EPFL

Thèse n° 8165

Trust as a Programming Primitive

Présentée le 29 septembre 2021

Faculté informatique et communications
Laboratoire des Systèmes de Centres de Calcul
Programme doctoral en informatique et communications

pour l'obtention du grade de Docteur ès Sciences

par

Adrien GHOSN

Acceptée sur proposition du jury

Prof. V. Kuncak, président du jury
Prof. E. Bugnion, Prof. J. R. Larus, directeurs de thèse
Dr G. Hunt, rapporteur
Dr A. Baumann, rapporteur
Prof. M. Payer, rapporteur

École
polytechnique
fédérale
de Lausanne

2021

In computing, turning the obvious into the useful is a living definition of the word 'frustration'
— Alan Perlis

To my parents and my cat...

Acknowledgements

This thesis is the result of five years of work that would not have been possible without a long list of people.

First and foremost, I would like to thank my advisors *Edouard Bugnion* and *James Larus*. Ed accepted me in the DCSL laboratory despite my obvious lack of interest for datacenter networking. He gave me the opportunity to work on the topics that truly interested me and provided me with the freedom to explore my own research ideas. Throughout my PhD, Ed was supportive, trusting, but also demanding, thus finding the right balance to keep me on track while I was slowly building myself up as a researcher. I am also grateful that I had the chance to have Jim as a second advisor. Inviting him to be part of the jury for my candidacy exam was probably one of the most decisive decisions I made during my PhD as it led to our continued collaboration after my first year. Jim has a beautiful mind that I admire. He has the ability to sleep through an entire presentation and still ask better questions than a room full of PhD students and professors; he can summarize, in one simple sentence, complex concepts that others are unable to explain in several paragraphs; he always finds the real research question hidden behind misleading details. Jim was supportive throughout my PhD, pushing me to do more than I orginally thought I could achieve. This PhD would not have been possible without the combination of these two amazing advisors.

At EPFL, I also had the chance to work with *Mathias Payer* on the enclosure project. He brought his security expertise to the project and is one of the friendliest and most approchable faculty members I met.

Next, I would like to thank my external jury members, Dr. *Galen Hunt* and Dr. *Andrew Baumann*. My PhD started with one of their papers [67] and ended with both of them taking part in my private defense. I feel lucky that such talented and renown researchers accepted to officiate as my jury members.

I also need to thank my "shadow" advisor *Marios Kogias*, who has been with me during my entire PhD as a colleague, office mate, a mentor, and a friend. Thanks for tolerating my constant interruptions in the office, for always taking the time to listen to my ideas and discuss them, for continuing to give me time and advices, even after you left EPFL. Thank you also for being a good friend and listening to my girls problems and random anecdotes, and putting up with my childish craziness. I am impatient to get my colleague (and good friend) back!

Of course, the PhD would not have been possible without the support of my family. My dad, *Sejean "Padre" Ghosn*, in his infinite wisdom, came up with a one sentence formula, “Tu peux pas regretter quelque chose qui est pour ton bien” that he generously prodigated during these

past 5 years. *Dorina "Quiche" Ghosn*, my mother whom I love. We both know that you are the one to thank for every single one of my achievements. You taught me how to be curious about everything, strong in the face of adversity, and proud enough to have self-respect but never quite enough to stop improving. I also thank (and blame) you for my hereditary craziness and my exaggerated love of disco music. I thank my sister, *Aline*, for being the moral compass of the family, and *Slomfy*, my cat, that I loved so much.
My English writing skills are not good enough to fit every single person that I would like to thank here in a well-constructed text. Thus, I will default to a list, thanking:
Margaret Church for being our "work" mom. Maggy sees every single one of us through the prism of motherly love, exaggerating our qualities and completely ignoring our imperfections. Thank you for all the love and support you gave me during these 5 years.
Tania Epars for always greeting me with a wide smile and taking the time to chat (your dog will miss me).
George "Daddy" Prekas for contaminating me with the need to have well-organized code that builds, runs tests, generates graphs and TeX entries, and makes a cup of coffee with a single `make` command. Thanks for the drinks at "Great Escape", it has not been the same without you.
Jonas Fietz for the non politically-correct discussions that became a ritual on Wednesdays. Where did you disappear to?
Mia Primorac for the regular breaks and ice-creams. I am still waiting for that truck full of ice-cream you promised me. In the meantime, thank you for always being available for a break and for being my friend.
Georg Schmid for being my friend since the Masters, for the holidays in Austria, and the concerts at "Les Docks" and "Lausanne Cite". I have not seen you quite as much as I would have liked during these five years, but we will make sure to remedy that in the future.
Sahand "Sachou" Kashani for being a friend and one of the nicest guys I ever met, for coming to Crossfit with me every morning at 6am, and for the "Zooburger" ritual. I am sure we will end up working together again. In the meantime, I count on you to Rx all the WODs and beat me at muscle ups.
Mahyar "El Chico" Emami, the Iranian John Snow, for being adorable. I will miss you Chico. I am counting on Sahand to convince you to come back to Crossfit.
Endri Bezati for the smoke/vape breaks. I admire your motivation to go up 2 floors while you could just vape in your office. Thanks for keeping me company on the BC terrasse and the fun conversations.
Rishabh Iyer and *Mark Sutherland*: you guys know the plan. Let's stick to it and see each other very soon.
Ogier and *Val* for always being up for a drink and a fun evening or a short break on campus.
All of my friends outside of EPFL, with a special thanks to *Guillaume Rollin*, my best friend for over 20 years. Having you close-by for the past two years was a huge moral support (and the occasion to make "gainz" on the weekends).
I thank *G.* for embellishing my days with a shy "hi" in the INM corridors for the past two years, and for all the laughs and support she gave me in the last two months of my PhD.

I will miss everyone at JSC Crossfit and more specifically my "Early Birds". I thank *Krista*, my crossfit bro, for all the support and good advice she gave me, and for always reminding me that I am a little snot that should lift heavier, run faster, and jump higher. While the words were harsh, the actions were sweet. Thanks for filling me up with "Mr. Hyde" when I was too lazy to workout, for giving me ice-cream when I had fever, and for being my unlicensed pharmacist. Now, for the unorthodox mentions, I thank Tinder for allowing me to have a social life in Lausanne, far away from computer science conversations; Zed for being the best wingman; my office "Pouf" for all the naps; the Bel Air Mac Donalds for my Sunday cheat meals; and the "Apero Du Captain" podcast for keeping me company and up-to-date with the latest Wazzuf.

Lausanne, July 20, 2021 Adrien Ghosn

Abstract

Programming has changed; programming languages have not. Modern software embraced reusable software components, i.e., public libraries, and runs in the cloud, on machines that co-locate applications from many different sources. This new programming paradigm leads to an unsafe world in which compromising a single public library or cloud server can potentially grant an attacker access to the sensitive data of tens or hundreds of applications.

Meanwhile, programming languages have not yet provided the mechanisms to address the insecurity and fragility inherent in modern software: (1) programs run in a single trust domain, thereby permitting unverified public library code to access their sensitive information and (2) the underlying operating system or hypervisor is able to access a program's sensitive information.

In this thesis, I present two programming abstractions and mechanisms that address these challenges. The first is secured routines, which protect user code and data from untrusted and potentially privileged code. The second is enclosures, a programming abstraction that splits a program into isolated trust domains, allowing safe execution of unverified public libraries. This research highlights the need for new software and hardware mechanisms to provide fine-grained (within an address space) isolation so that programs can be safely constructed from untrusted pieces of code and run in untrusted environments.

Résumé

La programmation a changé, mais pas les langages de programmation. Les programmes modernes englobent des composants logiciels réutilisables, appelées librairies publiques, et s'exécutent dans le Cloud, sur des machines qui regroupent des applications provenant de différentes sources. Ce nouveau paradigme de programmation conduit à un monde peu sûr, dans lequel la compromission d'une seule librairie publique ou d'un seul serveur dans le Cloud peut potentiellement permettre à un individu d'accéder aux données sensibles de dizaines ou de centaines d'applications.

Entre-temps, les langages de programmation n'ont toujours pas évolués afin de fournir les mécanismes permettant de remédier à l'insécurité et à la fragilité inhérentes aux logiciels modernes : (1) les programmes s'exécutent dans un seul domaine de confiance, permettant ainsi au code de librairies publiques non vérifié d'accéder à l'ensemble des informations sensibles et (2) le système d'exploitation ou l'hyperviseur sous-jacent est capable d'accéder aux informations sensibles d'un programme.

Dans cette thèse, je présente deux abstractions et mécanismes de programmation qui répondent à ces défis. La première s'appelle *secured routine* et protège le code et les données de l'utilisateur contre le code non fiable, même s'il exécute avec un niveau de privilège supérieur. La seconde, *enclosure*, est une abstraction de programmation qui divise un programme en domaines de confiance isolés, permettant l'exécution sécurisée de librairies publiques non vérifiées. Cette recherche met en évidence la nécessité de développer de nouveaux mécanismes logiciels et matériels pour fournir une isolation fine (dans un espace d'adresse) afin que les programmes puissent être construits en toute sécurité à partir de morceaux de code non fiables et exécutés dans des environnements potentiellement sous le contrôle d'individus mal intentionnés.

Contents

List of Figures

List of Tables

1 Introduction

Computer scientists did not wait for Facebook to "move fast and break things". Hacking is in our DNA and enabled the new field of Computer Science to make massive leaps forward rapidly. It took only 30 years to go from the first Turing-complete computer [11], which weighted over 5 tons, to the first highly successful mass-produced personal microcomputer, designed by a 27 years old hacker [5]. Half a century later, we are at a stage where our coffee machines, dish washers, or even lightbulbs pack more compute power and better connectivity than the technologies that broke the Enigma machine [17] or took us to the Moon.

Paradoxically, our systems and programming languages have not diverged a lot from the ones designed in the 1960's [35]. These were built to prioritize performance over security, in an era with limited hardware computational power and memory capacities, where most machines were under the control of a single entity, reserved to tech-savy users, and not on a world-wide network. Today, electronic devices are everywhere, used by everyone, are inter-connected by the Internet, and hold valuable private data. With this shift in paradigm, our systems and programming languages are starting to show their inability to address the growing security and privacy concerns of their users. Central to this disconnect between modern needs and legacy designs is one abstraction: the process.

Processes appeared in the 1960s and are still, nowadays, the main unit of scheduling and isolation in our systems. Processes enable concurrent and, on multi-core platforms, parallel execution of isolated programs potentially servicing multiple users.

The process is a schedulable unit of work, *i.e.,* an instance of a running program. Originally, especially on UNIX systems and due to the limitations of the hardware at the time, programs were small pieces of software, focused on providing one particular task, such as a shell or a pass of a compiler, that had to fit within the scarce memory resources at the time and embodied one independent step of an overall computation pipeline. Brian Kernighan and Rob Pike summarize this by describing the *UNIX philosophy* as "the idea that the power of a system comes more from the relationship among programs than from the programs themselves" [127].

The process is also the unit of isolation in the system. In MULTICS [168], the process abstraction is described as being intimately linked to the concept of address space, *i.e.,* each process runs in its own address space, independent from other address spaces.

Half a century later, the process is still the main unit of execution and isolation in computer systems. Google's Chromium browser, for example, leverages the process abstraction to sandbox rendering engines instances corresponding to different tabs and plugins [55]. The world, however, has changed and isolation requirements have evolved. Hardware improved, memory and cache resources got significantly larger, and, today, a single program running in a process can correspond to millions of lines of code, include dozens of libraries from various (unverified) origins, and occupy gigabytes of virtual memory. Programs are further connected to the Internet, deployed in the Cloud, on distant servers that belong to a foreign entity, and generate and have access to large amounts of sensitive data. All of this raises new concerns regarding the security of our systems and the way in which they are built, highlighting the inadequacy of relying on a decades-old isolation abstraction that we abused and which strained away from its original intended usage.

In this thesis, we focus on the modern security challenges within a process's address space, show that this decades old-abstraction, by itself, is not sufficient and too coarse-grained to provide the necessary isolation guarantees, and introduce new programming primitives to address this issue.

1.1 An Outdated Trust Model

The trust model embodied by modern systems is a vestigial reminder of a time where software security and the notion of trust therein closely corresponded to the process abstraction. Programming used to be simple. A programmer would write an application, compile it, and execute the resulting binary on a machine under his control, running the operating system of his choice. From the application's point of view, all code executing inside the application was provided by the developer and was equally trusted with access to the program's resources. The underlying operating system was trusted with access to the program's resources and relied upon to guarantee the application's isolation from other processes running on the same machine.

Interactions with the outside, *e.g.,* user-supplied arguments or data, were considered to be the main vector of attacks leading to a potential exploit and security breach, *i.e.,* unauthorized access, usage, modification, or leakage of sensitive resources. Malicious individuals exploited these interactions to trigger logical or memory corruption bugs [180]. Corresponding efforts were deployed to detect [64], mitigate [61, 164, 190], or prevent [153] corruptions by various communities, including hardware, system, languages, and application developers.

This decades-old trust model, represented in Figure 1.1, mistakenly leads us to assume that attacks happen during the lifetime of an application, *i.e.,* at run time, following an interaction

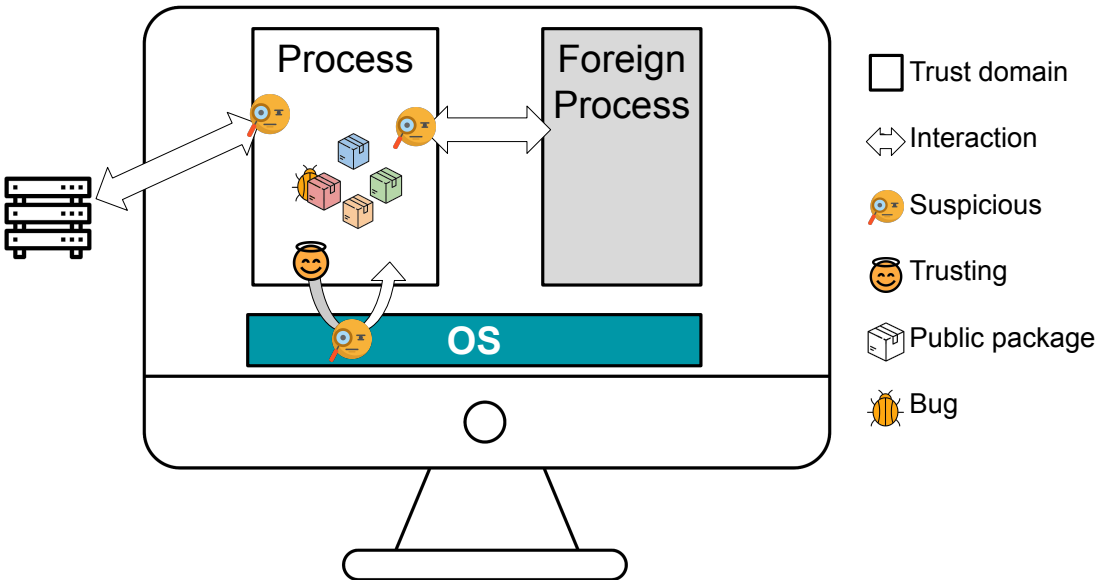

Figure 1.1 – The process centric trust model on current systems.

with an untrusted principal that corrupts/subverts the application. While often the case, this simplistic view of software security overlooks another serious threat to modern security and privacy, *i.e.*, attacks that come from within the application itself or the underlying system, potentially pre-dating its deployment and execution, and do not require any interaction with an untrusted entity.

As early as 1984, Ken Thompson warned programmers about the security implications of the increasing complexity of software [181] and more specifically the chain of trust necessary to bring an application to life. In his seminal Turing Award lecture, *Reflections on Trusting Trust*, Ken Thompson describes how a Trojan horse in a compiler's binary can automatically insert a backdoor in any compiled program, without being detected at the source code level. A similar attack happened recently, as a hacker compromised a server distributing PHP and inserted a backdoor in the programming language [91]. More related, some compiler bugs can lead to the unintentional insertion of vulnerabilities in otherwise correct application code [37, 53, 72]. One such example—that can be qualified as an instance of the correctness-security gap [95]— is the insecure compiler optimization that treated sensitive data scrubbing as dead stores and removes the corresponding instructions [37]. Anyone auditing the application's source code would assume sensitive data was properly zeroed-out and would probably not bother inspecting the generated machine-code, thus leaving sensitive information available in memory for anyone to read.

Thompson's lecture foresightfully warned that one cannot "trust code that you did not totally create yourself". As software development evolved to predominantly use public libraries, also known as *packages*; extensive runtimes; complex frameworks; and run on top of complex monolithic systems, code written directly by a developer only represents an overwhelmingly

small fraction of any application's code.

Challenge 1: *The current trust model for software is inherited from monolithic systems' process abstractions and assumes that threats come from interactions with external principals, thus failing to consider threats originating from the way in which applications are built and deployed.*

The next two sections focus on the description of threats delivered by an application's building blocks, *i.e.,* packages, and the systems they are deployed on.

1.2 Heterogeneous Trust in Applications

Programming has changed. Modern software development embraced reusable software components, *i.e.,* public libraries. Today's applications are built upon open-source libraries (aka packages) offering diverse functionality that increase programmer productivity and accelerate development cycles. Modern languages facilitate code sharing by providing tools to automatically publish, find, download, install, and use these packages [48–50, 111]. In the extreme, any modern application can be viewed as a collection of public packages orchestrated by application-specific code. These packages themselves are built in exactly the same way and require their own dependencies to be downloaded and installed, unbeknownst to the programmer. For example, the popular Python package `matplotlib` [27] requires seven public packages to be installed, among which `numpy` [28], a package with over one thousand contributors and a total of 300K lines of code, half of which are written in C.

Unfortunately, packages often come with few, if any, guarantees. They are not formally verified, their developers are typically unknown, their transitive dependencies hard to explore, and, most importantly, they execute in the proccess, and thus trust domain, as application-specific code. For the moment, package versionning seems to be the only real mechanism provided by package managers [47, 50, 111], but defaults to the most recently published version if not explicitely specified by developers. Thompson's observation thus becomes dramatically worrying, as only a small part of any application is written (or at least inspected) by the programmer himself, and any of the tens or hundreds of package dependencies [158, 159] can be used as a vector to insert a backdoor to access, modify, or steal sensitive information.

Malevolent individuals quickly capitalized on such a simple attack vector [77–82, 134, 145, 206]. Rather than focusing on the complex exploitation of a potential memory corruption bug or trying to subvert a language's compiler, which is usually published and distributed by a trusted entity, attackers can simply compromise the often less well guarded and audited popular public packages. An astonishingly common attack consists in publishing a modified clone of a valid package under a homoglyph or a slightly different name (often corresponding to common typing mistakes or expected names) [145]. For example, in 2019, a Python programmer discovered that PyPi [26], a Python package manager, was hosting a backdoored clone of `dateutil` under the name `python3-dateutil`. It was a malicious package that downloaded and ran an exploit binary hosted on a remote server. A more complex attack, similar to the PHP

one described above, consists in compromising a programmer's online account (*e.g.*, Github) to gain access to her repositories and introduce the desired backdoor. Recent research showed that gaining unallowed access to twenty high-level developers accounts could compromise half of the npm ecosystem [206].

Packages, despite being a vector for attacks, present interesting characteristics that could make a solution possible. First, they encapsulate a unit of trust, *i.e.*, a trust domain representing the package's publisher. Second, they are software components meant to be reused across various applications and are thus supposed to be context-unaware, *i.e.*, they do not need to know about the environment in which they execute and other parts of the application. Third, packages declare the code and data dependencies they require in order to properly implement their functionality. To properly function, packages require access to their own dependencies and arguments passed to them by the program, *i.e.*, a subset of the entire program's resources.

Modern applications' trust model is more complex than it used to be, as it includes diverse, potentially malicious, components for which the programmer has heterogeneous levels of trust and which should only be allowed access to subsets of a program's resources. Programmers currently do not have tools to control, at a fine granularity, which components can access, modify, or leak sensitive application information at run time. Programmers cannot trust their own applications and the problem sadly does not stop here.

Challenge 2: *Modern software requires intra-address-space isolation mechanisms to limit untrusted components' access to a program's sensitive resources.*

1.3 Heterogeneous Trust in Systems

The early 2000s saw the rise in popularity and rapid growth of Cloud services. Tech Giants, such as Amazon [1], Microsoft [24], and Google [19], made their infrastructures available for rent to other entities, ranging from companies as big as Netflix [40] to individual developers looking for flexible remote development environments [2] or access to specific hardware [7]. Cloud services expose different models (*e.g.*, IaaS, PaaS, SaaS, FaaS), with increasing levels of abstractions [8, 10, 25]. At its heart, the Cloud allows users to set aside the intricacies of IT infrastructure setups, reduce their deployment costs, scale their services at a push of a button, and more generally focus on their core business. Cloud service providers compete not only on pricing and performance, but also on the higher-level services they offer, their ease-of-use, and the extensibility of the ecosystem they propose (*e.g.*, elastic load balancing [9], fault tolerant deployments [18], lambdas [10] etc.).

Cloud services, while facilitating application deployment, scaling, and monitoring, come at a cost. Application developers and users have to implicitly extend their trust to the Cloud service providers, becoming susceptible to (1) vulnerabilities within the hypervisor [14], (2) exploits in libraries and Software as a Service (SaaS) infrastructures [12, 13, 15], (3) malicious employees with physical and administrative access to Cloud machines [85], and (4) intrusive surveillance

from governments [20, 36, 41]. According to a recent report by Ermetic [97], nearly 80% of the surveyed companies suffered a data breach in their Cloud-deployed systems within the last two years. Being able to guarantee the confidentiality and integrity of sensitive data comes out as a top priority among the vast majority of the participants and this can only be truly achieved if the potential threat of a compromised hypervisor is taken into account. Unfortunately, providing such strong guarantees in an adverserial environment is hard, requires establishing a root of trust and secured primites (*e.g.,* certificates for authentication and cryptographic keys for communication), and necessitates a level of expertise that most Cloud users do not have.

Challenge 3: *In modern software deployments, the operating system and, more generally, privileged code can no longer be trusted.*

1.4 Programming Languages Fall Behind

Programming has changed; programming languages did not. Modern software is going through a *trust crisis*. Programmers do not have any control over which parts of their applications or the systems hosting them can access, modify, or leak sensitive information. Despite the heterogeneous levels of trust that exist within applications and systems, programming languages still execute code from various origins inside the same address space, with uniform access to the program's resources. Some programming languages provide a simulacrum of isolation, more accurately described as information hiding, facilitating implementation modularity and often based on the type system. Examples include allowing object or package attributes and methods to be private and inheritance rules in certain PLs. This, however, is a weak form of isolation as most languages (1) allow unsafe behaviors to bypass the language's attribute modifiers, (2) provide a form of meta-programming to inspect code and data at run time despite type restrictions, (3) support interoperations with unsafe languages such as C/C++ that are unbothered by access restrictions, and (4) rely on the assumption that privileged code is trusted. Languages do not expose reliable mechanisms to restrict a package's access to a program's resources, especially in the presence of unsafe code, or abstractions to protect parts of an application from a potentially compromised host.

Nevertheless, programming languages have the potential to be the ideal layer to best express trust related policies. Trust is a relative notion highly dependent on the programmer's intent and is hence better addressed by the developer at the application level, *i.e.,* inside the application's code. In modern programming languages, public libraries are well-defined components that have clear boundaries in user code. Package dependencies are usually explicit, *i.e.,* written as `import` statements, and can be leveraged to construct the package dependence graph of an applications. Compilers could therefore be extended to automatically instrument application code to enforce user-defined restrictions at package boundaries, thus creating sub-compartments inside the application with differentiated access to the program's resources.

Challenge 4: *Programming languages only provide a weak form of intra-address space isolation unable to address the potential danger embodied by public libraries or compromised privileged code.*

Opportunity 1: *Programming languages can be extended with programming primitives to allow developers to address trust challenges within their applications. Implementing such mechanisms as a language extension has the following benefits: (1) it allows programmers to state their isolation requirements, (2) it is malleable and does not require modifications to the host, (3) it is backward compatible with legacy applications, and (4) it transparently abstracts the underlying mechanisms used to enforce isolation.*

While programming languages provide the expressibility and flexibility to describe intra-address-space isolation requirements, there is still a need to identify efficient mechanisms able to enforce such user-defined access rights, even in the presence of unsafe and unverifed code or software running at higher privilege levels. In the next section, we explore the modern opportunities provided by hardware security extensions.

1.5 Hardware Extension Opportunities

Hardware manufacturers slowly caught on the need to provide intra-address space isolation guarantees and expose a form of virtual memory management to user code. Recent hardware extensions such as Intel VT-x [185] (2005), Intel MPK [124] (2015), ARM Trustzone [30] (2013), Intel SGX [21] (2015), AMD SEV [39] (2016)/ AMD SEV-SNP [44] (2020), and the announced Intel TDX extension [125] (2020) present interesting foundations that could be used to address the trust crisis faced by modern software. In this section, we provide an overview of selected hardware extensions and highlight how they might help answer the challenges listed previously. We focus on Intel technologies as they are commonly available on commodity machines and are the ones explored in this thesis.

Intel VT-x: In 2005, Intel introduced hardware support for Extended Page Tables (EPT) with the Intel Virtualization Technology Extension [185] (VT-x). Intel VT-x extends the x86 ISA to facilitate user space hypervisor implementations and provide fast translation, in hardware, between guest and host addresses. The technology virtualizes privileged instructions and the registers for privileged code executing in non-root mode, thus reducing the guest OS's need for hypervisor intervention.

Intel VT-x, although marketed from a performance point-of-view (fast virtualization), can be considered as a security extension. In Cloud services, virtual machines are a common unit of deployment and ensure isolation between different users on multi-tenant platforms. As such, they must provide both strong isolation guarantees and low overheads.

Intel VT-x can be used to implement process-level virtualization, as in Dune [68]. This finer-grained form of memory management enables user-code to define in-application compart-

ments and implement a form of sandboxing with strong isolation guarantees enforced in hardware. In recent research work, Intel VT-x has been used in a multitude of areas to provide in-address space isolation. For example, Intel VT-x can protect and isolate hypervisors [175, 191], security monitors [174], kernel submodules [92, 155, 175, 179], or portions of applications [68, 144, 151] in specific frameworks.

On Linux, the Kernel-based Virtual Machine (KVM) module [45] exposes Intel VT-x and slightly simplifies the process of creating virtual machines via `ioctl` calls to its driver. Still, the technology requires careful configuration of the virtualized environment by expert engineers able to understand the various execution modes, setup the interrupt descriptor tables, and register memory mappings. In terms of performance, VM entries and exits, *i.e.,* transitions between the guest and the host, cost several microseconds and should be avoided on latency critical paths.

Intel MPK: Intel Memory Protection Keys [124] (MPK) is a recent hardware extension, only available on server-grade machines at the moment. With Intel MPK, page table entries are tagged with one of 16 available keys. A user-writable register, `PKRU`, encodes user-defined access rights for each key. This technology exposes a form of memory access right management to user code. System calls are provided to allocate and delete keys and tag memory regions. A switch between two MPK-defined compartments simply consists in a write to the PKRU register. Compared to Intel VT-x used for process virtualization, MPK exposes an easier-to-use API, as it does not require complex configuration of low-level hardware constructs (*e.g.,* execution mode, interrupt descriptor tables, etc.). Instead, Intel MPK only concerns itself with differentiated access rights to memory regions. Its interface is fairly simple and closely resembles `mprotect`. Intel MPK is, however, less flexible and provides weaker guarantees than Intel VT-x. First, the small number of available keys limits the direct adoption of Intel MPK as a mechanism to treat every single application's package as a separate trust domain. Some related work, such as *libmpk* [163], circumvent this limitation by virtualizing keys, at the cost of overheads induced by re-tagging page table entries. Second, the lack of protection of the PKRU register is both an advantage, as it allows switching between user-defined domains without requiring a mode switch, and a security concern, as other techniques (*e.g.,* static analysis of deployed code [118, 186]) must ensure that PKRU is only accessed by the intended code. Third, despite focusing on access right management, Intel MPK provides an incomplete solution as it only considers read and write access rights. A page marked as executable, for example, can only be made unreadable and thus does not prevent certain forms of return-oriented attacks invoking code outside of a compartment.

