Privagic : confidential computing made practical with secure typing

Subashiny Tanigassalame

INSTITUT POLYTECHNIQUE DE PARIS

NNT : 2024IPPAS004

Thèse de doctorat

Privagic: confidential computing made practical with secure typing

Thèse de doctorat de l'Institut Polytechnique de Paris
préparée à Télécom SudParis

École doctorale n°626 École doctorale de l'Institut Polytechnique de Paris (EDIPP)
Spécialité de doctorat : Informatique

Thèse présentée et soutenue à Palaiseau, le 5 Avril 2024, par

SUBASHINY TANIGASSALAME

Composition du Jury :

Marc Shapiro Directeur de recherche (émérite), Sorbonne Université, LIP6, Inria	Rapporteur
Laurent Réveillère Professeur, Université de Bordeaux	Rapporteur
Sonia Ben Mokhtar Directrice de Recherche, LIRIS CNRS	Examinatrice
Stéphane Ducasse Directeur de recherche, Inria	Président du Jury
Gaël Thomas Directeur de recherche, Inria	Directeur de thèse
Nicolas Anciaux Directeur de recherche, Inria	Invité

Privagic: confidential computing made practical with secure typing

Subashiny Tanigassalame

July 18, 2024

À Amma, Daddy et Kirthi

Remerciements / Acknowledgements / நன்றியுரை

Je souhaite exprimer ma sincère reconnaissance envers toutes les personnes qui ont contribué de manière significative à la réalisation de ce manuscrit de thèse. Leur soutien inestimable, leurs conseils éclairés et leur encouragement constant ont été des éléments essentiels pour mener à bien ma thèse.

Tout d'abord, je tiens à remercier mon directeur de thèse, Gaël Thomas, pour avoir proposé ce sujet de recherche et pour son mentorat depuis mon école d'ingénieur jusqu'à aujourd'hui. Son expertise, ses conseils, sa disponibilité et surtout sa confiance en mes capacités m'ont permis de surmonter tous les défis rencontrés pendant ma thèse. Je remercie chaleureusement Gaël pour son soutien continu dans la rédaction de ce manuscrit.

Je voudrais ensuite exprimer ma gratitude envers les deux rapporteurs, Marc Shapiro et Laurent Réveillère, pour leurs lectures attentives de ce manuscrit ainsi que pour leurs retours précieux. Je suis sincèrement reconnaissante pour le temps et l'attention qu'ils ont consacrés à examiner et évaluer mon travail avec tant de rigueur. Ensuite, je souhaiterais remercier Sonia Ben Mokhtar, Stéphane Ducasse et Nicolas Anciaux, pour avoir accepté de participer à mon jury. Je suis très honorée de compter ces cinq experts parmi les lecteurs de ce manuscrit.

Je tiens à remercier Amel Bouzeghoub pour son soutien inestimable tout au long de cette période. Ses attentions régulières et ses mots réconfortants ont été une source de réconfort et de motivation pour moi. Je souhaite également exprimer ma reconnaissance envers Denis Conan pour tous ses précieux conseils qu'il m'a apportés durant ma thèse.

Par la suite, j'aimerais remercier toutes l'équipe PDS et le département INF pour leur soutien, pour les nombreuses discussions enrichissantes, et pour tous les beaux souvenirs construits durant mes années de thèse. La liste des personnes, à qui je suis vraiment reconnaissante, est très longue, je voudrais néanmoins exprimer ma gratitude particulière envers François Trahay, Elisabeth Brunet, Amina Guermouche, Chantal Taconnet, Sophie Chabridon, Pierre Sutra, Mathieu Bacou, Michel Simatic, Pascal Hennequin, Olivier Berger et Valentin Honoré

Je voudrais aussi dire un grand merci à Brigitte Houassine et Marie Degli-Esposti, pour m'avoir apporté toutes les aides administratives nécessaires.

Je tiens à exprimer ma profonde gratitude à tous les doctorants pour leur soutien scientifique et moral. Leur présence a été précieuse dans les moments les plus hauts comme les plus bas de ma thèse. Plus particulièrement, je voudrais remercier Nabila Belhaj pour m'avoir guidé au début de mon doctorat et pour avoir partagé ses expériences avec moi. Un grand merci à Alexis Lescouet d'avoir généreusement fourni toutes ses astuces pour l'inscription à l'école doctoral et le kit de début de thèse. Je tiens également à remercier Alexis Colin pour les expertises C++ qu'il m'a apportées et pour toutes les sessions de debugging autour d'un cappuccino.

Je ne saurais assez remercier Yohan Pipereau. En plus de la grande contribution qu'il a apportée dans l'évaluation de Privagic, il a toujours été un support moral incroyable. Je lui suis grandement reconnaissante pour l'aide qu'il a apportée afin de bien terminer ma thèse.

Je voudrais ensuite remercier Adam Chader pour ses collaborations dans le projet et pour les moments de joie. J'aimerais remercier Mickaël Boichot pour sa bienveillance et pour avoir su me faire sourire dans toutes les circonstances. Un grand merci à Marie Reinbigler pour sa gentillesse et pour m'avoir introduit aux sujets du domaine biologique.

Je tiens également à exprimer ma reconnaissance envers Damien Thenot, Boubacar Kane, Remi Dulong, Anton Daumen, Pedro Borges, Jean-François Dumollard, Catherine Guelque et Nicolas Derumigny pour toutes les discussions enrichissantes, les fou-rires et les émotions partagées ensemble.

Furthermore, I would like to express my gratitude to my PhD sister, Jana Toljaga. First I am thankful for her contribution to the project. Then, during the final year of my thesis, she provided immense moral support, particularly during the writing of my manuscript. And finally, a special thanks to her for introducing me to the world of traveling.

Je souhaiterai remercier mes amis proches Mustapha Younsi, Ghita Jiar et Gayathri Nadessane-Gomathi, qui m'ont soutenue tout au long de cette aventure académique. Leur foi absolue en mes capacités et leur soutien constant ont été une source d'encouragement et de joie pendant les moments de stress. Je suis reconnaissante de les avoir dans ma vie.

Par ailleurs, je tiens à exprimer ma profonde gratitude envers ma famille pour leur soutien indéfectible, leur amour et leur compréhension tout au long de ce parcours. Leur soutien moral et leur encouragement ont été des élé-

ments essentiels dans les moments de doute et de difficulté. Un grand merci à ma famille pour m'avoir toujours enseigné l'importance de l'éducation.

என் பெற்றோர்கள் இருவருக்கும் என் இதயம் கனிந்த முதற்கண் நன்றிகள். நான் எடுத்த அனைத்து முடிவுகளிலும் எனக்கு உறுதுணையாக இருந்தீர்கள். நீங்கள் இருவரும் எனக்கு பின் துணை நிற்கும் நம்பிக்கையில்தான், என்னால் முன்னோக்கி ஓட இயன்றது.

நான் என் கல்வியை சிறப்பாக படித்து முடித்து விட்டேன். இதற்க்கு மேல் நான் படிக்க வேண்டும் என்று கோரி உங்களை தொல்லை செய்ய மாட்டேன். தங்கள் இருவரின் தைரியத்தில்தான், என்னால் PhD யை திறம்பட நிறைவு செய்ய முடிந்தது. என் அம்மா Valarmathi Tanigassalame க்கும், என் அப்பா Mounissamy Tanigassalame க்கும், என் மனமார்ந்த நன்றிகள்.

(Mes sincères remerciements à mes deux parents. Vous m'avez soutenu dans toutes mes décisions. Ce n'est qu'avec l'espoir que vous êtes tous les deux derrière moi que je pourrai avancer.

J'ai bien terminé mes études. Je ne vous dérangerai pas en exigeant de continuer plus que cela. C'est grâce au courage de vous deux que j'ai pu mener à bien mon doctorat. Ma profonde gratitude à ma mère Valarmathi Tanigassalame et mon père Mounissamy Tanigassalame.)

Je ne saurais terminer ce remerciement sans remercier ma petite sœur, Kiruthika Tanigassalame, qui a été, qui est, et qui restera toujours mon pilier. Merci pour ta joie de vivre et surtout pour la dose de folie que tu as apportée tout au long de la durée de ma thèse. Ta présence a été une source inestimable de soutien et de réconfort pour moi.

Enfin, je souhaite exprimer ma gratitude envers toutes les personnes qui, de près ou de loin, ont contribué au bon déroulement de ma thèse, même si je ne peux pas toutes les nommer individuellement.

Ensemble, vous avez tous joué un rôle crucial dans la réalisation de ce manuscrit de thèse, et je vous en suis infiniment reconnaissante.

Abstract

For more than twenty years, several tools have been proposed to automatically partition an application between a secure memory zone and a non-secure memory zone. These tools analyze the data flow of the application in order to identify the memory locations that may contain sensitive values. Most of these tools behave incorrectly in the presence of pointers. When they are correct, they are unable to handle threads because of the difficulty to track pointers in a multi-threaded application. The current tools are also unable to split an application in more than two partitions. This is caused by over-approximation, which leads to memory locations falsely shared between the two partitions.

In this thesis, instead of starting from data flow analysis, we propose to start from a more accurate technique: language typing. We introduce secure typing, which consists in embedding a partition identifier in the type system of a language. Based on secure typing, we designed a language-agnostic compiler based on LLVM. The compiler takes a legacy application enriched with secure types as input, and generates multiple partitions for Intel SGX. Our evaluation with micro- and macro-applications show that (i) secure typing can handle pointers, multiple threads and more than two partitions, (ii) adding secure types in a legacy application is easy, (iii) secure typing reduces the trusted computing base, and is more efficient than embedding a full application inside an enclave.

Privagic : Informatique confidentielle rendue pratique grâce au typage sécurisé

Résumé en français

L'informatique confidentielle consiste à protéger les données des utilisateurs lorsqu'elles sont traitées dans un système non fiable tel qu'une infrastructure cloud. Au niveau matériel, l'informatique confidentielle repose sur un environnement d'exécution fiable (TEE) (SGX d'Intel, SEV d'AMD ou TrustZone d'ARM). Un TEE est un environnement matériel qui isole une zone de mémoire, appelée enclave, d'un système d'exploitation ou d'un hyperviseur potentiellement compromis. Étant donné qu'il est difficile de partitionner manuellement une application entre une enclave et la mémoire non sécurisée, de nombreux outils de partitionnement automatique ont été proposés. Avec ces outils, le développeur annote certaines valeurs sensibles et l'outil analyse ensuite le code pour trouver les emplacements de mémoire dans lesquels les valeurs sensibles propagent. La plupart de ces outils se comportent incorrectement en présence de pointeurs. Lorsqu'ils sont corrects, ils ne parviennent pas à gérer les threads en raison de la difficulté à suivre les pointeurs dans une application multi-thread. Les outils actuels sont également incapables de diviser une application en plus de deux partitions. Cela est causé par une surapproximation, qui conduit à des emplacements mémoire faussement partagés entre les deux partitions. Sur la base de cette analyse, les outils répartissent ensuite les données et le code de l'application entre les parties sécurisées et non sécurisées. Pour un effort d'ingénierie modeste, ces outils limitent la surface d'attaque des codes placés à l'intérieur d'une enclave.

Le principal problème est que l'analyse du flux de données analyse une application de manière séquentielle. Par conséquent, il ne voit pas les modifications de pointeurs exécutées en parallèle par les autres threads. L'outil peut conclure à tort qu'un pointeur pointe vers une enclave, car cette observation est correcte si le code s'exécute de manière séquentielle, mais pas s'il s'exécute en parallèle. En présence de plusieurs threads, l'analyse du flux de données peut donc laisser s'échapper un élément sensible dans une mémoire non sécurisée par l'intermédiaire d'un pointeur identifié à tort comme pointant vers une enclave.

Étant donné que l'analyse du flux de données ne permet pas de gérer une application C multithreadée Nous proposons de partir d'un autre point de l'espace de conception, en laissant le développeur annoter explicitement tous les emplacements de mémoire qui contiennent des valeurs sensibles. Puisque le développeur annote explicitement tous les emplacements de mémoire sensibles, il n'est pas nécessaire d'analyser le code. Nous évitons ainsi par construction tout risque d'erreur d'analyse dans une application multithread. Cependant, en échangeant l'analyse automatique contre une annotation manuelle, nous échangeons également la facilité d'utilisation de l'analyse du flux de données contre un surcroît de travail pour le développeur. L'évaluation présentée dans cette thèse vise à vérifier que la charge imposée au développeur se traduit par un effort d'ingénierie raisonnable. Afin de permettre au développeur d'indiquer les emplacements de mémoire qui contiennent des valeurs sensibles, nous introduisons une nouvelle construction du langage appelée « type sécurisé ». Un type sécurisé est un type enrichi d'un identifiant d'enclave, que nous appelons une couleur. En ajoutant explicitement un type sécurisé à chaque emplacement sensible de la mémoire, le partitionnement du code devient simple.

Le typage sécurisé indique comment partitionner le code. En lui-même, le typage sécurisé ne fournit aucune garantie de sécurité. Pour renforcer la sécurité, nous proposons donc de compléter le typage sécurisé par des règles de typage. Le typage explicite de chaque emplacement de mémoire susceptible de contenir une valeur sensible rend possible le partitionnement d'une application multithread. L'ajout d'un type sécurisé à chaque emplacement mémoire sensible peut prendre beaucoup de temps au développeur. Pour cette raison, nous proposons de faciliter l'utilisation du typage sécurisé avec une forme simple d'inférence de type. En détail, nous proposons de déduire le type d'une variable locale non colorée, mais seulement si le code ne crée pas de pointeur sur la variable. Dans ce cas, la variable ne s'échappe pas de la portée d'une seule fonction, ce qui évite l'analyse inter procédurale. De plus, comme la variable ne s'échappe pas de la portée de sa fonction, elle ne peut pas être accédée par un autre thread. Avec cette restriction, la déduction d'un type sécurisé nécessite une simple analyse de la chaîne use-def, et le type déduit est correct même dans les applications multithread. Nous avons mis en œuvre notre principe de typage sécurisé dans le cadre Privagic pour

Intel SGX et le langage C.

Le compilateur Privagic s'appuie sur le compilateur LLVM, ce qui signifie qu'il ne s'appuie pas sur la sémantique C : il considère une représentation intermédiaire de bas niveau du code avec des types sécurisés ajoutés aux variables, aux arguments et aux champs des structures de données. Notre évaluation avec des micro- et macro-applications montre que (i) le typage sécurisé peut gérer les pointeurs, le multi-threads et plus de deux partitions, (ii) l'ajout de types sécurisés dans une application existante est facile, (iii) le typage sécurisé réduit la base de confiance et est plus efficace que l'intégration complète d'une application dans une enclave.

Contents

Chapter 1

Introduction

Billions of people use online applications on a daily basis. The key success of these applications is that they provide personalized services to their users: they return results related to users' interests. These preferences are calculated based on user profiles, retrieved from consumers' past queries or from their other online activities. A user profile contains user identity and data about the user, which is essential and specific to each application to provide the service. However, depending on the chosen application, these data may contain sensitive information about the consumer.

Today, protecting user profiles is difficult because they are deployed in cloud infrastructures. In detail, the majority of online services depend on a third party cloud server for storage and computational purposes. Data from those servers can be leaked by hackers or even by malicious system administrators. A leakage in user profiles may jeopardize user privacy. For instance, a user profile of a personalized geolocated application carries the user's mobility data. Numerous sensitive information including the user's home, workplace, religious or political preferences, can be derived from mobility data. Therefore a data breach menaces user's privacy.

Confidential computing consists of protecting user data when it is processed in an untrusted system such as a cloud infrastructure [4,18,19,40]. At a low level, confidential computing relies on a *trusted execution environment* (TEE). A TEE is a hardware environment that isolates a memory zone, called an *enclave*, from a potentially compromised operating system or hypervisor. For that, a TEE relies on remote attestation for authentication, and on hardware cryptography to enforce the integrity and the confidentiality of both the

code and data contained in an enclave.

TEEs have been commercially available for more than eight years in the CPUs of the main manufacturers. For Intel, it is SGX (Software Guard Extensions) [22], for AMD, it is SME (Secure Encrypted Virtualization) [32], and for ARM, it is TrustZone [3,42]. Because manually writing an application that uses a TEE, with the software development kits provided by the CPU manufacturers, remains difficult and error-prone, several tools were recently proposed to ease the use of a TEE. At one extreme, some tools propose to fully embed an application with its dependencies inside an enclave (e.g., Scone [6] or Graphene-SGX [53]). These tools ease the use of a TEE, but they lead to a large trusted computing base (TCB), which can easily reach tens of MiB (see §7.2.2). Such a large TCB leads to a large attack surface and a poor safety.

