings with channel communications.

The debate on the relative merits of the crossing-oriented abstractions of the Intel SDK and the communication-oriented abstraction of GOTEE is of course a new twist on the duality of shared memory and message passing [37]. While numerous systems have been built with the domain-crossing approach embodied in the Intel SDK (§2.2) [8, 17, 40], the current implementation of SGX favors an asynchronous, communication-oriented model, as demonstrated by GOTEE and Intel's own recent `switchless` [14]. Other mentioned solutions [14, 16, 23, 30, 31, 44] rely *internally* on message passing to avoid enclave crossings. GOTEE, however, leverages Go channels, an abstraction that is part of a language, type-safe, and widely used. The cross-domain channels extend the general channel programming abstraction and enable developers to use *explicit* cross-domain communication at the application-level. Internally, this single point of interaction allows to perform both static and dynamic safety checks in concordance with the language semantics.

As a general result, GOTEE shows that programming language support, with an appropriate abstraction and programming model, combines the best of previous approaches, *i.e.,* the fine-grained automatic partitioning, the message passing model precluding enclave crossings, as well as a higher level of isolation between the two domains, and provides an interesting testbed for future extensions, such as information flow control or user-defined system call filtering.

Microsoft used SGX in conjunction with machine-code modification and verification to ensure a property called information release confinement that guarantees that attackers can only see encrypted data [48]. Although their C++ programming model is crossing oriented, GOTEE would provide a better starting point as they impose and verify safety restriction on the C++ enclave code that would be unnecessary for a safe language such as Go. Similarly, the Microsoft VC3 [46] map-reduce system requires and checks at run-time an even stronger set of control-flow and memory-safety properties, which again are easily satisfied by Go programs.

Finally, there exist software solutions which rely on layered virtualization to remove any trust dependency from the operating system [27, 28, 34] or the cloud hypervisor [52]. GOTEE could provide a complementary application-level isolation.

8 Conclusion

What comes first, the processor or the programming model? Intel's SGX made a TEE generally available, and its SDK provides a thin veneer that exposes its hardware features as the programming model. As systems are constructed on SGX, it has become increasingly clear that the most effective use of this TEE is to have it execute only trusted operations and to run the bulk of an application outside of the enclave. This paper explores a new programming model to support this style of use. GOTEE provides language support for TEEs. It extends the Go programming language and uses the Go routine mechanism to invoke a function within the enclave. Our compiler uses a single annotation to distinguish trusted code and automatically partition a program and establish cross-domain communication.

GOTEE treats the enclave as a distinct computing entity and uses message passing to copy arguments to functions, which then execute securely in a distinct, secure domain. This alternative model has the advantage of not requiring expensive cross-domain control transfers, resulting in significantly higher performance than the standard option. Equally important, it reduces the close coupling between the trusted and untrusted domains and opens the possibility of new, more easily verified hardware implementations that can better isolate TEE cores and run faster.

Acknowledgments

We thank Mathias Payer, Pascal Felber, the ATC anonymous reviewers, and our shepherd Christof Fetzer for their detailed comments. Moreover, we would like to thank Marios Kogias, George Prekas, and Jonas Fietz for the many discussions and constant feedback that lead to this paper. This work was funded in part by a VMware Research Grant.

References

[1] CVE-2016-5195 - write to read-only memory mappings. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-5195.

[2] CVE-2017-1000366 - glibc vulnerability leading to arbitrary code execution. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-1000366.

[3] CVE-2017-4948 - vmware out-of-bound read leads to confidentiality violation. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-4948.

[4] CVE-2018-2727 - vulnerability in oracle financial services applications. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-2727.

[5] CVE-2018-7160 - node.js dns rebind leads to full code execution access. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-7160.

[6] Intel SGX - software guard extensions programming references. https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf.

[7] Intel SGX SDK - software guard extension, software development kit. https://software.intel.com/en-us/sgx-sdk.

[8] Rust sgx sdk. https://github.com/baidu/rust-sgx-sdk.

[9] Trustzone - arm. https://www.arm.com/products/security-on-arm/trustzone.

[10] The USA Patriot Act. https://www.justice.gov/archive/ll/highlights.htm, 2001.

[11] Intel Software Guard Extension (ISCA Tutorial). https://software.intel.com/sites/default/files/332680-002.pdf, 2015.

[12] CLOUD Act - H. R. 4943. https://www.congress.gov/bill/115th-congress/house-bill/4943/text, 2019.

[13] Ethereum project. https://www.ethereum.org/, 2019.