In recent related work, Intel MPK is used to create segregated memory allocation pools [186], isolate dataplanes [118], or implement intra-address space data encapsulation [131].

Intel SGX: Trusted Execution Environments (TEEs) are often presented as being the solution to the problem of trust for programs running on untrusted machines. They come in different flavors depending on hardware architectures (*e.g.,* Intel, ARM, AMD, and RISC-V processors).

These technologies have common features: (1) the introduction of a secure execution environment for trusted code, unaccessible from the untrusted host or applications, and (2) relying on the CPU as the root of trust of the system. Around that common base, architectures differ in the programming models and set of hardware features they expose. To name a few, attestation of loaded code and data in the trusted domain [21], encryption of trusted memory [21, 39], integrity protection [21, 44], register protection [3], and the ability to support different privilege levels [30, 39] or virtualization [3, 39, 44, 125] in secure mode of execution seem to be the main points of dissimilarity. In the next paragraph, we focus on Intel SGX.

Intel Software Guard Extension [21] (SGX) is Intel's implementation of a user-mode Trusted Execution Environment (TEE). Conceptually, Intel SGX provides intra-address space isolation of trusted user code and data executing inside an enclave by preventing accesses (reads, writes, and execution) from the untrusted part of an application or code running at a higher privilege level. While promising, Intel SGX lacks a clear programming model and requires highly skilled engineers who understand low-level technological intricacies and account for security pitfalls [58, 160]. Similar to Intel VT-x, Intel SGX requires non trivial configuration to register enclave memory pages and their content, define appropriate thread control structures (TCS), and handle transitions. Unlike Intel VT-x, the enclave's execution environment does not support the full x86 ISA and precludes certain instructions, such as `rdtsc` and `syscall` and thus requires trusted user code to be adapted to this restrictive environment. It further has non-trivial performance pitfalls, *e.g.,* enclaves entries and exits are several orders of magnitude more expensive than syscalls. The literature contains several interesting papers that explore the various higher-level abstractions, including operating systems [67, 83], containers [63], and compiler-driven partitioning of applications code [107, 138], that directly integrate with SGX and lower the bar to using these technologies. However, despite having been available for more than half a decade on commodity hardware, SGX is still seldomly adopted in practice. While research projects provide promising abstractions to protect trusted code execution, these solutions are not production-ready yet and still have non-negligible limitations, either in terms of performance or supported functionalities.

The next step of confidential computing seems to be heading towards confidential virtual machines, in the vein of AMD SEV [39] and Intel TDX [125]. Confidential virtual machines are an attempt, by hardware manufacturers, to provide a coarser-grained solution to the problem of trust in the Cloud than the one exposed by Intel SGX. These technologies allow Cloud users to automatically deploy their entire stack, including their applications and selected operating system, to a confidential execution environment inaccessible to a hypervisor. While potentially simpler to use than Intel SGX (which requires adapting application's code or deploying complex execution frameworks), confidential VMs put an entire operating system inside the Trusted Computing Base (TCB), thereby exposing a greater attack surface. There is also a paradox in the overall approach: the basic assumption is that users do not trust the Cloud service provider and try to protect against the hypervisor, but they trust the guest operating system. Where does the guest OS image come from? Oftentimes, Cloud services provide their own OS images for new VMs, instrumented or extended to bridge the semantic

gap [59, 60] between the guest OS and the hypervisor or provide flexible VM management.

All these hardware extensions introduce new opportunities to modify the decades old trust model embedded in the process abstraction. They provide mechanisms flexible enough to implement various programming models with intra-address space partitioning at different granularities. The question is now: what is the best way to employ these extensions and address modern software trust crisis? One obviously desirable feature for such a solution would consider the heterogeneity of hardware extensions and be compatible with different hardware setups (*e.g.*, Intel, AMD, and ARM). Another requirement is to hide the intricacies of the hardware technology and provide high-level solutions easily adoptable by non-expert developers.

Challenge 5: *Hardware manufacturers provide various extensions that allow to modify the isolation model inside a process address-space. These technologies are however hard to use, lack a clear programming model, and each come with their own vendor-specific subtleties.*

Opportunity 2: *Hardware extensions provide mechanisms enforcing intra-address space isolation. More specifically, these mechanisms allow to (1) protect users sensitive information from malicious privileged code, and (2) restrict public packages' access to a program's resources, even when they are written in unsafe languages.*

1.6 Thesis Statement

Above we identified five challenges and two opportunities in the trust crisis that modern software is facing. The high-level goal of this thesis is to explore the opportunities presented by hardware security extensions to make trust a manipulable programming primitive exposed to developers.

We first propose a programming abstraction to easily isolate trusted parts of the application. Second, we consider the problem of safely leveraging unverified public libraries by allowing developers to restrict access to the program's resources when invoking their functionalities. For both of these, we take a language-level approach, provide high level programming abstractions that transparently leverage hardware extensions to guarantee the desired intra-address space isolation. Finally, we take a step back and consider a software only solution that provides a flexible environment in which researchers can design and experiment with isolation abstractions.

Thesis Statement

Programming has changed, leading software into a seemingly inextricable trust crisis. As the way in which applications are built and deployed drastically changed to include entities with heterogeneous levels of trust, programming languages need to evolve to provide the proper

primitives allowing developers to correctly control which parts of their applications and systems can access, modify, and leak their private information. The combination of programming languages compiler techniques and recent hardware security extensions provides an unprecedented opportunity to construct easy-to-use and powerful higher-level abstractions able to address the problem of trust faced by modern software.

1.7 Thesis Contributions

In this thesis, I provide simple, easy-to-use programming abstractions that enable programmers to take into account the heterogeneous levels of trust that exist within their applications and the systems on top of which they are deployed. Trust is a relative notion better handled by the programmer herself and, thus, our proposed solutions should empower users and let them make isolation decisions, best expressed at the source code level. I present two programming abstractions and a software execution framework to help address these challenges.

Specifically, this thesis contributes the following two implemented, evaluated, and published systems:

- **Secured Routines: Language-based Construction of Trusted Execution Environments**

  The *secured routine* abstraction allows programmers to protect code and data from untrusted, potentially higher-privileged, code. Secured routines are a language-level approach to supporting Trusted Execution Environments. The abstraction closely integrates with the Go programming language, enables to easily leverage TEEs in legacy applications with minimal code refactoring, abstracts away the intricacies of the underlying hardware technology, and is designed to circumvent the hurdles and limitations of the hardware mechanism while providing good run time performance. The secured routine programming model replaces expensive enclave crossings with typed message passing and enhances memory isolation between trusted and untrusted code by preventing cross-domain memory references.

- **Enclosure: Language-Based Restriction of Untrusted Libraries**

  The *enclosure* abstraction splits a program into isolated trust domains and allows safe execution of unverified public libraries with limited access to the program's resources. Enclosures define a restricted execution environment which, by default, only grants access to the enclosed's code package dependencies derived from the package dependence graph and precludes system calls. Programmers can further extend or restrict the enclosure's accessible memory, at the granularity of packages, and selectively enable system calls. Enclosures can be ported to any programming language by leveraging LITTERBOX, our enclosure backend library that exposes a high-level language-agnostic API and implements isolation enforcement by leveraging hardware extensions.

This thesis further presents a discussion of the compatibility between these programming

abstractions, reflections on the process trust model and how hardware security extensions modify it, and describes an ongoing research effort:

- **Tyche: Software is the new Hardware** *Tyche* is a secured execution framework that implements and allows to compose isolation primitives for modern software. Hardware security extensions are hard to use, not amenable to modifications, constrained by the slow iteration pace of hardware development, and do not compose well. Tyche provides an environment to emulate hardware extensions in software, modify and compose them, and experiment with new primitives. At its heart, Tyche relies on security-oriented virtual ISAs to enforce trust domains isolation and control their accesses to a program's resources. Interactions between trust domains are only allowed via *typed gates* and controlled by a Trusted Intermediary.

This thesis highlights the need for new software and hardware mechanisms to provide fine-grained (within an address space) isolation so that programs can be safely constructed from untrusted pieces of code and run in untrusted environments.

1.8 Thesis Organization

The rest of this thesis is organised as follows:

Chapter 2 presents the *secured routine* [107] abstraction that executes user-level threads in a trusted execution environment unaccessible by untrusted software. This Chapter contains the paper published at ATC19, extended with an afterthought section.

Chapter 3 introduces *enclosures* [106], a programming abstraction that defines restricted execution scopes with limited access to a program's resources, expressed at package-granularity. This Chapter contains the paper published at ASPLOS21, extended with an afterthought section.

Chapter 4 reflects upon the composability of secured routines and enclosures and describes the challenges faced when trying to combine different hardware security extensions. This Chapter is an unpublished, original reflection on the work presented in the two papers published previously.

Chapter 5 describes, at a conceptual level, how hardware security extensions used in this thesis modify the trust model embodied by the process and suggests alternative approaches, notably in software.

Chapter 6 introduces Tyche, our current (unpublished) research effort that aims at providing an extensible safe execution framework in software to explore isolation schemes, means of establishing cooperation between mutually distrustful entities, and quickly develop new language and compiler approaches to address the problem of trust for modern software.

Chapter 7 contains future work and considers new programming languages and hardware opportunities, as well as a software refactoring of existing hypervisors to address the problem of trust in the Cloud.

Chapter 8 concludes this thesis.

1.8.1 Bibliographic Notes

This thesis was conducted under the supervision of my advisors Prof. Edouard Bugnion and Prof. James R. Larus, and parts of it were published in collaboration with Prof. Mathias Payer and my then colleague, Dr. Marios Kogias. The following publications are included in this thesis:

- *Secured Routines: Language-based construction of trusted execution environments*
  Adrien Ghosn, James R. Larus, Edouard Bugnion.
  In 2019 USENIX Annual Technical Conference (ATC19).
- *Enclosure: Language-based restriction of untrusted libraries*
  Adrien Ghosn, Marios Kogias, Mathias Payer, James R. Larus, Edouard Bugnion.
  In Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS21).

The following published work is outside of the scope of this thesis and will not be presented:

- *R2P2: Making RPCs first-class datacenter citizens*
  Marios Kogias, George Prekas, Adrien Ghosn, Jonas Fietz, Edouard Bugnion.
  In 2019 USENIX Annual Technical Conference (ATC19).

2 Secured Routines: Language-based Construction of Trusted Execution Environments

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.

2.1 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 [141] and Spectre [129] attacks, (2) vulnerabilities within the hypervisor [14], (3) exploits in libraries and Software as a Service (SaaS) infrastructures [12,13,15,16], (4) malicious employees with physical and administrative access to both computer and storage resources,

and (5) intrusive or extra-territorial government surveillance [36, 41, 70].

To address these concerns, processor vendors, following ARM's lead [30], 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 [63, 67, 137, 203]. In particular, Intel SGX [22] partitions hardware and software into two trust domains: a CPU, trusted code deployed by a particular user, 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, protected against accesses by privileged code, untrusted user software, and other enclaves deployed on the machine. 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. Before 2019, the private messaging application Signal [148] was one of the few applications to adopt enclaves and Microsoft Azure also was the first to offer a cloud solution to expose SGX features [146, 167]. 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 [63, 67] and (2) separate the portions of a program that require trusted execution [23, 29, 138]. Solutions in the first category provide an abstraction, such as an operating system [67] or a container [63], to execute unmodified applications in an enclave. The other alternative requires a developer to identify and partition [23, 29, 172], or provide annotations that a program analysis tool can use to partition [138], 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` [57]. 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 (§2.2), the secured routine abstraction (§2.3), the implementation of GOTEE (§2.4), and evaluate it using both microbenchmarks and three security-sensitive applications (§2.5). Finally, we discuss possible architectural improvements in §2.6, related work in §2.7, and conclude in §2.8.

2.2 Background

2.2.1 Intel Software Guard Extension

Intel Software Guard Extension (SGX) [22], 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. This section describes Intel SGXv1, *i.e.,* the implementation available on common Intel CPUs a the time this research was conducted (2018).

Figure 2.1 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

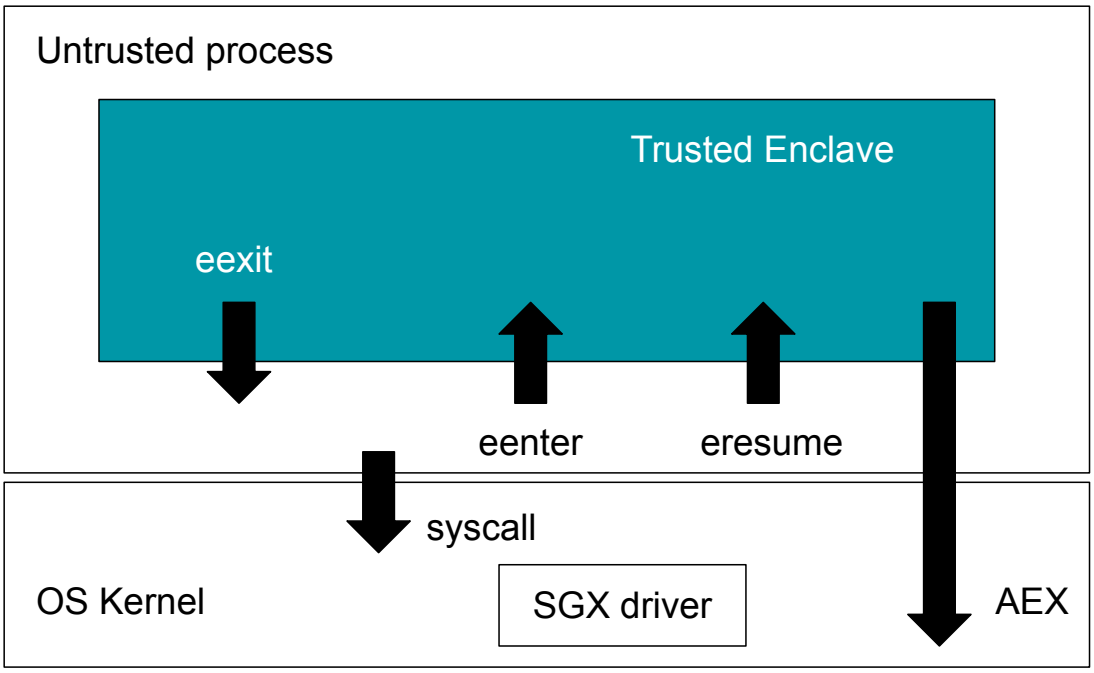

Figure 2.1 – Trusted Execution Environments with Intel SGX.

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, in hardware and via cryptographic signatures, ensuring the validity of the initial state of the enclave and allowing attestation with a remote entity.

SGX reserves, at boot time, a contiguous portion of physical memory, called the *Processor Reserved Memory* (*PRM*). The available SGX-enabled CPUs at the time this research was conducted (2018) have a maximal EPC 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 [38]. 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 [67, 205]. The CPU's Memory Encryption Engine (MEE) 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× [205].

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 2.1). 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.

2.2.2 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 a Windows library OS, Haven [67]; a Linux OS, Graphene [83]; and a container platform, SCONE [63]—running in SGX. While convenient for developers and effective at reducing expensive enclave crossings [205], 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 [173]).

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*) [23] 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 [63] and Eleos [161] both rely on message passing to implement asynchronous system calls and avoid expensive enclave exits. Following the same approach, Intel recently published `switchless` [43], a (under-development) mini-framework that provides a simple C++ messaging mechanism on top of the SDK.

Asylo [112] is a C++ framework, compatible with gRPC [113], 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 [138] 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.

2.2.3 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 [76]. These domain crossings are an order of magnitude more expensive than a system call [205], between ~2µs [205] and ~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 §2.5.2). 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.

2.3 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.

2.3.1 Threat Model

We follow the threat model of other work in SGX [63, 67, 83, 138] 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 [84] for GOTEE's runtime and system call interposition mechanism in Section 2.4.3.

Denial-of-service attacks, a known limitation of SGX [89], 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.

2.3.2 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 [120]. 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.

2.3.3 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.