Other tools automatically partition an application between a trusted part, which runs inside an enclave, and an untrusted part, which runs outside [13,28,31,34–37,45,54,58,60,61]. These tools minimize the TCB, but, as we show in §3, they remain impractical in the general case. At a high level, these tools statically analyze *sequentially* the data flow in order to identify the memory locations that may contain sensitive values. In the presence of pointers, most of these tools misbehave. When they do not misbehave, they are unable to handle threads because of the difficulty to track pointers in a multi-threaded application with sequential data flow analysis. These tools also significantly over-approximate the number of memory locations that may contain a sensitive value. They can thus not be used to partition an application into multiple enclaves, since some memory locations are falsely identified as containing values from two different enclaves. As a result, it is today impossible to automatically partition an application in multiple enclaves, e.g., in order to manage data from entities that do not trust each other, or in order to increase the difficulty for an attacker to steal useful data.

Additionally to the above mentioned limitations, our analysis also shows that, in addition to their impracticability in the general case, most of the existing tools are highly language-dependent. They depend on the semantics of the language such as the use of goroutines in Go or of classes in Java. The techniques proposed by most of the existing tools can thus not be reused for different languages, which leaves the problem of designing a practical

```
1. struct account {
2.   char color(blue) name[256];
3.   double color(red) balance;
4. };

5. struct account* create(char* name) {
6.   struct account* res = malloc(sizeof(*res));
7.   strncpy(res->name, name, 256);
8.   res->balance = 0.0;
9.   return res;
10. }
```

Figure 1.1: A simple example of the Privagic language in C.

partitioning tool for many languages.

Since data flow analysis remains fragile in the general case, we propose to start from another well-known language construct: explicit language typing. With explicit Language typing, a compiler is able to accurately give the nature of each memory location in a program, e.g., integer, float, pointer, etc. Explicit language typing is thus a perfect candidate to associate a security property to each memory location. Starting from this idea, we propose *secure typing*. We define a secure type as a classical type enriched with an enclave identifier, which we name a *color* Figure 1.1 illustrates the principle with the C language. Starting from a legacy application, the developer adds colors to the types of the fields `name` (line 2) and `balance` (line 3). These annotations mean that the `name` field lives in a blue enclave and that the `balance` field lives in a red enclave. With these secure types, analyzing the code in Figure 1.1 becomes obvious. At line 7, the expression uses a blue value and it has thus to be executed in the blue enclave. At line 8, the expression uses a red value and it has to be executed in the red enclave.

Thanks to explicit secure typing, a compiler can identify the enclave of any variable, of any field of any data structure, and of any instruction, even if an application is multi-threaded or manipulates pointers. Secure typing makes also possible the use of multiple colors because the compiler does not over-approximate the number of sensitive memory locations anymore. By allowing the use of multiple colors, secure typing helps to improve security. For example, in Figure 1.1, by using two colors, if an attacker compromises only a single enclave, the attacker can only steal relatively useless data: only

the customer names or only the account balances.

With secure typing, the compiler can also easily help the developer to enforce security by defining typing rules. To enforce confidentiality, for example, the compiler detects type errors such as storing a colored value in an uncolored memory location. For integrity, the compiler detects type errors such as storing an uncolored value in a colored memory location. With secure typing, the compiler can also detect other kinds of attacks such as Iago attacks [15], which consist in sending a poisoned value to an enclave. For that, the compiler simply has to detect other type errors such as using an uncolored value as input in an instruction that outputs a colored value.

Additionally to improving ease-of-use and enforcing security, secure typing also opens a new performance optimization opportunity. With secure typing, we observe that if two instructions access values with different colors, by construction, they do not have data dependencies. We can thus execute them in parallel. For example, in Figure 1.1, we can execute lines 7 and 8 in parallel, which increases performance as compared to a sequential execution. Running the code in parallel additionally avoids the prohibitive cost of entering and leaving an enclave [6, 47, 52, 56, 63]. Instead, the application can use communication in shared memory to transfer the control between the thread executed outside the enclaves and the thread executed inside.

We implemented our principle of secure typing in the Privagic framework. The framework contains a compiler and a parallel runtime. The compiler enforces security by enforcing typing rules and then partitions the application. The parallel runtime executes the code that accesses different colors in parallel. By default, the compiler runs in hardened mode. It enforces confidentiality and integrity, and prevents Iago attacks by totally isolating each enclave from the rest of the application. Since strong isolation can lead to a large trusted computing base and poor performance, the Privagic compiler also proposes a relaxed mode. This mode targets expert developers. It enforces confidentiality and integrity, but relaxes the color constraints that prevent Iago attacks. Relaxing these constraints forces the developer to carefully insert integrity checks when an instruction uses an uncolored value to compute a colored value, but allows the developer to still reduce the attack surface and optimize performance by executing code in an enclave only when absolutely necessary.

We implemented the Privagic framework for Intel SGX and the C language. Although we only added color annotations for the C language, the Privagic compiler itself is language agnostic. It relies on the LLVM compiler and can handle any language, provided that a front-end for this language exists for LLVM (e.g., Rust, Go, Haskell, Fortran, etc.).

We evaluate Privagic with a legacy application (memcached) and several data structures. Overall, our evaluation of Privagic shows that:

- Privagic can scale to a production-ready application such as memcached. Protecting memcached with Privagic requires the modification of 9 lines of code.
- Privagic divides the TCB by more than 200 as compared to fully embedding memcached in an enclave with Scone [6].
- The code generated with Privagic is up to 10 times more efficient than embedding the whole application in an enclave with Scone.
- The code generated with Privagic is sometimes more efficient than the unprotected code because of parallelism.

To summarize, this thesis makes thus the following contributions:

- It introduces explicit secure typing, which makes the use of multiple colors, multiple threads and pointers possible;
- It proposes the Privagic compiler, which relies on secure typing to automatically partition an application for Intel SGX, based on manually added colors;
- It proposes the FastSGX runtime, which executes in parallel an application generated by the Privagic compiler;
- It evaluates Privagic and shows that secure typing (i) offers a strong foundation to enforce confidentiality and integrity, (ii) eases the use of trusted execution environment in legacy applications, (iii) reduces the TCB as compared to fully embedding an application in an enclave, and (iv) improves performance by allowing the Privagic runtime to execute the generated code in parallel.

Organization of the thesis:

The rest of the thesis is organized as follows.

Background

Chapter 2 provides technical background about two major components of the thesis: Intel SGX-SDK and LLVM. The TEE on Intel processors is called SGX (Software Guard Extensions). The primary approach to using Intel SGX relies on a SDK (Software Developement Kit) provided by Intel. The Intel SDK manipulates enclaves, a protected memory zone. The first section of the chapter presents this toolkit. It describes the creation of enclaves and communication between them, and then presents SGX switchless call, which is a specific technique to enhance performance. The second section presents the LLVM compilers by introducing and providing an overview of LLVM IR, LLVM Pass, and annotations. Techniques and notions that are showcased in this chapter will be used in further chapters.

Related works and motivation

In chapter 3, our technique of secure typing is assessed with various existing methods, including those utilizing SGX and those dedicated to data partitioning. In this chapter, at first, we discuss briefly the threat model, where an attacker fully controls a machine including the operating system and hypervisor. Various applications rely on Intel SDK to use SGX. Some of them use switchless calls to reduce high communication call costs. However, these cases notoriously add a burden to the developer. Many frameworks propose to simplify the use of Intel SGX based on two techniques: (i) providing a new language abstraction, and (ii) automatic partitioning of the application. Nevertheless, these frameworks have a set of limitations. The second section of this chapter discusses the related works, different techniques used to simply the use of Intel SGX, their limitations, and finally compare them with our technique of explicit secure typing. The final section focuses on the data flow analysis technique, one of the major techniques used for the automatic partitioning of the application.

FastSGX: a message-passing based runtime for SGX

Chapter 4 depicts our first contribution, which is named FastSGX. FastSGX is a message-passing based runtime for SGX independent from Privagic. The FastSGX runtime is inspired from the switchless call of SGX-SDK. Switchless reduces communication call costs between the outside world and the enclaves by relying on communication channel located in shared memory. As switchless calls, FastSGX allows inter-enclave in shared-memory communication too. However, compared to switchless calls, FastSGX makes the threads and messages apparent to the developer, which allows the developer to significantly optimize the application, i.e., by avoiding wasting CPU cycles with idling threads, by allowing two enclaves to communicate directly, and by allowing the code located outside the enclave to run in parallel with the code located inside. At low-level, FastSGX implements the communication channels with lock-free data structures. It also implements a lock-free memory manager for the messages. We evaluate FastSGX using two classical data structures: a hashmap and a treemap. The evaluation compares the performance of the data structure (with one and two enclaves) executed using SGX-SDK, SGX-SDK with switchless calls and FastSGX. The main takeaways of this chapter are: (i) FastSGX has a simple usable interface with only four main functions, (ii) FastSGX can be used to design efficient multi-enclave applications, where existing switchless calls prove to be particularly inefficient. (iii) FastSGX systematically has better performance than SGX-SDK and SGX-SDK with switchless calls because of increased parallelism and of the use of lock-free data structures. This chapter discusses in detail FastSGX. First, it presents the related works, followed by the design of FastSGX, and finally presents the evaluation.

Colors and color management in Privagic

Moving forward, in the pivotal chapter 5, we unveil the intricacies of how Privagic analyses the application. Privagic operates on LLVM bitcode files, representing the entire LLVM Intermediate Representation (IR) of the application, generated conventionally by tools like clang for C language. The primary objective of Privagic is to analyze LLVM IR, focusing on the identification of confidentiality, integrity, and Iago attack vulnerabilities. The

analysis is facilitated by a pass derived from ModulePass. This chapter unfolds the intricacies of Privagic's analysis of LLVM IR, starting from the source code. The narrative progresses from the detection of an element's color, to type inference, color initialization, color compatibility, an overview of the analysis, explanations of typing rules, treatment of function calls, and concludes with potential error messages raised by Privagic.

Application partitioning

Chapter 6 describes the subsequent step, the partitioning of LLVM IR into distinct bitcode files, utilized for the generation of a partitioned executable through conventional compilation tools. FastSGX is adapted with a different set of functions to create Privagic runtime. This chapter first provides an overview of application partitioning, followed by a description of the various stages of the rewriting process.

Evaluation

Chapter 7 highlights the evaluation of Privagic. The evaluation aims to answer the questions related to the engineering effort while using Privagic, to the trusted computed base generated by Privagic, and to the performance of the application deployed using Privagic. We showcase the evaluation results of Memcached and of data structures.

Limitations and perspectives

In chapter 8, we explore the limitations of Privagic. Concurrently, we examine potential axes for the development of future works. The discussion encompasses various features, including the message-passing technique of our runtime, the number of threads used during execution, multi-colored structure, and multi-language aspects of Privagic.

Conclusion

Chapter 9 serves as the concluding chapter, summarizing our thesis work.

Chapter 2

Background

Contents

2.1 Developing with the Intel SDK

The most basic approach to using a TEE on Intel processors relies on a SDK (Software Developpement Kit) provided by Intel. The Intel SDK manages enclaves. As shown in Figure 2.1, an enclave is a protected contiguous memory zone located inside the virtual address space of a process. The trusted part of the application — both code and data — is placed inside the enclave. The untrusted part of the application stays in the memory region outside the enclave.

An Intel processor protects the memory of the enclaves by defining two processor modes: an enclave mode and a normal mode. In normal mode, the processor prevents any access to the memory of the enclaves. Preventing

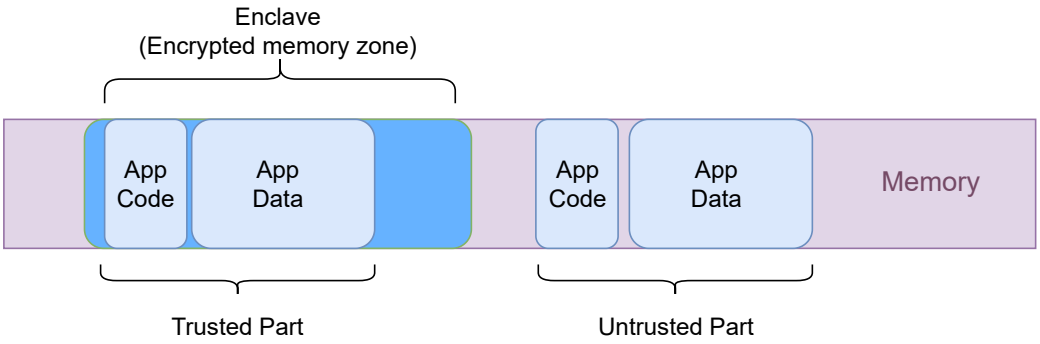


Figure 2.1: SGX enclave in memory

read access enforces confidentiality while preventing write access enforces integrity. When the processor enters the enclave mode, it gains access to a single enclave, which we call the active enclave. In this case, the processor can access untrusted memory, the memory of the active enclave, but not the memory of the other enclaves.

Preventing read and write access at the processor level is not enough to protect the enclaves. In detail, an attacker may gain full control of another hardware component, such as a PCIe device. The attacker can use this device to bypass the protection of the processor, i.e., to directly access the memory of an enclave in DMA. For this reason, when the processor operates in enclave mode, it enforces confidentiality by encrypting/decrypting the cache lines that belong to the active enclave when they cross the boundary of the CPU package. The processor also enforces integrity by maintaining a tree of cryptographic hashes [51], which is used to detect unintended writes.

2.1.1 Ecall & Ocall

A developer uses ecalls (enclave calls) and ocalls (outside calls) to switch the processor between the normal and the enclave mode. The untrusted part of an application makes an ecall to execute a function inside the enclave. Figure 2.2 illustrates the principle: an ecall is a call made from the untrusted part of the application toward the enclave (from the left side to the right side).

At a low level, an ecall behaves as a kind of system call. In detail, an enclave maintains an ecall table, which contains pointers to the functions

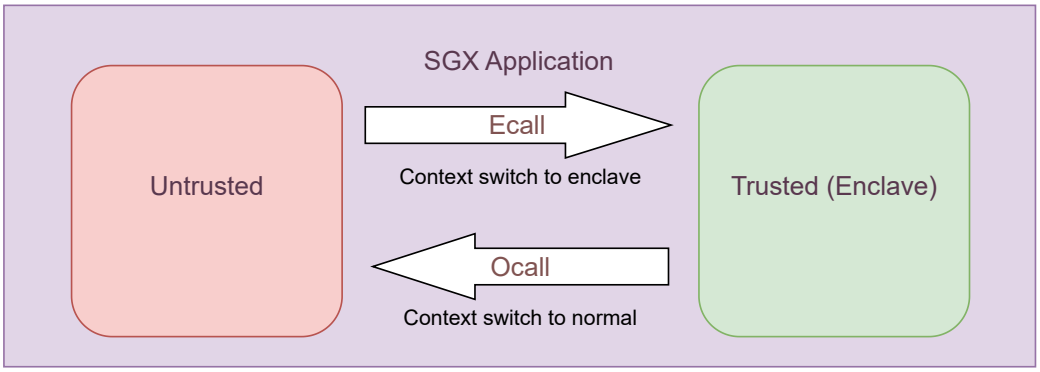


Figure 2.2: SGX Ecall and Ocall

```
1. enclave {
2.     trusted {
3.         public void ecall_inside([in] int *arg1, [user_check] int *arg2);
4.     };

5.     untrusted {
6.         void ocall_outside([in, out] int *arg3);
7.     };
8. }
```

Figure 2.3: A simple example of the SGX EDL.

located in the enclave and exposed to the untrusted part of the application. An ecall takes as argument an index `idx` in this ecall table, and the ecall transfers the control to the function at index `idx` after having switched the processor mode. As a result, the untrusted part of the application doesn't have any direct access to the function's code: it can only interact with an enclave through the ecall table.

An ocall is a call from within the enclave towards the outside part of the application (from the right side to the left side in Figure 2.2). Ocalls are mostly used because the operating system is unsafe and thus located in untrusted memory. When an enclave has to execute a system call, e.g., to access a file or to take a lock, it has thus to switch back the processor to normal mode. For that, it uses an ocall. The ocall switches from enclave mode to normal mode, and, similarly to an ecall, takes as argument an index in an ocall table, which gives the called function. The untrusted part of the

application executes those system calls and returns the results to the enclave for further execution of enclave code.

In order to generate the ecalls and ocalls, the developer uses a language named EDL (enclave description language). With EDL, a developer designs an application as a distributed system with two independent components: the unsafe and the enclave part of the application. The components communicate through an interface written in EDL. The interface describes the functions that the other components can call.

Figure 2.3 presents a simple example of an EDL file. The functions which will be placed inside the enclave are defined as trusted (for example `ecall_inside`). If the ocall function's or ecall function's arguments are pointer type, the user needs to specify at least one attribute from the following to define the nature of the pointer.

ECALL:

- [in]: In line 3, a buffer with the same size of `arg1` (size of int) will be allocated inside the enclave. The data from the pointer `arg1` is copied towards the allocated buffer. The modification will be done to the newly allocated buffer. The data pointed by `arg1` remains unchanged.

- [out]: In the case of [out] attribute, a buffer with the same size of the pointer will be allocated inside the enclave. The data from the pointer will not be copied, instead, the newly allocated buffer will be initialized to zero. After the end of the trusted function's execution, the data of the newly allocated buffer will be copied towards the argument pointer.