[14] Intel SGX Switchless - set of features to avoid expensive crossings. https://github.com/intel/linux-sgx/blob/master/sdk/switchless/, 2019.

[15] ADRIEN GHOSN, EPFL DCSL. GOTEE – a fork of Go with support for 'gosecure'. https://github.com/epfl-dcsl/gotee, 2019.

[16] ARNAUTOV, S., TRACH, B., GREGOR, F., KNAUTH, T., MARTIN, A., PRIEBE, C., LIND, J., MUTHUKUMARAN, D., O'KEEFFE, D., STILLWELL, M., GOLTZSCHE, D., EYERS, D. M., KAPITZA, R., PIETZUCH, P. R., AND FETZER, C. SCONE: Secure Linux Containers with Intel SGX. In *Proceedings of the 12th Symposium on Operating System Design and Implementation (OSDI)* (2016), pp. 689–703.

[17] BAUMANN, A., PEINADO, M., AND HUNT, G. C. Shielding Applications from an Untrusted Cloud with Haven. *ACM Trans. Comput. Syst. 33*, 3 (2015), 8:1–8:26.

[18] BERGHEL, H. Oh, What a Tangled Web: Russian Hacking, Fake News, and the 2016 US Presidential Election. *IEEE Computer 50*, 9 (2017), 87–91.

[19] BITTAU, A., MARCHENKO, P., HANDLEY, M., AND KARP, B. Wedge: Splitting Applications into Reduced-Privilege Compartments. In *Proceedings of the 5th Symposium on Networked Systems Design and Implementation (NSDI)* (2008), pp. 309–322.

[20] BRASSER, F., MÜLLER, U., DMITRIENKO, A., KOSTIAINEN, K., CAPKUN, S., AND SADEGHI, A.-R. Software Grand Exposure: SGX Cache Attacks Are Practical. In *Proceedings of the 11th USENIX Workshop on Offensive Technologies (WOOT)* (2017).

[21] BRUMLEY, D., AND SONG, D. X. Privtrans: Automatically Partitioning Programs for Privilege Separation. In *Proceedings of the 13th USENIX Security Symposium* (2004), pp. 57–72.

[22] BULCK, J. V., MINKIN, M., WEISSE, O., GENKIN, D., KASIKCI, B., PIESSENS, F., SILBERSTEIN, M., WENISCH, T. F., YAROM, Y., AND STRACKX, R. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *Proceedings of the 27th USENIX Security Symposium* (2018), pp. 991–1008.

[23] CHE TSAI, C., ARORA, K. S., BANDI, N., JAIN, B., JANNEN, W., JOHN, J., KALODNER, H. A., KULKARNI, V., DE OLIVEIRA, D. A. S., AND PORTER, D. E. Cooperation and security isolation of library OSes for multi-process applications. In *Proceedings of the 2014 EuroSys Conference* (2014), pp. 9:1–9:14.

[24] CHECKOWAY, S., AND SHACHAM, H. Iago attacks: why the system call API is a bad untrusted RPC interface. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XVIII)* (2013), pp. 253–264.

[25] CHONG, S., LIU, J., MYERS, A. C., QI, X., VIKRAM, K., ZHENG, L., AND ZHENG, X. Secure web application via automatic partitioning. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)* (2007), pp. 31–44.

[26] COSTAN, V., AND DEVADAS, S. Intel SGX Explained. *IACR Cryptology ePrint Archive 2016* (2016), 86.

[27] CRISWELL, J., DAUTENHAHN, N., AND ADVE, V. S. Virtual ghost: protecting applications from hostile operating systems. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XIX)* (2014), pp. 81–96.

[28] DONG, X., SHEN, Z., CRISWELL, J., COX, A. L., AND DWARKADAS, S. Shielding Software From Privileged Side-Channel Attacks. In *Proceedings of the 27th USENIX Security Symposium* (2018), pp. 1441–1458.

[29] ETHEREUM. Go Ethereum. https://github.com/ethereum/go-ethereum, 2019.

[30] GOLTZSCHE, D., SIEBELS, T., AND KAPITZA, R. Trustscript: Language support for partitioning trusted web applications. https://www.eurosys2019.org/wp-content/uploads/2019/03/eurosys19posters-abstract100.pdf, 2019.

[31] GOOGLE LLC. Asylo. https://asylo.dev/, 2019.

[32] GOOGLE LLC. gRPC. https://grpc.io/, 2019.

[33] HOARE, C. A. R. Communicating Sequential Processes. *Commun. ACM 21*, 8 (1978), 666–677.

[34] HOFMANN, O. S., KIM, S., DUNN, A. M., LEE, M. Z., AND WITCHEL, E. InkTag: secure applications on an untrusted operating system. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XVIII)* (2013), pp. 265–278.