Listing 2.1 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

```
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 2.1 – Using secured routines to isolate a secret key within the TEE.

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 [23, 29, 138], 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 2.1.

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 [138], where the trusted and untrusted 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 2.1, 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 supplied as an argument, *i.e.,* if a memory access outside of the argument frame is required. The mechanism automatically tracks copied objects to detect cyclic references, can enforce limitation on the per-copy amount of bytes consumed, and implements defensive checks (*e.g.,* accounts for TOCTOU attacks and ensures that memory references are within the bounds of the sending domain).

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. Worth noting, this specificity of Gotee makes it compatible with other TEE models [30, 135] that do not provide implicit memory access from an enclave to an untrusted parent address space.

2.3.4 Runtime Cooperation

The secured routine abstraction requires two separate trust 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 2.1, the trusted runtime spawns a new routine that invokes the `InitSymKey(done)` closure.

Figure 2.2 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.

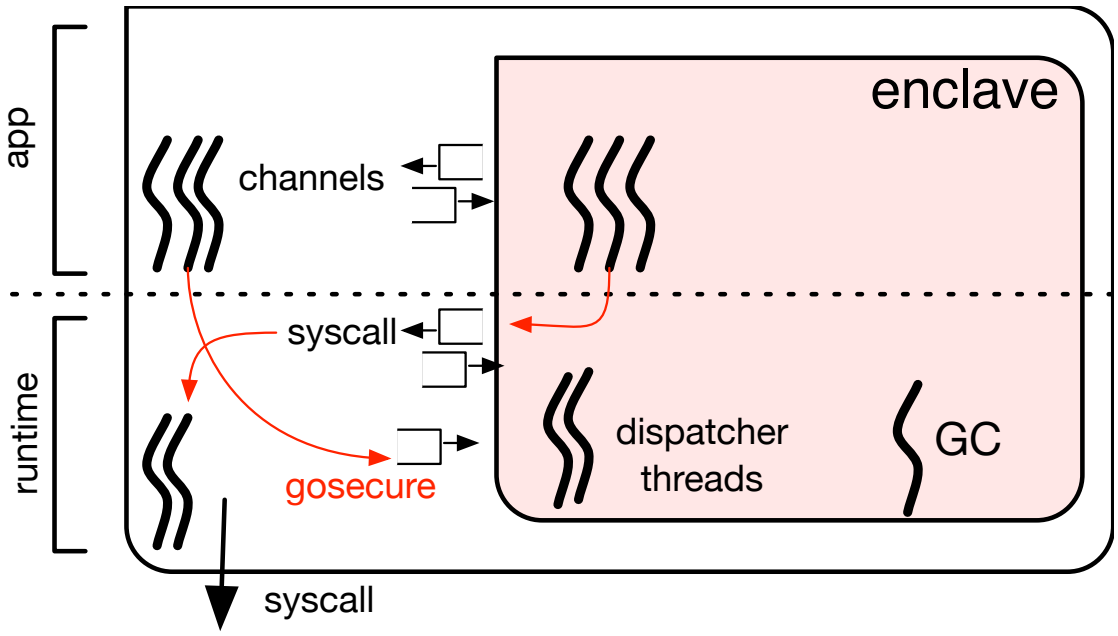

Figure 2.2 – Channel-based cooperation between runtimes.

2.3.5 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 the two trust 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.

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 [63, 83, 138], 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.

2.4 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.

2.4.1 Compiler Support for `gosecure`

The GOTEE compiler is responsible for partitioning code and data according to the design of §2.3.3. 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 2.3 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-

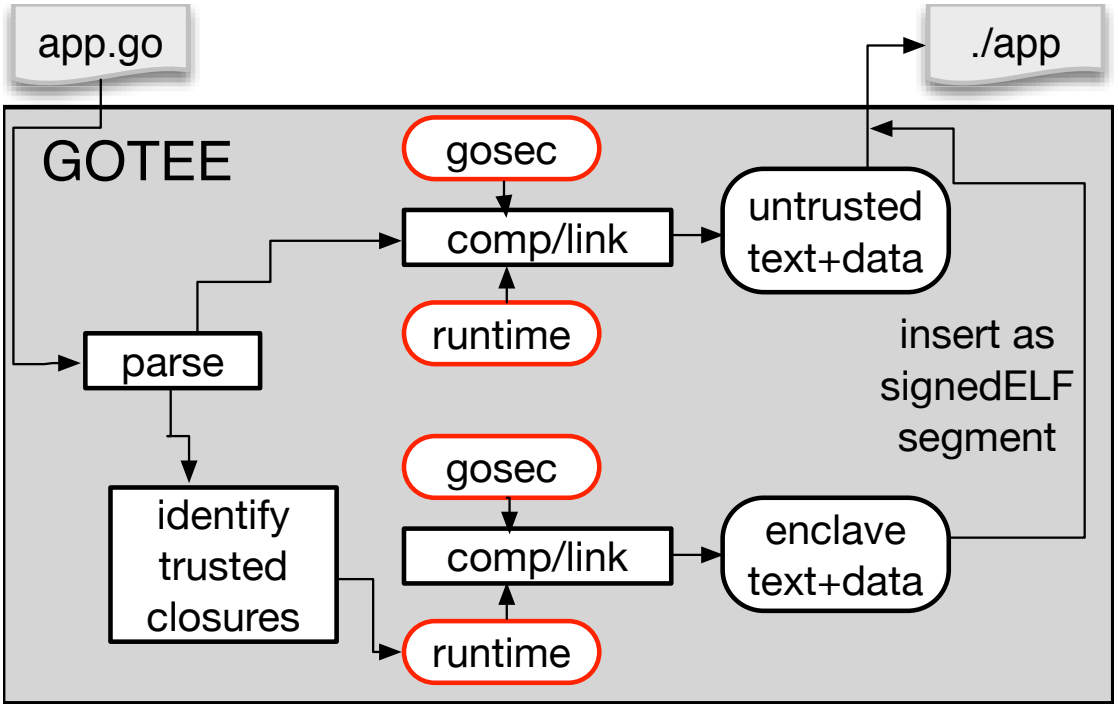

Figure 2.3 – The GOTEE compilation pipeline.

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. GOTEE also supplies functions to generate a measurement and signature of the trusted code at run time for debug purposes.

2.4.2 `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 [89] that delivers the token required to perform the initialization (`einit`, see §2.2.1). 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 at compile time. A similar environment variable is available to limit the number of threads concurrently executing inside an enclave at run time. By default, the loader adds only two TCS to the enclave: one to execute the user code, the other to support garbage collection.

Loading and initiliazing an enclave with the early SGX hardware is a series of distinct steps that involve the SGX driver (to execute privileged instructions), the SGX daemon (to retrieve a `EINITTOKEN` cryptographic token), the measurement byte array generated by the `gosec` library while creating the enclave [89] (§2.4.1), 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: Some of the asynchronous exits from the enclave, *e.g.,* faults and exceptions, are handled in two steps. First, they 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. The same mechanism allows in-enclave fault handlers to run by letting the AEX handler perform an `eresume` if the exit was not triggered by a panic or a GOTEE exception.

2.4.3 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 pre-allocates 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. Changing the TLS base, however, requires interactions with the untrusted operating system. GOTEE circumvents this problematic case 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 and avoids leaking information via the thread TLS value.

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

	Workload	Text	Data	RO-data	Total	Main Package Dependencies	App 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	++ `crypto, gnet, encoding, golang.org/x/crypto/*`	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 2.1 – 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.

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.

2.5 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 Linux kernel driver 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 (§2.5.1); (2) the performance, latency, and throughput of secured routines and their cost in comparison to the crossing-oriented approach of the Intel SDK (§2.5.2); (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(§2.5.3), by performing a fine-grained partitioning of a standard Go package(§2.5.4), and by extending a real-world application with

a TEE-specific implementation(§2.5.5).

2.5.1 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 2.1 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 2.1 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.

2.5.2 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

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 2.2 – Latency microbenchmarks in µs.

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.
- `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 2.2 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.

Note, however, that in both cases GOTEE requires a second thread running outside of the enclave for the untrusted part of the code and to service system calls.

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 2.4 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

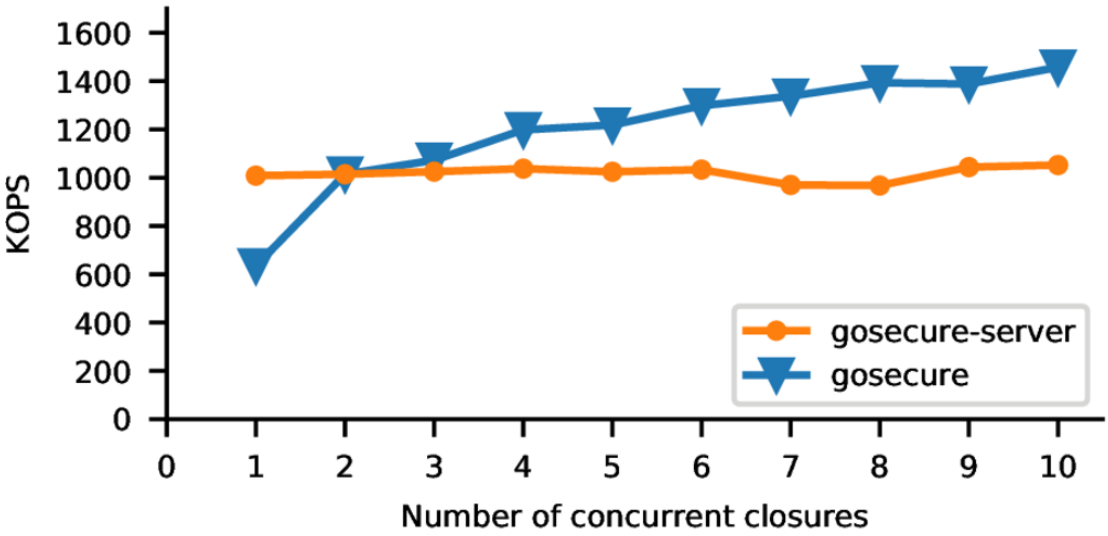

Figure 2.4 – Synchronous closure execution rate for secured routine multiplexed on top of a single enclave thread.

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 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.

2.5.3 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 2.1. 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 struc-

tures 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.

2.5.4 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. We did not implement, in the benchmark, a mechanism to safely deliver the certificate to an attested enclave (which would be required in a real-world deployment) and focused, instead, on measuring the overhead of having secured encryption operations solely in the enclave.

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 [138] reported only 60% of native throughput due to the cost of frequent enclave crossings.

2.5.5 Keystore based on go-ethereum

The go-ethereum [98] project is the official implementation of the Ethereum protocol [42] 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 2.1. 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.

2.6 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 [38, 89]. SGX has already been shown to be vulnerable to side-channel attacks based on caches [73], page faults [196], branch shadowing [136], and processor side-channel attacks ("Foreshadow" [76], a variant of Spectre [129] and Meltdown [141]).

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.

2.7 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 [123], 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 [202] system used security types and information-flow analysis to partition programs so that secure computations could be distributed and executed on trusted processors. Swift [87] partitioned a web app to run its trusted computation on a server. Wedge [71] 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 [74] 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 [108] 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 [138] 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 [133]. While numerous systems have been built with the domain-crossing approach embodied in the Intel SDK (§2.2.2) [29, 67, 138], including a

solution for the Rust programming language [29], the current implementation of SGX favors an asynchronous, communication-oriented model, as demonstrated by GOTEE and Intel's own recent `switchless` [43]. Other mentioned solutions [43, 63, 83, 108, 112, 161] 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 [177]. 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 [172] 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 [90, 94, 121] or the cloud hypervisor [204]. GOTEE could provide a complementary application-level isolation.

2.8 Chapter 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.

2.9 Afterthoughts

Secured routines was published at the USENIX Annual Technical Conference 2019 (ATC19) [107]. In this section, we take a step back and summarize the lessons learned, 2 years later, and place our work in perspective by considering the evolution of the technologies, their usage in practice, and the accumulated feedback we received after the project's publication.

The secured routine programming model does not make attestation [21,112,114] a central part of its design and only briefly discusses it in the paper. As described in the paper, attestation can be added to the `gosec` library to be automatically performed at enclave load time. Our library could, for example, accept a configuration file or annotation that defines the local or remote entity in charge of validating the enclave's measurement. One might wonder why we did not enforce a particular attestation scheme within GOTEE, *e.g.,* by providing attested cross-domain channels, or similar to Asylo [112]'s secured channels on top of gRPC [113], and why it was not a central part of our design. The answer is simple, we considered that attestation was an end-to-end problem, with no clear communication scheme, and was too tightly linked to the application's own logic. GOTEE is a compiler that supports *any* pure Go code. As such, it can be used for both local programs with no network access and for network-facing applications. We thus did not want to impose, within our programming model, any form of communication with an external entity that would have required access to IPCs or the network interface. However, as demonstrated both by the SSH-server and the trusted Go-ethereum wallet experiments, our programming model supports communication with remote entities and does not impede on one's ability to establish secured authenticated communication channels with the enclave.

GOTEE exposes a binary trust model as it assumes only one enclave per application. Theoretically, GOTEE could be extended to support multiple enclaves within the same program address space by, for example, allowing the `gosecure` keyword to take an enclave identifier as a parameter. However, we made a decision, early on, to focus on providing only support for a single trusted domain per application for 3 reasons. First, we wanted to keep the model simple in order to make it usable by the average programmer. A manichean approach of the trust model thus seemed like the best option and was easier to reason about. Second, we knew that this alternative approach would not scale gracefully to tens or hundreds of enclaves. Each enclave requires its own copy of the modified Go runtime, *i.e.,* ~800KB for a limited usable EPC of 128MB, and must be statically linked to a different range of address within the program's virtual memory. Third, we believe that splitting enclaves accross processes rather than craming them within a single address space makes more sense and is, of course, possible

with GOTEE.

GOTEE, by design, circumvents Intel SGX limitations and performance pittfalls. A valid critic could thus be made about GOTEE's timeliness, considering futur iterations of SGX hardware. For example, Intel SGX version 2 extends the enclave's execution environment to allow dynamic updates to the enclave's memory regions (shrinking and extending), and increases EPC size to potentially 2TB. We can also expect the cost of transitions to decrease. As a result, GOTEE's enforcement of static linking, aggressive dead code elimination (DCE) to reduce memory consumption in the enclave, and message passing communication could be considered obsolete as they would no longer be essential to enclave's execution and performance. We believe, however, that our design is a principled approach to providing a secured and strong separation of trust domains and would still be relevant in SGX v2. By relying on cross-domain channels as the sole mean of communication between trusted and untrusted code, we provide a single point of interaction that can be carefully implemented and instrumented to avoid accidental leakage of information and adopt defensive programming checks. We would also like to point out that hardware development cycles are extremely slow. Intel SGX v2 was announced before GOTEE's publication and is only available on certain CPUs from 2019. There is thus an undeniable need to address the current's version limitations and support it for the years to come. We acknowledge, however, that GOTEE would benefit from an improved SGX implementation, notably by being able to dynamically increase the enclave's trusted heap at run time.

A trade-off exists between the freedom and control exposed to the programmer and the strength of the security guarantees provided by the system. In GOTEE, we made the decision to implement deep-copy to allow sending complex structures across the trust domain boundary and rely on compiler emitted warnings to guide the developer's implementation and highlight potential pointer de-references. It is therefore the programmer's responsibility to read warnings, think about which structures will be copied into the untrusted space, and ensure that no sensitive information is part of the copied data. In retrospect, we think that a practical solution should probably be more restrictive and only perform one level of copy, *i.e.*, disallow types that require pointer de-referencing in the copying process. Put differently, simpler types might be better to avoid unintended leakage of sensitive data.

Despite being extremely easy to leverage, GOTEE still only appeals to expert engineers. The vast majority of Cloud users, who are the main target of TEE technologies, do not have the maturity and expertise (yet) required to adopt a multi-trust domain execution environment. TEE technologies will take a while to make it into the mainstream world of programming. In the meantime, a successful TEE-based product should be as simple as a push of a button or selecting an option in the Cloud service provider's UI when creating a virtual machine. Users have a coarse understanding of trust, confidentiality, and integrity and mostly treat the entire VM as one single domain [97]. Intel, after a few years spent on SGX, has realized this and is starting to propose confidential virtual machines, similar to other vendors [30, 44, 125]. At the same time, some companies and startups are trying to provide frameworks to transparently

lift Cloud applications to TEEs on the current hardware. Anjuna security [4] takes unmodified applications and deploys them in the client's selected Cloud platform. While GOTEE is not relevant to their clients, Anjuna engineers expressed interest in the project as a useful tool to quickly prototype and test their products or deploy microservices for their infrastructures.

GOTEE is a double-edge sword leading to a paradoxical situation. Simplifying code deployment inside the enclave also implies that developers pay less attention to what code ends up executing in the TEE. With secured routines, leveraging any public library inside the enclave becomes as easy as using it outside of a TEE, as it does not require any source code modification or instrumentation. As a result, developers might end up including buggy or malicious code inside the trusted domain. This problem is not specific to TEEs and plagues all modern software that heavily relies on reusable public components called packages. The next Chapter focuses on this particular issue and is not tied to TEEs technologies.

3 Enclosure: Language-Based Restriction of Untrusted Libraries

Programming languages and systems have failed to address the security implications of the increasingly frequent use of public libraries to construct modern software. Most languages provide tools and online repositories to publish, import, and use libraries; however, this double-edged sword can incorporate a large quantity of unknown, unchecked, and unverified code into an application. The risk is real, as demonstrated by malevolent actors who have repeatedly inserted malware into popular open-source libraries.

This paper proposes a solution: *enclosures*, a new programming language construct for library isolation that provides a developer with fine-grain control over the resources that a library can access, even for libraries with complex inter-library dependencies. The programming abstraction is language-independent and could be added to most languages. These languages would then be able to take advantage of hardware isolation mechanisms that are effective across language boundaries.

The *enclosure* policies are enforced at run time by LITTERBOX, a language-independent framework that uses hardware mechanisms to provide uniform and robust isolation guarantees, even for libraries written in unsafe languages. LITTERBOX currently supports both Intel VT-x (with general-purpose extended page tables) and the emerging Intel Memory Protection Keys (MPK).

We describe an *enclosure* implementation for the Go and Python languages. Our evaluation demonstrates that the Go implementation can protect sensitive data in real-world applications constructed using complex untrusted libraries with deep dependencies. It requires minimal code refactoring and incurs acceptable performance overhead. The Python implementation demonstrates LITTERBOX's ability to support dynamic languages.

3.1 Introduction

Programming has changed; programming languages have not. Modern software development has embraced abstraction and reusable software components. Today, applications are built

using open-source libraries (aka packages) that offer diverse, tested functionality that increases programmer productivity. In the extreme, an application can become a collection of libraries orchestrated by application-specific code. To facilitate code sharing, modern languages provide tools and online repositories to publish, find, download, access, and update public libraries, *e.g.,* Python modules [48], Go packages [111], Ruby gems [49], and Rust crates [50].[1]

Although languages embrace libraries, few, if any, provide mechanisms to address the insecurity and fragility inherent in their use: (1) packages come without a formal specification of what they do, and do not do; (2) their developer is typically unknown, thus untrusted; (3) they can import other unknown and untrusted packages, and lack traceable dependence management; and (4) most important, programs run in a single trust domain that does not isolate code or data from different packages.

In general, a developer's trust in a public package often appears to be based on its popularity or a shallow code review. Careful inspection is both impractical, since importing a single package may incorporate hundreds or thousands of transitively dependent packages [158, 159], or infeasible, as a package's code may change frequently. As a result, an application can become a patchwork of code from untrusted and unverified sources.

Malevolent individuals have been quick to exploit the opportunity to insert malicious code in popular packages [77, 78, 82, 206], to modify an IDE to insert the code [183], or to substitute modified clones [79–81, 134, 145]. These attacks are easy to implement and provide unimpeded access into hundreds, if not thousands, of applications for malicious code that steals private information or opens backdoors. For example, malicious Python packages stole SSH and GPG keys from the local file system [78, 81]. More generally, even legitimate third-party libraries may implement undocumented functionality that operates outside of its advertised scope. For example, the Facebook iOS SDK to identify users also shared device information with Facebook without user consent [200].

Although the systems, programming languages, and security communities have long studied software isolation [68, 71, 92, 118, 122, 138, 142–144, 151, 155, 174, 175, 179, 186, 190, 191], previous approaches do not address the full range of requirements for package isolation and security: (1) low-level systems abstractions may not match programming language requirements [68, 92, 142, 151, 174, 175, 191] or may require extensive application refactoring [71, 143, 144]; (2) pure language approaches, *e.g.,* Rust or JavaScript isolates, are limited to a single language and programs written only in the language; (3) mixed approaches, *e.g.,* Erim [186], Hodor [118], and Glamdring [138], ignore package structure, are unaware of dependencies among packages, or lack expressive access rights.

While packages are at the root of security and fragility problems, their unique characteristics also facilitate solutions. Specifically, packages consist of code and data written to run as part of any program, which means they must have clearly defined entry points, not be dependent

[1] Frameworks, such as node.js, also support library repositories. This paper considers them from a language-specific perspective, e.g., as JavaScript packages.

on a program's environment, and explicitly declare and import their dependencies. They can run in isolation given access to their input data and the packages they depend upon. The clear boundaries of a package can help partition a program's memory address space into isolated regions that prevent code in the package from improperly accessing the rest of the program's environment.

We propose a new programming language construct that gives a developer fine-grain control over the package resources that a computation can access, even for packages with complex dependencies. It introduces a dynamically-scoped set of restrictions on which parts of the address space can be accessed and which system calls can be invoked. The abstraction is language-independent and could be added to most languages. Its implementation relies on hardware isolation mechanisms that provide trustworthy, fine-grain access control within a virtual address space [21, 62, 124, 185, 195].

This construct is called an *enclosure*. It implements an isolation policy for a closure by binding it to a memory view and a set of permitted system calls, which restricts access to program resources by the code invoked in the closure, regardless of which package contains it. The current system starts with a view that limits access only to the resources in the packages that the closure invokes. A developer can restrict or extend this view by selectively enabling read, write, or execute access rights for a specific package. The developer can also selectively allow system calls. The policy is dynamically scoped and applies to all code executed by the *enclosure*, which in turn can invoke other *enclosures* that further restrict the accessible resources.

LITTERBOX enforces *enclosure* policies at run time. It is a language-independent framework that uses hardware mechanisms to provide uniform and robust isolation guarantees, even for packages written in unsafe languages. LITTERBOX exposes a high-level API that is reusable across programming languages. The isolation is built on one of several different hardware technologies, and LITTERBOX hides the hardware complexity.

This paper makes the following contributions:

- *Enclosure* is a dynamically scoped programming language construct that imposes user-defined access policies on code invoked within it. These policies restrict, at package granularity, what parts of a program (data and code) this code can access and which system calls it can invoke. Untrusted packages, even those with deep dependence graphs, can be isolated from sensitive information.
- LITTERBOX is a language-independent framework to enforce *enclosure* policies with robust hardware isolation mechanisms. LITTERBOX currently supports both Intel VT-x (with its general-purpose extended page tables) and the emerging Intel Memory Protection Keys (MPK).
- We describe an efficient implementation of *enclosures* and LITTERBOX for the Go language. It has low overheads for real-world applications and can drastically reduce a program's trusted codebase.
- We describe a prototype implementation of *enclosures* and LITTERBOX for the dynamic programming language Python.
- We present experiments demonstrating that the overhead can be as low as 1.02x for real applications.

LITTERBOX and both language frontends are open-sourced [56].

3.2 Enclosure construct

An *enclosure* is a programming language construct that enables a developer to restrict code's access to program resources to prevent untrusted code from accessing, modifying, or leaking sensitive data. It limits the code to access only functions and data from specified program packages (the memory view) and to execute only explicitly allowed system calls. These restrictions are dynamically scoped, so they apply to the closure's body and the code invoked by it.

3.2.1 Definitions

A *package* can export four items for use by other packages: (1) functions (code), (2) variables (mutable data), (3) constants (immutable data), and (4) arena (heap). Variables are either (1) static variables (*e.g.,* pre-allocated globals) or (2) dynamic variables (dynamically allocated objects). A package's functions allocate dynamic variables within the package's arena.

A *program* is a collection of packages, organized as a directed package-dependence graph. This graph is statically determinable from the packages' import statements. A package `Foo`