- [in, out]: In the case of [in, out] attribute, a buffer with the same size of the pointer will be allocated inside the enclave. The data from the pointer will be copied towards the buffer. After the end of the trusted function's execution, the data of the buffer will be copied towards the argument pointer.

- [user_check]: In line 3, the pointer `arg2` will not be verified, the user needs to make sure of the safety of the data passed through this pointer. The buffer of the pointer `arg2` is not copied to the enclave.

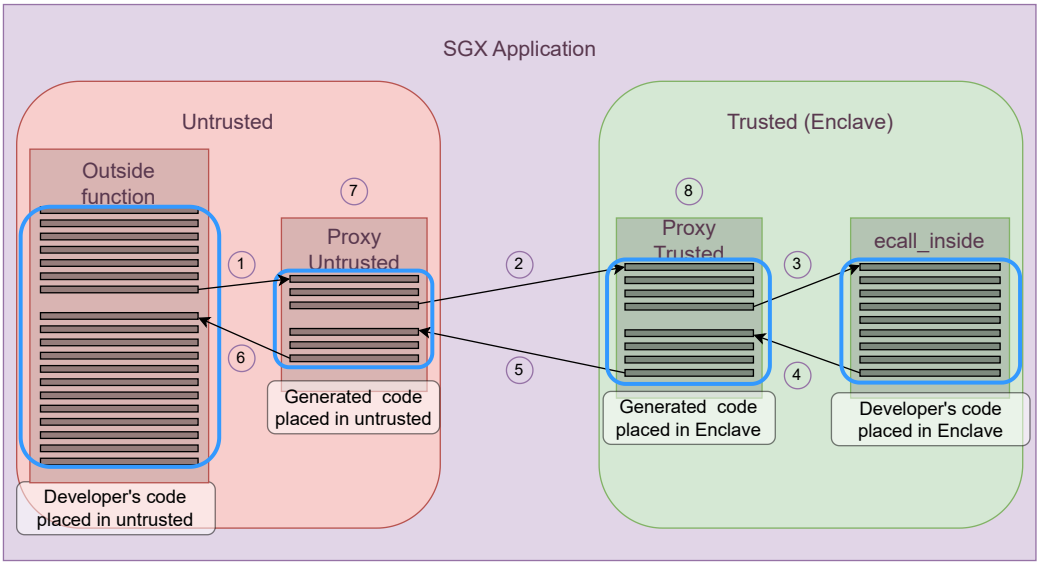
OCALL:

- [in]: A buffer with the same size of the pointer will be allocated in the outside memory (untrusted). The data from the pointer is copied towards the allocated buffer. The modification will be done to the newly allocated buffer. The data inside the enclave remains unaffected.
- [out]: In the case of [out] attribute, a buffer with the same size of the pointer will be allocated in the outside memory. The data from the pointer will not be copied, instead, the newly allocated buffer will be initialized to zero. After the end of the untrusted function's execution, the data of the newly allocated buffer will be copied towards the argument pointer inside the enclave.
- [in, out]: In line 6, a buffer with the same size of the pointer `arg3` will be allocated outside. The data from the pointer will be copied towards the buffer. After the end of the untrusted function's execution, the data of the buffer will be copied towards the argument pointer `arg4` inside the enclave.
- [user_check]: The pointer will not be verified, and the data pointed by the pointer will not be copied to outside. If the pointer points towards enclave memory, the outside function cannot use it to access the enclave memory.

At line 3 of Figure 2.3, the argument `arg1` of an ecall function `ecall_inside` has an attribute `in`. A single argument can have multiple attributes as in line 6, where `arg3` has `in` and `out` as its attributes.

The developer compiles an EDL interface with a compiler provided by Intel SDK. This compiler generates a stub to call the functions provided by the other components, and a skeleton in charge of receiving the call and dispatching it to the actual implementation provided by the developer. The stubs in the untrusted part are generated in `Enclave_u.h` and `Enclave_u.c` (Represented as Proxy Untrusted in Figure 2.4). The stubs in the trusted part are generated in `Enclave_t.h` and `Enclave_t.c` (Represented as Proxy trusted).

Figure 2.4 shows the way an ecall is executed. `ecall_inside` from Figure 2.3 is taken as an example. The Outside function from the left side



1. `ecall_inside(enclave_id, arg1, arg2)`
2. `sgx_ecall(enclave_id, ecall_number, &ocall_table_Enclave, &ms)`
3. `ecall_inside(arg1, arg2)`
4, 5, 6. Propagates back the return value
7. `Enclave_u.h, Enclave_u.c`
8. `Enclave_t.h, Enclave_t.c`

Figure 2.4: An example of ecall execution in SGX

makes an ecall(1). The Outside function is written by the developer. It calls the ecall `ecall_inside` with `enclave_id` as the first argument, followed by its actual arguments. `enclave_id` represents a unique identification number associated with an enclave (as SGX can manipulate multiple enclaves simultaneously). A proxy generated by the compiler in the untrusted part (7) receives the ecall `ecall_inside`. The untrusted proxy function will call the function inside the enclave through the trusted proxy generated inside the enclave (2). The context will be switched from normal to the enclave. The trusted proxy function (8) will call the actual trusted function `ecall_inside` written by the developer (3). The return value will also be passed through the proxies (4, 5, 6). The Proxies manage the encryption/decryption of arguments and return values according to provided attributes. The context will again be switched from enclave to normal. The Outside function will

continue its execution from the point where ecall was made.

Ocall follows the same pattern, where a trusted function initiates the call from inside the enclave.

2.1.2 Sgx Switchless

The cost of entering or leaving an enclave makes designing an efficient application for Intel SGX is difficult. This cost is prohibitive: while a standard call costs only a few cycles, entering or leaving an enclave costs 7000 cycles [41, 55, 56].

In order to reduce the cost to enter or leave an enclave, a developer can use a technique named *switchless call* [6, 52, 56, 62, 63]. A switchless call consists of leveraging *worker threads* in order to avoid switching the processor from the normal mode to the enclave mode. In detail, each worker thread runs in a *security domain*: either the non-secure domain or in an enclave. In order to perform a call from one domain to another, a worker thread of one domain sends a message to a worker thread in the other. To send this message, the worker thread simply writes a value in a shared memory zone named an activation zone.

Transferring the control between one domain and another costs a single cache miss: the cache miss that loads the activation zone from the core of the sender thread into the core of the receiver thread. Since transferring a cache line costs a few hundred cycles instead of several thousand, a switchless call significantly improves performance compared to switching the processor mode.

2.2 LLVM

LLVM(Low Level Virtual Machine) is an open-source compiler infrastructure. Figure 2.5 gives an overview of the global LLVM infrastructure. It can be divided into two major components: frontend and backend. LLVM plays a role as a backend of the compiler. As shown in Figure 2.5 the compiler takes source code from different languages like C and C++ and provides executables for different instruction set architectures (ISA) of processors like x86, and ARM. The frontend of the compiler, clang, clang++ converts the source

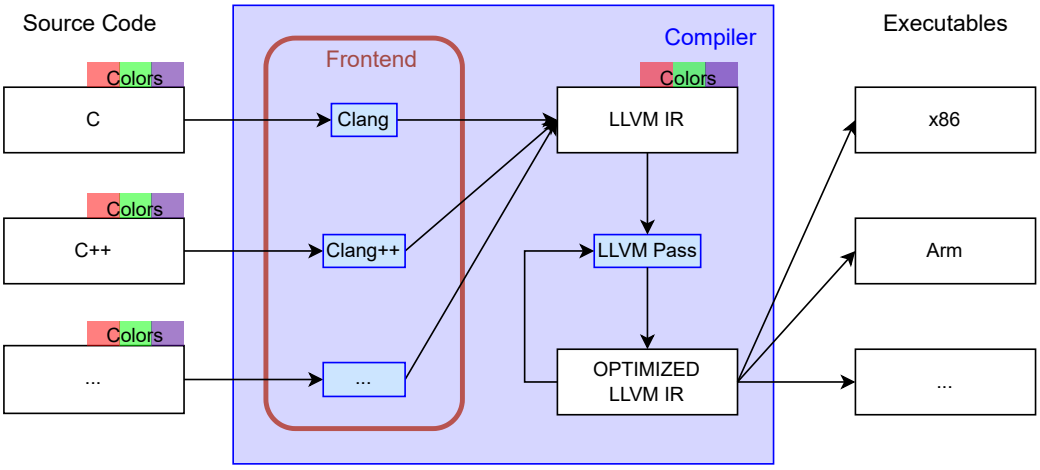


Figure 2.5: Overview of LLVM Compiler

code into LLVM IR (intermediate Representation). The backend takes LLVM IR as an input and applies multiple optimization passes called `LLVMPass` in order to generate an optimized LLVM IR. Finally, the compiler generates a binary specific to the given architecture. Considering `n` number of languages and `m` number of architectures, instead of requiring `n*m` different compilers, only `n` frontends and `m` backends are needed.

To implement secure typing we work on LLVM, the backend of the compiler, precisely we work with LLVMPass. We first discuss LLVM IR in §2.2.1, then LLVM Pass in §2.2.2 and conclude with annotations in §2.2.3

2.2.1 LLVM IR

LLVM considers a machine with a memory and an infinite number of typed registers. An LLVM instruction takes registers as input and outputs a new register. A register is assigned once, which means that an instruction and its output register are equivalent. Figure 2.6 presents an example of LLVM code. LLVM IR has multiple granularity to represent the code. Few among them are the following:

- **Module:** represents the entire compiled code unit. It serves as the highest-level container for all other objects in the LLVM IR.
- **Function:** represents a function or method in the source code. In-

```
1. @y = global i32 0              ; int y = 0;

2. define i32 @test(i32 %0)  {  ; int test(int a) {
3.   %2 = alloca i32             ;   int x;
4.   %3 = add i32 %0, 42         ;
5.   store i32 %3, i32* %2       ;   x  =   a + 42;
6.   store i32 %3, i32* @y       ;   y  =   a + 42;
7.   %4 = call i32 @f(i32* %2)  ;
8.   ret i32 %4                  ;   return f(&x);
9. }                              ; }

10. declare i32 @f(i32*) #1       ; extern int f(int*);
```

Figure 2.6: Registers and memory in LLVM.

ternally, LLVM represents a function by a control flow graph of basic blocks [1, 2]. The control flow graph of a function consists of the oriented graph of the basic blocks of the function connected by the jumps and conditional jumps. A function starts its execution in a special basic block named the *entry point* of the function. It also holds information about its type and return type. Lines 2 to 9 of Figure 2.6 show an example of a function. There are different kinds of functions in LLVM IR, some of the majorly used kinds are:

- **Internal Function:** the function code is defined in the module.
- **External Function:** the function code is defined outside the module (eg. line 10 of Figure 2.6)
- **Intrinsic Function:** the function code is replaced, in place, by a machine-specific sequence of instructions when LLVM emits the native code. LLVM uses the intrinsic functions when it can replace a function call by a machine-specific optimized code. This is, for example, the case of a call to the function `memcpy`, which can be replaced inline by an optimized sequence of `rep/movsd` on an Intel processor. An intrinsic function always starts with a `'llvm.'` prefix and is always defined externally. The body of an intrinsic function is, by construction, empty. A few samples of LLVM intrinsic functions are `llvm.va_start` (initializes arguments list), `llvm.prefetch` (insert prefetch instruction) and `llvm.sin.f32`

(return `sin` of a float value) etc.

- **BasicBlock:** is a sequence of instructions without a jump and that do not contain instructions that are the target of a jump, except for the first and the last instruction. Therefore it has a single entry point and exit point (terminator). Terminator instructions are either `BranchInst` or `ReturnInst` (e.g., line 8 of Figure 2.6). If a function has multiple basic blocks, each entry point is associated with a label, which is used by instructions or by other basic blocks to refer to it. Basic block also holds information about its predecessor basic blocks if they exist.

- **Instruction:** represents a single operation in the program. A typical instruction is the `add` instruction at line 4, which adds 42 to the register `%0` (the parameter of the function). This instruction outputs its result in the register `%3`. In addition to the registers, a instruction can access memory. It can create new memory locations by calling a `malloc` function. Following are some of the major instruction types:

  - **AllocaInst:** The code can also create a local variable with the `AllocaInst` (line 3 in Figure 2.6). It allocates memory in the stack.

  - **LoadInst:** In order to access memory, a function uses the `load` instruction: `r = load(p)` loads the value pointed by the register `p` in the register `r`

```
%1 = load i32, i32* @global
```

  - **StoreInst:** In order to write in the memory, a function uses the `store` instruction: `store(v, p)` stores the value of the register `v` in the value pointed by the register `p` (e.g. lines 5 or 6 in Figure 2.6).

  - **GetElementPtrInst:** is used to retrieve the address of a subelement of an aggregate data structure such as arrays, structs, and vectors. The following example shows that: the type of register `%1` is `struct.A*`; `getelementptr` instruction get the second element (`i32 1`) of the `struct.A*`; and returns the register `%2` that points to the element.

```
%struct.A = type { i32, i64 }

%2 = getelementptr %struct.A, %struct.A* %1, i32 0, i32 1
; struct_a->second
```

– **CallInst:** represents a function call. It encapsulates information like called function, arguments, and any associated attributes.

```
declare i32 @f(i32*) #0   ; extern int f(int*);

%4 = call i32 @f(i32* %3) ; int result = f(&a);
```

– **ReturnInst:** returns a value (possibly void) at the end of function execution. A function can carry multiple `ReturnInst`. It is a terminator instruction, hence it is always placed at the end of a basic block.

– **BranchInst:** transfers the control flow from the current basic block to another basic block inside the same function. It is also terminator instruction. `br` instructions are either conditional or simple jump. In the following example, if the result carried by the register `%1` is true, execution jumps to the basic block labeled as `true_label` otherwise to the basic block labeled as `false_label`

```
br i1 %1, label %true_label, label %false_label
```

– **PHINode:** is used in the case of SSA (Single Static Assignment) graph representation of LLVM IR. When a basic block has multiple predecessors, `phi` instruction selects a value based on the control flow graph. It selects the result based on the branch taken. All the `phi` instructions will be placed at the top of the basic block, no `non-phi` instruction is allowed between the first and the last `phi` instructions of a given basic block. In the given example `phi` instruction returns `a_val` if the taken branch is `a_label`, and returns `b_val` if the taken branch is `b_label`. The result of `phi` instruction is placed in register `%10`

```
%10 = phi i32 [ %a_val, %a_label ], [ %b_val, %b_label ]
```

- **GlobalVariable:** is created by the keyword `global` (line 1 in Figure 2.6). `global` returns a register that points to the variable.

2.2.2 LLVMPass

An LLVMPass is a unit of compilation applied to an LLVM IR. LLVM passes are a major component of the backend of the compiler. A pass takes a LLVM IR as an input and outputs another IR. As shown in Figure 2.5, multiple LLVM passes can operate during the compilation process before the final output. A LLVMPass can perform various tasks. LLVM passes can be applied for the transformation, analysis, or optimization of the LLVM IR.

- **Analysis:** An analysis pass does not alter the LLVM IR but gathers information. This information may be used by later passes. Some examples are Basic CallGraph Construction (basiccg), Dependance Analysis (da) and Dominator Tree Construction (domtree).

- **Transform:** This kind of pass transforms the IR often in order to improve performance. LLVM provides some predefined transformation passes focused on code optimization. Some common optimization passes are Dead Code Elimination (dce), Dead Store Elimination (dse) and Global Variable Optimizer (globalopt)

- **Utility:** A utility pass performs various additional tasks without impacting Analysis or Transform passes. This kind of pass is often used by developers for understanding and debugging LLVM IR. Some instances include View CFG of function (view-cfg), Module Verifier (verify) and Assign names to anonymous instructions (instnamer)

LLVM provides an interface to write customized LLVM passes. These passes could be inherited from different classes depending on the needed granularity. Within the scope of this thesis, we concentrate on ModulePass. ModulePass is a type of pass that is applied to the whole module. It uses the entire module i.e. the entire program as a single unit, in order to analyze or transform the LLVM IR. It can access global objects such as Global Variable or Functions without any particular order. Therefore this pass allows to add or remove entities at the global level. ModulePass can access and apply function level passes (e.g. dominators to retrieve the dominator tree of the

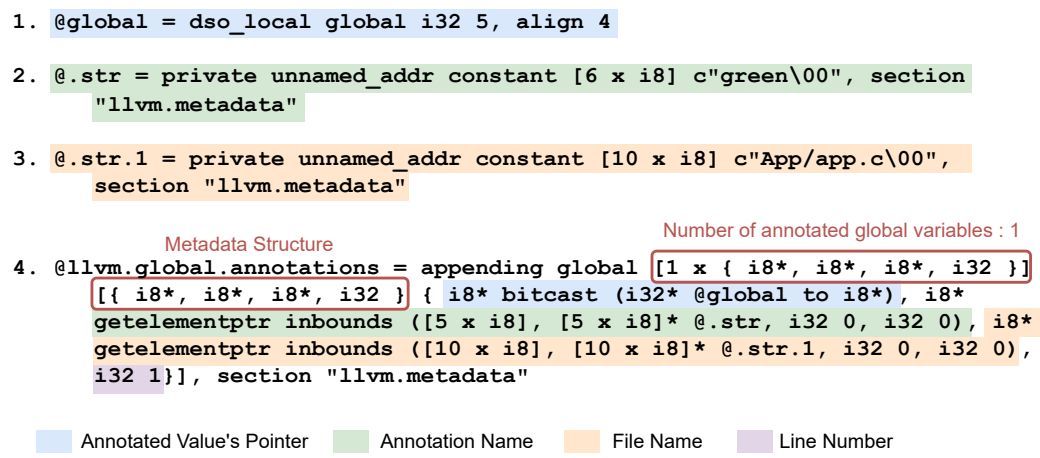


Figure 2.7: Example of Global Variable Annotation

given function as an analysis result). Modifications performed in the LLVM IR should be carefully handled. Even a small mistake will result in a broken LLVM IR. Nevertheless, ModulePass offers great liberty to developers to rewrite the code. Every element of LLVM IR can be added or removed, including metadata information. Modification is prohibited for some elements such as structure type. Once initialized a structure type cannot be modified.