[35] HUNT, G. C., AND LARUS, J. R. Singularity: rethinking the software stack. *Operating Systems Review 41*, 2 (2007), 37–49.

[36] KOCHER, P., GENKIN, D., GRUSS, D., HAAS, W., HAMBURG, M., LIPP, M., MANGARD, S., PRESCHER, T., SCHWARZ, M., AND YAROM, Y. Spectre Attacks: Exploiting Speculative Execution. *CoRR abs/1801.01203* (2018).

[37] LAUER, H. C., AND NEEDHAM, R. M. On the Duality of Operating System Structures. *Operating Systems Review 13*, 2 (1979), 3–19.

[38] LEE, S., SHIH, M.-W., GERA, P., KIM, T., KIM, H., AND PEINADO, M. Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing. In *Proceedings of the 26th USENIX Security Symposium* (2017), pp. 557–574.

[39] LIANG, X., SHETTY, S., ZHANG, L., KAMHOUA, C. A., AND KWIAT, K. A. Man in the Cloud (MITC) Defender: SGX-Based User Credential Protection for Synchronization Applications in Cloud Computing Platform. In *Proceedings of the 10th IEEE International Conference on Cloud Computing (CLOUD)* (2017), pp. 302–309.

[40] LIND, J., PRIEBE, C., MUTHUKUMARAN, D., O'KEEFFE, D., AUBLIN, P.-L., KELBERT, F., REIHER, T., GOLTZSCHE, D., EYERS, D. M., KAPITZA, R., FETZER, C., AND PIETZUCH, P. R. Glamdring: Automatic Application Partitioning for Intel SGX. In *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)* (2017), pp. 285–298.

[41] LIPP, M., SCHWARZ, M., GRUSS, D., PRESCHER, T., HAAS, W., MANGARD, S., KOCHER, P., GENKIN, D., YAROM, Y., AND HAMBURG, M. Meltdown. *CoRR abs/1801.01207* (2018).

[42] MACKIE, K. Azure Confidential Computing Project Getting Added Partner Support. https://redmondmag.com/articles/2018/05/10/azure-confidential-computing-partners.aspx, 2018.

[43] MARLINSPIKE, M. Technology preview: Private contact discovery for signal. https://signal.org/blog/private-contact-discovery/.

[44] ORENBACH, M., LIFSHITS, P., MINKIN, M., AND SILBERSTEIN, M. Eleos: ExitLess OS Services for SGX Enclaves. In *Proceedings of the 2017 EuroSys Conference* (2017), pp. 238–253.

[45] RUSSINOVICH, M. Introducing Azure confidential computing. https://azure.microsoft.com/en-us/blog/introducing-azure-confidential-computing/, 2017.

[46] SCHUSTER, F., COSTA, M., FOURNET, C., GKANTSIDIS, C., PEINADO, M., MAINAR-RUIZ, G., AND RUSSINOVICH, M. VC3: Trustworthy Data Analytics in the Cloud Using SGX. In *IEEE Symposium on Security and Privacy* (2015), pp. 38–54.

[47] SHACHAM, H. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *ACM Conference on Computer and Communications Security* (2007), pp. 552–561.

[48] SINHA, R., COSTA, M., LAL, A., LOPES, N. P., RAJAMANI, S. K., SESHIA, S. A., AND VASWANI, K. A design and verification methodology for secure isolated regions. In *Proceedings of the ACM SIGPLAN 2016 Conference on Programming Language Design and Implementation (PLDI)* (2016), pp. 665–681.

[49] XU, Y., CUI, W., AND PEINADO, M. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *IEEE Symposium on Security and Privacy* (2015), pp. 640–656.

[50] ZDANCEWIC, S., ZHENG, L., NYSTROM, N., AND MYERS, A. C. Untrusted Hosts and Confidentiality: Secure Program Partitioning. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)* (2001), pp. 1–14.

[51] ZEGZHDA, D. P., USOV, E. S., NIKOL'SKII, V. A., AND PAVLENKO, E. Use of Intel SGX to ensure the confidentiality of data of cloud users. *Automatic Control and Computer Sciences 51*, 8 (2017), 848–854.

[52] ZHANG, F., CHEN, J., CHEN, H., AND ZANG, B. CloudVisor: retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)* (2011), pp. 203–216.

[53] ZHAO, C., SAIFUDING, D., TIAN, H., ZHANG, Y., AND XING, C. On the Performance of Intel SGX. In *IEEE WISA* (2016), pp. 184–187.