```
1  package main
2
3  import(
4    "img”
5    "libFx”
6    "os"
7    "secrets"
8  )
9
10 var key := "myPrivateKey"
11
12 func main() {
13   original := secrets.LoadImage("myimg.png")
14
15   rcl := with["secrets:R","none"]func(o *img.Png) *img.Png {
16     return libFx.Invert(o)
17   }
18
19   inverted := rcl(original)
20   res := img.Watermark(inverted, key)
21   os.Write("signed.png", res.Bytes())
21 }
```

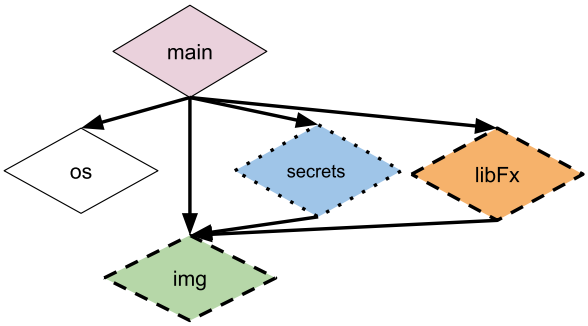

Figure 3.1 – The `rcl` *enclosure* prevents the call to the public package `libFX` from modifying or leaking sensitive information. The top-right corner shows the application's package-dependence graph, with `rcl`'s natural dependencies in dashed borders, and its extended read-only view to `secrets` in dotted borders. Color-coding of variables highlights which package arena holds the corresponding value.

has a *direct* dependence on package `Bar` if `Foo` imports `Bar`. Package `Foo` has a *transitive* dependence on `Bar` if there is a directed path from `Foo` to `Bar` of length greater than 1 in the graph. A package's *natural* dependencies is the set of packages contained in its direct and transitive dependencies. A package `Bar` is *foreign* to `Foo` if it is not part of `Foo`'s natural dependencies.

A *closure* is a function combined with an environment that holds the bindings for its free variables. A closure belongs to the package that defines it, and it shares some of the package's natural dependencies.

An *enclosure* binds a dynamically scoped *memory view* and set of allowed system calls to a closure. The memory view defines access rights to the program's packages by the code invoked in the closure. By default, *enclosures* prohibit all system calls and limit memory views to only the resources in packages in the closure's *natural* dependencies. User-defined policies can selectively authorize system calls and restrict or extend the memory view.

3.2.2 Enclosure Expression

Enclosures are declared with the following syntax:

Stmt	::=	**with** [Policies] ClosureDef
ClosureDef	::=	**func** (args) resultType { body }
Policies	::=	MemModifiers, SysFilter
MemModifiers	::=	(pkg : U | R | RW | RWX)*
SysFilter	::=	none | all | (net | io | file | mem | ...)*

The *enclosure* expression returns a closure that is permanently associated with a memory view and system call filter. The closure can be bound to a variable and reused throughout the program's lifetime. The memory view and system call filter will be enforced during every execution of the closure.

`MemModifiers` and `SysFilter` specify the *enclosure*'s memory view and authorized system calls, respectively. `MemModifiers` extend or restrict the closure's memory view by specifying access rights, similar to those in the Unix file system, to a package: `R` grants read-only access to a package's data and constants, `RW` grants read access to its constants and read-write access to variables, `RWX` gives full access to its resources: *i.e.,* read for constants, read-write for variables, and the ability to invoke functions. `U` unmaps a package, so it is completely inaccessible in the *enclosure*.

When an *enclosure* manipulates data or functions from a foreign package, *e.g.,* passing one of its functions as a callback, the developer must explicitly specify the policies governing the closure's access. Explicit access specifications prevent accidental sharing of the foreign package's data.

`SysFilter` allows programmers to specify which system calls a closure can invoke. System

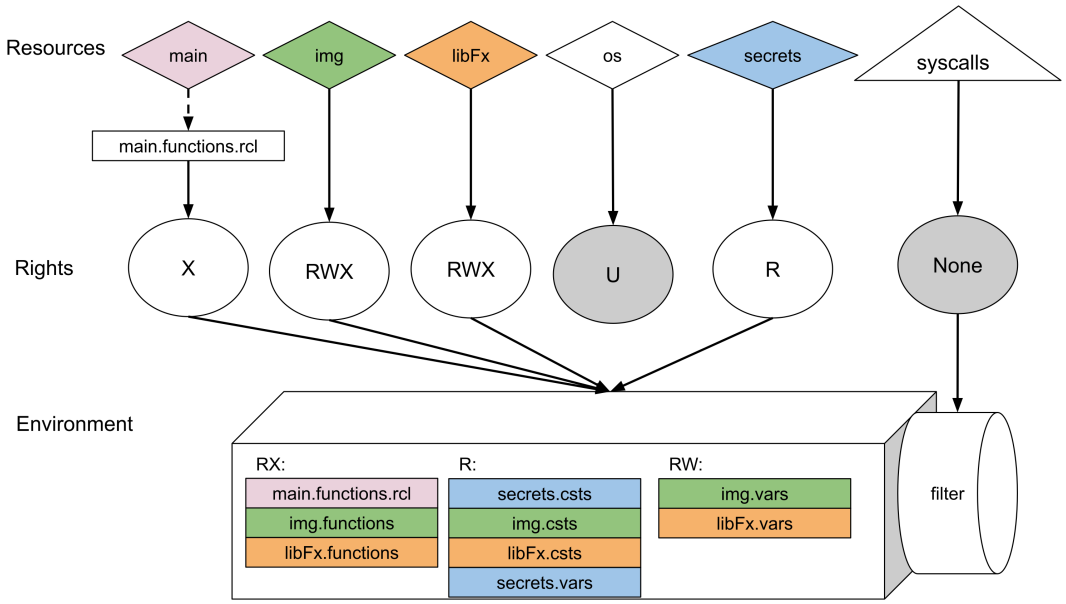

Figure 3.2 – Programs resources made available while executing the `rcl` *enclosure* defined in Figure 3.1.

calls are grouped into categories around logical services, *e.g.*, `file` for filesystem operations, `net` for network access, or `mem` for calls such as `mmap` and `mprotect`. A category included in `SysFilter` is allowed in the *enclosure*.

A call to an *enclosure* triggers a *transition* into a dynamically scoped environment restricted by its memory view and system call filter. These transitions are called *switches*. The closure runs inside this restrictive environment until it returns, thereby triggering a switch back to the caller's environment. *Enclosures* nest dynamically, but a switch can only enter an equal or more restrictive environment, preventing an escalation of privileges. It can return to a less restrictive environment. An *enclosure* faults if it violates the policies defined by its memory view and system call filter. A fault stops the execution of the closure and aborts the program.

Figure 3.1 presents an example of an *enclosure* in a small Go program and its corresponding directed package-dependence graph. Line 15 defines the `rcl` *enclosure* that calls the `Invert` function from the public package `libFx`. The *enclosure*'s natural dependencies are `img` and `libFx`. The `R` memory modifier extends `rcl`'s memory view to include the foreign package `secrets`, with read-only access. The `none` system call filter explicitly prohibits all system calls. At line 19, the *enclosure* computes and returns the inverse of the `original` image. As `original` belongs to `secrets`' arena, `rcl` is unable to modify it. Furthermore, `rcl`'s memory view does not include `main` or `os`, and so it would fault if it tried to access the `key` private key.

Figure 3.2 shows which resources belong to which package. The *enclosure* memory view and system call filter define which access to each package's resources are permitted. The `rcl` closure runs in an execution environment in which these restrictions are enforced.

3.2.3 Threat Model

We make no assumptions about the logic of the code running inside a restricted environment. Code from packages running in the environment can be implemented in unsafe languages, access raw memory, and execute system calls. *Enclosures* ensure that this code faults if it tries to access a package outside of its memory view or perform a prohibited system call. The developer is responsible for declaring *enclosures* to properly encapsulate untrusted code in their applications.

Enclosures assume that packages have a well-defined layout, *i.e.,* that their functions, variables (including heap), and constants can be identified and inspected to verify that they follow the format allowed by the LITTERBOX backend (see §3.5). In particular, packages cannot share memory pages. Enforcing this assumption is the responsibility of the compiler and is verified by LITTERBOX at run time.

We assume that the underlying operating system and hardware are correct. Side channels such as rowhammer [128] or microarchitectural flaws [129] that modify or leak memory content are out of scope.

3.3 Enclosure Policies

Enclosures can impose both fine-grain isolation policies on a single function invocation as well as program-wide policies on all uses of a package. They are similar in many aspects to program sandboxes: explicit transition into an isolated environment, a possibility of nesting restrictions, and explicit control of sharing exceptions. Because *enclosures* nest, they can be declared at any level of an application (*e.g.,* main program, a framework, or a package), which allows fine-grained tuning of isolation to the specific requirements of the code invoking the *enclosure*.

3.3.1 Default Policy

An *enclosure* associates a memory view, a collection of packages and respective access rights, and a system call filter with a closure and the code it invokes. The dynamic scope of this construct imposes its restrictions on the closure's natural dependencies, the code in the packages invoked by the closure. This dynamic behavior not only allows a given package to be subject to different restrictions when two *enclosures* use it, but it also allows enclosures' authors to restrict blackbox code whose source is unknown, unavailable, or too complex to manually inspect.

By default, *enclosures* prevent system calls and limit the memory view only to allow access to resources in a closure's natural dependencies. This default policy was chosen for its simplicity and usability. Other designs are possible, for example: disabling access to all packages and requiring a programmer to supply an allowed package list or allowing access to all packages

and expecting a denied list. However, both alternatives require in-depth knowledge of a program's package-dependence graph and extensive, brittle annotation. By contrast, *enclosure*'s policy allows isolation of a complex subsystem without an understanding of its potentially complex and evolving dependence graph of transitively invoked packages. It treats packages as a blackbox, yet provides them with a sufficient environment to run normally.

Enclosure's default policy disables all system calls. This decision forces programmers to state their assumption of which system services a package and its dependents might reasonably execute. Once again, this choice is not intrinsic. The proposed `SysFilter` syntax could be changed to allow finer-grained filtering, for example, by filtering on system call arguments.

3.3.2 Program-Wide Policies

Enclosures are a *local* mechanism that can enforce higher-level, program-wide policies. These policies are restrictions that apply across the full execution of a program. For example, package `Foo` should never have access to package `Bar`. An *enclosure* whose memory view unmaps `Bar` will enforce this restriction. To impose a program-wide policy, all calls into `Foo` must be enclosed. Currently, a programmer must manually insert an *enclosure* statement at each call site or provide wrappers for `Foo`'s functions that encapsulates them in *enclosures*. A compiler could automate this process by wrapping all calls into `Foo` in *enclosures* that do not allow access to `Bar`.

Program-wide policies implemented with *enclosures* allow enforcement of high-level security requirements, such as guaranteeing sensitive information's confidentiality and integrity. Confidentiality of a package's data is enforced by enclosing calls to other untrusted packages that should not access this information. Alternatively, these packages can be prevented from leaking information by disabling all system calls. A package's integrity can be ensured by mapping it read-only in the enclosed code. In Figure 3.1, `secrets`'s confidentiality is guaranteed by disabling all system calls for `rcl` and integrity is enforced by making `secrets` read-only. §3.6 contains other examples, including one showing how *enclosures*' non-disruptive integration with programming languages can implement secured-callbacks.

3.3.3 Limitations

Enclosures have a few inherent limitations.

Because they operate at package granularity, *enclosures* cannot selectively share a subset of a package's code or data. This could present challenges when a particular package holds a sensitive state and is shared by mutually distrustful packages. A possible solution is to refactor the program's code to extract the package's state and split it into separate packages that can be independently shared with each distrustful parties.

A second limitation relates to information flow control. As explained above, *enclosures* can

enforce the confidentiality of selected data by either not sharing it with untrusted code or disabling the *enclosure*'s system calls, thus preventing leakage. However, when enclosed code requires access to sensitive data and system calls, *enclosures* cannot guarantee that no information will be leaked. This is a challenging problem because any system call can be used as a side-channel to exfiltrate sensitive data shared through the *enclosure*. Section 3.6.5 provides a specific example of this situation and a mitigation.

Third, as mentioned in §3.2.3, side-channels attacks and microarchitectural flaws are not addressed.

3.4 LITTERBOX Design

Enclosures consist of two separate parts: (1) *frontend* language-specific support, provided by a language's compiler and runtime, and (2) the *backend* that uses hardware to enforce a closure's memory view and filter system calls.

LITTERBOX is a language-independent backend for *enclosures*. It supports diverse frontends with a simple API that offers transparent control over multiple hardware isolation technologies. Figure 3.3 presents a general overview of how a language frontend interacts with the LITTERBOX backend to implement *enclosures*.

3.4.1 LITTERBOX Abstractions

LITTERBOX defines simple system-level abstractions to represent a program's resources as sections, packages, and *enclosures*.

A *section* is a contiguous, page-aligned virtual memory region in the program's address space. Its start address, size, and default access rights (*i.e.,* read (R), write (W), execute (X)) characterize it. Sections can be dynamically allocated at run time, *e.g.,* with `mmap`.

A *package* is a collection of non-overlapping sections. It has a unique name and typically contains one or more *text* (RX), *rodata* (R), and *data* (RW) sections. A package's arena, §3.2.1, is part of its data sections and is not shared with other packages.

An *enclosure* consists of a unique identifier, the virtual address of its closure, its memory view as a set of package names and associated access rights, and its system call filter. The closure resides in its own text section owned by the package that declares it.

As packages partition the program's address space, LITTERBOX uses package dependencies to compute an *enclosure*'s complete memory view. This operation occurs at startup time for compiled statically-linked languages and package-load time for dynamic languages.

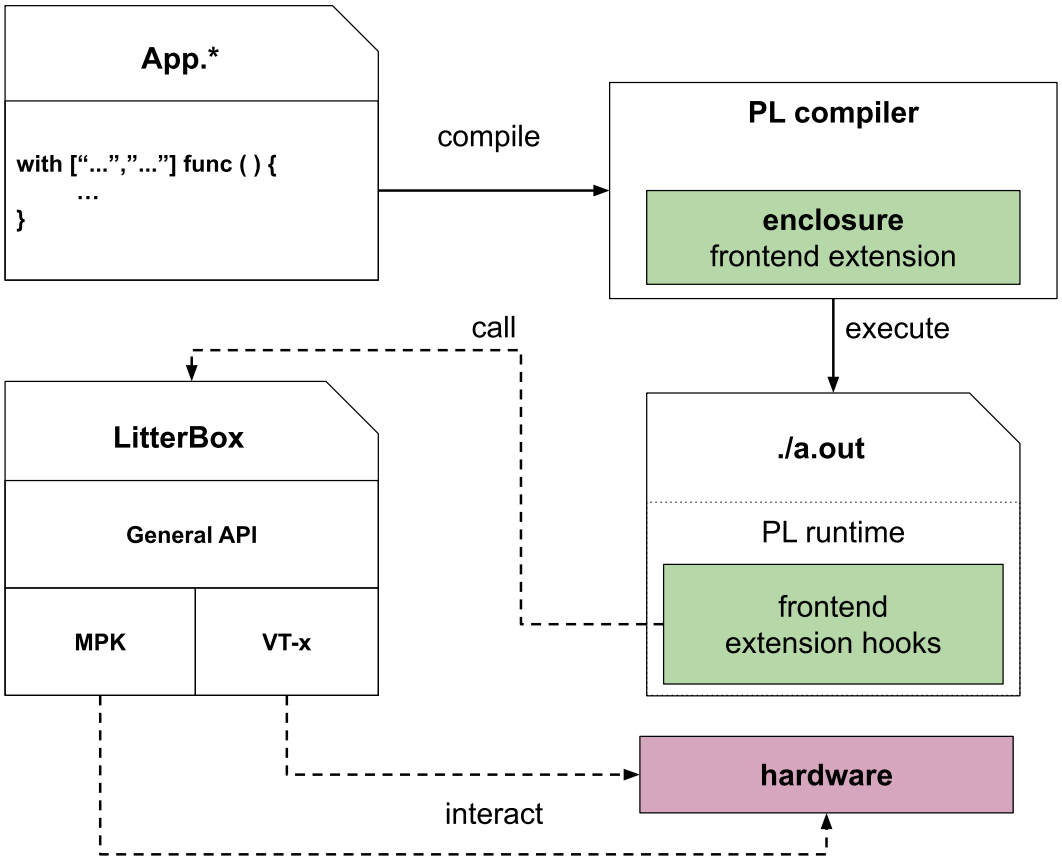

Figure 3.3 – Overview: language support for *enclosures* with *frontend* extension inside the PL's compiler, and runtime hooks to call the language-independent LITTERBOX *backend*.

3.4.2 LITTERBOX API

LITTERBOX exposes a small API to the frontend language implementation. LITTERBOX supports *enclosure*-defined operations with four functions: (1) `Init`, (2) `Prolog`, (3) `Epilog`, and (4) `FilterSyscall`. It provides two additional functions for language runtimes: (5) `Transfer` for dynamic memory management and (6) `Execute` for user-level scheduling.

A runtime's initialization code in statically-linked languages or the package import and *enclosure*-parsing code for dynamic languages calls `Init`. It takes a description of the program's packages and *enclosures* and computes the memory views. LITTERBOX initializes the underlying hardware to create, for each *enclosure* memory view, the corresponding restrictive execution environment.

`Prolog` and `Epilog` provide the switch mechanism that allows a program to enter and leave an *enclosure*'s execution environment.

`FilterSyscall` is called when an *enclosure* attempts to perform a system call. It either permits the call to execute or rejects it and raises a fault.

Memory allocators require a mechanism to shrink or extend a package's arena. The `Transfer` function dynamically repartitions heap memory by transferring a section from one package to another. It permits memory allocators to reuse freed memory sections for a subsequent allocation, even across packages.

Some modern languages, such as Go, provide user-level threading to support concurrent execution. The language's runtime implements a scheduler that yields execution from one user thread to another. `Execute` enables run-time scheduling of user-level threads by providing a switch mechanism between two unrelated protection environments. The language's scheduler calls `Execute` to transition from one user thread execution environment to another. Thus, the scheduler can preempt, block, or resume an *enclosure*'s execution in the correct execution environment.

3.5 Implementation

Language support for *enclosures* requires changes to a programming language's syntax, compiler, and runtime. The full Go extension for *enclosures* is a 1,000 LOC patch to the Go compiler and runtime. The patch is modular and self-contained, so it can be easily maintained. The Python extension prototype is a fork of CPython with 600 LOC changed to introduce a multi-segmented heap for dynamic memory management. LITTERBOX is 6,500 LOC written in Go.

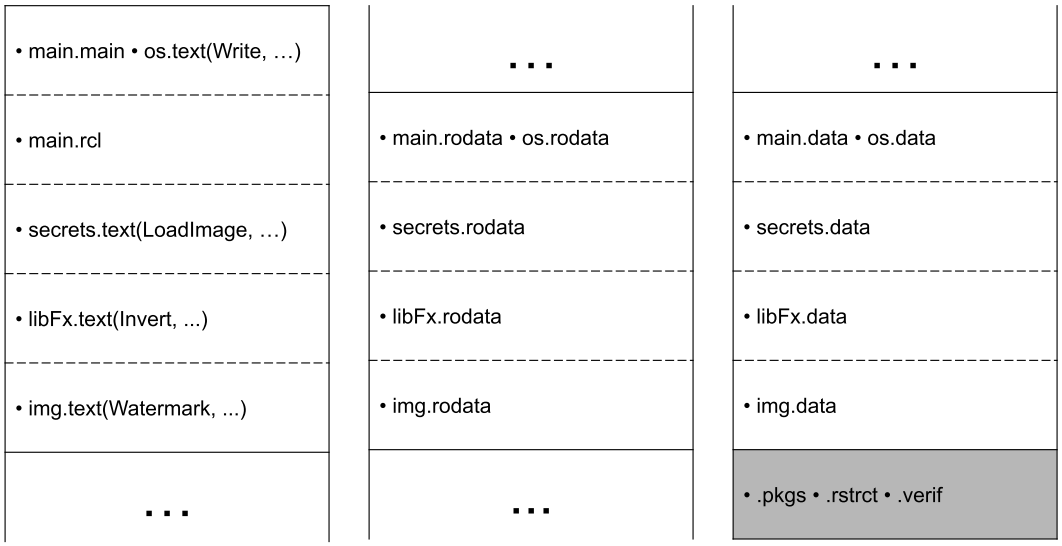

Figure 3.4 – Figure 3.1's final executable content produced by Go's frontend support for *enclosures*. ELF sections from left to right: `.text (RX)`, `.rodata(R)`, and `.data (RW)`. Dashed lines represent intra-ELF section page-aligned symbol addresses, and greyed out entry the frontend's generated ELF sections for LITTERBOX.

3.5.1 Go Frontend

Parsing: We extend Go's syntax to accept the *with* keyword using the syntax in Section 3.2.2. *Enclosure* policies are parsed as literals, *i.e.,* string constants. This allows the compiler to validate their satisfiability at compile time. Packages can define `init` functions to be executed at package load time, and additional syntactic sugar is needed to tag package import statements with *enclosure* policies. This encloses the execution of the package's `init` function inside an *enclosure*. The parser also registers per-package *enclosures* and assigns unique identifiers.

Compiling: The compiler relies on the type checker to identify and register an *enclosure*'s direct dependencies and insert the `Prolog` and `Epilog` calls. It also augments calls to the dynamic allocator (`mallocgc`) with the caller's package identifier. The compiler outputs one *code object* per package that contains the expected `.text` (functions), `.data` (global variables), and `.rodata` (constants) sections, as well as a `.rstrct` section containing the package's *enclosures* configurations and direct dependencies.

Linking: The linker has global knowledge of the program's package-dependence graph and assembles packages' code objects into a single executable. For each code object, it extracts the `.rstrct` sections, computes every *enclosure*'s memory view, and marks packages that appear in at least one *enclosure*. The linker's symbol address-assignment algorithm segregates marked packages resources in separate sections so that no two marked packages overlap. *Enclosure* closure functions are isolated into their own memory sections.

The linker outputs three distinguished ELF sections as part of the executable. The `.pkgs` and

`.rstrct` sections hold descriptions of packages and *enclosures* to be passed to LITTERBOX's `Init` during runtime initialization. A `.verif` ELF section stores the call-sites to LITTERBOX hooks, which LITTERBOX uses to filter API calls at run time. Figure 3.4 illustrates the executable corresponding to Figure 3.1's program.

Runtime: Go's dynamic memory allocator divides the heap into class-size sections, called spans, cached per hardware thread and used to satisfy allocations based on the requested size. The *enclosure*-extension adds a level of indirection by dynamically assigning spans to packages' arenas. After adding a span to a given arena, the runtime calls LITTERBOX's `Transfer`.

Go scheduler *enclosure*-extension maintains a mapping between a routine and the corresponding execution environment and relies on split-stacks to isolate frames preceding the *enclosure*'s call. To avoid escalation of privilege attacks, execution environments are transitively inherited by goroutine creation so that user-level threads created inside an *enclosure*'s environment continue to execute in the same environment. The scheduler uses the `Execute` hook to switch between goroutines associated with different environments. Similarly, garbage collection needs full access to the program's resources but executes on top of runtime goroutines associated with a trusted execution environment.

3.5.2 Python Frontend

LITTERBOX can support dynamic programming languages. Our Python prototype is based on a fork of CPython 3.9.1. Rather than repeating implementation details common to the Go frontend (parser extension, or instrumenting *enclosures* bodies), this section focuses on challenges attributable to: 1) Python's dynamic behavior, and 2) CPython's implementation.

Dynamic behavior: Python is a dynamic language that accumulates and processes knowledge about a program during its execution. Modules (packages) are lazily imported when a file is parsed and functions are compiled only when needed. As a result, and unlike Go, LITTERBOX must accept multiple calls to `Init`, each of which provide only partial information about a program. CPython's import mechanism registers modules and their direct dependencies with LITTERBOX. Similarly, the compiler registers *enclosures* and their direct dependencies as they are compiled.

In this dynamic setting, and unlike Go, LITTERBOX, not the compiler, must compute the transitive dependencies of modules and *enclosures* full memory views. Furthermore, the execution of an *enclosure* can trigger new imports, so LITTERBOX's default policy makes these new packages available to the executing *enclosure*, unless explicitly restricted by user policies.

Python provides little control over dynamic memory allocation. The language does not offer an equivalent of Go's `new` or C's `malloc` functions to identify dynamic allocation. Without explicit allocation, it is difficult for a programmer to encapsulate data in a specific module. To allow a programmer to express this intent, we implemented `localcopy`, a function similar to

Python's `copy.deepcopy`, which creates an object copy in the caller's module.

CPython internals: CPython is the reference implementation for the Python programming language. It is a highly optimized, complex system that presents some challenges in providing strong isolation guarantees.

CPython's default object memory allocator is a singleton whose state resides in global static variables. Its design is very similar to Go's, in that it manages `mmap`ed arenas divided into class-sizes. We made some small changes to encapsulate the allocator's state in a structure, which allowed multiple allocator instances to co-exist with non-overlapping arenas. This in turn enabled us to assign a memory allocator per module and segregate objects allocated by different modules on distinct memory pages. Our memory allocator further distinguishes functions (code) and objects (data) in separate arenas within one module. This allows LITTERBOX to hide a module's functions when the module is mapped without execution rights, while still allowing access to its data.

For performance reasons, CPython co-locates data and metadata, specifically the reference counting counters in the headers of objects. While efficient, this implementation decision makes it difficult for an isolation mechanism to enforces read-only semantics on an object, as it would preclude updating reference counts. Similarly, the CPython generational garbage collector (GC) embeds a linked list's `next` pointer inside object, which might be inaccessible within an *enclosure*. To circumvent these problems, our CPython extension performs a controlled switch to a trusted environment, with full access to program resources, to modify reference counts in read-only objects or enqueue on the GC linked lists. While sufficient for a prototype, this approach is expensive as the full cost of two switches is incurred on every access to read-only objects. In the future, our Python extension will separate objects' data and metadata.

3.5.3 LITTERBOX Implementation

LITTERBOX provides support for two hardware-enforced isolation mechanisms: Intel VT-x (LB^{VTX}) and Intel MPK (LB^{MPK}). While these two technologies differ greatly, LITTERBOX provides a common implementation and only differentiates between the selected hardware for three operations: (1) creating and enforcing an execution environment (`Init`, `FilterSyscall`), (2) extending a package's arena (`Transfer`), and (3) performing a switch between execution environments (`Prolog`, `Epilog`, `Execute`).

Common Aspects: LITTERBOX validates the configuration passed to `Init` by ensuring that sections are aligned and non-overlapping and that the memory views and authorized system calls can be satisfied. At that point, LITTERBOX performs an important optimization by clustering the packages across all memory views that have the same access rights. This clustering creates larger, logical *meta-packages* that can be efficiently managed. LITTERBOX derives the set of memory sections and associated access rights that, together with system call

filters, define the *enclosure* execution environment *description*.

LITTERBOX code and data consist of two packages, namely `user` and `super`. The `user` package is available in all execution environments and provides authorized access to `Prolog`, `Epilog`, `Execute`, and `Transfer` hooks. The `super` package contains the *enclosures* definitions, the verification list of allowed call-sites to the API, and the descriptions. It also handles the logic that validates calls to the LITTERBOX API, modifies execution environments, and performs switches.

Both hardware implementations use execution environment descriptions to initialize their underlying hardware and create different execution environments.

LB$^{\text{VTX}}$: Intel Virtualization Technology extension (VT-x) [185] extends the x86 ISA to simplify hypervisor implementation. It relies on an extended-page-table (EPT) in hardware to map host virtual (HVA) and guest physical addresses (GPA). It defines VMX root for a hypervisor with unmodified CPU behavior and non-root mode for guest operating systems with restricted CPU behavior. The non-root mode has access to the virtual machine's CR3 register and can manage its guest virtual (GVA) to GPA mappings.

LB^{VTX} relies on Linux's Kernel-based Virtual Machine (KVM) module [45] for Intel VT-x to create a virtual machine (VM) in which the application executes. An execution environment in the context of LB^{VTX} is a page table mapping that enforces the *enclosure* description, in other words, its memory resources are associated with the correct access rights in user-space. LB^{VTX} creates a separate page table for each *enclosure*. It also allocates one trusted page table with user-access to all packages except LITTERBOX's `super` to run non-enclosed code. Finally, `super` is mapped in the guest kernel address space (non-root kernel mode) and implements the guest operating system. For simplicity, LB^{VTX} strives to preserve $GPA == GVA == HVA$ whenever possible. It only breaks the invariant $GPA == GVA$ when necessary to circumvent VT-x's 40 bits physical address space. Once all execution environments are initialized, LB^{VTX} enters the VM and resumes the application's execution in non-root user mode, with the trusted page table mappings.

To perform a switch, LITTERBOX functions perform a specialized system call to our guest operating system. The system call handler has access to the `super` package, and checks that the call-site to LITTERBOX's API corresponds to the program's specifications supplied to the `Init` function, which is found in the `.verif` ELF section. If the transition is authorized, the guest operating system switches the VM's CR3 register (the page table root) to the target execution environment and returns (`iret`). Using a single VM per application and implementing switches as system calls, rather than instantiating a VM per *enclosure*, reduces both the complexity of LITTERBOX management of KVM state and the overhead of switches because a syscall is less costly than a VM EXIT.

`Transfer` is also implemented as a system call that updates the relevant execution environments' page tables. Similar to switches, the call-site and the validity of the arguments are

checked before applying the desired modifications.

The handler filters system calls according to the current execution environment's filter. If authorized, system calls are passed through to the host [68] via a hypercall (`VM EXIT`). The system call is performed in root user mode, which then returns to the VM with the results (`VM RESUME`).

A fault triggers a `VM EXIT`, prints a trace of the root-cause, and stops the program's execution.

LB^{MPK}: Intel Memory Protection Keys (MPK) [124] extend the x86 ISA to enforce memory page protections without domain switches. Page table entries are tagged using 4 previously ignored bits to encode 16 different tags, called keys. A new user-writable and readable register, `PKRU`, uses two bits per key (32 bits total) to encode access and write capabilities for pages tagged with the corresponding keys. Hardware enforces PKRU permissions on data access. The Linux kernel provides system calls to manage keys, *i.e.,* allocate and free, and the `pkey_mprotect` system call to tag a range of addresses with a key.

LB^{MPK} relies on Intel MPK to isolate *enclosures*. It allocates one key for each meta-package. In practice, clustering packages results in fewer than 16 meta-packages whose views fit into the 16 keys. Libmpk [163]'s key virtualization could be used to overcome Intel MPK's limitation if the need arises. Similar to Erim [186], LB^{MPK} scans the program to ensure that only the LitterBox package modifies the PKRU register. As in LB^{VTX}, all calls to the API are checked against the verification information stored in `super`.

An execution environment for LB^{MPK} is simply the PKRU register's value that encodes access rights for all meta-packages.

A switch validates the transition using `super`'s verification and writes the PKRU register.

A transfer is slightly more complicated as it must invoke a `pkey_mprotect` system call to update the relevant page table entries' key.

System calls are filtered by translating the `FilterSyscall` function into a BPF filter loaded via seccomp [51, 140], which indexes the current environment (from the PKRU value) to a mask of permitted system calls. We use a Linux kernel patch [152] to expose the `PKRU` register to seccomp. As Intel MPK is an emerging technology, we consider it a reasonable assumption that future versions of the kernel will incorporate a similar mechanism. An alternative would be to implement techniques similar to Erim [186] or rely on a BPF map updated upon switches.

A fault in LB^{MPK} stops the program's execution.

3.6 Evaluation

This evaluation is performed on an Intel(R) Xeon(R) Gold 6132 CPU @ 2.60GHz running Ubuntu 20.04 LTS with Linux kernel version 5.4.0-42-generic, and a patch [152] to access

Table 3.1 – Microbenchmarks results in nanoseconds.

	Baseline	LB^{MPK}	LB^{VTX}
call	45	86	924
transfer	0	1002	158
syscall	387	523	4126

PKRU value in seccomp. We report numbers for the Go frontend implementation based on LITTERBOX. The evaluation is divided into two parts: (1) microbenchmarks to measure the cost of LITTERBOX's fundamental operations with both hardware enforcement mechanisms, and (2) macrobenchmarks to study *enclosure*'s usage in realistic applications, divided into a qualitative and quantitative studies.

As a baseline, we report unmodified Go performance, noted as Baseline, where *enclosures* are replaced by vanilla closures. LITTERBOX's Intel MPK hardware enforcement is reported as LB^{MPK}, and Intel VT-x as LB^{VTX}. All benchmarks run single threaded in order to accurately quantify the overheads of domain crossings (*i.e.,* switches).

3.6.1 Microbenchmarks

We rely on microbenchmarks to answer the following questions: (1) What is the cost of performing a call to an *enclosure*? (2) What is the basic cost of memory management calls to the `transfer` LITTERBOX's hook? (3) What overheads does LITTERBOX impose on system calls?

To answer each of these question, this evaluation uses three microbenchmarks to measure LITTERBOX's overheads:

- `call`: measures the time required to call and return from an empty *enclosure*.
- `transfer`: calls LITTERBOX's `Transfer` on a 4-page memory section.
- `syscall`: an *enclosure* performs a `getuid` system call in a loop.

We run each microbenchmark a million times and report the median latency value, in nanoseconds, in Table 3.1. These latencies are shared by the Go and Python frontends, as they both use LITTERBOX's backend.