2.2.3 Annotation

The compiler Clang LLVM permits to add annotation to some of the LLVM IR elements. In the thesis, we intensively use annotation to add colors to the variables, e.g., to indicate to which enclave belongs a variable.

An example to add an annotation to a variable in C language is the following:

```
1. int __attribute__((annotate ("green"))) global = 5;
```

The frontend of the compiler, clang, translates annotations in the LLVM IR. Clang doesn't intercept these annotations, it is forwarded towards the backend of the compiler, LLVM. LLVM uses annotation for special purposes, for example, code optimization. These annotations are ignored by the compiler unless any LLVM passes explicitly use them. Therefore, annotation does not disturb the compilation chain and does not alter the generated executable.

```
1. %15 = bitcast i32* %14 to i8*
2. %16 = call i8* @llvm.ptr.annotation.p0i8(i8* %15, i8* getelementptr
    ↪ inbounds ([6 x i8], [6 x i8]* @.str, i32 0, i32 0), i8*
    ↪ getelementptr inbounds ([10 x i8], [10 x i8]* @.str.1, i32 0, i32
    ↪ 0), i32 15)
3. %17 = bitcast i8* %16 to i32*
```

Figure 2.8: Example of Pointer Annotation

In LLVM IR annotations are primarily represented in three ways: Global annotation, Pointer annotation, and Variable annotation.

Global Annotation

Figure 2.7 shows an example of global variable annotation. Here the Global Variable named `global` is annotated with the annotation `'green'` (see the annotation example above). At line number 1 the global variable `global` is declared and initialized as 5. At line number 2, `@.str` is a global string that is declared and initialized as `'green'`. This is the annotation string. At line number 3, `@.str.1` is a global string that is declared and initialized as `'App/app.c'`. This is the file name of the code. `llvm.global.annotation` at line 4 is a Global Variable which englobes annotation details of all Global Variables. As shown in the figure it is an array of `Metadata Structure` specific to annotation. In this example, only one global variable is annotated (`[1 x i8*, i8*, i8*, i32 ]`). In the case of `n` annotated Global Variables, the first part will be `[n x i8*, i8*, i8*, i32 ]`. The first element of the metadata structure contains a pointer to the annotated value (line 1), the second contains the pointer to the annotation string (line 2), the third contains the pointer to the file name (line 3), and finally, the fourth contains the line number of C code where the variable is declared (line number where `global` is declared in C file is 1). The Global Variable `llvm.global.annotation` will be processed by LLVM Pass to associate Global Variables with their annotation. The annotation string `'green'` can be retrieved from the global string `@.str`, by getting its initialization value.

Pointer Annotation

```
1. %6 = alloca i32, align 4
2. %10 = bitcast i32* %6 to i8*
3. call void @llvm.var.annotation(i8* %10, i8* getelementptr inbounds ([6
   ↪  x i8], [6 x i8]* @.str, i32 0, i32 0), i8* getelementptr inbounds
   ↪  ([10 x i8], [10 x i8]* @.str.1, i32 0, i32 0), i32 21)
```

Figure 2.9: Example of Variable Annotation

Figure 2.8 illustrates an example of Pointer annotation. This annotation is used to convey annotation in specific cases like annotating an element of the structure. `llvm.ptr.annotation` is an intrinsic function that takes four arguments and returns a pointer type texttti8* (line 2). The function arguments represent the same data as the metadata structure from llvm.global.annotation except for the first argument. In this case, the first argument is `i8* %15`. The register `%15` is a BitcastInst which casts the register `%14` from `i32*` to `i8*` (line 1). The annotated value is actually the register `%14`. As the first argument should be `i8*`, it passes by a BitCastInst. Then again the return value of `llvm.ptr.annotation` which is stored in register `%16` passes through another BitCastInst (line 3), to cast back to the annotated values type. For further execution, the register from `%17` (line 3) will be used instead of the actual register `%14`

Variable Annotation

Figure 2.9 shows an example of Variable annotation. It is used to annotate local variables which are defined using AllocaInst (line 1). `llvm.var.annotation` is also an intrinsic function that takes four arguments similar to `llvm.ptr.annotation`, whereas the return type is `void` (line 3). Resembling to Pointer Annotation, the annotated value, register `%6` is passed by a BitCastInst (line 2), before used as the first argument. Whereas there is no return value for the function call. For further execution, the register `%6` will be used completely independently to `llvm.var.annotation`

Chapter 3

Motivation and related work

Contents

In the remainder of the manuscript, we concentrate on the TEE, Intel SGX. Different techniques are proposed to use SGX. These techniques have their own set of advantages and drawbacks. As presented in the introduction §1, we propose Privagic, which leverages secure typing to automatically partition an application.

Privagic ensures application security by enforcing typing rules and by automatically partitioning the application according to data access. As the data type is enriched with an enclave identifier in the form of a color, both colored data and the code accessing those data are placed inside the enclave of the corresponding color.

In this chapter, we compare our technique with some of the existing techniques, both for using SGX and for data partitioning. This chapter gives a brief presentation of the threat model, discusses these previous works related to Privagic, and finishes by focusing data flow analysis technique.

Tool	Technique	Language	Starting point	Partitioning granularity		Multiple partitions	Multiple threads	Language coverage
				Code	Data			
Glamdring [34]	Abstract interpretation [23]	C	Func. args.	Function	Global variable	No	No	Complete
Privtrans [13]	Use-def chains [1]	C	Function	Function	Incorrect[1]	No	No	Incomplete[1,4]
Treillis [37]	Call graph [1]	C	Func./glob. var.	Function	Incorrect[1]	No	No	Incomplete[1]
ProgramCutter [58]	In-vitro execution	C	None	Function	Incorrect[1]	No	No	Incomplete[1,4]
SeCage [36]	Taint analysis	C	Local variable	Function	Incorrect[1]	No	Yes	Incomplete[1]
Montsalvat [61]	Points-to analysis [5]	Java	Function	Java class	Java class	No	No	Complete
Civet [54]	Points-to analysis [5]	Java	Java class	Java class	Java class	No	No	Complete
Uranus [31]	Symbol table	Java	Function	Java class	Java class	No	Yes[2]	Complete
Rubinov at al. [45]	Taint analysis [7]	Java	Variable	Java class	Java class	No	No	Incomplete[3]
GoTEE [28]	Prog. dependence graph [26]	Go	goroutine	goroutine	Global variable	No	Yes	Complete
PtrSplit [35]	Prog. dependence graph [26]	Agnostic	Global variable	Function	Global variable	No	No	Complete
SecV [60]	In-vitro execution	Agnostic	Function	Function	Object	No	No	Incomplete[4]
Our contribution	Language typing	Agnostic	Type	Instruction	Field	Yes	Yes	Complete

[1]: does not support pointers
[2]: does not support thread local storage
[3]: the paper states that the technique can only handle 86% of the applications
[4]: does not inspect all the code

Table 3.1: Automatic partitioning tools. No previous tool handles both multi-threaded applications and explicit pointers. They also do not support multiple enclaves because they over-approximate the variable locations that may contain secure values.

3.1 Threat model

We consider an attacker that fully controls a machine (operating system and hypervisor included). We assume however that the attacker cannot read or write the memory of an enclave protected by Intel SGX. Therefore, we trust that the processor, the Privagic runtime, and the software development kit provided by Intel to use SGX are correct and do not contain bugs.

Privagic can run in two modes: relaxed and hardened mode. In relaxed mode, Privagic ensures the confidentiality and integrity of the sensitive values of an application. In hardened mode, Privagic additionally prevents Iago attacks [15] against the enclaves.

The application is not trusted and may contain bugs. Privagic has the goal of minimizing the TCB in order to reduce the probability of bugs inside the enclave.

Privagic does not address the problem of side-channel attacks, since Intel SGX does not address this attack vector.

3.2 Related work

Various works propose to simplify the use of Intel SGX. Some application relies only on the Intel SDX to use SGX. Many frameworks are also proposed to make SGX more user-friendly. Some frameworks use different techniques to partition applications for security purposes. Hereafter, we first discuss applications only depending on Intel SDK, then we present some frameworks for using Intel SGX and different partitioning techniques. We discuss the limitations of these techniques. Finally discuss the use of type information.

3.2.1 Using Intel SDK

As shown in Table 3.1, many applications rely only on the Intel SDK (§2.1) to use SGX. [12, 16, 25, 33, 48, 59, 66]. These applications offer a high-level of safety because they have a small TCB and because the developers of these projects manually ensured their security. However, manually ensuring that sensitive data never escapes from an enclave is difficult. Designing an application as a distributed system with components that communicate through an interface also adds a burden on the developer. These applications are as

efficient as they can be. However, they often heavily depend on ecalls/ocalls, which are notoriously costly (see our evaluation in §4.3.2). Optimizing the communication between an enclave and the unsafe part of an application is possible by using switchless calls [6, 41, 47, 52, 56, 62, 63], but adds even more burden on the developer.

3.2.2 Frameworks that simplify the use of Intel SGX

In order to ease the use of SGX, several frameworks propose to run a complete application with its dependencies in an enclave [6, 10, 38, 43, 53]. This approach leads to a large unsafe TCB that can reach tens of megabytes (see §7.2.2).

In order to avoid a large TCB, two techniques are proposed. The first technique consists of defining new language abstractions or new programming models [47, 49]. This approach may ensure a high level of safety, but they require a complete rewriting of the application, this makes them inadequate for legacy applications with a large code base.

In a different context, language abstractions were proposed to ensure the confidentiality of sensitive values in a distributed system. Some tools automatically partition applications written in the Jif language [17, 64, 65]. Jif is a language based on Java, in which the developer explicitly indicates how data flows between partitions that do not trust each other. This also requires a complete rewrite of the application.

The second technique automatically partitions an application. Privagic belongs to this category. Table 3.1 gives an overview of the automatic partitioning techniques used in previous works. These works target Intel SGX, AMD's SME, ARM Trustzone, but also privilege separation [46], i.e. isolating dangerous functions in another process. All these works are based on data flow analysis techniques. The developer annotates sensitive variables or functions, a tool analyzes how the data flows in the application: by using various techniques that we do not detail here, such as, use-def chains [1], abstract interpretation [23], in-vitro execution [58, 60], points-to analysis [5, 9, 24, 44, 50, 57], program dependence graphs [26] or taint analysis [7, 36].

A first limitation of the above framework is that data flow analysis analyzes an application *sequentially*, inspecting instructions one after the other.

Because of the difficulty to track pointers in the presence of threads most of the tools do not handle multi-threaded applications (see, e.g., [14] for the abstract interpretation engine used in Glamdring [34], [36] or [7] for taint analysis, or [57] for points-to analysis). The only exceptions are GoTEE [28] for the Go language and Uranus [31] for the Java language. These tools rely heavily on specific language constructs: goroutine for GoTEE and Java classes for Uranus. They do not address languages with explicit pointers. In summary, no technique is currently available to handle multiple threads in the presence of explicit pointers.

A second limitation is that, data flow analysis has difficulty with pointers or aliasing [5,14,34,35]. Several works simply behave incorrectly in the presence of pointers or references. Those that do correctly handle pointers and references, over-approximate the memory locations that may contain sensitive values. As a result, these techniques are not able to split an application into more than two partitions, since the same memory location maybe falsely marked as belonging to two partitions.

3.2.3 Using typing to enforce isolation

With secure typing, we avoid the limitations of data flow analysis techniques. The secure type of a value directly indicates in which enclave the value lives, which avoids any over-approximation, and allows Privagic to know the enclave of any memory location, regardless of the presence of pointers or threads.

Using language typing to enforce memory isolation has already been explored in several works [8,11,27,30] These systems enforce isolation by using typing: they ensure that a process cannot forge a pointer to an unauthorized memory. Previous works rely on classical typing to prevent unauthorized memory access. In contrast we propose to embed security information directly in the type system.

3.3 Focus on glamdring

Several frameworks rest upon data flow analysis to partition the application. Data flow analysis identifies where the application stores sensitive

```
1. struct shared {
2.   int other;
3.   int secret;
4. }* shared;

5. #pragma glamdring sensitive-source(secret)
6. void inside(int secret) {
7.   shared = malloc(sizeof(*shared));
8.   shared->secret = secret;
9. }

10. void outside() {
11.   // *shared allocated in enclave => SIGSEGV
12.   shared->other = 42;
13. }
```

Figure 3.1: Limitation of data flow analysis.

data. However, tools that rely on data flow analysis become fragile when they partition the application.

In this section, we discuss the limitations of data flow analysis through the example of Glamdring framework [34]. Figure 3.1 presents a code for which Glamdring misbehaves. At line 5, the developer tags `secret` as a sensitive source. Glamdring relies on the eva/impact plugins of frama-c [21] to compute the locations in the source code that are impacted by the value of `secret`. By using abstract interpretation [23], the plugin adequately identifies that lines 3 and 8 are impacted. Glamdring then partitions the code by relying on the slicing plugin of frama-c. This plugin identifies the statements required to correctly execute the code identified by the impact plugin. It identifies that the declaration of the `shared` variable at line 4 and that lines 7 are required to execute line 8. Since Glamdring partitions the application at the function granularity, it puts thus the `shared` variable and the code of `inside` in the enclave.

At this step, Glamdring misbehaves because the `outside` function is executed outside the enclave. This function accesses the `shared` variable, which is allocated inside the enclave at line 7, and thus is not accessible by `outside` since outside is executed in normal mode.

The issue comes from the `shared` data structure. This structure contains both sensitive and non-sensitive values, which are not handled by a data

flow analysis tool. In order to solve the issue, a transformation tool should associate additional information to tag `secret` as sensitive and `other` as non-sensitive. Adding this information is the very goal of secure typing. Secure typing consists of tagging a type with a color, which indicates whether a value is sensitive or not. With secure typing, a compiler can adequately partition the application presented in Figure 3.1: by re-organizing the `shared` data structure as we do with Privagic (see §6.4), or by promoting `other` as sensitive and putting all the code that accesses `other` in an enclave.

Chapter 4

FastSGX: a message-passing based runtime for SGX

Contents

Our first contribution is FastSGX a message-passing based runtime for SGX. FastSGX is used to implement the Privagic runtime, but is independent to Privagic.

As discussed in the §3 entering and leaving an enclave consumes a huge number of CPU cycles. The cost is prohibitive: while a standard function call costs only a few cycles, entering or leaving an enclave costs approximately 7000 cycles [41,55,56]. Because of this cost, designing an efficient application for Intel SGX is difficult.

Several research works show that we can avoid this cost by using *switchless calls* [6, 52, 56, 62, 63]. A switchless call leverages *worker threads* in order to

avoid switching the processor from the non-secure mode to the secure mode. In detail, each worker thread runs in a single *security domain*: either the non-secure domain or in an enclave. In order to perform a call from one domain to another, a worker thread of one domain sends a message to a worker thread in the other. To send this message, the worker thread simply writes a value in a shared memory zone named an activation zone. Transferring control from one domain to another costs a single cache miss: the cache miss that loads the activation zone from the core of the sender thread in the core of the receiver thread. Since transferring a cache line costs a few hundred cycles instead of several thousand, a switchless call significantly improves performance.

Although the switchless call is well known, using this technique efficiently remains challenging for three reasons.

The first is that the worker threads are hidden to the developer and configured statically. In detail, the SGX runtime uses worker threads internally to optimize the time to transfer the control from/to an enclave. A worker thread consumes CPU resources even when idle because it actively spins on the activation zone. Unfortunately, the developer can configure the number of worker threads only statically. This is inadequate if the workload evolves over time. In such a case, either the developer over-provisions the number of worker threads, which wastes CPU resources when the workload is low, or under-provisions the number of worker threads, which leads to inefficiencies when the workload increases [62].