call: The cost for the Baseline is $45ns$ and $86ns$ and $924ns$ for LB^{MPK} and LB^{VTX} respectively. This translates to an overhead of $\sim 40ns$ (86-45) per *enclosure* call for LB^{MPK} and less than 1 µs (924-45) for LB^{VTX}. LB^{MPK} is thus able to perform a single switch in approximately $20ns$ by writing the PKRU register. LB^{VTX} switch depends on a system call to change the CR3 register, so effectively we measure the cost of two system calls.

transfer: The relative performance of each backend changes when it comes to memory management. LB^{VTX} is able to efficiently transfer a memory span by toggling the presence

Table 3.2 – Macrobenchmarks results.

	Baseline	LB^{MPK}		LB^{VTX}		Benchmark information				
	raw	raw	slowdown	raw	slowdown	App TCB #LOC	Enclosed #LOC	#Stars	#Contributors	#Public deps
bild	13.25ms	14.88ms	**1.12x**	13.91ms	**1.05x**	32	166K	2.9K	15	1
HTTP	16991reqs/s	16738reqs/s	**1.02x**	9560.14reqs/s	**1.77x**	31	-	-	-	-
FastHTTP	22867reqs/s	22025reqs/s	**1.04x**	11375reqs/s	**2.01x**	76	374K	13.1K	100	3

bits in the corresponding page tables. LB^{MPK}, however, requires a `pkey_mprotect` system call, which is ~6 times slower than LB^{VTX}.

syscall: LB^{MPK} incurs negligible overheads as system call filtering requires a few operations to accept or reject a system call based on the PKRU value. LB^{VTX} relies on hypercalls to service system calls and pays the full cost of a `VM EXIT` of ~4µs. This approach is similar to other container technologies such as gVisor [199].

These benchmarks suggest that both implementations impose reasonable overheads when it comes to an *enclosure* call, as these can potentially be amortized by the closure's service time. While LB^{VTX} more efficiently handles memory sections being transferred between packages, LB^{MPK} wins when it comes to filtering and executing system calls. Thus, depending on application characteristics, users can make an informed decision on which version of LitterBox to use.

3.6.2 Macrobenchmarks

This section uses popular Github Go packages to benchmark the performance of small applications derived from each package's "hello world" sample, to determine the worst-case performance overheads of LitterBox. In these applications, *enclosures* are used, in very different ways, to safely leverage the *unmodified* public package. Table 3.2 reports the achieved raw performance with Go Baseline, LB^{MPK}, and LB^{VTX}, and respective slowdowns for each benchmark.

Reducing the application's TCB: *Enclosures* drastically reduce the trusted codebase (TCB), *i.e.,* code executing with full access to the program's address space and syscall API. As shown in Table 3.2, each application consists of less than a hundred lines of code (LOC) that import thousands of LOC from public dependencies. In every macrobenchmark, a single *enclosure* declaration, using the default policy, completely encloses public library code and its transitive dependencies, thus preventing any public package from accessing and leaking sensitive information held by the application.

Processing Sensitive Images with Bild: Bild [176] is a popular Go Github public package for parallel image processing. While presenting attractive functionalities, such as an `Invert` function for images, bild silently drags-in over 160K lines of code of unverified origin. This is an daunting quantity of potentially harmful code to examine while writing a simple 32 LOC application that loads and inverts an image. We declare an *enclosure* to enclose the call to

bild's `Invert` function. We further disallow all system calls and extend the *enclosure* memory view with read-only access to the `main` package that holds the sensitive image to invert. This simple benchmark is the textbook example of how sensitive information can be safely exposed to an untrusted package, while preventing modifications of program state or leakage (*e.g.,* via system calls).

The benchmark is purely computational and memory-intensive as it allocates and computes an inverted image. LB^{VTX} shows a mere 5% slowdown. As predicted by microbenchmarks in §3.6.1, LB^{VTX} overhead to call an *enclosure* is absorbed by the closure's service time and its efficient mechanism for transfers. LB^{MPK} achieves a respectable 12% slowdown, attributable to the cost of frequent transfers to populate the arena with memory spans of various sizes to satisfy bild's dynamic memory allocations.

Securing an HTTP server: Go provides an HTTP server implementation in the `net/http` package. A typical concern in web-facing applications with TLS support is to protect private keys and certificates from potential attacks delivered via user requests. Such attacks can, for example, attempt to trigger a buffer-overflow in the request-handler to leak sensitive data. This benchmark defines the request handler as an *enclosure* with no access to the packages used by `net/http` and no system calls. To measure raw overheads, the handler's logic only selects a 13KB in-memory static HTML page to service the request. This is a typical use of *enclosures* to prevent potentially harmful code from accessing sensitive resources.

Once again, the observations made in §3.6.1 are confirmed. As the benchmark is primarily dominated by socket operations, LB^{VTX}'s high overhead in servicing system calls introduces a 1.77× slowdown. This time, as the *enclosure* does not perform dynamic memory allocations, LB^{MPK} is able to perform almost as well as the baseline.

Using a Public HTTP Framework: FastHTTP [187] is an industry-grade Github public Go package that implements a performance-oriented HTTP server. FastHTTP offers high throughput, as long as we can trust over a 100 programmers and more than 350K LOC. To prevent FastHTTP from accessing an application's sensitive resources, we create and run the server in an *enclosure*, only allowed to perform `net`-related system calls (*i.e.,* socket operations). The *enclosure* forwards requests to a trusted handler goroutine via go channels. This benchmark shows how trusted callbacks can easily be implemented. To measure overheads precisely, the trusted handler simply returns 13KB static HTML pages as before. In a more realistic deployment, the handler would access a private database or other sensitive information, which would be completely unavailable to the *enclosure* running the FastHTTP server.

Similar to the simple HTTP experiment, LB^{MPK} achieves a throughput comparable to the baseline. We observe a small slowdown that seems to be due to the server's consumption of dynamic memory. This cost is however greatly diminished by FastHTTP efficient usage of memory, *e.g.,* `HTTPRequest` object reuse across requests. This allows LB^{MPK} to avoid numerous costly transfers. LB^{VTX} has a 2× slowdown due to system calls. Note that the LB^{VTX} slowdown

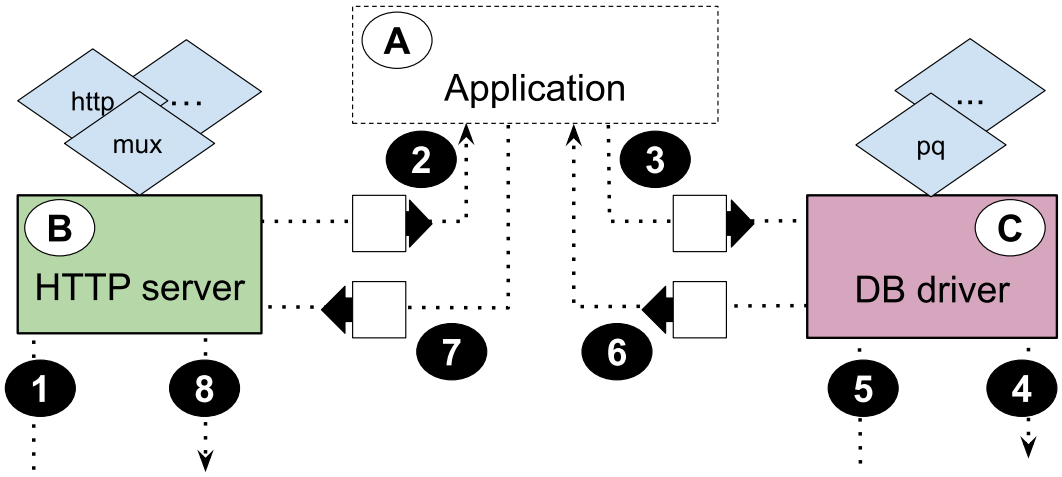

Figure 3.5 – *Enclosures* isolating the HTTP server and the database driver in a wiki-like web application.

in FastHTTP is larger that in HTTP. This is not due to an increase in the frequency of system calls as FastHTTP and HTTP have a similar system call trace. However, FastHTTP service time to accept connections and parse requests is significantly smaller, while the system call overhead remains the same.

3.6.3 Usability

We consider a wiki-like web-app [52, 178] that stores its pages in a Postgres database, as depicted in Figure 3.5. This web-app is written in Go, relies on the deprecated `pq` [99] public library as a Postgres driver and on the `mux` [182] package to route HTTP requests consisting of `GET` (read pages) and `POST` (create a page). Together, `pq` and `mux` incorporate 44 public Github packages as dependencies.

To prevent any public package from subverting our application, we rely on two *enclosures* that communicate with trusted code via Go channels.

First, the HTTP server Ⓑ consisting of `mux` and its transitive dependencies is enclosed without access to the database, the file-system, or the rest of the application holding sensitive information, *e.g.,* page templates and the database password. It is however authorized to create and read ①/write ⑧ to its own network sockets. Similar to the FastHTTP experiment, HTTP handlers forward parsed requests to trusted code on a private Go channel ②.

Second, `pq` and its natural dependencies are isolated in an *enclosure* Ⓒ, acting as a proxy server only allowed to communicate with Postgres via a pre-defined network socket. This enclosure has no access to the HTTP server's logic, the file-system, and network operations on other sockets. This database proxy server accepts SQL requests on a Go channel ③, communicates them to Postgres, ④ and ⑤, and returns the result to trusted code ⑥.

The trusted code base, *i.e.*, non-enclosed Ⓐ, consists of the application's glue code, responsible for reading requests forwarded by the enclosed handlers ②, contacting the enclosed database proxy server ③, validating the SQL query result ⑥, and generating and forwarding ⑦ the HTML response.

The throughput slowdown is similar to the one in the FastHTTP experiment.

3.6.4 Python Enclosures

The CPython extension is an unoptimized prototype. While §3.6.1 presents reasonable overheads for LITTERBOX basic operations, we would like to study how the challenges described in §3.5.2 affect the performance of Python programs. This section quantifies the impact of these limitations on the Python *enclosures* performance and provide insight into improving *enclosure* on this platform.

Consider a Python program with a single *enclosure* that encapsulates the use of the `matplotlib` module. User sensitive data from a `secret` module is shared read-only with a closure that generates a plot from the data and writes the result to disk. The experiment runs using LB^{VTX} to understand the relative impact of each overhead, including system calls.

We first try a conservative approach where each reference count operation and garbage collection triggers a switch to a trusted environment before returning to the *enclosure*, as described in §3.5.2. This experiment shows a ~18x increase in execution time for *enclosures*, as compared to standard Python. We measure nearly 1M switches due to reference counting and garbage collection. The delayed initialization of the *enclosure* environment, including computation of package dependencies, *enclosures* memory views, and configuration of the underlying hardware mechanism (KVM), represents 4.3 percent of the measured slowdown. System call overheads (requiring a VM exit) account for less than 1 percent of the slowdown. This can be explained since the system calls (futex and write) have a service time larger than the system call overhead measured in §3.6.1.

We run a second experiment to simulate changes that allow updating a read-only object's reference count without a switch. To do so, the `secret` module is mapped with read-write access and switches for reference count operations are disabled. The measured slowdown is now ~1.4x and is dominated by the delayed initialization cost. Note that this cost has to be paid once, at the first invocation of an *enclosure* and can be amortized if the *enclosure* is called multiple times.

From this experiment, we believe that decoupling CPython data and metadata would enable more efficient support of *enclosures* and should be the main focus of future work.

3.6.5 Security

This section shows that *enclosures* can address the threats from the malicious packages cited in §3.1 [78–81]. To this end, we re-created Python and Go packages that perform the same attacks as the original malicious ones. These attacks mostly access local secrets, either within the program's memory or on the local file system (*e.g.,* private SSH keys), and attempt to exfiltrate them via the network or open backdoors on the local system.

Enclosures easily detect and protect against most attacks with a basic configuration, *i.e.,* the default memory view and limited system calls, while still allowing valid behaviors to run successfully. However, a few packages [78, 80] presented a challenge. These packages provide a valid functionality that requires access to a secret and system calls that could be used to exfiltrate sensitive data. For example, the `ssh-decorater` package [78] allows SSHing to a given IP address and executing python commands on the remote server. The public library was, however, infected with malicious code exfiltrating user credentials to another server, via a `POST` request. To prevent this attack, we modify the application code to pass a pre-allocated socket and private key to the enclosed `ssh-decorater` public package, therefore enabling us to disable socket creation and file-system access. Another solution extends the sysfilter categories to only allow connect system calls to a list of pre-defined IP addresses, allowing us to grant socket creation and file-system access to `ssh-decorater`, while preventing it from contacting a malicious server. Note, however, that the valid remote host can still be used as a relay to send the credentials to the malicious server.

A similar issue arose with malicious clones of the Python Django framework. To protect against these, we took an approach similar to the one used in FastHTTP with secured callbacks.

3.7 Discussion

Granularity: Packages' composition and size make possible effective isolation inside an application's address space. Their coarser granularity is far easier to manage than individual objects, and better fits the granularity of page-based hardware isolation mechanisms. Moreover, packages can often be clustered into efficient meta-packages, as explained in §3.5.3. Clustering reduces the number of keys needed to tag an entire address space and, in many cases, fits into the 16 possible Intel MPK keys.

Explicit scoping: *Enclosures* utilize dynamic scoping to control the application of restrictions on program resources. *Enclosures* isolate untrusted packages by explicitly enclosing invocations of their functions. An alternative approach to protect against public packages could automatically enforce a transition to a restricted execution environment on every single untrusted package function invocation. This approach is, however, limiting as compared to *enclosures* for 3 reasons: 1) it requires modification of untrusted package code, 2) it imposes a switch per call into a package, prevents programmers from controlling switches, and might result in large overheads, and 3) *enclosures* can emulate this approach, as mentioned in §3.3.2.

Hardware enforcement: As the need for in-application isolation grows, hardware should evolve to provide efficient, reliable, and easy-to-use enforcement mechanisms. Intel MPK offers a first-generation solution that exploits unused bits in the PTE to store protection keys on existing hardware. However, Intel's decision to permit key modification by unprivileged code is debatable, especially given system calls' low cost. An ideal solution would combine MPK's low overheads and ease-of-use with VT-x's robust protection model, scalability to multiple address spaces (each with 16 keys), and the ability to filter system calls in a protected library operating system [68]. This last aspect is similar to the gVisor/Sentry mechanism for containers [199].

Capabilities: Capability support, as proposed by CHERI [195] or CODOMs [188], is an attractive approach that offers simplicity, expressiveness, and strong guarantees at the cost of more substantial hardware changes. LITTERBOX, with its decoupling of the API and hardware implementations, could support capabilities in the future.

3.8 Related Work

Isolation is the combination of a policy (what is isolated) and mechanism (how is the isolation enforced). The system may restrict interactions for various reasons, such as limiting error propagation or constraining untrusted components. LITTERBOX focuses on untrusted packages that share the same address space as a trusted program. Intra-address space isolation has been studied along different dimensions, we list here key differentiating factors.

Operating system mechanisms: The most common software isolation mechanism is operating system processes. While the application and its packages could be partitioned into separate processes, the cost of IPC and the complexity of argument marshaling along with the high implementation complexity limit this approach to a few examples such as web browsers [143].

Previous work proposed incorporating some process isolation and control mechanisms within an address space [71, 122, 142]. Wedge's sthreads restrict memory accesses and system call capabilities associated with a thread [71]. LWC's light-weight contexts extend this approach by providing control over the memory view, system call capability, and execution state inside an object [142]. SMV's secure memory views provide a more straightforward approach, a uniquely identified memory domain that can be accessed when it is attached to the current thread [122]. Common to all these approaches is that they require code refactoring to use the new mechanisms.

By contrast, *enclosures* offer a more natural separation that closely integrates with the language. Unlike LWC, a developer does not need to be aware of all of a package's transitive dependencies or their layout in memory and can, with a single line of code, completely encapsulate them. Moreover, LITTERBOX could employ these systems to enforce memory isolation and syscall filtering. LWC presents an interesting OS abstraction and could provide an alternative LITTERBOX backend that does not require specialized hardware (*e.g.*, Intel VT-x).

Virtualization mechanisms: Virtualization enables a hypervisor to apply different access permissions to a collection of memory pages shared among several virtual machines. These VMs can encapsulate code running in the same address space, with different access rights. Dune [68] uses Intel VT-x [185] to virtualize the process abstraction and isolate software components. SIM [174] with Intel VT-x protects a trusted security monitor running in an untrusted guest. Nexen [175] and HyperSafe [191] focus on protecting and isolating hypervisors. LXD [155], Nooks [179], and Nested Kernel [92] isolate kernel submodules. TrustVisor [151] is a thin hypervisor that isolates portions of an application. SeCage [144] takes a data-driven approach, by automatically partitioning an application into security domains, based on the secrets they access, and isolates them from each other with Intel VT-x.

Unlike Secage, *enclosures* enforce security domain boundaries based on packages. This simplifies an application's partitioning, especially for environments where static and dynamic analysis is hard. *Enclosures* make it easier for developers to reason about isolated compartments and prevent accidental sharing of sensitive data, *e.g.,* via valid pointer references passed to the untrusted package. *Enclosures* further allow fine-grain schemes, such as exposing sensitive data to untrusted packages, while preventing it from being modified.

Virtualization generally incurs (high) performance overhead due to extended page tables and hypercalls. Intel MPK provides more specialized hardware support with lower overheads. For example, Hodor [118] shows that Intel MPK [124] isolates data-plane libraries with far smaller overheads than Intel VT-x.

Similarly, *enclosures* based on LITTERBOX support both Intel VT-x and Intel MPK as isolation backend mechanisms.

Programming language and runtime mechanisms: Programming languages provide information hiding and abstraction mechanisms, a weak form of isolation. These are software-engineering mechanisms meant to decouple components, rather than robust run-time isolation mechanisms. The frequent escapes from unsafe languages or run-time reflection demonstrates that this cat-and-mouse game is not a real security solution.

Software Fault Isolation (SFI) [190] adds memory isolation to unsafe languages by inserting dynamic checks on accesses. Control-Flow Integrity (CFI) [54] guarantees that only valid code areas can be executed. Together, CFI and SFI guarantee that only a subset of the application may access the protected memory region. However, these techniques require non-trivial static and dynamic analyses with non-negligible costs in terms of complexity and overheads. An alternative solution embodied by NaCl [197], XFI [207], WebAssembly [117], and CloudFlare's [201] use of V8 isolates [110] are akin to Proof-Carrying code [157] and enforce restrictions on which machine instructions can access segmented memory [170].

Enclosures built on top of LITTERBOX combine several of these approaches. Their close integration in a programming language allows statically delimited memory view boundaries and control of the generated code. Similar to PCC, the resulting binary abides by a certain

format, verified by LITTERBOX along call-sites to its API, and leveraged to efficiently apply hardware-enforced isolation.

Language-based hardware-enforced mechanisms: Isolation can be provided by extending a language with security domains and enforcing isolation with specialized hardware. SOOAP [116] is a security analysis framework that relies on annotations to help refactor and compartmentalize existing applications. It could be used in conjunction with LibMPK [163], a library that virtualizes and manages MPK keys, to provide strong isolation guarantees. Erim [186], using Intel MPK, and Shreds [86], using ARM memory domains [62], expose isolated memory pools and associate code allowed to access them. Memsentry [131] provides an LLVM pass to implement data encapsulation, selectively enforced by MPK or other technologies. Similarly, Glamdring [138] relies on annotations to mark sensitive data and isolate code accessing it inside Intel SGX [21] enclaves. Glamdring and *enclosures*' specifications differ as they solve two (different) problems. *Enclosures* annotate the entry points of top-level packages, which are the fundamental abstraction they isolate, as secrets are for Glamdring. Gotee [107] isolates trusted code with its own memory pool inside SGX enclaves. JITGuard [104] relies on Intel SGX to protect jitted code.

In essence, most of these solutions focus on data encapsulation. *Enclosures* take a different approach: (1) resource partitioning closely follows the natural static package-dependence graph of the program; (2) *enclosures* promote packages to a higher-level construct embodying a basic unit of resources programmatically manipulated by developers to be isolated or shared according to strict policies. Instead of a strict "all-or-nothing" partitioning, *enclosures* provide a way to compose packages to form the memory view exposed to a closure.

Hardware extensions: Appropriate hardware support would make it easy to isolate packages. Multics's segments [69], implemented on the vintage GE 645, provide fine-grain access control appropriate for *enclosures*. More recent work has proposed extensions to modern processor to control memory accesses within a single address space. Mondrian memory protection [193] (MMP) implements word-granularity hardware-enforced memory isolation. Its permission control granularity would be ideal for isolating program objects. IMIX [105] and MicroStache [154] extend the Intel x86 ISA with instructions to access safe memory regions. CODOMs [188] tags code pages with keys delimiting the memory resources and privileged instructions they are allowed to use. Capability Hardware Enhanced RISC Instructions [195] (CHERI) is a hardware extension that allows fine-grained compartmentalization and enforces spatial, referential, and temporal memory safety. CHERI operates at the object-level and requires a deeper understanding and instrumentation of third-party packages than *enclosures*.

As LITTERBOX exposes a stable high-level API and hides hardware details, any of these technologies could be added as a hardware-enforced isolation mechanism without requiring any changes to the application or the programming language itself. CODOMs and CHERI expressiveness makes them particularly appealing candidates. CHERI could be used as a non-page based LITTERBOX backend, which would reduce memory fragmentation or allow to

discriminate access to CPython's data and metadata while keeping them co-located (§3.6.4).

Enclosures' most similar project is Verona [115], a recently open-sourced Microsoft project that introduced a safe, infrastructure programming language. Like *enclosures*, it provides linear regions that compartmentalize legacy components by encapsulating their code, data, and dynamic allocations. The language executes unsafe components in sandboxes and looks to using CHERI in the future. Unlike *enclosure*, and to the best of our knowledge, Verona does not provide developers with a fine-grain method to composing access policies to the program's resources on a per code invocation-basis, and requires the application's main logic to be rewritten in a different language.

3.9 Chapter Conclusion

Enclosures provide a mechanism to execute untrusted packages inside a restricted environment, easily tunable by programmers, that limits access to a program's memory and its system resources. Using packages, and their transitive dependencies, as the basic unit of shareable resources results in easy-to-understand and manipulate isolation boundaries within an application.

Enclosures are language-independent and make no assumptions about the safety of the code. Instead, LITTERBOX provides support for *enclosure* policies based on hardware-based isolation mechanisms.

Our evaluation proves that *enclosures* can be efficiently added to Go, a language with a complex runtime, and provide robust isolation guarantees using both Intel VT-x and Intel MPK. Our Python implementation confirms the generality of the approach and support for dynamic languages.

3.10 Afterthoughts

Enclosures was published at the Internation Conference on Architectural Support for Programming Languages and Operating Systems 2021 (ASPLOS21) [106]. In this section, we leverage the feedback received both at the conference and during guest talks given at Microsoft Research and VMWare research to put the paper's work into perspective, expand on some of the design choices, their limitations, and the main take aways.

Enclosures ease-of-use and efficiency depends on the application's modularity. In the paper, we identified packages as being natural boundaries defining sub-compartments that can be manipulated independently and encompass a trust domain by construction. This assumption, however, might not hold in poorly designed projects, with highly connected package dependence graphs and potential circular dependencies. Such instances are rare in Go, as it usually precludes circular dependencies and most often has a lattice package dependence graph (*i.e.,*

`main` at the top and `runtime` at the bottom). Even in the presence of highly connected groups of packages, memory allocations can be extracted to a user-defined package, *e.g.,* by providing an allocator or copy API, and only require small refactoring efforts.

Enclosures overhead per backend can be hard to predict. In the paper, we showed that LB^{VTX} incurred high overheads for system calls and that LB^{MPK} was mostly affected by memory pages transfers between packages. The latter can be hard to predict, as it requires intimate knowledge of the allocation profile of included libraries, and can be impacted by the language's runtime heuristics (*e.g.,* when to trigger a garbage collection). For LB^{VTX} and system calls, the problem is much more subtle than simply looking at the `strace` of the program. First, for system calls with a low service time, the introduced overhead will be more noticeable. Second, certain system calls exhibit high service time variability, *e.g.,* lock operations can be extremely fast when contention is low, or arbitrarily long to return if they require blocking. Third, we observed with the HTTP and Fast-HTTP experiments that two programs with exactly the same `strace` can experience very different slowdowns (1.77x and 2.04x). In HTTP, the application level service time is much higher and thus hides the introduced overheads for system calls. In Fast-HTTP, user code is optimized and thus, in comparison, presents a bigger (relative) slowdown. LB^{VTX}'s overheads are thus a complex function of system call service times relative to cycles spent in user code.

Compared to secured routines, enclosures do not leverage the type system. The question of leveraging types as compartments boundaries and enforce access rights to their attributes, rather than focusing on packages, came up several times while presenting enclosures. Types would allow finer-grain control of accesses to resources and enable compiler analysis to detect isolation violations at compile time (rather than run time). This represents a data-centric approach to isolation, where we would have protected access, modification, and method invocation to structures or objects attributes. However, the goal for enclosures was to support as many languages as possible and enable interoperability between languages, including unsafe ones. As different languages do not have the same type system, and considering that all components might not be part of the same compilation unit (and thus not amenable to analysis), we quickly eliminated the option of relying on types. At the same time, I do not personally believe that types encapsulate trust domains very well. Types define the common API of data passed between function calls in a program. They therefore spread throughout the entire program and do not clearly define trust domain boundaries. For example, consider Figure 3.5: both `mux` (the web server) and `pq` (the DB driver) leverage structures representing a network socket, but should not have access to one another's structure. Allocations, on the other hand, present higher degrees of locality and are often accessed/used close to their allocation sites, with only a few of them escaping their allocation scope (*e.g.,* globals and objects returned from a function) and even fewer escaping their package.

4 Combining Secured Routines and Enclosures

4.1 Abstractions Compatibility

From a programming point of view, the secured routine and enclosure abstractions can be combined to address the heterogeneous levels of trust that exist both within an application and on the system on top of which it executes. Specifically, portions of the application that access sensitive information can easily be executed in a Trusted Execution Environment thanks to secured routines. With GOTEE's automation, public libraries, *e.g.,* image processing or ML packages, can be incorporated in the trusted code base to operate on secrets. With enclosures, a programmer can further control such public libraries' accesses to secrets and prevent leakage or unauthorized modification of sensitive data.

The code snipet 4.1 shows an example of a secured routine receiving private data from a socket and processing it inside an enclosure to extract features. The enclosure does not grant the untrusted `ML` public library access to the rest of the enclave's memory, prevents modifications to the data held in `mpkg`, and disallows system calls.

More generally, the two abstractions provide orthogonal functionalities that allow them to easily combine. Secured routines split the application's address space into two fully separated domains, trusted (unaccessible by the host) and untrusted, while enclosures enable, within each of this domains, to restrict resources accessible by a portion of the domain's code.

Unfortunately, while the proposed abstractions work well with each other at a conceptual level, the underlying technologies supporting them are more difficult to combine.

4.2 Hardware Incompatibilities

Hardware security extensions, such as Intel SGX [21], are hard to use. In secured routines, we described all the challenges that came with using Intel SGX or more generally TEEs. Our abstraction attempts to address the lack of a programming model for such technologies and exposes a clean easy-to-use language abstraction, which exhibits good performance by

```
1  ...
2  func analyzeLoop(addr net.IP, port net.Port) {
3    sock := net.Connect(addr, port)
4    ...
5    bytes := sock.Receive()
6    data := mpkg.Parse(bytes)
7    with["mpkg:R", "none"] func(d Data) {
8      ...
9      features := ML.Process(d)
10     ...
11   }(data)
12 }
13
14 func main() {
15   ...
16   gosecure analyzeLoop(net.IP{...}, net.Port{...})
17   ...
18 }
```

Listing 4.1 – A secured routine encapsulates code receiving secret data and safely processes it inside an enclosure, using an untrusted `ML` public library to extract features, without granting it the ability to modify the received data or perform system calls.

carefully avoiding the technology's hurdles. Similarly, with enclosures, we build a simple programming abstraction that abstracts away the underlying technologies (Intel MPK [124] and Intel VT-x [185]), that otherwise require low-level expertise and a non-negligible amount of configuration to be properly used. While all of these hardware technologies are challenging to use individually, they seem to be even harder to combine.

Despite the fact that SGX, MPK, and VT-x are all Intel technologies, potentially available on the same CPU, there is no clear way to conjugate two of these together. Worse, the security guarantees provided by each of these technologies do not seem to be additive. For example, consider Intel SGX and Intel MPK. While Intel MPK can be used in the untrusted domain to create sub-compartments in the application, there is no clear way to leverage the technology inside the trusted one, *i.e.,* the enclave. Specifically, as the untrusted operating system is responsible for key allocation and tagging the corresponding page table entries, enclave code cannot safely rely on MPK to create subcompartments inside the enclave. One could imagine that page table tags could be part of the enclave's measurement, and the underlying technology would produce an error in the trusted CPU if a malicious OS modifies a page table tag at run time. This solution is however incomplete, as it does not allow re-tagging page table entries after the enclave's initialization. Another approach could provide dedicated instructions to modify and verify tags for a given range of addresses inside the enclave's memory, from the trusted domain. This second approach, however, would add complexity to an already complex technology for the specific case of using Intel MPK inside the enclave, *i.e.,* Intel MPK would have to be used differently depending on the CPU's mode of execution. Similarly, Intel VT-x requires privileged code intervention to setup the underlying EPT and enclaves do not support

ring 0 code execution (even in non-root mode). A hint towards the intrinsic incompatibility of Intel SGX and VT-x in their current form is the proposal of Intel TDX [125] extension. Intel TDX provides hardware support for confidential VMs and can be seen as dedicated silicon for an execution environment combining the virtualization support of Intel VT-x, with the confidentiality and integrity of Intel SGX. If the two technologies had been easily composable, *i.e.,* if enclaves had support for non-root ring 0 code, TDX would be redundant.

4.3 Current Status and Possible Solutions

While secured routines and enclosures are compatible from a programming abstraction point of view, the underlying technologies employed to provide hardware isolation guarantees are not. This is a frustrating result that derives partially from the lack of maturity of these hardware security extensions. There is therefore hope that future iterations on the hardware will address these compatibility issues. For SGX, for example, Intel announced a version 2 of the hardware with more flexibility in the management of the enclave's address space at run time, allowing dynamic modifications of its memory layout and access rights. Unfortunately, even Intel SGX version 2 does not allow running non-root ring 0 code inside an enclave thus preventing us from combining secured routines and enclosures, even on more recent hardware.

Both secured routines and enclosures abstract the underlying hardware intricacies behind high-level programming abstractions. This allows to easily change the supporting hardware isolation mechanisms. For enclosures, LitterBox is built to provide a unified API for both supported backends. In GOTEE, a similar approach could be adopted by the `gosec` library. As a result, a potential solution to Intel hardware extensions lack of composability would be to explore solutions proposed by other hardware vendors, while leaving the programming abstractions and their semantics unchanged. Recent work on ARM platforms highlighted the benefits of rethinking the split, in terms of responsibility, between software and hardware and relying on formal verification to acquire strong levels of trust in the former. Komodo [100] implements a formally verified monitor for a TEE environment. This monitor manages the resources attributed to the secure domain, exposes functionalities such as attestation and dynamic allocation of memory to the trusted domain and could be extended with new features. The hardware, on the other hand, simply enforces the split of physical resources. Another example is Keystone [135] on RISC-V processors, which seems to provide a flexible implementation of TEEs that could be augmented with a mechanism similar to Intel MPK or VT-x. Keystone provides the notion of trusted memory management plugins that implement and expose extra features to code executing inside the TEE. One could therefore implement a mechanism that resembles Intel MPK, in a similar fashion to the enclave's page eviction mechanism described in the paper [135]. Keystone further allows both user and supervisor (privileged) code to run inside the enclave, thus allowing intra-enclave memory management and system call interposition, two of the features we exploited in Intel VT-x.

Hardware development is slow. It will be several years before processors equiped with more

flexible technologies become available at scale on commodity machines. The slow pace of hardware evolution forces us to explore alternative solutions in software able to address the software trust crisis on the current generation of machines available in the Cloud. This path is explored in our proposal called Tyche(§6).

5 The Process & Hardware Extensions

5.1 What is in a process?

The process is the favored abstraction usually synonymous with a program instance running on top of an operating system. A process encapsulates a program's resources and run time attributes, *i.e.,* the program's memory, file descriptors, and execution state. Processes enable a form of virtualization of the central processing unit (CPU) and the implementation of time-sharing [168] as well as multi-tasking. As such, a process represents a unit or group (*e.g.,* software threads) of schedulable elements as well as the default isolation mechanism in any operating system. In other words, a process is both the unit of scheduling and the unit of isolation in the system.

Looking within a process, the program's address space can be divided into two parts: (1) the user space and (2) the kernel space. A common technique consists in mapping the kernel's address space at the top of the program's virtual memory, with `super` access. A process executing in kernel mode has full access to the program's address space, while user code is limited to the subset of the address space with pages tagged as `user` accessible. A process thus defines an asymmetric memory model that embodies the commonly accepted trust model of modern systems: the kernel is trusted and has full access to the resources it manages on behalf of a user program's instance. This is represented on Figure 5.1(a) at a high conceptual level with the OS managing and accessing resources exposed to the application's instance.

5.2 Hardware security extensions

In this section, we describe, conceptually, the three hardware extensions used so far in this thesis, *i.e.,* Intel SGX [21], Intel MPK [124], and Intel VT-x [185], and how they affect the process abstraction's trust model described above.

Intel SGX: As a Trusted Execution Environment, Intel SGX [21] provides confidentiality and integrity for user code and data running inside the enclave. This is achieved, in hardware, by

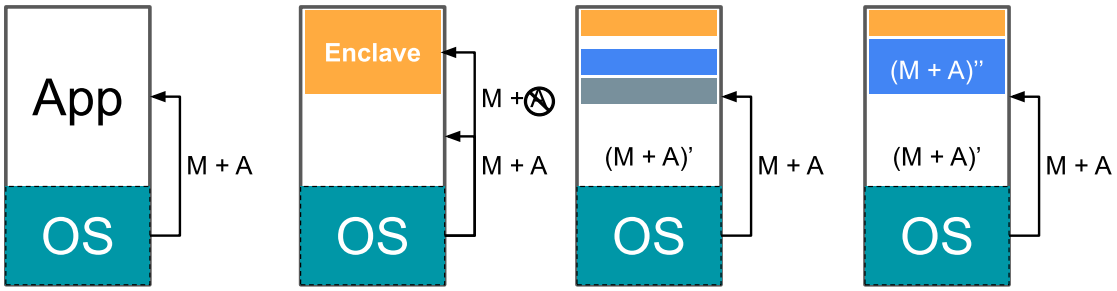

Figure 5.1 – From left to right: (a) Process, (b) SGX, (c) MPK, (d) VT-x. Each figure represents the program's address space, including the underlying operating system (OS). The OS manages (M) and has access (A) to the application's memory resources.

encrypting, integrity protecting, and ignoring unallowed reads and writes to enclave memory from code running in non-enclave mode, including the OS. However, enclave memory still requires the proper virtual mappings to be installed by the OS in the process' page tables.

Conceptually, Intel SGX is a hardware mechanism that provides support to decouple memory management from accesses within the process abstraction. Specifically, SGX still relies on the OS to manage the enclave's virtual memory (including the handling of page faults), but prevents the OS from accessing the corresponding underlying physical memory pages. This is represented in Figure 5.1(b). Furthermore, Intel SGX subdivides the user space into two domains, *i.e.,* (protected) trusted memory and (unprotected) untrusted one. Similar to the user/kernel space split, it is an asymmetrical memory model that mirrors the trust relationship between the two domains: untrusted code only has access to untrusted memory, while the enclave can access both domains' memory.

Intel MPK: Intel MPK [124] allows user code to create sub-compartments inside the application's address space with different access rights to tagged memory regions.

Conceptually, Intel MPK is a hardware security extension that introduces a mechanism enabling a form of memory access right management performed mostly in user-space. User code decides which memory regions should be tagged with a given MPK key and has access to the PKRU register allowing to switch between the defined intra-address space compartments with different memory access rights. Intel MPK is thus a mechanism that delegates a form of resource access-right management to user code running inside a process. It is worth noting that Intel MPK only allows to reduce access rights installed by the OS and still relies on the OS to tag page table entries. It thus does not modify the default process trust model between user and kernel spaces. Intel MPK is represented in Figure 5.1(c).

Intel VT-x: Intel VT-x [185] differs from the two previous extensions as it was marketed through a performance argument rather than as a security extension. Intel VT-x provides hardware support for extended page tables (EPTs), *i.e.,* host virtual to guest physical mappings, in hardware, as well as exposing privileged instructions and registers, *e.g.,* `CR3`, to guest operating

systems. Among the three Intel extensions presented in this thesis, Intel VT-x is the oldest, the most successful one in terms of adoption, and the only one to primarly target a performance improvement.

Conceptually, however, Intel VT-x shares some of Intel MPK's properties. It is a mechanism that allows to delegate a form of management for a subset of the program's resources to user code. This user code is then able to expose and manage resources for a guest operating system that in turn manages and exposes resources to a guest program. Also similar to Intel MPK, Intel VT-x does not break the trust model between user and kernel space, *i.e.,* guest page table access rights can only be equal or a subset of the host's access rights. Figure 5.1(d) presents these delegations of management.

Take-aways: All three technologies pertain to intra-address space isolation, *i.e.,* they introduce a flexibility in the management of resources inside a process. More specifically, they all enable to create sub-compartments inside the process address space to execute certain components of the application with differentiated access to the program's resources. We identify two other operations performed by these extensions: (1) decoupling management from the ability to access the managed resources (SGX), and (2) delegating management of resources to user code.

5.3 A Paradoxical Paradigm

Efficient intra-address space isolation mechanisms are required to address the trust crisis faced by modern software. Any application can be viewed as a collection of modules, components, from disparate provenance and heterogeneous levels of trust, that need to have restricted access to the program's resources. In Section 5.2, we highlighted the conceptual contributions of hardware technologies that provide intra-address space isolation: (1) decoupling management from accesses, (2) delegating management of resources to entities that are not the OS. However, as explained in 4.2, these extensions are not widely adopted, hard to leverage, and not amenable to composition. They further add complexity to already hard-to-model architectures, riddled with side-channels [76, 128, 129, 196] that endager even the proposed security guarantees. A more viable solution to addressing modern software isolation needs should rely on simpler primitives, easier to reason about, and able to resist side channel attacks.

Modern hardware security extensions are an attempt at introducing more flexibility in the process abstraction, breaking its antic and rigid trust model by introducing new isolation primitives, and reducing the OS prerogatives. It is quite paradoxical to rely on hardware, which is slow to develop and hard to change, to introduce flexibility in a software abstraction, *i.e.,* the most malleable element of our execution stack. A more reasonable approach would reconsider the limitations of the process abstraction, central to current operating system designs, and propose alternative abstractions before burning new complex extensions in silicon.

Previous work explored the limitation of the process abstraction, proposed alternative designs to decouple its role as a schedulable unit from memory isolation [71, 142], and introduced intra-address space isolation. Other solutions [123, 147, 156] took a language approach, relying on programming languages features such as type systems to implement intra-address space isolation and enforce high-level program-wide security policies. An obvious alternative to hardware security extensions would thus revisit operating systems to deconstruct the monolithic process abstraction and provide new lighter-weight primitives. Such abstractions need to provide intra-address space isolation, *i.e.,* the ability to create software compartments inside a program's instance, with differentiated access to the program's resources and low transition overheads. A second design objective for new OS abstractions requires an efficient mechanism to safely delegate the management of resources to entities that are not the operating system, while preserving the OS' ability to revoke/reclaim resources (akin to what was proposed in the Exokernel design [96, 126]). This, in particular, would allow to replace technologies such as Intel MPK or Intel VT-x. Finally, due to the high heterogeneity of trust in applications and the systems they execute on, new operating system designs should strive to decouple the management of resources from the ability to access them. Ideally, we would like the operating system to retain its ability to allocate, manage, and revoke resources from applications, while guaranteeing it will not be able to access, modify, or leak sensitive information. This is a challenging problem that can be approached in various ways, both in hardware and software. As hardware is slow to evolve, we take the software route and highlight a potential approach based on a software layer of indirection.

5.4 A Software Layer of Indirection

Introducing a level of indirection is system engineers' oldest and most abused trick in the book [1]. Throughout this thesis, we relied on hardware security extensions as the root isolation mechanism, due to what we call the *native execution assumption.* Secured routines and enclosures assume that application code executes natively and implements arbitrary behaviors, *e.g.,* raw memory accesses and invokes any instruction. As a result, hardware isolation enforcement was the only viable solution. But what if, instead, we preclude the execution of native code? This is far from being a novel idea, it is as old as language virtual machines [139, 166, 171], was used in related work to secure and optimize full systems [123, 147], and is already leveraged to safely download, validate, and execute code inside the kernel (*e.g.,* eBPF [51, 150]). Getting rid of the *native execution* assumption by executing common software on top of a virtual ISA would allow to (1) build security and isolation guarantees directly into the ISA, (2) provide the flexibility required to explore new isolation primitives, and (3) be compatible with legacy code and hardware platforms.

Recent projects, such as Webassembly [117] (Wasm), propose a basic specification for a virtual ISA with a focus on security. Wasm differs from language virtual machines, such as the JVM [139]: it does not provide complex high-level abstractions or features (*e.g.,* Object

[1] According to Dr. Dave Eckhardt, my OS professor from CMU

Oriented support, Garbage Collection) reflecting a specific PL's properties, but rather focuses on a simple abstraction of an instruction set. Many languages, including C/C++, Java, Go, and Rust [6] support Wasm as a compilation target.

In the next chapter, we propose to use virtual machines as the basic unit of isolation to replace both inter and intra address spaces. Rather than proposing yet another virtual ISA, we focus on the specification of an execution environment that presides over the scheduling of multiple VMs, mediates their interactions, and allows to independently extend and optimize the execution environment and each VM's internal implementation. Isolation is enforced by each VM as they encapsulate the set of accessible resources of a trust domain and only allow interactions with the outside world via controlled mediated channels provided by our runtime. The runtime itself focuses on: (1) initializing VMs and mapping their resources, (2) scheduling units of execution, and (3) mediating VM interactions.

6 Tyche: Software is the New Hardware

Disclaimer: *This Chapter describes an ongoing research project in collaboration with Dr. Marios Kogias and for which we do not have a full prototype yet.*

The recent proliferation of hardware security extensions such as Intel SGX [21], TDX [125], MPK [124], ARM Trustzone [30], and AMD SEV [39], addresses the primary challenge facing modern software: trust. Unfortunately, so far, adoption has been slow. These security extensions are constrained by ISA backward compatibility, are hard to leverage, slow to evolve, do not compose well, and lack clear high-level programming models.

In Chapter 5, we suggested that each of these security extensions modifies, in hardware, the decades-old trust model exposed by the process abstraction to introduce intra-address space isolation guarantees and breaks the assumption of a trusted host imbued with unrestricted access to user code and data. We argue, however, that these technologies are still at an early stage and too inflexible to fully solve the problem of trust. More specifically, we believe that trust is a problem that requires high-level semantics and complex specification of acceptable behavior that are simply not expressible with the current hardware mechanisms.

Here, we advocate for a different approach: as we still lack a clear programming model to address the problem of trust faced by modern software, solutions and isolation primitives should first be explored in a malleable, easily extensible execution environment that allows the creation and composition of new isolation guarantees. We propose Tyche, a software execution environment that composes trust domains running in separate (language) virtual machines and connects them via a trusted intermediary. We demonstrate how Tyche emulates and composes existing intra-address space hardware isolation mechanisms and sketch out how a Wasm [117] embedder provide a base for the implementation of our proposed execution environment.

We believe that being able to easily extend and modify Tyche will enable a rapid exploration of isolation solutions, permit experimentation with various programming models, and help converge on a well-defined set of requirements and primitives that hardware could use to

address the modern problem of trust.

6.1 Introduction

Hardware vendors have begun offering security features such as Intel MPK [124, 163], SGX [21], ARM Trustzone [30], memory domains [62], and AMD's SEV [39]. These extensions are partial solutions to the primary challenge facing modern software: *trust*. This problem assumes many forms, but its solutions typically require isolating and reasoning about relationships among software components, a capability missing from conventional processors. Components – ranging from libraries in a single address space, processes in different address spaces, or complete virtual machines – need isolation from other components or even from the underlying host. Hardware offers different mechanisms to isolate these system components.

Researchers enthusiastically welcomed these hardware extensions, which yielded a plethora of publications [67, 68, 86, 88, 104, 105, 107, 109, 118, 131, 135, 138, 151, 155, 161–163, 175, 179, 186].[1] These papers applied these extensions in hypervisors [68, 151, 155, 175, 179], library operating systems [67, 68, 118], programming languages [104, 107, 109, 138, 162], and browsers [109].

Many of these hardware extensions were immature and (unavoidably) not widely tested or applied, and quickly exposed their limitations. Researchers had many opportunities to demonstrate imagination and creativity in formulating ways to circumvent or correct some of these flaws [107, 118, 161–163, 186]. Unfortunately, side-channel attacks [76, 128, 129] undercut the promised isolation by exposing serious flaws in the processors' micro-architecture which rendered the extension's guarantees, at best, questionable. Paradoxically, Software mitigations [184], with non-negligible performance slowdowns, were necessary to ensure the isolation of hardware security extensions, leaving many wondering about the usefulness of the extensions in the first place.

Considered broadly, recent hardware security extensions, in their current state, are unable to become widely-used standards in our development and deployment stacks. Trusted Execution Environments, for example, are seldomly used in practice despite being available in the Cloud [167], and their programming model is still evolving. After years spent focusing on providing a TEE at a user-level [21], Intel has recently shifted towards confidential VMS [125], that encapsulate full VMs rather than just carefully selected portions of application-level code. Technologies such as Intel MPK [124] are only available on certain high-end microprocessors and, for the moment, cannot be combined with Intel's TEE. At the same time, while virtualization is at the heart of modern software deployment in the Cloud and provides the basic unit of isolation among tenants, most security extensions on commodity hardware fail to properly integrate with virtualization technologies. In Chapter 4, we described our inability to combine Intel MPK or Intel SGX with Intel VT-x, thus preventing us from composing secured routines and enclosures. This incompatibility illustrates an important problem: despite being

[1] Software developers presumably were less thrilled by another complex ISA feature.

aware of the problem of trust in modern software, we are still unsure about the appropriate programming model and primitives to address it.

Moreover, these complex architectural features incur considerable cost. They add complex, poorly specified semantics on top of already overloaded ISAs, are inflexible and limited in functionality, incur a long-term burden of backward compatibility, and interoperate in unpredictable ways. Recent work [65] proposed that CPU vendors could deliver new features as microcode updates or provide a new privilege level sitting beneath the OS (similar to RISC-V machine mode [135]) to increase flexibility and mitigate *current* hardware limitations.

We observe that existing hardware security extensions continue the *native ISA assumption*, that a processor executes an arbitrary sequence of instructions, without a higher-level specification of acceptable or unacceptable behavior. The few exceptions in most processors are low-level errors, such as divide by zero. Trust extensions attempt to raise the semantic level by coarsely prohibiting some instruction behaviors; for example, code running in an enclave should, in general, not interact with code running outside. Unfortunately, the semantics of the restriction are typically described only in terms of a processor implementation, which is heavily constrained by compatibility, and consequently may not meet software requirements.

We consider a different approach. We believe that processor implementation is the wrong level of the system stack to experiment and evolve new features and mechanisms. It is difficult and time-consuming to implement, extend, and validate new ideas in silicon. Software, by contrast, is easily malleable and capable of prototyping robust mechanisms with acceptable performance. Widespread experimentation with software-implemented security mechanisms can expose opportunities for hardware performance enhancement, which will properly drive the design of new hardware features.

Several recent trends support the direction of our proposal. First, datacenter networking has followed a similar path with great success, where switching is now largely defined in software [132] and endpoints implement low-latency protocols in userspace without support from NIC vendors [149]. Second, Apple's emulation layer (with some hardware-support for strong memory consistency) for their M1 CPU demonstrates that x86 code ran as a virtual ISA can equal or outperform native code [119]. Finally, WebAssembly [117] (*Wasm*) is a newly introduced, virtual ISA aiming to provide verified isolation with acceptable performance overheads.

In this Chapter, we revisit the problem of isolating mutually distrustful components that make up an application (§3). This time, rather than providing a single programming abstraction, we focus on the design of a flexible and extensible execution environment that allows to explore compartmentalization strategies, understand the isolation requirements of modern applications, and derive secure interaction mechanisms between trust domains. Specifically, we aim at providing an execution environment that satisfies the following goals:

- Provide strong, fine-grained intra-address space isolation primitives.

- Emulate and compose existing hardware security hardware support.
- Support execution on heterogeneous hardware platforms without requiring specific extensions.
- Be easily extensible via software updates.

We believe that this execution environment would enable us to explore compiler and language-driven approaches to automatically instrument applications with strong isolation boundaries. We also hope that this approach will identify the basic mechanisms and abstractions desirable from hardware. Taken broadly, our approach in this Chapter works in reverse compared to §2 and §3: rather than providing a programming model for existing hardware security extensions, this chapter defines an execution environment that can help identify better hardware extensions.

Tyche is a software execution framework for user applications. It relies on virtual machines as the primary unit of isolation among mutually distrustful domains. The virtual machines run a virtual ISA and define *execution modules* that implement isolation guarantees even for software components written in unsafe languages and running in a shared address space. Tyche provides a *trusted intermediary* [169] that enforces modularity by managing and scheduling the different modules and mediating the interactions among them.

Tyche does not rely on specific hardware features and can be implemented and deployed on commodity hardware. However, Tyche, as all virtual ISA emulators, would benefit from proven and well-known architectural features such as segments and call gates [168], which ironically were both dropped during the x86 architectural transition from 32 to 64-bit and the introduction of virtualization support [75, 102, 185, 197].

Tyche does not require changes to programs' source code. A program is compiled to a virtual ISA to run inside one or several execution modules, *i.e.,* virtual machines with the appropriate properties. We sketch an implementation of Tyche-support for a Wasm embedder. Wasm provides a virtual ISA with appropriate security features, making it a very attractive compilation-target to replace native execution of modules written in unsafe languages. We show that Tyche can emulate existing ISA isolation abstractions and compose them while providing a flexible testbed to explore new isolation primitives.

Philosophically, Tyche takes an approach similar to Singularity [123], *i.e.,* it is an experimation to explore new designs ideas, not a solution in itself. Specifically, Tyche's goals span across both software and hardware as we hope to identify (1) techniques to automatically partition applications built from heterogeneous untrusted software components, (2) understand how to safely allow interactions and pass information between separate trust domains, (3) explore new trust models, and (4) derive desirable hardware primitives to support such isolation schemes without hurting performance. The goal of this research is, however, not to burn Tyche as is into silicon, but rather to understand the set of guarantees better addressed in hardware

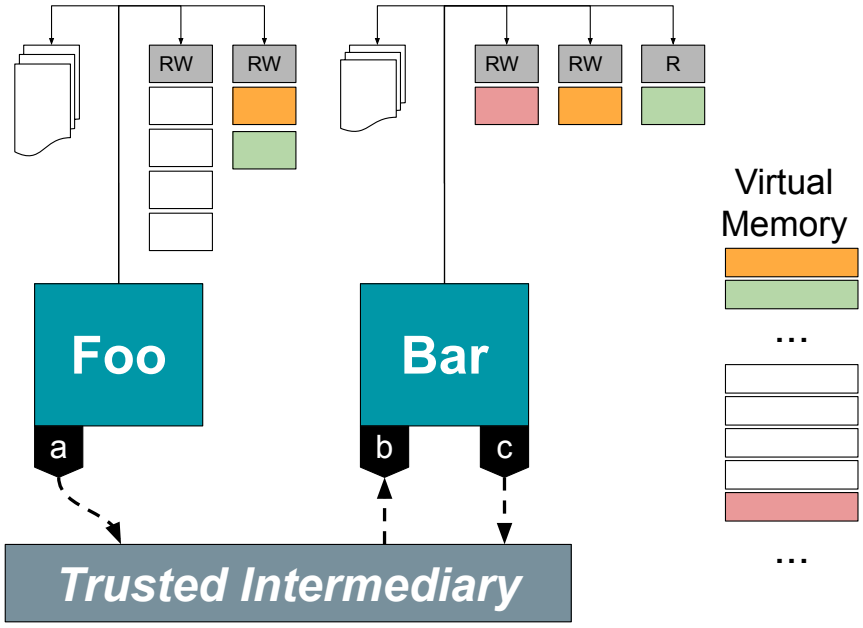

Figure 6.1 – Tyche Overview: user application with modules `Foo` and `Bar` running inside a single process address space in Tyche, managed by the trusted intermediary. Each module encapsulates a trust domain, with access to code, data, and gates (a)(b)(c).The right-side depicts the actual process virtual memory layout of all the program's segments.

or operations that could be sped up with the appropriate silicon support.

6.2 Tyche

Tyche relies on three simple elements, derived from basic systems design principles [169]: (1) execution modules, (2) gates, and (3) a trusted intermediary. This section describes each of the elements and the guarantees they must provide. It then sketches how WebAssembly [117] (Wasm) is used to secure modules written in unsafe languages.

6.2.1 Execution module

An execution module declares pre-defined access rights to a trust domain's available resources and ensures they are enforced for the code in the module. A trust domain incarnates a single unit of trust, usually corresponding to a single principal. The set of resources available to a trust domain consists of three categories: (1) code, (2) data, and (3) gates. Code is the set of functions that belong and can run inside the domain. The data is the set of readable (and

optionally writable) memory accessible by code running inside the domain. Note that the intersection of different domains' resources can be non-empty since the same data can be shared across domains. Gates control interactions with other trust domains. We distinguish between imported gates, that permit requests of functionality from foreign domains, and exported gates, that permit other domains to invoke a functionality in the local trust domain.

The execution module is responsible for ensuring a trust domain's encapsulation. It is the sandboxing mechanism that exposes and restricts a domain's access to its own resources. It also ensures that gates are the sole mean of explicit interaction with other domains.

In Tyche, execution modules are (language) virtual machines [46, 117] that ensure the desired isolation. Apart from that requirement and the compatibility with Tyche's execution model, the virtual machine can implement any virtual ISA or bytecode. The choice to enforce isolation and restricted access to resources in software is a trade-off that sacrifices performance, compared to native code execution restricted via hardware, for flexibility in the granularity at which guarantees are provided and extensibility.

6.2.2 Gates