The second issue comes from the function semantic exposed by the SGX runtime provided by Intel. With a call semantic, the caller is suspended during a call. The caller uselessly wastes a CPU while actively waiting for the termination of the call. The developer can thus not use the wasted CPU resource to execute useful code in parallel.

The third issue comes from the design of the current SGX runtimes, which prevent a direct call from one enclave to another. Instead, a worker thread has to first indirectly transfer control to a non-secure worker.

In FastSGX, we use the switchless call technique more efficiently with a new programming model. In detail, we propose to explicitly design an SGX application as a distributed system with worker threads that communicate by exchanging messages. Each worker thread runs in a single security domain. However, the developer can create and destroy worker threads on the fly to

adapt the number of worker threads to the workload. Moreover, instead of a function-call abstraction, we expose a message-passing abstraction. Thanks to this, a worker thread can send a message and continue to execute while another worker thread processes the message. This avoids wasting the CPU of the sender during a call. Finally, by exposing to the developer an interface to send and receive messages, a developer can send a message from any worker thread to any other worker thread. This avoids indirects an inter-enclave call through the non-secure domain.

Our message-passing runtime FastSGX implements communication channels with lock-free data structures, and also implements a lock-free memory manager to allocate and free messages.

We evaluate FastSGX with two classical data structures: a hashmap and a treemap. We evaluated versions of these data structures with one and two enclaves. Our evaluation with different access patterns shows that:

- The interface of FastSGX, with its four main functions, is simple enough to be usable in practice,
- FastSGX supports multi-enclave applications, in contrast to current switchless call runtimes, which are especially inefficient in this case.
- The data structures implemented with FastSGX consistently outperform the equivalent data structures implemented with the Intel SDK.

The remainder of the chapter is organized as follows: §4.1 discusses related work, §4.2 presents the design of FastSGX, §4.3 details our evaluation, and §4.4 concludes.

4.1 Related work

Many applications rely directly on the Intel SDK to use SGX [12, 16, 25, 33, 48, 59, 66]. Since using the Intel SDK can be complex for legacy applications, several frameworks support running the complete application, with all its dependencies, inside an enclave [6, 10, 38, 43, 53]. This approach leads to an overly large trusted computing base. Other tools propose to automatically partition an application by starting from variables or functions annotated as sensitive [13, 28, 31, 34–37, 45, 54, 58, 60, 61]. These tools ease the development

while minimizing the trusted computing base. These tools are complementary to FastSGX: they could rely on FastSGX to optimize the time to transfer the control from/to an enclave.

As presented in the introduction, several runtimes rely on switchless calls to avoid the cost of switching the processor from/to secure mode [6, 52, 56, 62, 63]. These runtimes hide the worker threads to the developer, which makes the dynamic optimization of the number of worker threads difficult, prevents the execution of code in parallel in the worker threads, and is sub-optimal for a multi-enclave application. FastSGX exposes the worker threads and a message-passing interface to the developer, thus avoiding these three limitations.

EActors [47] designs a SGX application as a set of actors. EActors runs worker threads in the enclaves, which executes eactors, and communicate by messages. Using EActors requires a whole redesign of an application in order to implement the application with actors. Using FastSGX is more straightforward since only adding calls to send and receive messages is required. Moreover, with FastSGX, the developer explicitly creates on the fly the worker threads, which allows the developer to dynamically adjust the number of worker threads to the workload. This is not the case with EActors. With EActors, the number of worker threads is configured statically, which is inadequate for an application with a dynamic workload because worker threads may become useless in case of a low workload, or saturated in case of a high workload.

4.2 Design of FastSGX

FastSGX is a message-oriented runtime for Intel SGX. It manages a set of enclaves. FastSGX associates an enclave with a communication channel, implemented as a lock-free FIFO queue stored in unsafe memory [29]. FastSGX also implements a lock-free memory allocator in order to allocate and free the messages. For that, FastSGX uses a simple lock free stack [29].

FastSGX also manages a set of worker threads. A worker thread executes within a single enclave. It receives messages through the communication channel. If several worker threads are associated to the same enclave, they share the communication channel. Each message is received by a single

```
1. // initialize the runtime with nenclaves enclaves
2. int initialize(size_t nenclaves, ...);

3. // create a worker thread in the enclave eid
4. int new_worker(pthread_t* tid, size_t eid);

5. // send the value to the enclave eid with the message id mid
6. void send(size_t eid, size_t mid, union value value);

7. // receive a message with the message id mid from eid
8. void recv(size_t eid, size_t mid, union value* value);
```

Figure 4.1: Main functions of the FastSGX interface.

worker thread.

4.2.1 Interface

Figure 4.1 presents the main functions provided by FastSGX. To initialize the runtime, a developer calls the initialize function. Its arguments are number of enclaves to be created and, for each enclave, a path to the binary to be loaded in it (eid). The initialize function returns an enclave identifier for each enclave, by starting at eid 1. FastSGX considers that the pseudo-enclave with the eid 0 represents the code and data located in unsafe memory. It also considers that the main thread of a process, i.e., the thread that called initialize, is a worker thread associated to enclave 0.

To start executing code in the enclaves, the developer calls the `new_worker` function. This creates a new worker thread, in the enclave given as an argument, by executing the function named `start_routine`, located in the binary loaded at initialization.

A worker thread in an enclave can send and receive by exchanging messages, with the `send` and `recv` functions. The arguments of `send` are the destination enclave (eid), a message identifier (mid), and a content (value). The message identifier is used to address a message to a specific `recv` function, which is useful to avoid confusing two messages with two different meanings received in parallel. The value is a union that has the size of a machine word (64 bits on Intel).

The `recv` function has the same three parametern types. If eid is equal to

```
1. struct message* dequeue(struct mbox* mbox) {
2.   struct message* res;
3.   struct message* expected;

4.   do {
5.     res = mbox->head;
6.     expected = res;
7.   } while(res == NULL
8.           !atomic_compare_exchange_strong(&mbox->head,
9.                                           &expected,
10.                                          res->next));

11.  return res;
12. }
```

Figure 4.2: Basic incomplete lock free dequeue function.

-1, the function receives messages from any enclave, otherwise, it only receives messages from eid. If mid is equal to -1, the function receives messages with any mid, otherwise, it only receives messages with the mid given as a parameter. The `recv` function blocks until a matching message is received. The `recv` function removes the message from the channel, fills the value with the content of the message, and returns.

4.2.2 Hazard pointers

The ABA problem [39] appears with lock-free data structures implemented with compare and swaps, such as FastSGX's lock-free queues.

To illustrate ABA problem, we first present a naive algorithm used to implement a lock-free queue in Figure 4.2. Suppose the queue contains two messages A and X. To dequeue A, a receiver thread t_1 loads a pointer to A from the head of the queue (line 5). It then loads a pointer to X from the `next` field of A, and replaces the head by the pointer to X (line 8). In case another receiver thread would also dequeue A after t_1 the reads head on line 5 but before the update at line 8, t_1 updates the head from A to X only if the head is still equal to A. For that, t_1 uses an atomic compare-and-swap instruction, which atomically checks that head is A (`expected` at line 9), and replaces head by X (`res->next` at line 10). This simple algorithm is correct only if a message is never freed. In FastSGX, a receiver frees a message after

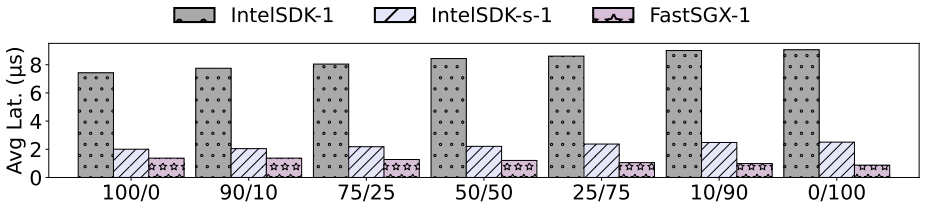
it is consumed, in order to avoid a memory leak. This leads to the ABA problem described next.

Suppose again that the queue contains two messages A and X. The receiver thread t_1 loads A and X, and then, another receiver thread t_2 is scheduled. t_2 dequeues A and X, and frees the messages. Then, a sender thread t_3 inserts a new message B. At this step, the queue contains a single message: B. When t_1 is re-scheduled, the compare-and-swap is supposed to fail because B is not A. However, this is not necessarily the case: since A is free when t_3 allocates a message, B may be allocated at the same location as A. In such a case, the compare and swap of t_1 succeeds. Now t_1 incorrectly thinks that A is still the head of the list. t_1 thus installs a pointer to X as the head of the queue, which is incorrect because X was freed by t_2.

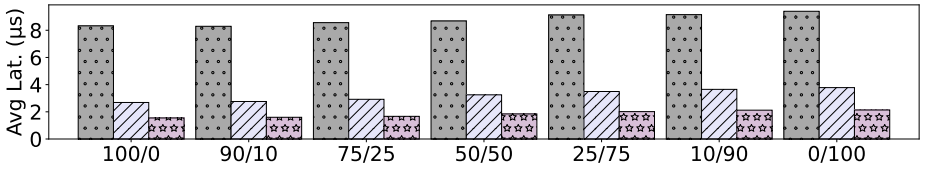
FastSGX avoids the ABA problem by using hazard pointers [39]. When a receiver loads the head pointer, it signals to the other threads that freeing the pointed message is unsafe. For this, the receiver thread records the head pointer in an array named the hazard pointer array. Then, to free a message, FastSGX uses two lists: a purgatory list and a free list. When the application frees a message, FastSGX adds the message in the purgatory list. When an application allocates a message, FastSGX uses the free list. If the free list is empty, FastSGX tries to move a batch of N messages from the purgatory list to the free list. It inspects eash message in the purgatory list, and, if the message is still referenced by the hazard array, FastSGX ignores the message since a receiver may still use the message. Otherwise, FastSGX moves the message to the free list.

4.3 Evaluation

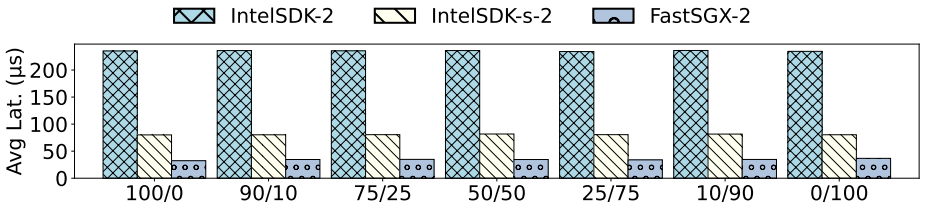
We evaluate the performance of FastSGX on an Intel i5-9500 CPU 3 GHz with 16 GiB memory. This 6-core CPU ships SGX version 1 with a maximal memory size usable by the enclaves of 93 MiB. The machine runs Linux 5.15.0, glibc 2.31, clang 10.0.0, and Intel SGX SDK 2.19.100.



(a) Hashmap with 1 enclave (zipfian access pattern)



(b) Treemap with 1 enclave (zipfian access pattern)



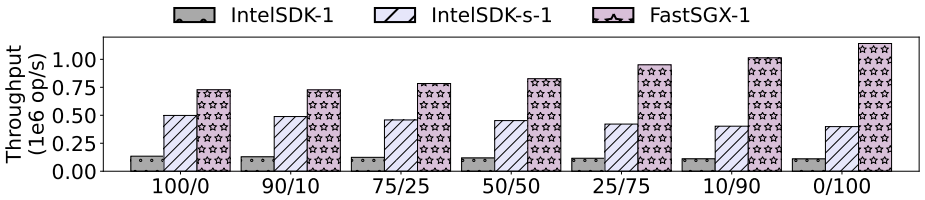
(c) Hashmap with 2 enclaves (uniform access pattern)

Figure 4.3: Latency of the data structures. X/Y: get/put ratio.

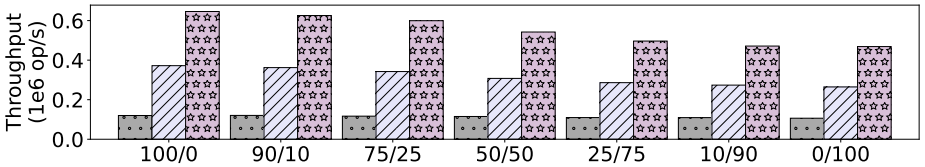
4.3.1 Micro-benchmarks

We first evaluate two maps that associate keys to values: a hashmap and a treemap. The hashmap uses a separate chaining algorithm: it is designed as an array of linked-lists, in which each linked-list contains the keys that collide. The treemap is implemented as a red-black tree, which ensures that the tree remains balanced.

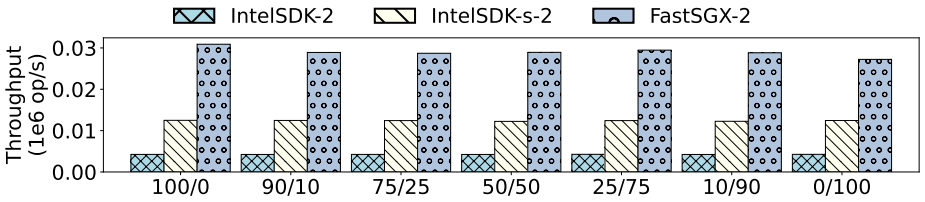
We evaluate three versions with a single enclave: IntelSDK-1, IntelSDK-s-1 and FastSGX-1. These versions store the whole map in a single enclave. The two IntelSDK versions expose the `put`/`get` functions to the non-secure domain. IntelSDK-1 switches the processor mode during a call, while IntelSDK-s-1 uses switchless calls. FastSGX-1 is implemented with FastSGX. For a `get`, the worker thread in the non-secure domain sends a message to execute the get in the enclave, and waits for the result. For a `put`, since the result of a `put` is not used, the worker thread in the non-secure domain sends a message to execute `put`, but continues its execution in parallel.



(a) Hashmap with 1 enclave (zipfian access pattern)



(b) Treemap with 1 enclave (zipfian access pattern)



(c) Hashmap with 2 enclaves (uniform access pattern)

Figure 4.4: Throughput of the data structures. X/Y: get/put ratio.

We also evaluate the same three versions with two enclaves. These versions store the keys in one enclave and the values in another, which makes the communication pattern between the enclaves more complex. As with a single enclave, we evaluate (i) IntelSDK-2, which switches the processor mode, (ii) IntelSDK-s-2, which uses switchless calls, and (iii) FastSGX-2, which uses message passing.

Figure 4.3 reports the latency and Figure 4.4 the throughput with different `put`/`get` ratios. With one enclave, we execute 100 000 operations in a map with 100 000 keys, and with two enclaves, 20 000 operations in a map with 20 000 keys. A key is 8-bytes long and a value 1024-byte long.

With one enclave, we observe that, as expected, IntelSDK-s-1 consistently performs better than IntelSDK-1. We also observe that FastSGX-1 is consistently better than IntelSDK-1 and IntelSDK-s-1. The better performance of FastSGX-1 comes from two complementary phenomenons.

First, with a high number of puts, a put is executed in parallel with

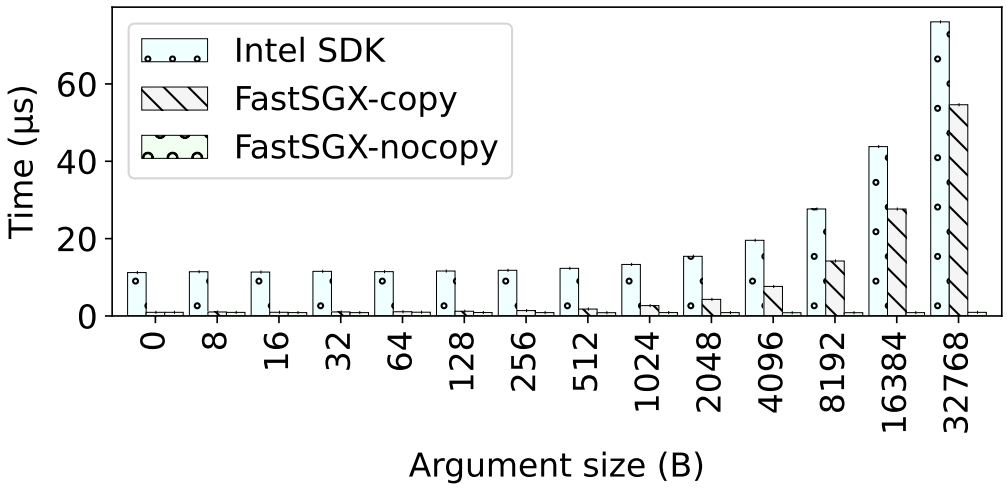


Figure 4.5: Ping-pong time with Intel SDK and with FastSGX.

the load injector in FastSGX-1. This is not the case with IntelSDK-s-1, which suspends the caller during a call. The higher parallelism of FastSGX-1 explains why FastSGX-1 performs better than IntelSDK-s-1 with a high number of puts (right of the curves).