Gates allow explicit interactions with resources outside of a trust domain. They are typed interfaces that restrict control transfers between domains. A gate specifies the name and type signatures of a function that a domain exports as well the form of control transfer it accepts. The type defines the memory layout of the data that is transferred across the domains when the gate is invoked. We distinguish between exported gates and imported ones. A domain exports gates to selectively allow other domains access to the functionality it implements. A domain imports gates to invoke functionality implemented by other domains. In terms of semantics, the invocation of gate triggers a synchronous or asynchronous control transfer into the exporting domain and specifies how typed arguments are transferred across the domains boundaries (*e.g.,* value or reference passing). We also assume that both the caller and the callee domain of a gate can uniquely identify the other party.

6.2.3 Trusted Intermediary

The trusted intermediary is the privileged part of Tyche relied upon by all domains to (1) correctly instantiate the execution modules, (2) bind exported and imported gates, and (3) enforce their semantics. The trusted intermediary is further trusted not to expose one domain's resources to another without the owner's consent. More generally, it should not access, modify, or leak a trust domain's resources.

The trusted intermediary is a mediator between mutually distrustful entities. When a module invokes an imported gate, it yields control to the trusted intermediary. The intermediary validates the request, selects the module that exports the appropriate gate, and performs the control transfer to execute the function in the other domain.

The trusted intermediary manages and schedules user-level threads which are the unit of execution and concurrency in Tyche. These user-level threads flow across virtual machines via gates. The intermediary cooperates with the execution module to keep track of a thread's state within a virtual machine, save it upon the invocation of a synchronous gate, and restore it upon return.

Figure 6.1 depicts a user program in Tyche, running in a single-process address space. Module `Foo` invokes the imported gate (a) to synchronously transfer control to `Bar`'s exported function (b). Since it is a synchronous call, the trusted intermediary deschedules `Foo` and schedules `Bar` to run `Bar`'s function exported via (b). `Foo`'s second segment is shared with `Bar` as two separate segments, with read-write (orange) and read-only (green) access rights.

6.2.4 Tyche & Wasm

In Tyche, we plan to use the Wasm ecosystem to support the safe execution of modules written in unsafe languages (*e.g.,* C), languages for which no virtual machine with the desired guarantees exists, or binaries for which no source code is available.

Wasm Overview: WebAssembly (Wasm) [117] is a binary instruction format for a stack-based virtual machine meant to be simple and compact. Most popular languages compilers support Wasm as a compilation target [6]. Wasm comprises a *core specification* as well as *embedding interfaces*. The core specification [31] is a document that defines the semantics of WebAssembly independent of its embedding, *i.e.,* the environment in which Wasm is leveraged. Embeddings are consensual interfaces and APIs that specify how to make Wasm interact with various environments, including JavaScript [32], Web browsers [34], and system call interfaces [33].

The Wasm ISA relies on segmentation to represent mutable memory resources, *i.e.,* contiguous arrays called *linear memories* accessed via typed load/store instructions with a tuple `(linear memory id, offset)`. The Wasm format supports external and imported types (`externtype`) or references to functions (`externref`). Host functions can thus be embedded into a WebAssembly module to allow interactions with the surrounding environment or other module instances.

A Wasm module is the basic unit of code deployment. It defines the module's functions, tables, memories, globals, imports, exports, names, types, as well as initialization data. At run time, a module is instantiated in an embedder, *i.e.,* an execution environment, validated, initialized, and executed.

Adapting a Wasm embedder to Tyche : We are currently extending the Gasm [198] embedder to be compatible with Tyche's execution model. Gasm is a simple virtual machine that runs Wasm modules to completion. Our current implementation extends the embedder to support multiple user-level threads executing inside a single module and allows Tyche's trusted intermediary to preempt their execution. The implementation also relies on `externref` exposed to

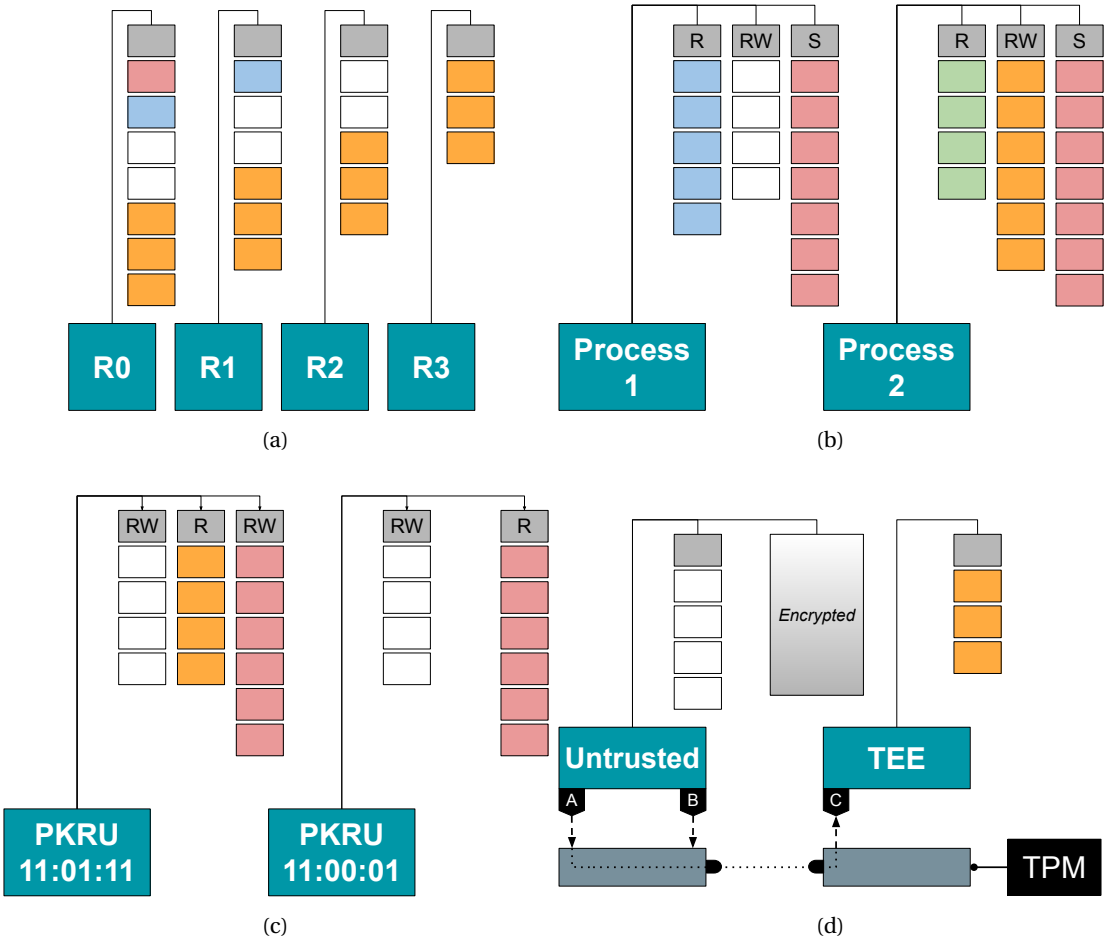

Figure 6.2 – (a) Rings (b) Processes (c) MPK (d) TEE

the module to implement gates. The current prototype executes multiple user-level threads per virtual machine and handles transitions via gates between execution modules.

6.3 Emulating & Composing Hardware Isolation

This section demonstrates that Tyche can emulate the most common isolation patterns provided by hardware and security extensions. Figure 6.2 summarizes these implementations. To demonstrate Tyche's ability to compose such abstractions, §6.3.3 sketches how to implement secured routines and enclosures in Tyche and how to compose them.

6.3.1 Common abstractions

Rings: Ring-based isolation are nested trust domains (4 in the x86) that limit code running in a ring to access only a subset of the memory regions and the instructions available in lower

rings.

Tyche can implement an arbitrary number of rings. The memory resources are accessible segments and privileged instructions are gates. Nesting rings therefore: 1) provide access to a subset of the most privileged ring's memory segment and 2) limit the gates exposed at each ring level.

Figure 6.2 (a) depicts Tyche implementation of 4 nested rings. Ring 0 has full access to the segment, while each subsequent ring is only allowed access to a subset of it. More complicated schemes, involving multiple memory segments or gaps in the address space, *e.g.*, making the green region unavailable to ring 1, are also possible.

Processes: A process is an instance of a program execution scheduled by the operating system. A process encapsulates the program's code, data, execution state, and access to system resources (*e.g.*, file-system, network). Inter-process communication (IPC) occurs through shared memory regions or IO such as pipes, files, or network messages.

Tyche's execution modules provide an abstraction similar to processes. They encapsulate a domain's code, data, and execution state and can be independently scheduled on different threads. Access to system resources, pipes, files and network occurs through gates, which can easily mimic IPCs. Figure 6.2 (b) shows a simple example that mimics two processes with read-only (R) and read-write (RW) segments and a shared (S) third memory region for IPCs.

6.3.2 Hardware Extensions

Virtualization: Virtualization, as defined by Popek and Goldberg [165], relies on a virtual machine monitor (*VMM*) to expose hardware resources to a guest virtual machine (*VM*). It ensures that a program running inside a VM behaves as it would on a physical machine (*equivalence*), that the VMM has complete control of resources (*resource control*), and that most of the guest program runs without VMM intervention (*efficiency*).

Hardware extensions, such as Intel VT-x, extend ISAs with mechanisms to facilitate and accelerate the execution of virtual machines, notably by providing extended-page-tables (EPT) and virtualizing privileged instructions and registers (*e.g.*, `CR3`) to safely expose them to guest operating systems.

At its core, virtualization reduces to nested ring isolation. One could run the host kernel in ring 0, the host user in ring 1, the guest kernel in ring 2, and the guest user in ring 3. As described in §6.3.1, Tyche supports any number of nested rings and thus can provide virtualization and, similar to Fluke [103], nested virtualization isolation. Equivalence and efficiency properties are guaranteed by the execution module virtual machine's implementation. Resource control is either performed directly by the trusted intermediary, or exposed to a module via gates.

Intel MPK: Intel Memory Protection Keys (MPK) tags page table entries with one of 16 possible

keys. A user-writable and readable register, PKRU, uses two bits per key (32 bits total) to encode memory access rights (write|read) enforced by hardware. In essence, Intel MPK restricts access to memory resources based on the current execution context, encoded in PKRU. The clear limitations of Intel MPK are: 1) the limited number of keys, and 2) that user code can modify PKRU and grant itself access to a tagged memory region.

Tyche can mimic Intel MPK by sharing segments with appropriate access rights with a trust domain and dynamically changing the set of available segments and access permissions at run time. Figure 6.2 (c) simulates a PKRU register with 3 keys and selectively enable read and write accesses to the tagged segments for different modules. Tyche can improve upon Intel MPK by supporting any number of keys and can rely on gates to control whether a domain can modify its access rights.

Trusted Execution Environments: Trusted Execution Environments (TEEs), such as Intel SGX [21], ARM Trustzone [30], and AMD SEV [39], provide confidentiality and integrity of user code and data deployed on potentially compromised Cloud servers (see §1.5).

TEEs split a machine's resources between two mutually distrustful domains each with their own CPU (*e.g.,* enclave or non-enclave mode) and memory (*e.g.,* Enclave Page Cache). The same model can be implemented by physically splitting a machine's resources, *e.g.,* by relying on a trusted co-processor private to the trusted application, with its own tamper-proof memory resources, inaccessible from CPUs executing untrusted code (*i.e.,* the OS).

As described in §6.3.1, Tyche can implement full isolation between two domains, and we could imagine running untrusted parts of an OS in Tyche (similar to work on trusted hypervisors [90]). A more literal approach to resource partitioning could run trusted code on separate hardware, far from the OS's reach. Inspired by work on GPUs [189], and with Tyche exposing system resources as gates, trusted code could run directly on a dedicated co-processing unit with non CPU-accessible on-chip memory and no OS, *e.g.,* an FPGA. Untrusted code runs in Tyche on the server's CPU and OS, and uses gate A-C to load encrypted code to a Tyche runtime flashed on the FPGA, able to decrypt and execute trusted code, and uses gate B to manage it's resources without accessing them.

6.3.3 Composing & Nesting Abstractions

To illustrate how various isolation abstractions can be combined in Tyche, we focus on secured routines and enclosures implementation and combination in our execution environment. Implementing support for secured routines and enclosures in Tyche is straight forward.

For secured routines [107], trusted code and data are deployed in the secured environment as described above. Cross domain channels are implemented via gates with copy semantics. Attestation can be performed by measuring the module's initial state (*e.g.,* the Wasm module corresponding to trusted code and data) and relying on the attached TPM to sign the

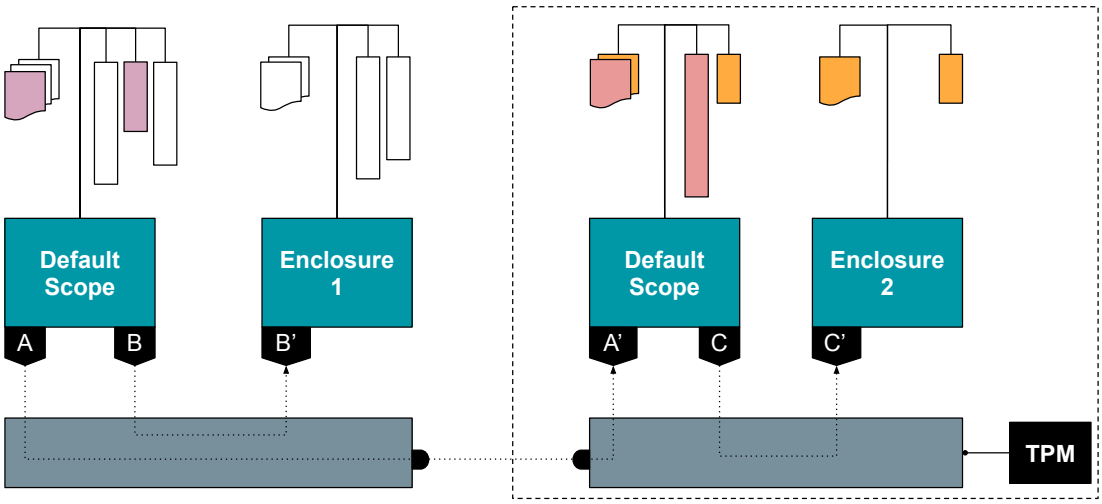

Figure 6.3 – Secured routines leveraging an enclosure to execute an untrusted library safely inside the TEE. The dashed rectangle corresponds to the TEE.

measurement.

Enclosures [106] support is straight forward as well: for each execution scope defined inside the application (*i.e.,* default scope and each enclosure scope), we create a different execution module. An execution module's resources is the collection of segments and associated access rights that correspond to the meta-packages accessible by the scope. The actual memory backing up these segments is shared across execution modules while access rights are enforced by the VM itself. For system call filtering, we install (*i.e.,* import) the allowed gates that expose the authorized underlying system resources, or simply filter syscalls at the callee module. A switch between scopes is implemented through synchronous gates, *i.e.,* an execution module imports a gate that corresponds to the entry point to another scope (*i.e.,* an enclosure's closure). In terms of actual implementation on top of Wasm, the main challenge would be to implement compiler support for multiple linear memories. We however see an easy path towards a solution. In Go, at link time, our Go enclosure frontend extension segregates symbols per meta-packages by assigning a start address and a length for each package section, *i.e.,* data, rodata, code, and heap (see §3.5.1). An obvious solution is thus to simply replace the start address with a linear memory id at that stage, and use it during Wasm byte code generation in place of the virtual address.

Combining secured routines and enclosures is trivial, as it simply entails having multiple modules in each of the completely isolated high-level trust domains (trusted and untrusted worlds). Figure 6.3 shows the combination of secured routines and enclosures. The untrusted application executing on the left side of the Figure uses gate `A` to trigger the execution of a secured routine inside the TEE (via the exported gate `A'`). As per the description of secured routines, no memory segment is shared between the two domains. Both the untrusted and trusted domains leverage enclosures, via gates `B` and `C` respectively, to isolate portions of the

memory resources when invoking public libraries.

6.4 Beyond Hardware

Tyche's flexible design enables the implementation and exploration of more ambitious isolation schemes than the ones constrained by hardware mechanisms.

6.4.1 Securing Applications at Compile/Link time

Tyche can be used to implement safe language inter-operability. Modules running in separate virtual machines might be written in different languages, providing different safety properties internally. Interactions between these modules are, however, mediated by Tyche and follow strict common APIs, while their execution is encapsulated by the surrounding virtual machine. This property could be used, for example, in GOTEE [107], to provide safe support for C libraries. In GOTEE, we precluded C interactions as we heavily relied on the strong typing of the Go programming language (*e.g.,* for cross-domain channels), and disallowed unsafe memory accesses (*e.g.,* raw pointers) to prevent cross-domain memory references. With Tyche's execution environment, the former is required only at the interfaces between languages (gates), while the latter is enforced by the virtual machine itself (*e.g.,* by executing C code in a Wasm embedder).

Another use-case for Tyche is the exploration of compiler and linker techniques that would automatically partition applications at packages boundaries and run them in separate execution modules. In §3, the programmer is still responsible for identifying and enclosing unsafe parts of the application, which can be cumbersome. With Tyche, we could study the feasibility of automating this process and experiment with the granularity at which data is made available from one package to another (*e.g.,* share single objects rather than a package's full heap).

6.4.2 Tyche Distributed Deployments

Tyche can be deployed as a distributed system on heterogeneous hardware setups with several compute units. As explained at the end of §6.3.2, and similar to Swift [87], Tyche can transparently distribute an application's components on different compute units (*e.g.,* a CPU and an FPGA). The trusted intermediary is responsible for transparent remote communications through gates, *e.g.,* by use of an efficient RPC protocol [130]. We could also envision a more aggressive form of distributed setup, where remote instances of Tyche's trusted intermediary cooperate, via network communications, to schedule execution modules transparently.

6.4.3 Tuning Semantics & Inter-operability

In Tyche, execution modules' virtual machine implementations can differ as long as they guarantee the required isolation properties and respect gates types. Each virtual machine can be independently tuned to use different primitives or abstractions internally. More specifically, this allows the behavior of the virtualized ISA to vary between modules, be fine-tuned to the specific VM's requirements, and selectively enable extra features on a per-VM basis. For example, unlike Intel CET, shadow stacks can be selectively enabled in VMs that do not provide control-flow-integrity, thus allowing paying the associated cost only in domains where these security mechanisms are required.

6.4.4 Granularity & Revocation

Tyche does not impose alignment restrictions on the segments' start addresses or sizes. In the limit, this allows, in Wasm, allocating an object in its own linear memory and, similar to Mondrian memory [193], to provide object-granularity access rights enforcement. Support for object-granularity isolation enables the implementation of ownership semantics across execution modules. Gates can transfer an object's ownership from one execution module to another and enforce mutual exclusion on said object.

Tyche can expose resource management and revocation safely across modules. Appropriate gates allow a module to transfer sensitive information into an untrusted module solely for the duration of a computation and prevent leaks of the information, for example by restricting the module's access to the network. The untrusted module could then be destroyed upon return, preventing it from storing and later leaking the information. Other schemes can be envisioned, such as relying on real-time timers to determine how long a given resource should be shared with a concurrently executing module.

6.5 Summary

Hardware manufacturers have a long track record of successfully providing performance improvements. However, they are yet to prove that they can correctly build support for flexible software isolation primitives. In this Chapter we proposed to give up on native ISA execution of user applications to provide solutions to the general problem of trust exhibited by software. Tyche builds strong isolation guarantees as primitives in the virtual machines running user code and relies on a trusted intermediary to safely enable module interactions and expose system resources.

7 Future Directions

7.1 Opportunities With New Programming Languages & Hardware

In this thesis we exploited the synergy that exists between certain programming languages abstractions, *e.g.,* user-level threading, typed channels, packages, and hardware security extensions to address the heterogeneous levels of trust that exist within our applications and systems. Recent advances in both programming languages and hardware features could bring interesting new opportunities to design mechanisms to secure our software stack.

Verona [115] is a new safe infrastructure programming language. Verona schedules behaviors, *i.e.,* pieces of code, on named memory regions. Code executing in a behavior can only access memory belonging to the acquired regions. One end goal for Verona is to allow the sandboxed execution of legacy software components, wrapped in a Verona-compatible layer. How this will be enforced, whether it will involve specific hardware mechanisms or rely on a safe virtual ISA (*e.g.,* Wasm [117]), is still open. Another interesting problem to consider is whether techniques we want to explore in Tyche to automatically partition legacy software into trust domains could also be applied to safely run applications in the model proposed by Verona.

In terms of hardware extensions, CHERI [195], a hardware capability implementation, presents interesting features to encapsulate unsafe software components [93, 101, 192, 194]. We already identified CHERI as a promising backend for enclosures (§3.7) that would provide control at a finer granularity than page-based hardware mechanisms and thus reduce the memory fragmentation in our implementation. CHERI also seems promising in the context of Tyche. It could be used as a segmentation mechanism, *i.e.,* CHERI pointers could replace the Wasm embedder linear memories, with access rights enforcement implemented by the hardware. This would allow to generate efficient native code whose memory accesses are expressed as an offset of CHERI pointers, in an approach closer to the one proposed in NaCl [197].

7.2 Mechanisms to Safely Expose Management of Resources

In this thesis we studied the problem of guaranteeing the confidentiality and integrity of user code and data deployed on untrusted machines. Generally, this problem is a consequence of the popularity of Cloud deployments and the increased concern towards user privacy. We are, however, not convinced that new hardware extensions are required for this, or at least, not in the form proposed by TEEs.

In Chapter 5, we identified the separation of resource management from the ability to access them as a desirable feature. The problem of trust in Cloud deployment includes two important challenges: (1) preventing a curious hypervisor from accessing a user's resources, and (2) providing a guarantee to clients that their code and data is properly loaded and protected. Both of these challenges must be addressed while preserving the hypervisor's own requirements: (1) the ability to effect management decisions, and (2) protecting itself from untrusted user code.

We advocate that responsibilities and privileges should be redistributed among the entities involved in Cloud deployments, *e.g.,* the hypervisor and the guest virtual machines, to provide stronger guarantees to Cloud users. We believe this is achievable in software, without new (complex) hardware mechanisms. We draw inspiration from related work [59, 60, 66, 90, 96] to devise a trusted monitor acting as an intermediary between hypervisors and virtual machines.

Via techniques similar to Virtual Ghost [90] or by de-privileging the hypervisor, we would reserve the ability to access hardware configuration (*e.g.,* EPT or page table mappings) to a trusted monitor. The monitor would be a (minimal) trusted intermediary between an untrusted hypervisor –making placement and management decisions on behalf of the CSP– and the guest machines deployed by Cloud users. As demonstrated in Komodo [100], trust in the monitor for all parties involved (the de-privileged hypervisor and the guest) can be derived from a combination of formal verification and attested measurement (*e.g.,* a signed boot measurement).

The trusted monitor validates hardware configuration changes with both parties before applying them. Specifically, it provides two crucial guarantees: (1) the hypervisor can, at any point, reclaim a resource, and (2) guest machines control how their resources are shared with the hypervisor or other VMs (*e.g.,* can ensure exclusive access to a physical page or allow selective sharing with specific VMs).

To orchestrate the cooperation between the hypervisor and guest machines, while ensuring both of the above guarantees, we rely on *hyperupcalls* [59]. Hyperupcalls are pieces of software that belong to the guest machine and can be invoked by the hypervisor to perform a control transfer into the VM's environment to execute the associated procedure without involving the VM's threads of execution. While they were originally designed to bridge the semantic gap between hypervisors and guest operating systems, we see in this mechanism an opportunity to establish a safe mean of cooperation between two mutually distrustful entities. Specifically, guest machines would ship with hyperupcalls as extensions to the guest operating system,

similar to related work [66]. These can be measured and their functional correctness can be validated by the hypervisor before instantiating the VM. These extensions accept as input a signed token from the hypervisor that represents a desired change in configuration. They can then either accept the change or refine it with extra requirements such as making a memory page exclusively accessible by the VM. The token and the VM's signed refinements have to be forwarded to the trusted monitor that validates them before applying the desired changes or reporting an error. Hyperupcalls for resource revocation, *e.g.*, reclaiming a page, cannot fail but provide the VM with the opportunity to implement defensive measures such as zeroing-out a page before returning it to the hypervisor. Of course, mechanisms should be devised to prevent controlled-channel attacks [196] or mitigate them in the implementation of hyperupcalls. We imagine a set of standard hyperupcalls packaged as an OS extension library, with a fixed API, that exposes all the management functionalities required by a hypervisor and removes the burden of providing formally verifiable implementations from the Cloud users.

8 Conclusion

In this thesis we focus on the problem of trust faced by modern software. As programming evolved to incorporate and heavily rely on unverified reusable software components, called packages, and as applications migrated to the Cloud, programming languages failed to adapt and provide appropriate abstractions to address the heterogeneous levels of trust that exist both within an application and on the systems on which they run. Developers do not have the tools to control which parts of their execution stack can access, modify, or leak their sensitive information.

We recognized the opportunities presented by recent hardware security extensions that enable to operate on the traditional trust model embodied in the process. We proposed new programming abstractions that closely integrate with these hardware extensions, while masking their inherent intricacies. Specifically, we presented secured routines, a language-level approach to supporting trusted execution environments that allows programmers to easily protect parts of their application's code from a potentially malicious host. Then, we introduced enclosures, an abstraction that allows developers to control resources made accessible, at package-granularity, to portions of their applications. Enclosures allow to safely leverage unverified public packages and protect an application's sensitive resources.

The combination of secured routines and enclosures should have allowed to incorporate complex logic in a trusted execution environment while guaranteeing that public libraries included in the trusted computing base would not weaken the security of code executing inside the TEE. Alas, the underlying hardware security extensions in charge of enforcing both abstractions' isolation seem, for the moment, too hard to compose. In the light of this result, we took a step-back and considered the conceptual modifications to the process' trust model brought by these hardware technologies. We introduced our current work, Tyche, that explores the design of an execution environment flexible enough to implement the modern isolation requirements between mutually distrustful trust domains.

Bibliography

[1] Amazon web services (aws): Cloud computing services. https://aws.amazon.com/.

[2] Amazon's simple cloud server. https://aws.amazon.com/lightsail/.

[3] Amd sev-es: Guest-hypervisor communication block standardization. https://developer.amd.com/wp-content/resources/56421.pdf.

[4] Anjuna security: Secure enclaves. https://www.anjuna.io/.

[5] Apple ii. https://en.wikipedia.org/wiki/Apple_II.

[6] Awesome WebAssembly Languages - a list of languages that compile to wasm. https://github.com/appcypher/awesome-wasm-langs.

[7] Aws and intel. https://aws.amazon.com/intel/.

[8] Aws bare metal machines. https://aws.amazon.com/about-aws/whats-new/2019/02/introducing-five-new-amazon-ec2-bare-metal-instances/.

[9] Aws elastic load balancing. https://aws.amazon.com/elasticloadbalancing/.

[10] Aws lambda. https://aws.amazon.com/lambda/.

[11] Colossus computer. https://en.wikipedia.org/wiki/Colossus_computer.

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

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

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

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

[16] 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.

[17] Enigma machine. https://en.wikipedia.org/wiki/Enigma_machine.

[18] Fault-tolerant components on aws. https://docs.aws.amazon.com/whitepapers/latest/fault-tolerant-components/amazon-elastic-compute-cloud.html.

[19] Google cloud: Cloud computing services. https://cloud.google.com/.

[20] Google: Global requests for user information. https://transparencyreport.google.com/user-data/overview?t=table.

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

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

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

[24] Microsoft azure: Cloud computing services. https://azure.microsoft.com/.

[25] Microsoft azure services. https://azure.microsoft.com/en-us/services/.

[26] Pypi: Python package manager. https://pypi.org/.

[27] Python matplotlib. https://matplotlib.org/.

[28] Python numpy. https://github.com/numpy/numpy.

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

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

[31] Webassembly: Core specification. https://webassembly.github.io/spec/core/.

[32] Webassembly: Javascript api. https://webassembly.github.io/spec/js-api/index.html.

[33] Webassembly: Wasi. https://github.com/WebAssembly/WASI.

[34] Webassembly: Web api. https://webassembly.github.io/spec/web-api/index.html.

[35] Linux is obsolete. https://groups.google.com/g/comp.os.minix/c/wlhw16QWltI, 1992.

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

[37] Insecure compiler optimization. https://owasp.org/www-community/vulnerabilities/Insecure_Compiler_Optimization, 2008.

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

[39] Amd memory encryption. https://developer.amd.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf, 2016.

[40] Netflix on aws. https://aws.amazon.com/solutions/case-studies/netflix-case-study/, 2016.

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

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

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

[44] Amd sev-snp: Strenghtening vm isolation with integrity protection and more. https://www.amd.com/system/files/TechDocs/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf, 2020.

[45] Linux Kernel-based Virtual Machine. https://www.linux-kvm.org, 2020.

[46] Oracle Java programming language. https://www.oracle.com/java/, 2020.

[47] Pip the pypa recommended tool for installing python packages. https://pypi.org/project/pip/, 2020.

[48] Python Packge Index. https://pypi.org/, 2020.

[49] Rubygems stats. https://rubygems.org/stats, 2020.

[50] Rust the cargo book. https://doc.rust-lang.org/cargo/commands/, 2020.

[51] Seccomp BPF. kernel.org/doc/html/latest/userspace-api/seccomp_filter.html, 2020.

[52] Writing Web Applications golang. https://golang.org/doc/articles/wiki/, 2020.

[53] Cve gcc: Security vulnerabilities. https://www.cvedetails.com/vulnerability-list/vendor_id-72/product_id-960/GNU-GCC.html, 2021.

[54] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the 2005 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 340–353, 2005.

[55] Charles Reis Adam Barth, Colling Jackson. The security architecture of the chromium browser. https://seclab.stanford.edu/websec/chromium/, 2008.

[56] Adrien Ghosn. Enclosures: language-based restriction of untrusted libraries. https://github.com/aghosn/enclosures, 2020.

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

[58] Fritz Alder, Jo Van Bulck, David F. Oswald, and Frank Piessens. Faulty Point Unit: ABI Poisoning Attacks on Intel SGX. pages 415–427, 2020.

[59] Nadav Amit and Michael Wei. The Design and Implementation of Hyperupcalls. In *Proceedings of the 2018 USENIX Annual Technical Conference (ATC)*, pages 97–112, 2018.

[60] Nadav Amit, Michael Wei, and Cheng-Chun Tu. Hypercallbacks: Decoupling Policy Decisions and Execution. In *Proceedings of The 16th Workshop on Hot Topics in Operating Systems (HotOS-XVI)*, pages 37–41, 2017.

[61] Ingo Molnar Arjan Van de Ven. Exec shield. https://static.redhat.com/legacy/f/pdf/rhel/WHP0006US_Execshield.pdf, 2004.

[62] ARM. Arm1136jf-s and arm1136j-s technical reference manual. https://developer.arm.com/documentation/ddi0211/latest/, 2020.

[63] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, André Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark Stillwell, David Goltzsche, David M. Eyers, Rüdiger Kapitza, Peter R. Pietzuch, and Christof Fetzer. SCONE: Secure Linux Containers with Intel SGX. In *Proceedings of the 12th Symposium on Operating System Design and Implementation (OSDI)*, pages 689–703, 2016.

[64] Arash Baratloo, Navjot Singh, and Timothy K. Tsai. Transparent Run-Time Defense Against Stack-Smashing Attacks. In *USENIX Annual Technical Conference*, pages 251–262, 2000.

[65] Andrew Baumann. Hardware is the new Software. In *Proceedings of The 16th Workshop on Hot Topics in Operating Systems (HotOS-XVI)*, pages 132–137, 2017.

[66] Andrew Baumann, Dongyoon Lee, Pedro Fonseca, Lisa Glendenning, Jacob R. Lorch, Barry Bond, Reuben Olinsky, and Galen C. Hunt. Composing OS extensions safely and efficiently with Bascule. In *Proceedings of the 2013 EuroSys Conference*, pages 239–252, 2013.

[67] Andrew Baumann, Marcus Peinado, and Galen C. Hunt. Shielding Applications from an Untrusted Cloud with Haven. *ACM Trans. Comput. Syst.*, 33(3):8:1–8:26, 2015.

[68] Adam Belay, Andrea Bittau, Ali José Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. Dune: Safe User-level Access to Privileged CPU Features. In *Proceedings of the 10th Symposium on Operating System Design and Implementation (OSDI)*, pages 335–348, 2012.

[69] A. Bensoussan, C. T. Clingen, and Robert C. Daley. The Multics Virtual Memory: Concepts and Design. *Commun. ACM*, 15(5):308–318, 1972.

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

[71] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. Wedge: Splitting Applications into Reduced-Privilege Compartments. In *Proceedings of the 5th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 309–322, 2008.

[72] Bojan. A new fascinating linux kernel vulnerability. https://isc.sans.edu/forums/diary/A+new+fascinating+Linux+kernel+vulnerability/6820/, 2009.

[73] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software Grand Exposure: SGX Cache Attacks Are Practical. In *Proceedings of the 11th USENIX Workshop on Offensive Technologies (WOOT)*, 2017.

[74] David Brumley and Dawn Xiaodong Song. Privtrans: Automatically Partitioning Programs for Privilege Separation. In *Proceedings of the 13th USENIX Security Symposium*, pages 57–72, 2004.

[75] Edouard Bugnion, Scott Devine, Mendel Rosenblum, Jeremy Sugerman, and Edward Y. Wang. Bringing Virtualization to the x86 Architecture with the Original VMware Workstation. *ACM Trans. Comput. Syst.*, 30(4):12:1–12:51, 2012.

[76] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *Proceedings of the 27th USENIX Security Symposium*, pages 991–1008, 2018.

[77] Catalin Cimpanu. Somebody tried to hide a backdoor in a popular javascript npm package. https://www.bleepingcomputer.com/news/security/somebody-tried-to-hide-a-backdoor-in-a-popular-javascript-npm-package/, 2018.

[78] Catalin Cimpanu. Backdoored Python Library Caught Stealing SSH Credentials. https://www.bleepingcomputer.com/news/security/backdoored-python-library-caught-stealing-ssh-credentials/, 2019.

[79] Catalin Cimpanu. Malicious Python libraries targeting Linux servers removed from PyPi. https://www.zdnet.com/article/malicious-python-libraries-targeting-linux-servers-removed-from-pypi/, 2019.

[80] Catalin Cimpanu. Twelve malicious Python libraries found and removed from PyPi. https://www.zdnet.com/article/twelve-malicious-python-libraries-found-and-removed-from-pypi/, 2019.

[81] Catalin Cimpanu. Two malicious Python libraries caught stealing SSH and GPG keys. https://www.zdnet.com/article/two-malicious-python-libraries-removed-from-pypi/, 2019.

[82] Catalin Cimpanu. Malicious npm packages caught installing remote access trojans. https://www.zdnet.com/article/malicious-npm-packages-caught-installing-remote-access-trojans/, 2020.

[83] Chia che Tsai, Kumar Saurabh Arora, Nehal Bandi, Bhushan Jain, William Jannen, Jitin John, Harry A. Kalodner, Vrushali Kulkarni, Daniela A. S. de Oliveira, and Donald E. Porter. Cooperation and security isolation of library OSes for multi-process applications. In *Proceedings of the 2014 EuroSys Conference*, pages 9:1–9:14, 2014.

[84] Stephen Checkoway and Hovav Shacham. 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)*, pages 253–264, 2013.

[85] Adrian Chen. Gcreeep: Google engineer stalked teens, spied on chats. https://gawker.com/5637234/gcreep-google-engineer-stalked-teens-spied-on-chats.

[86] Yaohui Chen, Sebassujeen Reymondjohnson, Zhichuang Sun, and Long Lu. Shreds: Fine-Grained Execution Units with Private Memory. In *IEEE Symposium on Security and Privacy*, pages 56–71, 2016.

[87] Stephen Chong, Jed Liu, Andrew C. Myers, Xin Qi, K. Vikram, Lantian Zheng, and Xin Zheng. Secure web application via automatic partitioning. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)*, pages 31–44, 2007.

[88] Matthew Cole and Aravind Prakash. Simplex: Repurposing Intel Memory Protection Extensions for Information Hiding. *CoRR*, abs/2009.06490, 2020.

[89] Victor Costan and Srinivas Devadas. Intel SGX Explained. *IACR Cryptol. ePrint Arch.*, 2016:86, 2016.

[90] John Criswell, Nathan Dautenhahn, and Vikram S. Adve. 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)*, pages 81–96, 2014.

[91] Dan Goodin. Hackers backdoor php source code after bearching internal git server. https://arstechnica.com/gadgets/2021/03/hackers-backdoor-php-source-code-after-breaching-internal-git-server/, 2021.

[92] Nathan Dautenhahn, Theodoros Kasampalis, Will Dietz, John Criswell, and Vikram S. Adve. Nested Kernel: An Operating System Architecture for Intra-Kernel Privilege Separation. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XX)*, pages 191–206, 2015.

[93] Brooks Davis, Robert N. M. Watson, Alexander Richardson, Peter G. Neumann, Simon W. Moore, John Baldwin, David Chisnall, James Clarke, Nathaniel Wesley Filardo, Khilan

Gudka, Alexandre Joannou, Ben Laurie, A. Theodore Markettos, J. Edward Maste, Alfredo Mazzinghi, Edward Tomasz Napierala, Robert M. Norton, Michael Roe, Peter Sewell, Stacey D. Son, and Jonathan Woodruff. CheriABI: Enforcing Valid Pointer Provenance and Minimizing Pointer Privilege in the POSIX C Run-time Environment. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXIV)*, pages 379–393, 2019.

[94] Xiaowan Dong, Zhuojia Shen, John Criswell, Alan L. Cox, and Sandhya Dwarkadas. Shielding Software From Privileged Side-Channel Attacks. In *Proceedings of the 27th USENIX Security Symposium*, pages 1441–1458, 2018.

[95] Vijay D'Silva, Mathias Payer, and Dawn Xiaodong Song. The Correctness-Security Gap in Compiler Optimization. In *IEEE Symposium on Security and Privacy Workshops*, pages 73–87, 2015.

[96] Dawson R. Engler, M. Frans Kaashoek, and James W. O'Toole Jr. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)*, pages 251–266, 1995.

[97] Ermetic. Top identity and data access risks. https://l.ermetic.com/wp-idc-survey-results?utm_campaign=IDC%20Survey%20Highlights&utm_source=Press%20release.

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

[99] Daniel Farina. pq - A pure Go postgres driver for Go's database/sql package. https://github.com/lib/pq, 2020.

[100] Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. Komodo: Using verification to disentangle secure-enclave hardware from software. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, pages 287–305, 2017.

[101] Nathaniel Wesley Filardo, Brett F. Gutstein, Jonathan Woodruff, Sam Ainsworth, Lucian Paul-Trifu, Brooks Davis, Hongyan Xia, Edward Tomasz Napierala, Alexander Richardson, John Baldwin, David Chisnall, Jessica Clarke, Khilan Gudka, Alexandre Joannou, A. Theodore Markettos, Alfredo Mazzinghi, Robert M. Norton, Michael Roe, Peter Sewell, Stacey D. Son, Timothy M. Jones, Simon W. Moore, Peter G. Neumann, and Robert N. M. Watson. Cornucopia: Temporal Safety for CHERI Heaps. In *IEEE Symposium on Security and Privacy*, pages 608–625, 2020.

[102] Bryan Ford and Russ Cox. Vx32: Lightweight User-level Sandboxing on the x86. In *Proceedings of the 2008 USENIX Annual Technical Conference (ATC)*, pages 293–306, 2008.

[103] Bryan Ford, Mike Hibler, Jay Lepreau, Patrick Tullmann, Godmar Back, and Stephen Clawson. Microkernels Meet Recursive Virtual Machines. In *Proceedings of the 2nd*

Symposium on Operating System Design and Implementation (OSDI), pages 137–151, 1996.

[104] Tommaso Frassetto, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. JITGuard: Hardening Just-in-time Compilers with SGX. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 2405–2419, 2017.

[105] Tommaso Frassetto, Patrick Jauernig, Christopher Liebchen, and Ahmad-Reza Sadeghi. IMIX: In-Process Memory Isolation EXtension. In *Proceedings of the 27th USENIX Security Symposium*, pages 83–97, 2018.

[106] Adrien Ghosn, Marios Kogias, Mathias Payer, James R. Larus, and Edouard Bugnion. Enclosure: language-based restriction of untrusted libraries. In *Proceedings of the 26th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXVI)*, pages 255–267, 2021.

[107] Adrien Ghosn, James R. Larus, and Edouard Bugnion. Secured Routines: Language-based Construction of Trusted Execution Environments. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, pages 571–586, 2019.

[108] David Goltzsche, Tim Siebels, and Rüdiger Kapitza. Trustscript: Language support for partitioning trusted web applications. https://www.eurosys2019.org/wp-content/uploads/2019/03/eurosys19posters-abstract100.pdf, 2019.

[109] David Goltzsche, Colin Wulf, Divya Muthukumaran, Konrad Rieck, Peter R. Pietzuch, and Rüdiger Kapitza. TrustJS: Trusted Client-side Execution of JavaScript. In *Proceedings of the 10th European Workshop on Systems Security (EUROSEC)*, pages 7:1–7:6, 2017.

[110] Google. Chromium V8 isolates. https://chromium.googlesource.com/chromium/src/+/master/third_party/blink/renderer/bindings/core/v8/V8BindingDesign.md#Isolate, 2020.

[111] Google. Golang add dependencies to the module and install them. https://golang.org/cmd/go/#hdr-Add_dependencies_to_current_module_and_install_them, 2020.

[112] Google LLC. Asylo. https://asylo.dev/, 2019.

[113] Google LLC. gRPC: A high performance, open source universal RPC framework. https://grpc.io/, 2019.

[114] Franz Gregor, Wojciech Ozga, Sébastien Vaucher, Rafael Pires, Do Le Quoc, Sergei Arnautov, André Martin, Valerio Schiavoni, Pascal Felber, and Christof Fetzer. Trust Management as a Service: Enabling Trusted Execution in the Face of Byzantine Stakeholders. In *Proceedings of the 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 502–514, 2020.

[115] Microsfot Confidential Computing group. Project Verona: a programming language for the modern cloud. https://www.microsoft.com/en-us/research/project/project-verona/, 2020.

[116] Khilan Gudka, Robert N. M. Watson, Jonathan Anderson, David Chisnall, Brooks Davis, Ben Laurie, Ilias Marinos, Peter G. Neumann, and Alex Richardson. Clean Application Compartmentalization with SOAAP. In *Proceedings of the 2015 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1016–1031, 2015.

[117] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and J. F. Bastien. Bringing the web up to speed with WebAssembly. In *Proceedings of the ACM SIGPLAN 2017 Conference on Programming Language Design and Implementation (PLDI)*, pages 185–200, 2017.

[118] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L. Scott, Kai Shen, and Mike Marty. Hodor: Intra-Process Isolation for High-Throughput Data Plane Libraries. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, pages 489–504, 2019.

[119] Henry T. Casey. MacBook M1 benchmarks are in — and they destroy Intel. https://www.tomsguide.com/news/macbook-pro-m1-benchmarks-are-in-and-they-destroy-intel, 2020.

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

[121] Owen S. Hofmann, Sangman Kim, Alan M. Dunn, Michael Z. Lee, and Emmett Witchel. 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)*, pages 265–278, 2013.

[122] Terry Ching-Hsiang Hsu, Kevin J. Hoffman, Patrick Eugster, and Mathias Payer. Enforcing Least Privilege Memory Views for Multithreaded Applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 393–405, 2016.

[123] Galen C. Hunt and James R. Larus. Singularity: rethinking the software stack. *ACM SIGOPS Oper. Syst. Rev.*, 41(2):37–49, 2007.

[124] Intel. Intel®64 and IA-32 Architectures Software Developer's Manual, 2020.

[125] Intel. Intel®Trust Domain Extensions, 2021.

[126] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Héctor M. Briceño, Russell Hunt, David Mazières, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth Mackenzie. Application Performance and Flexibility on Exokernel Systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP)*, pages 52–65, 1997.

[127] Brian W. Kernighan and Rob Pike. *Unix programming environment*. Prentice Hall, 1984.

[128] Yoongu Kim, Ross Daly, Jeremie S. Kim, Chris Fallin, Ji-Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In *Proceedings of the 41st International Symposium on Computer Architecture (ISCA)*, pages 361–372, 2014.

[129] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. In *IEEE Symposium on Security and Privacy*, pages 1–19, 2019.

[130] Marios Kogias, George Prekas, Adrien Ghosn, Jonas Fietz, and Edouard Bugnion. R2P2: Making RPCs first-class datacenter citizens. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, pages 863–880, 2019.

[131] Koen Koning, Xi Chen, Herbert Bos, Cristiano Giuffrida, and Elias Athanasopoulos. No Need to Hide: Protecting Safe Regions on Commodity Hardware. In *Proceedings of the 2017 EuroSys Conference*, pages 437–452, 2017.

[132] Diego Kreutz, Fernando M. V. Ramos, Paulo Jorge Esteves Veríssimo, Christian Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig. Software-Defined Networking: A Comprehensive Survey. *Proc. IEEE*, 103(1):14–76, 2015.

[133] Hugh C. Lauer and Roger M. Needham. On the Duality of Operating System Structures. *ACM SIGOPS Oper. Syst. Rev.*, 13(2):3–19, 1979.

[134] Lawrence Abrams. Malicious RubyGems packages used in cryptocurrency supply chain attack. https://www.bleepingcomputer.com/news/security/malicious-rubygems-packages-used-in-cryptocurrency-supply-chain-attack/, 2020.

[135] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanovic, and Dawn Song. Keystone: an open framework for architecting trusted execution environments. In *Proceedings of the 2020 EuroSys Conference*, pages 38:1–38:16, 2020.

[136] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing. In *Proceedings of the 26th USENIX Security Symposium*, pages 557–574, 2017.

[137] Xueping Liang, Sachin Shetty, Lingchen Zhang, Charles A. Kamhoua, and Kevin A. Kwiat. 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)*, pages 302–309, 2017.

[138] Joshua Lind, Christian Priebe, Divya Muthukumaran, Dan O'Keeffe, Pierre-Louis Aublin, Florian Kelbert, Tobias Reiher, David Goltzsche, David M. Eyers, Rüdiger Kapitza,

Christof Fetzer, and Peter R. Pietzuch. Glamdring: Automatic Application Partitioning for Intel SGX. In *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)*, pages 285–298, 2017.

[139] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1997.

[140] Linux. SecComp Load Filter. https://man7.org/linux/man-pages/man3/seccomp_load.3.html, 2020.

[141] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown. *CoRR*, abs/1801.01207, 2018.

[142] James Litton, Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, Bobby Bhattacharjee, and Peter Druschel. Light-Weight Contexts: An OS Abstraction for Safety and Performance. In *Proceedings of the 12th Symposium on Operating System Design and Implementation (OSDI)*, pages 49–64, 2016.

[143] Lei Liu, Xinwen Zhang, Guanhua Yan, and Songqing Chen. Chrome Extensions: Threat Analysis and Countermeasures. In *Proceedings of the 2012 Annual Network and Distributed System Security Symposium (NDSS)*, 2012.

[144] Yutao Liu, Tianyu Zhou, Kexin Chen, Haibo Chen, and Yubin Xia. Thwarting Memory Disclosure with Efficient Hypervisor-enforced Intra-domain Isolation. In *Proceedings of the 2015 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1607–1619, 2015.

[145] Lukas Martini. Fake version of dateutil and jellyfish. https://github.com/dateutil/dateutil/issues/984, 2019.

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

[147] Anil Madhavapeddy and David J. Scott. Unikernels: the rise of the virtual library operating system. *Commun. ACM*, 57(1):61–69, 2014.

[148] Moxie Marlinspike. Technology preview: Private contact discovery for signal. https://signal.org/blog/private-contact-discovery/.

[149] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena E. Olson, Erik Rubow, Michael Ryan, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. Snap: a microkernel approach to host networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, pages 399–413, 2019.

[150] Steven McCanne and Van Jacobson. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *USENIX Winter*, pages 259–270, 1993.

[151] Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil D. Gligor, and Adrian Perrig. TrustVisor: Efficient TCB Reduction and Attestation. In *IEEE Symposium on Security and Privacy*, pages 143–158, 2010.

[152] Michael Sammler. seccom: Add pkru into seccomp data. https://marc.info/?l=linux-api&m=154039581615478&w=2, 2018.

[153] Microsoft. /safeseh (safe exception handlers). https://docs.microsoft.com/en-us/cpp/build/reference/safeseh-image-has-safe-exception-handlers?redirectedfrom=MSDN&view=msvc-160, 2016.

[154] Lucian Mogosanu, Ashay Rane, and Nathan Dautenhahn. MicroStache: A Lightweight Execution Context for In-Process Safe Region Isolation. In *Proceedings of the 21st International Symposium on Research in Attacks, Intrusions, and Defenses(RAID)*, pages 359–379, 2018.

[155] Vikram Narayanan, Abhiram Balasubramanian, Charlie Jacobsen, Sarah Spall, Scotty Bauer, Michael Quigley, Aftab Hussain, Abdullah Younis, Junjie Shen, Moinak Bhattacharyya, and Anton Burtsev. LXDs: Towards Isolation of Kernel Subsystems. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, pages 269–284, 2019.

[156] Vikram Narayanan, Tianjiao Huang, David Detweiler, Dan Appel, Zhaofeng Li, Gerd Zellweger, and Anton Burtsev. RedLeaf: Isolation and Communication in a Safe Operating System. In *Proceedings of the 14th Symposium on Operating System Design and Implementation (OSDI)*, pages 21–39, 2020.

[157] George C. Necula. Proof-Carrying Code. In *Proceedings of the 24th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 106–119, 1997.

[158] Nick Nikiforakis, Luca Invernizzi, Alexandros Kapravelos, Steven Van Acker, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. You are what you include: large-scale evaluation of remote javascript inclusions. In *Proceedings of the 2012 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 736–747, 2012.

[159] Nikola Đuza. JavaScript Growing Pains: From 0 to 13,000 Dependencies. https://blog.appsignal.com/2020/05/14/javascript-growing-pains-from-0-to-13000-dependencies.html, 2020.

[160] Alexander Nilsson, Pegah Nikbakht Bideh, and Joakim Brorsson. A Survey of Published Attacks on Intel SGX. *CoRR*, abs/2006.13598, 2020.

[161] Meni Orenbach, Pavel Lifshits, Marina Minkin, and Mark Silberstein. Eleos: ExitLess OS Services for SGX Enclaves. In *Proceedings of the 2017 EuroSys Conference*, pages 238–253, 2017.

[162] Meni Orenbach, Yan Michalevsky, Christof Fetzer, and Mark Silberstein. CoSMIX: A Compiler-based System for Secure Memory Instrumentation and Execution in Enclaves. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, pages 555–570, 2019.

[163] Soyeon Park, Sangho Lee, Wen Xu, Hyungon Moon, and Taesoo Kim. libmpk: Software Abstraction for Intel Memory Protection Keys (Intel MPK). In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, pages 241–254, 2019.

[164] Team PaX. Adress space layout randomization. https://pax.grsecurity.net/docs/aslr.txt, 2001.

[165] Gerald J Popek and Robert P Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17(7):412–421, 1974.

[166] Martin Richards. The Portability of the BCPL Compiler. *Softw. Pract. Exp.*, 1(2):135–146, 1971.

[167] Mark Russinovich. Introducing Azure confidential computing. https://azure.microsoft.com/en-us/blog/introducing-azure-confidential-computing/, 2017.

[168] Jerome H. Saltzer. Protection and the Control of Information Sharing in Multics. *Commun. ACM*, 17(7):388–402, 1974.

[169] Jerome H Saltzer and M Frans Kaashoek. *Principles of computer system design: an introduction*. Morgan Kaufmann, 2009.

[170] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

[171] William F. Schmitt. The UNIVAC SHORT CODE. *IEEE Ann. Hist. Comput.*, 10(1):7–18, 1988.

[172] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. VC3: Trustworthy Data Analytics in the Cloud Using SGX. In *IEEE Symposium on Security and Privacy*, pages 38–54, 2015.

[173] Hovav Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *Proceedings of the 2007 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 552–561, 2007.

[174] Monirul I. Sharif, Wenke Lee, Weidong Cui, and Andrea Lanzi. Secure in-VM monitoring using hardware virtualization. In *Proceedings of the 2009 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 477–487, 2009.

[175] Le Shi, Yuming Wu, Yubin Xia, Nathan Dautenhahn, Haibo Chen, Binyu Zang, and Jinming Li. Deconstructing Xen. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*, 2017.

[176] Anthony N. Simon. Bild: A collection of parallel image processing algorithms in pure Go. https://github.com/anthonynsimon/bild, 2020.

[177] Rohit Sinha, Manuel Costa, Akash Lal, Nuno P. Lopes, Sriram K. Rajamani, Sanjit A. Seshia, and Kapil Vaswani. A design and verification methodology for secure isolated regions. In *Proceedings of the ACM SIGPLAN 2016 Conference on Programming Language Design and Implementation (PLDI)*, pages 665–681, 2016.

[178] Soham Kamani. Adding a database to a Go web application. https://www.sohamkamani.com/blog/2017/10/18/golang-adding-database-to-web-application/, 2020.

[179] Michael M. Swift, Steven Martin, Henry M. Levy, and Susan J. Eggers. Nooks: an architecture for reliable device drivers. In *ACM SIGOPS European Workshop*, pages 102–107, 2002.

[180] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. SoK: Eternal War in Memory. In *IEEE Symposium on Security and Privacy*, pages 48–62, 2013.

[181] Ken Thompson. Reflections on Trusting Trust. *Commun. ACM*, 27(8):761–763, 1984.

[182] Gorilla Web Toolkit. mux - Request router and dispatcher. https://github.com/gorilla/mux, 2020.

[183] Trend Micro. The XCSSET Malware: Inserts Malicious Code Into Xcode Projects, Performs UXSS Backdoor Planting in Safari, and Leverages Two Zero-day Exploits. https://documents.trendmicro.com/assets/pdf/XCSSET_Technical_Brief.pdf, 2020.

[184] Paul Turner. Retpoline: a software construct for preventing branch-target-injection. https://support.google.com/faqs/answer/7625886.

[185] Rich Uhlig, Gil Neiger, Dion Rodgers, Amy L. Santoni, Fernando C. M. Martins, Andrew V. Anderson, Steven M. Bennett, Alain Kägi, Felix H. Leung, and Larry Smith. Intel Virtualization Technology. *Computer*, 38(5):48–56, 2005.

[186] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK). In *Proceedings of the 28th USENIX Security Symposium*, pages 1221–1238, 2019.

[187] Aliaksandr Valialkin. FastHTTP: Fast HTTP implementation for Go. https://github.com/valyala/fasthttp, 2020.

[188] Lluís Vilanova, Muli Ben-Yehuda, Nacho Navarro, Yoav Etsion, and Mateo Valero. CODOMs: Protecting software with Code-centric memory Domains. In *Proceedings of the 41st International Symposium on Computer Architecture (ISCA)*, pages 469–480, 2014.

[189] Stavros Volos, Kapil Vaswani, and Rodrigo Bruno. Graviton: Trusted Execution Environments on GPUs. In *Proceedings of the 13th Symposium on Operating System Design and Implementation (OSDI)*, pages 681–696, 2018.

[190] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient Software-Based Fault Isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP)*, pages 203–216, 1993.

[191] Zhi Wang and Xuxian Jiang. HyperSafe: A Lightweight Approach to Provide Lifetime Hypervisor Control-Flow Integrity. In *IEEE Symposium on Security and Privacy*, pages 380–395, 2010.

[192] Robert N. M. Watson, Jonathan Woodruff, Peter G. Neumann, Simon W. Moore, Jonathan Anderson, David Chisnall, Nirav H. Dave, Brooks Davis, Khilan Gudka, Ben Laurie, Steven J. Murdoch, Robert M. Norton, Michael Roe, Stacey D. Son, and Munraj Vadera. CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization. In *IEEE Symposium on Security and Privacy*, pages 20–37, 2015.

[193] Emmett Witchel, Josh Cates, and Krste Asanovic. Mondrian memory protection. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)*, pages 304–316, 2002.

[194] Jonathan Woodruff, Alexandre Joannou, Hongyan Xia, Anthony C. J. Fox, Robert M. Norton, David Chisnall, Brooks Davis, Khilan Gudka, Nathaniel Wesley Filardo, A. Theodore Markettos, Michael Roe, Peter G. Neumann, Robert N. M. Watson, and Simon W. Moore. CHERI Concentrate: Practical Compressed Capabilities. *IEEE Trans. Computers*, 68(10):1455–1469, 2019.

[195] Jonathan Woodruff, Robert N. M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert M. Norton, and Michael Roe. The CHERI capability model: Revisiting RISC in an age of risk. In *Proceedings of the 41st International Symposium on Computer Architecture (ISCA)*, pages 457–468, 2014.

[196] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *IEEE Symposium on Security and Privacy*, pages 640–656, 2015.

[197] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native Client: a sandbox for portable, untrusted x86 native code. *Commun. ACM*, 53(1):91–99, 2010.

[198] Takeshi Yoneda. Gasm: a minimal wasm virtual machine implemented in go. https://github.com/mathetake/gasm.

[199] Ethan G. Young, Pengfei Zhu, Tyler Caraza-Harter, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. The True Cost of Containing: A gVisor Case Study. In *Proceedings of the 11th workshop on Hot topics in Cloud Computing (HotCloud)*, 2019.

[200] Eric S. Yuan. Zoom's Use of Facebook's SDK in iOS Client. https://blog.zoom.us/zoom-use-of-facebook-sdk-in-ios-client/, 2020.

[201] Zack Bloom. Cloud Computing without Containers. https://blog.cloudflare.com/cloud-computing-without-containers/, 2020.

[202] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. Untrusted Hosts and Confidentiality: Secure Program Partitioning. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, pages 1–14, 2001.

[203] Dmitry P. Zegzhda, E. S. Usov, V. A. Nikol'skii, and Evgeny Pavlenko. Use of Intel SGX to ensure the confidentiality of data of cloud users. *Autom. Control. Comput. Sci.*, 51(8):848–854, 2017.

[204] Fengzhe Zhang, Jin Chen, Haibo Chen, and Binyu Zang. 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)*, pages 203–216, 2011.

[205] Chongchong Zhao, Daniyaer Saifuding, Hongliang Tian, Yong Zhang, and Chunxiao Xing. On the Performance of Intel SGX. In *IEEE WISA*, pages 184–187, 2016.

[206] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. Small World with High Risks: A Study of Security Threats in the npm Ecosystem. In *Proceedings of the 28th USENIX Security Symposium*, pages 995–1010, 2019.

[207] Úlfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. XFI: Software Guards for System Address Spaces. In *Proceedings of the 7th Symposium on Operating System Design and Implementation (OSDI)*, pages 75–88, 2006.

Adrien **Ghosn**

PhD student in Computer Science

ghosn.adrien@gmail.com | https://people.epfl.ch/adrien.ghosn | aghosn | aghosn

Education

Ecole Polytechnique Federale de Lausanne(EPFL) — *Lausanne, Switzerland*

Computer Science Engineering — *Sep. 2010 - 2021*

- 2016 – 2021: PhD in Datacenter System Laboratory, Prof. Edouard Bugnion and Prof. James Larus
- 2013 – 2016: Master Degree, Foundations of Software specialization (avg 5.75/6)
- 2010 – 2013: Bachelor Degree

Northeastern University(NEU) — *Boston, U.S.A.*

Master Thesis — *Sep. 2015 - Mar. 2016*

- Supervised by Prof. Jan Vitek in the Programming Languages Laboratory

Carnegie Mellon University(CMU) — *Pittsburgh, U.S.A.*

Exchange Year, Bachelor Degree in Computer Science — *Aug. 2012 - Jul. 2013*

- Dean's list School of Computer Science for QPA > 3.75/4

PhD Internships

Summer Internship — *Kirkland, USA*

Google Asylo team - Matt Gingell — *June - August 2019*

- Asylo team, Trusted Execution environments, SGX
- Explored potential designs to support higher-level programming languages in SGX enclaves
- Delivered a prototype that allowed HLPL code to run inside SGX

Research & Publications

Focus Areas

Programming Languages, Systems, Security
Isolation of mutually distrustful software components
Hardware-enforced isolation

Ongoing Research — *Lausanne, Switzerland*

EPFL, DCSL - Prof. Edouard Bugnion, Prof. James Larus, Prof. Mathias Payer — *Aug. 2019 - Present*

- Web-assembly as unit of intra-address space isolation

Enclosures: Language-based restriction of untrusted libraries [1] — *Lausanne, Switzerland*

EPFL, DSCL - Prof. Edouard Bugnion, Prof. James Larus, Prof. Mathias Payer — *Sep. 2019 - Oct. 2020*

- New fine-grain programming abstraction to restrict public libraries access to program resources
- Frontend extensions to Go and Python PLs
- Backend support for hardware isolation enforcement (Intel VT-x & Intel MPK)
- Intra-address-space isolation, Sandboxing, Compiler, Linker, Runtime

Secured Routines: Language-based construction of TEEs [2] — *Lausanne, Switzerland*

EPFL, DSCL - Prof. Edouard Bugnion, Prof. James Larus — *Jun. 2018 - May 2019*

- Extended Go programming language to support executing goroutines inside Intel SGX.
- Intel SGX, Confidentiality, Intergrity, Go, Compilers, Code partitioning, Hardware Extensions

Light-Weight Contexts in Dune — *Lausanne, Switzerland*

EPFL, DSCL - Prof. Edouard Bugnion — *Sep. 2016 - Jul. 2017*

- Process virtualization with Dune
- Intra-address space isolation, protecting secrets, memory snapshots
- 5x speed improvement over a Linux fork
- Intel VTX, Dune, Virtualization, Kernel module, Virtual Memory Management

117

Efficient Runtime Deoptimization for R(Master Thesis)

Boston, U.S.A.

Northeastern University - Prof. Jan Vitek

Sep. 2015 - Mar. 2016

- Speculative optimizer for an R JIT compiler
- Removes performance bottlenecks due to the language semantics
- Ensures correct run-time behavior.
- On-stack replacement, speculative optimizations, runtime de-optimization, R, LLVM, JIT compiler

Aperiodic-Event Support in FASA

Baden, Switzerland

ABB Corporate Research - Dr. Manuel Oriol

Feb. 2015 - Aug. 2015

- Fixed-priority servers, data-driven events, real-time control applications
- kernel design, dynamic linking/loading & software updates, pi-calculus

Scalameta: AST Persistence & Obey: Code Health

Lausanne, Switzerland

EPFL, LAMP - Prof. Martin Odersky & Dr. Eugene Burmako

Jan. 2014 - Feb. 2015

- Obey: Scala-linter for user-defined rules enforced at compile-time
- AST Persistence: typed-AST format for Scala
- Resolves compiler version incompatibilities and provides IDE macros expansion support

Projects

Operating Systems & Design 15-410

CMU

Undergraduate

Jan. 2013 - Jul. 2013

- Implementation of a x86 Unix like Kernel in C and ASM
- Design and implementation of thread library, scheduler, virtual memory, various drivers, system calls

Tweet Aggregator

EPFL

Graduate

Jan. 2014 - Jul. 2014

- Big Data web application to gather and display real-time tweets on a map, according to user-defined keywords
- Filtering and clustering tweets according to zoom-level and selected geographical areas

Compiler & Advanced Compiler

EPFL

Graduate

Sep. 2013 - Jul. 2014

- Design & implementation of compilers for JVM-based Lisp-like languages
- Mark & sweep garbage collector and optimization phases including DCE-CSE, constant folding, closure hoisting

Skills

Programming	Go, C/C++, Java, Shell scripting, asm, Python
Knowledge in	Compilers & PL design, Language runtimes & virtual machines Operating System design, Virtualization, KVM, Intel VT-x, Intel MPK Software security, Hardware Security extensions, Trusted Execution Environments Theoretical CS, Concurrent & Distributed Algorithms

Teaching

Semester Projects	Go Intel MPK library (Charly Castes) System call filtering in Go & Python enclosure support (Elsa weber)
Teaching Assistant	Functional Programming (2020), Introduction to Operating Systems (2019) Introduction to Java Programming (2018), Systems for Data Science (2017-2020) Introduction to C Programming (2016-2017), Concurrent Programming (2015) Student Volunteer at ECOOP (2016)

Personnal

Languages Fluent in French & English

118

References

[1] Adrien Ghosn, Marios Kogias, Mathias Payer, James R. Larus, and Edouard Bugnion. Enclosure: language-based restriction of untrusted libraries. pages 255–267, 2021.

[2] Adrien Ghosn, James R. Larus, and Edouard Bugnion. Secured Routines: Language-based Construction of Trusted Execution Environments. pages 571–586, 2019.