Second, FastSGX is designed with lock-free data structures whereas Intel SDK uses a blocking scheme. In detail, with IntelSDK-s-1, a thread takes a spin-lock when it accesses an activation zone. FastSGX-1 does not take a lock since it uses lock-free data structures. Thanks to the use of lock-free data structures, FastSGX-1 is also better than IntelSDK-s-1 with a high number of get operations (left of the curve).

With two enclaves, we observe similar results. With two enclaves, the relative difference between FastSGX and the Intel SDK versions is even higher because FastSGX allows the two enclaves to communicate directly. This is not the case with the Intel SDK versions, which pay an additional transfer of control to the non-secure domain for each operation. This result confirms that using an explicit message-passing scheme is important to optimize a multi-enclave application.

4.3.2 Ping-pong

In this experiment, we compare the cost of switching the processor with the cost of exchanging messages. For that, we evaluate a ping pong application in three configurations: FastSGX/copy, FastSGX/reference and Intel SDK.

With FastSGX/copy and FastSGX/reference, we run two worker threads: one worker thread in the non-secure domain and one worker thread in an enclave. The non-secure worker thread sends a ping to the enclave worker thread, which answers with a pong message. The ping message contains a pointer to an array of bytes. In FastSGX/copy, the application copies the pointed array in a buffer before replying with a pong. In FastSGX/reference, the application ignores the argument.

With Intel SDK, the non-secure thread of the application calls a ping function provided by the enclave. The non-secure thread of the application is blocked during the execution of the ping function, which is equivalent to waiting for a pong. With Intel SDK, the buffer is copied from the non-secure domain into the enclave.

Figure 4.5 reports the results of the experiment. We first observe that, when the size of the argument is equal to 0, using message passing instead of mode switching divides the function execution time by 12 (11.2 µs for Intel SDK versus 0.9 µs for FastSGX/copy or FastSGX/nocopy). This result highlights the benefit of using messages instead of switching the processor mode.

We also observe that, when the argument size increases, the cost of copying the buffer becomes larger than the cost of transferring the control. With a buffer of 32 KiB, we observe, however, that using message passing still saves 28% of the time (76.1 µs for Intel SDK versus 54.6 µs for FastSGX/-copy). This result shows that, even for an application that protects large user data sets by copying them in an enclave, using message passing remains interesting.

4.4 Conclusion

Our first contribution is FastSGX, a message-based runtime for SGX. FastSGX relies on the switchless call principle, but avoids the limitations of the current implementations. In detail, FastSGX (i) allows the developer to adjust the

number of worker threads to the actual workload on the fly, (ii) allows the developer to execute code in parallel in the sender while a receiver proceeds a message, and (iii) allows the developer to directly transfer the control from an enclave to another. Thanks to these properties, our evaluation with different data structures and workloads shows that FastSGX consistently outperforms the SGX development kit of Intel, and with one or two enclaves. Our evaluation also shows that FastSGX, with its 4 main functions, is simple and usable in practice.

Chapter 5

Colors and color management in Privagic

Contents

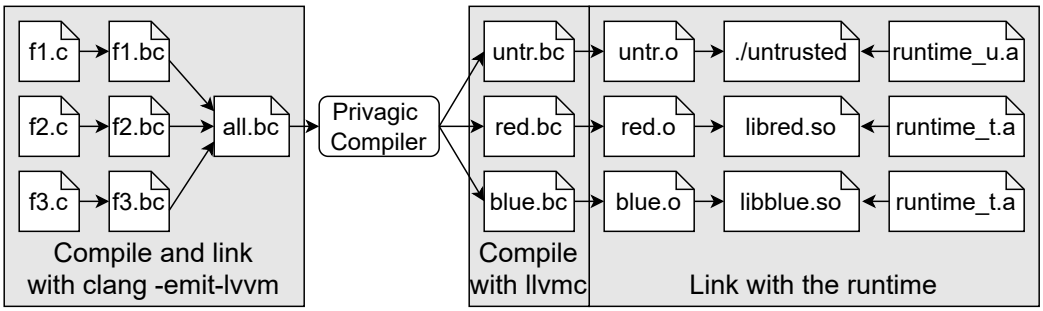


Figure 5.1: Overview of Privagic

Privagic relies on explicit secure typing to partition an application. Secure typing consists on a type enriched with a domain identifier named a color. Since secure typing is explicit, Privagic does not have to analyze the application to find the locations that may contain sensitive values. Thanks to that, secure typing avoids the limitations of the current analysis tools, which are unable to analyze a multi-threaded application because they cannot explore all the possible thread inter-leavings.

Since explicitly modifying the type of each variable and each function argument of a legacy application is painful, Privagic also implements a simplified form a type inference. In detail, Privagic is able to deduce by inference the colors of the local variables and of the arguments of the functions, but only if the code does not create a pointer to these memory locations. Thanks to this limitation, the variable or argument is only used by a single thread, and only within a single function. Using type inference in this case is straightforward: a simple data flow analysis performed at the scale of the function is enough to identify the color of a variable or argument. Note that, in the general case, type inference is a specific instance of the more general taint analysis problem. Even if theoretically, the two problems are equivalent, pragmatically, by restricting type analysis to local variables and arguments, we make the technique usable in practice since Privagic does not have to explore the different thread inter-leavings. This is not the case of the previous taint analysis tools (e.g., Glamdring [34]), which try to solve the problem of taint analysis in the general case, and can thus not handle multi-threaded applications since they cannot explore all the possible thread inter-leavings.

In the remainder of the chapter, we first present an overview of Privagic, and we then delve into various aspects of our system's analysis process. Section §5.2 introduces how our system detects an element's color. Following this, §5.3 illustrates the type inference, while §5.4 details the color initialization process. In §5.5 we discuss the color compatibility and §5.6 provides an overview of the analysis. The chapter further explores typing rules in §5.7, elucidates the treatment of function calls in §5.8, and finally §5.9 concludes with possible error messages raised by Privagic.

5.1 Overview

Privagic is designed as a part of the backend of the LLVM compiler. As shown in Figure 5.1, Privagic takes a LLVM bitcode file as input. This LLVM bitcode file contains the whole LLVM Intermediate Representation (IR) of the application. It is generated by a classical tool chain, e.g., `clang` for the C language. Privagic first analyzes the LLVM IR in order to detect confidentiality, integrity, or Iago attack issues. Privagic then partitions the LLVM IR into several bitcode files, which are used to generate an executable with a classical compilation tool chain.

Privagic can run in two modes. In hardened mode, Privagic enforces confidentiality and integrity, and additionally prevents Iago attacks: an enclave never uses an unsafe value computed outside an enclave. In this mode, Privagic totally isolates an enclave from the rest of the application. The rest of the application cannot access the enclave, and the enclave cannot access the memory of the rest of the application, neither in read nor in write.

The hardened mode is sometimes too restrictive. The developer may for example want to store profiling data in unsafe memory directly from an enclave in order to avoid crossing the enclave boundary to update them. The developer may also want to access global data structures located outside the enclave after having checked their integrity instead of storing the data structures inside the enclave in order to minimize the enclave size. For this reason, Privagic also proposes the relaxed mode. In this mode, Privagic ensures confidentiality and integrity, but it does not prevent against Iago attacks. In other words, the rest of the application cannot access an enclave (confidentiality and integrity), but an enclave can access unsafe memory,

either in read or in write.

5.2 Color detection

As presented in section §2.2.1, LLVM considers a machine with a memory and an infinite number of typed registers. A LLVM instruction takes registers as input and outputs a new register.

Privagic allows the developer to enrich a type with a color. Figure 1.1 (see §1) illustrates an example of adding colors to a type. The `color` keyword is a macro. It transforms the color declaration into a generic C annotation.[1] The `clang` frontend does not handle by itself the annotation: it simply emits the annotation in the LLVM IR. Privagic uses these annotations to identify the colors associated with the types.

Colors can be added to:

- Global Variables
- Local Variables
- Structure Fields

As described in section §2.2.3, according to annotated types, clang translates the annotations in three different forms in LLVM IR: Global annotation, pointer annotation, and variable annotation. Colors are associated with global variables using global annotation and local variables using variable annotation.

5.2.1 Structure Fields

Detecting the color that is added to the structure field is not as straightforward as detecting the colors of global or local variables. Figure 5.2 shows a simple example of colored structure in C. Figure 5.3 is the LLVM IR representation of the code for line number 5 of Figure 5.2. In Figure 5.3, line 1 loads the global variable `sa` of type `struct A`. Then line 2 gets the first field of the structure `struct A`. The type of register in line 2 is `i32*`. In order to add pointer annotation to the register, line 3 casts the register from `i32*` to

[1] E.g., `__attribute__((annotate ("blue")))`.

```
1. struct A {
2.   int color(green) a;
3. };
4. struct A* sa;
5. sa->a = 5;
```

Figure 5.2: A simple C example of colored struct

```
1. %13 = load %struct.A*, %struct.A** @sa, align 8
2. %14 = getelementptr inbounds %struct.A, %struct.A* %13, i32 0, i32 0
3. %15 = bitcast i32* %14 to i8*
4. %16 = call i8* @llvm.ptr.annotation.p0i8(i8* %15, i8* getelementptr
   ↪ inbounds ([6 x i8], [6 x i8]* @.str, i32 0, i32 0), i8*
   ↪ getelementptr inbounds ([10 x i8], [10 x i8]* @.str.1, i32 0, i32
   ↪ 0), i32 15)
5. %17 = bitcast i8* %16 to i32*
6. store i32 5, i32* %17
```

Figure 5.3: Example of structure field's color

`i8*`. Line 4 calls `llvm.ptr.annotation` function and adds the color to the register `%15` (line 3). The section §2.2.3 explains in detail the specifications of pointer annotation. Then at line 5, the colored register is cast from `i8*` to its previous type `i32*`. For further use, the register `%17` (from line 5) is taken. Therefore the register `%17` is used by the StoreInst in line 6. In order to associate the color to the field, Privagic has thus tracked backward from the pointer annotation at line 4 up to the first field of the structure `struct A`.

5.3 Type inference

In order to ease the use of Privagic in a legacy application, Privagic can infer the color of the local variables and registers. Privagic only infers the color of a local variable if the code does not create a pointer to the local variable. Moreover, Privagic does not try to infer the color of global variables or fields of the data structures. Thanks to these restrictions, Privagic only has to infer the type of a variable locally accessed by a single function and a single thread. These restrictions allow thus Privagic to scale to multi-threaded

Color	Given to	Compatible with
F (Free)	Registers and instructions	Any other color
U (Unsafe)	Memory locations	No color
S (Shared)	Memory locations	No color (but becomes F when loaded)

Table 5.1: Initial colors given to the uncolored elements.

applications.

Privagic implements this design by first unifying the registers and the local variables, and only if the code does not create a pointer to the local variable. For that, Privagic starts by promoting the local variables (i.e., the `alloca`) into registers by using the `mem2reg` pass of LLVM [1]. This pass exactly implements our need: it promotes a local variable only if the code does not create a pointer to the local variable. After this transformation pass, Privagic thus only has to infer the colors of the registers in order to infer the colors of both the registers and local variables (without pointers to the local variable).

5.4 Initial colors

Initially, Privagic gives a color to each register, instruction, and memory location. When a register or a memory location has an explicit color in the code, Privagic uses it. When the color is not explicit, Privagic uses the three special colors presented in Table 5.1.

The F color is the initial color of the instructions and of the registers without explicit color. The F color is the only color compatible with any other color. The F color indicates that a register or an instruction is not bound to an enclave, i.e., that it can safely be computed and used in any enclave. The F color is also used for type inference: Privagic can change the color of a F element when it visits an instruction.

The U and S colors are used for the memory locations without explicit color. U and S indicate that the memory location is located in unsafe memory. Both colors are incompatible with any other color. However, the S color has a special property: it becomes F when it is loaded from memory into a register.

In hardened mode, Privagic colors an uncolored memory location in U,

which protects against Iago attacks since the U color is incompatible with the colors of the enclaves. In relaxed mode, Privagic colors the memory location in S, which allows an enclave to use a value loaded from a S memory location since the value becomes F when it is loaded.

Note that a developer can use the S color explicitly for some variable in hardened mode to relax the Iago constraints, or the U color explicitly in relaxed mode to add Iago constraints.

5.5 Color compatibility

While Privagic analyzes the code, it enforces typing rules. For that, Privagic verifies the compatibility between colors. Two colors $\overline{x}$ and $\overline{y}$ are compatible, which we note $\overline{x} \sim \overline{y}$, if they are equal or if one of them is F. When Privagic detects that two colors are incompatible while they should be, Privagic reports an error to the developer.

5.6 Overview of the analysis

Privagic analyzes the code by inspecting the LLVM instructions one after the other.

When Privagic inspects a direct call to a *local function*, i.e., a function for which the IR is available in the analyzed file, Privagic specializes the called function with the colors of the arguments. For example, in line 7 in Figure 2.6, since `%2` is F, Privagic generates a specialized version of `f` for a F argument. Thanks to the specialized versions, Privagic considers that the colors of the arguments of a function are always known when it visits a function. While Privagic visits a specialized function, it verifies that the returned values are all compatible.

Privagic starts the analysis of an input file from entry points. An entry point is a function that may be called from another codebase. In order to handle the case where Privagic generates a library, Privagic considers by default that any function with the `extern` attribute is an entry point. The developer can also explicitly give the entry points by using annotations (e.g., only the `main` function). For each entry point, Privagic considers that it is

Mode	Instruction	Color propagation		#
		Registers	Instruction	
both	$r = load(p)$	$\overline{*p} \neq \text{S} \Rightarrow r \leftarrow \overline{*p}$	$ins \leftarrow \overline{r}$	1
	$r = op(x_0,\ ...)$	$\forall i, r \leftarrow \overline{x_i}$	$ins \leftarrow \overline{r}$	2
	$store(p, v)$	$\overline{v} \sim \overline{*p}$	$ins \leftarrow \overline{*p}$	3
	$x_n = ins(x_0,\ ...)$	$ins \in \mathcal{B} \Rightarrow x_n \leftarrow \overline{\mathcal{B}}$	$ins \leftarrow \overline{\mathcal{B}}$	4
hardened	$r = load(p)$	$\overline{p} \sim \overline{*p}$	No effect	5
	$store(v, p)$			

$\overline{x} \sim \overline{y} \Leftrightarrow ((\overline{x} \neq \overline{y}) \wedge (\overline{x} \neq \text{F}) \wedge (\overline{y} \neq \text{F}) \Rightarrow \text{error})$
$x \leftarrow \overline{y} \Leftrightarrow (\overline{x} \sim \overline{y}) \wedge (\overline{x} = \text{F} \Rightarrow x \text{ takes the color } \overline{y})$
ins: any instruction *op*: *ins* that is not a store, a load or a call

Table 5.2: Secure type system ($\overline{x}$ is the color of x).

called from U and that its arguments are U in hardened mode, and F in relaxed mode.

When Privagic analyzes an instruction, it propagates the colors. This propagation can change the input colors of instructions that are already visited. That may happen in the case of a recursive call or of a backward jump. For that reason, Privagic uses a fixpoint algorithm. In detail, Privagic runs one or several full analysis passes on the whole IR of the application. Each pass propagates the colors by starting from the entry points. At the end of a pass, if Privagic inferred new colors during the pass, it restarts a new analysis pass. Otherwise, the fixpoint algorithm stops.

5.7 Typing rules

The code analysis has three different goals. First, the analysis has the goal of inferring the colors of the F registers. Then, the analysis has the goal of enforcing the correctness of the code by enforcing confidentiality and integrity in relaxed mode, and by additionally preventing Iago attacks in hardened mode. Finally, the analysis has the goal of associating a color to each instruction. Privagic uses this color in the partitioning phase to know in which enclave it has to emit an instruction.

```
1. int x = 0, y = 0;
2. int color(blue) b;

3. void f() {
4.   if(b == 42)        // basic block A
5.     x = 1;           // basic block B (indirect leak)
6.   y = 2;             // basic block C (non sensitive)
7. }
```

Figure 5.4: Indirect color propagation.

In order to achieve these goals, Privagic enforces the rules presented in Table 5.2, which are discussed in the remainder of the chapter.

5.7.1 Confidentiality

Privagic prevents a direct leak with rules 1 to 3. These rules first enforce confidentiality by ensuring that if an instruction accesses a value colored in C, then the instruction is executed in the enclave C (fourth column of Table 5.2). Additionally, these rules enforce confidentiality by propagating, instruction after instruction, the colors from the loads to the stores. In detail, for a load, Rule 1 ensures that the output takes the color of the loaded memory location. This rule has a special case when the memory location is S. In that case, the output register remains F, which makes the register compatible with any other color. Rule 2 propagates the colors from the right to the left for an operation that is not a load, a store or a call. Rule 2 also prevents the use of two inputs with incompatible colors. Without this constraint, executing the instruction without leaking a colored value would be impossible since one of the enclaves would have had access to a value with another color. Rule 3 prevents a direct leak through a store by reporting an error when a value is stored in a memory location with an incompatible color.

Rule 4 handles indirect leaks. An indirect leak happens when a conditional jump is controlled by a colored value. Figure 5.4 illustrates the issue. In this code, if an attacker observes that `x` is equal to 1, the attacker can deduce that `b` is equal to 42.

Without giving the implementation details, rule 4 prevents indirect leak by assigning a color to each basic block.[2] When Privagic visits a conditional

[2]A basic block is a sequence of instructions without jump and that do not contain

jump controlled by a colored value, it assigns the color of the value to the basic blocks of the "if" and "then" branches (basic bloc $\mathcal{B}$ in Figure 5.4). It does not propagate the color to the join point of the "if" (e.g., $\mathcal{C}$ in Figure 5.4) since this basic block does not carry sensitive information anymore. Rule 4 ensures that if a basic block $\mathcal{B}$ has the color C, then the output register of any instruction that belongs to $\mathcal{B}$ has a color compatible with C. For example, in Figure 5.4 in relaxed mode, Privagic reports an error: `x` cannot take the blue color of $\mathcal{B}$ at line 5 since `x` is S (line 1). Rule 4 also ensures that the instructions that belong to a basic block with the color C are executed in enclave C.

5.7.2 Integrity

Privagic enforces integrity by ensuring that only a code executed in enclave C can modify a value with the color C. For that, first, if an instruction outputs a C value, Privagic generates the instruction in the C enclave (rules 1 and 2). Then, Privagic generates a store to a C memory location in the enclave C (rule 3).

5.7.3 Iago attacks

In order to prevent Iago attacks, Privagic ensures that an instruction generated in some enclave C uses only values located or computed in the enclave C. For that, Privagic first ensures that using a F register in an enclave is safe by replicating the computation of the F registers in each enclave. Then, Rule 1 ensures that a value loaded from unsafe memory takes the color U in hardened mode, which ensures that the unsafe value cannot be used to compute a C register thanks to rule 2. Rule 5 complements the other rules by preventing the use of cross-enclave pointers. In detail, this rule ensures that if the value pointed by p has the color $\overline{*p}$, then $\overline{p}$ is compatible with $\overline{*p}$.

instructions that are the target of a jump, except for the first and the last instruction [1].

5.8 Function calls

As already stated, Privagic handles a direct call to a local function by generating a specialized version of the function. This section discusses the other forms of calls.

5.8.1 Direct call to an external function

A function is external if Privagic cannot analyze its code, i.e., if its IR is not contained in the analyzed IR file. Privagic considers by default that an external function belongs to the untrusted part of the application. When Privagic visits an external call, it ensures that the arguments are compatible with U (U or F). If one of the arguments is a pointer, Privagic also ensures that the pointed memory location is U.

If a function is available from within the enclave, the developer can use the `within` annotation. This is the case of the functions of the mini-libc provided by Intel SGX SDK. This mini-libc library is included in the Privagic runtime, which is embedded in each enclave. It contains basic functions such as `malloc` or `memcpy`. For a `within` function, Privagic ensures that a call executed in the enclave C only receives C and F arguments. Privagic also verifies that if an argument is a pointer, then the pointed memory location is C. This ensures that the code cannot inadvertently let a sensitive value escape, e.g., by calling `memcpy` with a U pointer while the `memcpy` is executed in C.

5.8.2 Communication with the outside

In hardened mode, an enclave is totally isolated from the rest of the world. In order to allow the enclave to communicate with the outside, Privagic provides a special annotation of the form `within_r_a*`. This annotation can only be used for an external function available from within the enclave. The annotation gives color constraints: `r` gives a constraint on the result of a call, and `a*` on the arguments. A constraint is equal to c (color) or f (free or void for the returned value). Privagic handles such an annotation by ignoring the f arguments when it checks the compatibility of the colors. An f argument can thus have any color and can point to any location.

For example, a developer can annotate the `encrypt(plain, len, key, iv, cypher)` function of the `libssl` library with `within_f_ccccf`. Thanks to this annotation, `cypher` can point to a U or S location, even if `plain` is C, which allows the function to write the encrypted result in unsafe memory. Privagic also provides by default a `within_c_cffc` function `classify(char* safe, char* unsafe, int len, int max)` to copy an unsafe buffer in a safe buffer located in the enclave, and a `within_c_fcc` function `declassify( char* unsafe, char* safe, int len)` to copy a safe buffer in unsafe memory.

5.8.3 Indirect call

Handling an indirect call is more complex because Privagic cannot always identify the called function during compilation. Because of that, Privagic uses a conservative approach. Privagic considers an indirect call as a direct call to an external function located in the untrusted part of the application. As for any external call, Privagic verifies that the arguments of an indirect call have the F or the U color. If an argument is a pointer, Privagic also verifies that the pointer points to a U or S location. Accordingly, when an instruction loads a function pointer, Privagic loads a pointer to a version of the function specialized for U arguments.

5.9 Error messages

Privagic raises errors to the developer when it cannot partition the code, i.e., when it detects a color incompatibility. In order to help the developer solve a color incompatibility, Privagic carefully reports why colors are incompatible. In detail, whatever the error, Privagic first reports the line number and the name of the file where the error occurs to the standard error:

Errors occurred on line number 42 of file test.c:

Additionally, Privagic indicates the call chain that leads to an error and why the error occurs:

- **The line has at least two incompatible colors: blue, green ...**

This error message indicates that one of the instructions generated by the corresponding line of code has two different colors. This error occurs regardless of the function's argument color.

- **The argument number 1 of function func should be compatible with blue whereas it is green**
  The analysis of the function `func` identifies that argument number 1 should be compatible with blue. This error is raised at the callsite of the function `func` where the function is called with the green argument number 1.

- **The Colors of arguments number 1 and 3 of the function func should be compatible, whereas they are blue and green respectively**
  The analysis of the function `func` identifies that a dependency between argument number 1 and argument number 3 is not satisfied (e.g., because the code computes a value by using these arguments, see Rule 2). This error is raised at the callsite of function `func` where the function is called with the blue argument number 1 and the green argument number 3.

- **The return values of the function func have at least two incompatible colors**
  When a function has two or more `ReturnInst`, all the `ReturnInst` should have compatible colors.

- **The given argument number 1 of the function func contains error**
  This error is raised when an incompatibility is detected while treating the `CallInst`: the color of the argument 1 is not compatible with the color that the parameter should have.

- **The current BasicBlock should be blue, whereas the called function func manipulates green**
  This error is raised at the callsite of the function `func` because the called function returns a green value, while the basic block is colored in blue (indirect leak).

- **A trusted data of color blue cannot be stored in an untrusted variable**
  This error is specific to `StoreInst`. It happens because of a direct leak.

- **By calling the function func with argument number 1 uncolored and argument number 2 colored, a trusted data of color blue is stored in an untrusted variable**
  Error comes from line number 123 of file test.c (callsite)
  This error is raised at the callsite of function `func`, while line number 42 belongs to function `func`. This error indicates that, because of the colors of the arguments, the function will let a blue value escape in untrusted memory. By indicating both the incriminated line in `func` and the callsite, the developer can easily understand the call chain and modify either `func` or the call site.

Chapter 6

Application partitioning

Contents

After having computed the colors, the Privagic compiler partitions the application by rewriting it. It rewrites the application in three steps: it rewrites the global variables, the multi-color structures, and finally the functions. Once the application is rewritten Privagic relies on our FastSGX runtime to execute the partitions in SGX. In the rest of this chapter, we begin by having an overview (§6.1), then we discuss adapting FastSGX for use in Privagic (§6.2), and we describe the different steps of rewriting.

6.1 Overview

LLVM program consists of modules. The module is the top level structure that encapsulates all the other elements of the LLVM IR (see §2.2.1). The module contains a list of global variables, a list of functions, a list of structure types etc. The single LLVM IR file that Privagic gets as an input consists of one unique Module. This module represents the entire code of the application that has to be processed. As shown in Figure 5.1 Privagic generates one LLVM IR file for each color based on the cloned module. For this reason, during the rewriting phase Privagic makes several clones of the input module, one per color. The cloned module contains only the elements that are to be placed inside the corresponding enclave.

6.2 Adaptation of FastSGX

Privagic relies on FastSGX to deploy the partitioned application on Intel SGX. We added to FastSGX the `spawn` and `cont` functions, derived from `send` (described in §4.2.1). `Spawn` sends a message to start a function and `cont` sends a message to continue the execution of the function. We also add a `wait` function, derived from `recv` (described in §4.2.1), which unblocks a thread blocked in `wait`.

6.2.1 Chunks and function spawn

Since a function can access multiple colors, the FastSGX may have to execute a function across different enclaves. For example, in Figure 1.1 (see

Introduction), `create` executes in both the blue and red domain since line 7 accesses a blue field and line 8 a red field. Therefore, FastSGX compiler splits such a function into sub-functions, which we call *chunks*. A chunk with color C only contains instructions with the color C or F. It can run in a single enclave of color C.

In order to start a chunk from another enclave, Privagic uses the `spawn` message. This message takes as an argument a function number. When the worker thread of an enclave with the color C receives a `spawn` message for a function `g`, it starts the execution of the chunk C of `g`.

6.2.2 Synchronization between the chunks

Chunks may have to synchronize and exchange messages. This first happens in relaxed mode to send or receive function arguments (see §6.5 below). Privagic may also have to synchronize the chunks when a function executes an instruction that has an effect, e.g., a write in S in relaxed mode or the execution of an external function.

In order to synchronize enclaves or exchange messages, Privagic uses the newly derived functions `cont` and `wait`. The `cont` function takes a 64-bit value as an argument and sends it in a `cont` message to an enclave through its FIFO queue. The `wait` function waits for a `cont` message and returns the 64-bit value contained in the message.

6.3 Global variables

As a first step, Privagic rewrites the global variables. For the variables that are not S, Privagic places them in their enclaves. During the rewriting phase, Privagic clones the C variables in their enclaves and the instructions using those variables remain unchanged and placed inside C enclaves. The variables that are placed inside the enclave cannot be modified from outside or from another enclave. The S variables, which only exist in relaxed mode, should be accessed and modified by all enclaves. Therefore Privagic moves them in a single shared data structure stored in unsafe memory. Each enclave has a pointer to this global data structure, which allows any enclave to access the shared variables. While initializing the enclaves the pointer is also initialized

```
1. // From:
2. %1 = load i32, i32* @a, align 4, !dbg !25

3. // To:
4. %2 = load %globals_type*, %globals_type** @globals
5. %3 = getelementptr inbounds %globals_type, %globals_type* %2, i32 0, i32 4
6. %4 = load i32, i32* %3, align 4, !dbg !25
```

Figure 6.1: Example of global variable load rewriting.

inside the enclave. The pointer that is placed inside the enclave remains unchanged, hence all the enclaves can access and modify the S variables.

In order to create the shared data structure, a new structure type is defined in LLVM and placed inside the cloned modules. The element types of this new structure type are the types of each S global variable. A global variable of this new data structure is created in unsafe memory and is placed only in Untrusted. Each element of the data structure is initialized with the initializer of the S global variables. A pointer to this global data structure is created as another global variable and is placed in all enclaves. Accordingly, Privagic rewrites the LLVM IR in order to replace the accesses to the S global variables with accesses to this shared global data structure.

6.3.1 Example

Figure 6.1 illustrates an example of how those accesses are rewritten in the case of instruction. A global variable is accessed by a function through a load instruction. Line 2 of Figure 6.1 shows an example of a load instruction in LLVM IR. This instruction loads the global variable `a` in register `%1`. Considering the global variable `a` to be S, the load instruction should be rewritten. At first, the pointer of the global data structure should be loaded in register `%2` (line 4). Then, the element corresponding to the global variable `a` should be retrieved from the global data structure in register `%3` (line 5). Finally, the global variable is loaded in register `%4` as the exact type of global variable `a` (line 6). The register `%4` is used throughout the function instead of the register `%1`. The register `%1` is removed from the function in order to avoid disturbance during the execution.

```
1. struct account {
2.   char color(blue)* name[256];
3.   double color(red)* balance;
4. };

5. struct account* create(char* name) {
6.   struct account* res =
7.     malloc_in(U, sizeof(*res));
8.   res->name =
9.     malloc_in(blue, sizeof(*res->name));
10.  res->balance =
11.    malloc_in(red, sizeof(*res->balance));

12.  strncpy((*res)->name, name, 256);
13.  (*res)->balance = 0.0;
14. }
```

Figure 6.2: Data structure rewriting.

6.4 Multi-color structures

As a second step, Privagic rewrites the structures with multiple colors. These structures can only exist in relaxed mode because using a multi-color structure requires the use of cross-enclave pointers.

For a multi-color structure, since an enclave is contiguous in the virtual address space, Privagic cannot keep the fields packed in memory. For that reason, Privagic introduces an indirection level. Instead of storing directly the colored values in the structure, Privagic stores pointers to the colored values. Therefore for each structure containing a colored value, a new structure type is created. The element types of the new structure type are:

- If the element is F, the element type is the same as the old structure's element type.
- If the element is colored, the element type will be the pointer of the old structure's element type.

Lines 1-4 of Figure 6.2 illustrate in C how Privagic rewrites the account structure presented in Figure 1.1. As both elements of the old structure are colored differently, both elements of the new structure are pointers toward the old elements.

6.4.1 Allocation

When Privagic rewrites a data structure, Privagic first analyzes the code in order to associate each allocation site (e.g., a call to `malloc`) to a data structure. Then, for the multi-color data structures, Privagic rewrites the allocation site in order to allocate the data structure in unsafe memory, and the colored fields in their enclave (lines 6-11). For that, Privagic generates calls to the `malloc_in` function, which allocates memory from the enclave given as a first argument.

Privagic allows the data structure to be stored in an enclave, different from its elements' colors. As FastSGX provides inter-enclave communication, it is feasible. In the case where the data structure is of color C, its F elements take C color. Therefore, F elements of the data structure are also allocated in C enclave.

6.4.2 `malloc_in`

The `malloc_in` function spawns a local function `malloc_out` inside the adequate enclave. It then sends the memory size that needs to be allocated to the `malloc_out` function, through a continue message. `malloc_out` then executes the actual `malloc` function inside the enclave, and sends back the allocated pointer to the `malloc_in` function. Then `malloc_in` function finally returns the allocated pointer to the caller function. Note that even if the pointer leaks outside the enclave, the caller cannot access the pointed value since SGX prevents memory access to an enclave.

6.4.3 Access

Privagic also rewrites all the accesses to the multi-colored data structure in order to handle the indirection (lines 12-13). In the case where the structure is stored in an enclave different from its element, the pointer of the element will be sent to the adequate enclave to proceed with the computation. In line 6 `*res` is allocated in U and the element `balance` of the structure `*res` is allocated in the enclave red. Line 13 manipulates a variable of color red, hence it will be executed in enclave red. To execute this line, U will send the pointer `balance` (which is saved in U) to the enclave red. Enclave red will execute the line 13 and update the `balance`.

6.5 Code rewriting

Finally, Privagic partitions the code itself. While it partitions the code, Privagic leverages the fact that two instructions with different colors do not have data dependencies in order to generate a parallel code. For that, Privagic supposes that the Privagic runtime runs a worker thread in each enclave for each thread of the application. The remainder of the section explains how Privagic partitions the code step by step.

6.5.1 Color set and chunks

In order to generate a parallel code, Privagic first computes the *color set* of each function. The color set of a function is the set of colors used by a function, F excluded. Figure 6.3 shows an example. In this example, the color set of `main` (line 4 in Figure 6.3) is equal to $\{blue, \mathrm{U}\}$ because `main` uses the U color at line 5, and the blue color at line 6. For the specialized version of `f` that receives a blue parameter (line 9), its color set is $\{blue\}$ because `f` receives a blue argument, calls a function without using color, and returns the F value 42. For `g`, its color set is $\{red, blue, \mathrm{U}\}$ because it uses the red and blue colors at lines 14 and 15, and then executes a call to an external function, which is colored in U during the type analysis phase.

For each color of a color set, Privagic generates a colored version of the function, which we call a *chunk*. For example, as shown in Figure 6.4, Privagic generates three chunks for the function `g`, one per color of the color set.

6.5.2 Simple cases

We first consider an IR without loads, stores, or direct calls to local functions. With this IR, each instruction is mono-colored. Privagic partitions the code by generating the C instructions of a function in its C chunks. Privagic replicates the F instructions in each chunk, which ensures that using a F value in a chunk is safe (see §5.7.3). If a F instruction is uselessly replicated, a dead-code-elimination pass [1] eliminates it afterward.

Global variables

```
1. int color(U) unsafe = 0;
2. int color(blue) blue = 10;
3. int color(red) red = 0;
```

Function `main`
$colorset = \{blue, \mathrm{U}\}$

```
4. int main() {
5.   unsafe = 1;
6.   int x = f(blue);
7.   return x;
8. }
```

Function `f`
$colorset = \{blue\}$

```
9. int f(int y) {
10.   g(21);
11.   return 42;
12. }
```

Function `g`
$colorset = \{red, blue, \mathrm{U}\}$

```
13. void g(int n) {
14.   blue = n;
15.   red = n;
16.   printf("Hello\n");
17. }
```

Figure 6.3: A complete example

6.5.3 Loads and stores

We now consider the loads and stores. If $\overline{*p} = \mathrm{C}$ with $\mathrm{C} \neq \mathrm{S}$, Privagic generates the load or the store in C. For example, as shown in Figure 6.4, Privagic generates `unsafe = 1` (line 5 in Figure 6.3) in U, `blue = n` (line 14) in blue, and `red = n` (line 15) in red. In relaxed mode, when p and $*p$ do not belong to the same enclave, Privagic additionally uses `wait`/`cont` in order to send the value p from its enclave to C. C will proceed with the computation of $*p$

If $\overline{*p} = \mathrm{S}$, Privagic can generate the load or the store in any enclave.

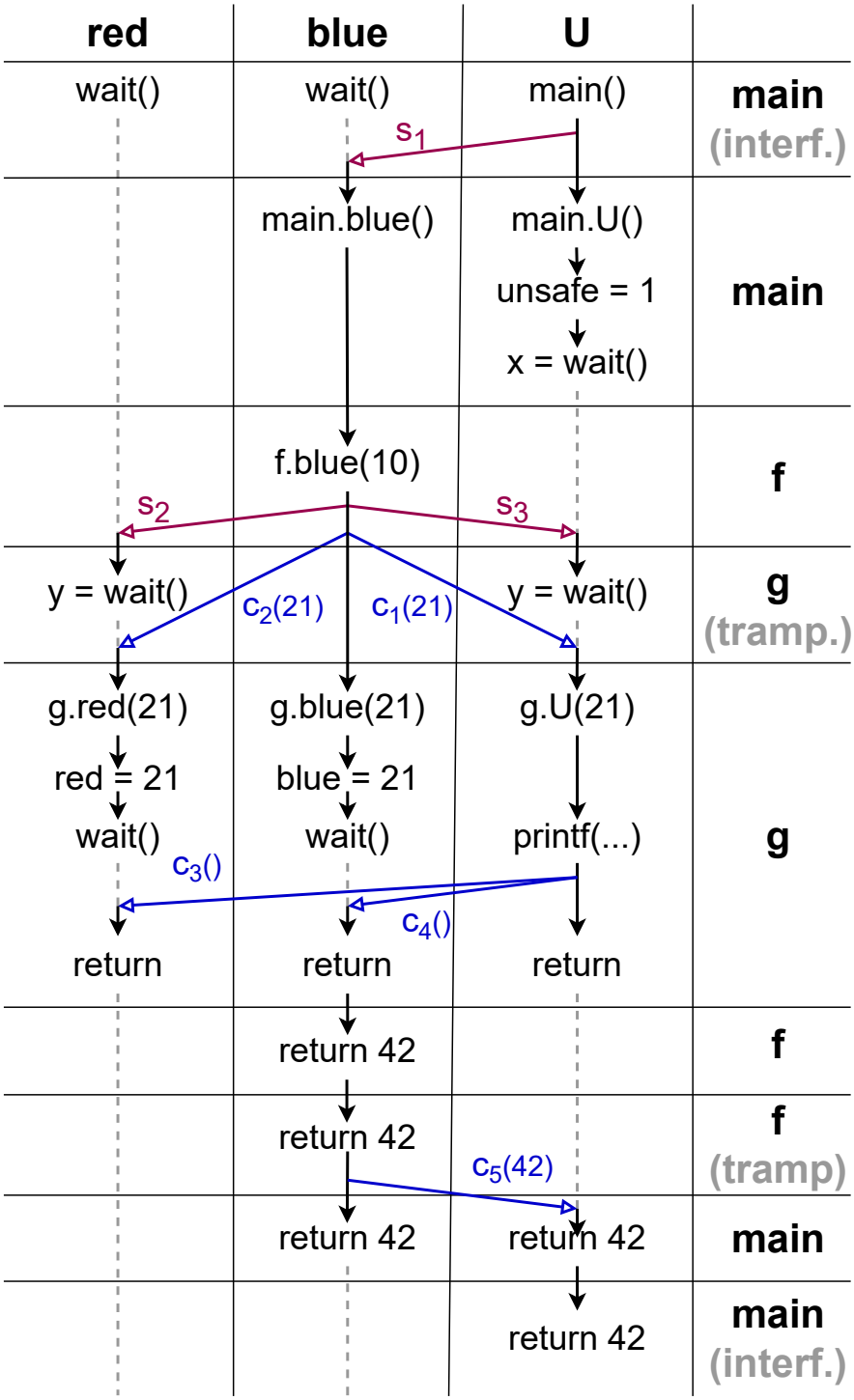


Figure 6.4: Execution of the example given in Figure 6.3

For a load, Privagic replicates the load in each chunk because loading a value from an enclave is more efficient than receiving it from another enclave with a `wait/cont`. For a store in S, Privagic has to synchronize the chunks because a store in S has a visible effect (see §6.5.5 below). Replicating the instruction in each chunk is thus inefficient since this would lead to many synchronizations and is prone to error if multiple enclaves try to store the same value. Privagic thus selects one of the chunks of the function as a *master chunk* and generates the stores to S in this chunk. A color is selected as the *master chunk* if it has the maximum number of instructions. In order to synchronize, the *master chunk* sends a `cont` message to all the colors in the *color set* except itself. On the other hand, the chunks of those colors call `wait` to wait for the `cont` message from the *master chunk*.

6.5.4 Function call

For each call site, Privagic compares the color set of the caller with the color set of the callee. If a color C is common between the two color sets, Privagic generates a call to the chunk C of the callee directly in the chunk C of the caller. In that case, the chunk of the caller calls the chunk of the callee with the C and F arguments, but not with the others arguments. For example, in Figure 6.4, `main.blue` directly calls `f.blue` with the blue argument (10).

In order to handle the case where one of the colors of the callee is not in the color set of the caller, the Privagic runtime manages communication channels between the enclaves. In order to start a missing chunk, the Privagic runtime generates a `spawn` message, which takes as an argument, the identifier of the missing chunk. The missing chunk may need F arguments, which are only computed in the caller. When this is the case, Privagic reports an error in hardened mode since using a value computed by another enclave is unsafe. In relaxed mode, using an unsafe value is authorized, and the Privagic runtime generates thus the `cont` message to send a F argument, and the `wait` function to wait for a message.

Since a chunk receives its F arguments as a parameter, the caller cannot directly spawn the execution of a missing chunk if the missing chunk has F parameters. Instead, Privagic generates trampolines in charge of receiving the F arguments and calling the chunk. Figure 6.4 illustrates how Privagic leverages trampolines to start `g` from `f`. In this example, `f.blue` sends the

spawn messages s_2 and s_3 in order to start trampolines for the chunks `g.red` and `g.U`. Then, `f.blue` sends the F argument 21 with the `cont` messages c_1 and c_2. Finally, the trampolines receive the argument 21 and call the chunks.

6.5.5 Synchronization barriers

Some instructions have a visible effect. This is the case for a store to S or for a call to an external function located in U. Misordering the execution of such instructions would lead to incorrect behaviors: a read to S while the corresponding write in sequential is not yet executed, or the inversion of two `printf`. For this reason, when an instruction has a visible effect, Privagic generates a synchronization barrier by using `wait`/`cont`. The `cont` messages are sent by the chunk that executed the instruction with a visible effect. In other chunks, the instruction is replaced by a `wait`.

Figure 6.4 illustrates how Privagic generates a synchronization barrier for the `printf` of `g` at line 16 in Figure 6.3. In the red and blue chunks of `g`, Privagic replaces the call to `printf` by a call to `wait`, which suspends the execution of the red and blue enclaves. Accordingly, in U, Privagic generates calls to `cont` after the call `printf`. These calls emit the c_3 and c_4 messages to the red and blue enclaves, which unblock the enclaves.

6.5.6 Entry points and indirect calls

For each entry point and each function indirectly called, Privagic considers that it executes only in U, and that it receives U arguments in hardened mode, and F arguments in relaxed mode. For that reason, Privagic generates an *interface version* of these functions. An interface version keeps the original name of the function. It is in charge of starting the missing chunks. For example, in Figure 6.4, the interface function of `main` starts the execution of `main.blue` by sending a `spawn` message, s_1, to enclave blue, and then directly calls `main.U`.

Chapter 7

Evaluation

Contents

In this chapter, we focus on evaluating Privagic. The evaluation has the goal of answering the following questions:

- Engineering effort – what is the difficulty of using Privagic,
- Trusted computed base – what is the size of the TCB with Privagic,
- Performance – what is the performance of an application generated with Privagic.

We describe the hardware and software setting in §7.1, the evaluation results of Memcached in §7.2, and finally the evaluation results of data structures in §7.3.

Workload	Pattern	read	update	scan	insert	rmw‡
A	zipfian	50%	50%	0%	0%	0%
B	zipfian	95%	5%	0%	0%	0%
C	zipfian	100%	0%	0%	0%	0%
D	latest	95%	0%	0%	5%	0%
E†	uniform	0%	0%	95%	5%	0%
F	zipfian	50%	0%	0%	0%	50%

†: not evaluated because our applications do not implement scan
‡: read-modify-write

Table 7.1: YCSB workloads

7.1 Hardware and software setting

We evaluate Privagic on two machines. Machine A is an Intel i5-9500 CPU 3 GHz with 16 GiB memory. This 6-core CPU ships SGX version 1 with a maximal EPC size usable by the enclaves of 93 MiB. Machine B is an Intel Xeon Gold 5415+ with 16 CPUs, 120 GiB of memory and 22.5 MiB of last level cache. This processor supports SGX version 2 with a maximum EPC size of 8131 MiB.

Machines A and B runs Linux 5.15.0, glibc 2.31, clang 10.0.0 and Intel SGX SDK 2.19.100.

7.2 Memcached

We first evaluate a complete legacy application: memcached 1.6.12 (24841 lines of C code). Memcached is an in-memory cache widely used in production. It is designed as an event-based system with multiple threads to handle the requests.

In order to inject the load, we use the standard Java version of the YCSB benchmark [20]. Figure 7.1 presents the workloads. YCSB and memcached communicate through the loopback network on the same machine in order to minimize the network overheads. YCSB and memcached are carefully bound to different physical cores. YCSB is configured to use 6 worker threads, which simulate 6 clients concurrently accessing the cache. YCSB uses a record size of 1024 B and runs $8,000,000$ operations.

We configure memcached with 7 threads: a worker thread, a network listener thread and some miscellaneous background threads, e.g., in charge

	Modified (C locs)	TCB (KiB)	Unsafe code (LLVM locs)
Scone	0	51271	78106 + libraries
Privagic	9	268	1 238

Table 7.2: Memcached metrics (locs: lines of code)

of maintaining the least recently used key/value pairs in memory.[1]

We evaluate YCSB with data sets ranging from 1 MiB to 32 GiB by changing the number of records. We evaluate three configurations of memcached on machine B. Unprotected consists in running memcached in a docker container without SGX. Scone consists in using Scone [6] v5.7.0 to run memcached in a container fully embedded in an SGX enclave. Scone calls the operating system by using switchless calls [6]. Privagic is our generated version of memcached. It protects the central data structure of memcached (a map) with a single color.

7.2.1 Engineering effort

Column "Modified" of Table 7.2 reports the number of modified lines of code. Scone does not require any modification since memcached is fully embedded in an enclave. For Privagic, we modified 9 lines of code: 2 to add the colors to the central map, and 7 to declassify the values. This result shows that using Privagic in a legacy application requires a modest engineering effort.

7.2.2 TCB size

As shown in the "TCB" column of Table 7.2, the binary code loaded in the enclave with Privagic is roughly 200 times smaller than the binary code loaded with Scone. With Scone, this code includes memcached (349 KiB), the musl C library (14.7 MiB) and the OS library shipped with Scone (36.2 MiB). Any bug from any of these components may lead to a threat inside the enclave.

The 268 KiB of Privagic includes the Intel SDK runtime and the Privagic runtime, which are not supposed to contain bugs. If we exclude this trusted code, the unsafe code, which consists in the code generated by Privagic and

[1] https://github.com/memcached/memcached/blob/master/doc/threads.txt

embedded in the enclave, contains 1238 lines of LLVM code (column "Unsafe code"). We cannot report the exact unsafe code with Scone because Scone does not disclose its source code. However, even if we ignore the musl C library and the library OS of Scone, the code of memcached embedded in an enclave is already 63 times larger (78106 lines of LLVM code).

Overall, these results show that, Privagic significantly reduces the TCB as compared to fully embedding the application in the enclave, and for a modest engineering effort.

7.2.3 Performance

Figure 7.1 reports the throughput of memcached on machine B (see §7.1).

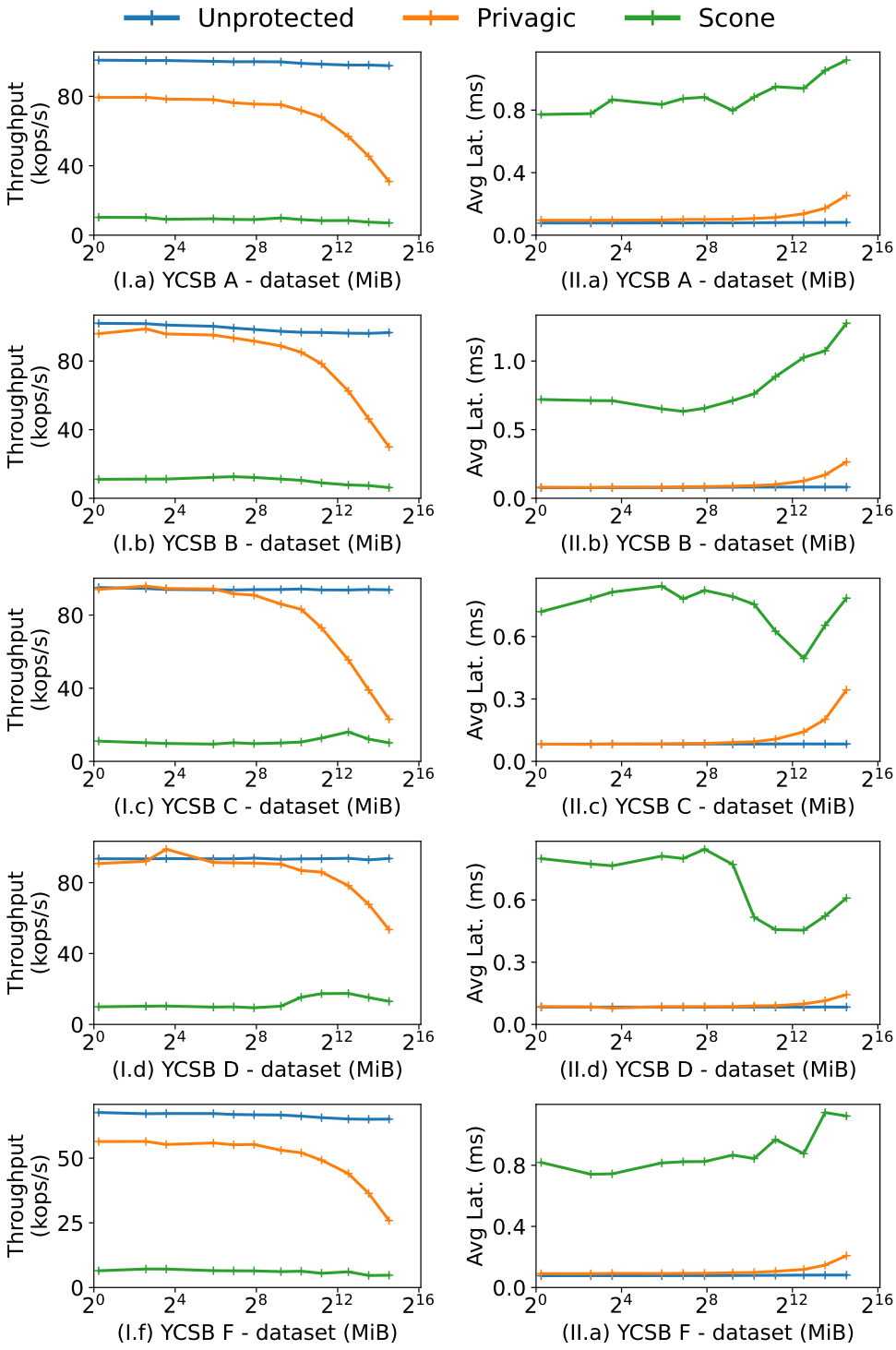
Small data set. For a small dataset (less than 200 MiB), the throughput of Privagic is between 8.5 to 10.0 better than the throughput of Scone.

With Scone, the overhead is larger than with Privagic for two reasons. First, the time to enter and leave the enclave to execute a request is larger with Scone than with Privagic. Then, since Scone fully executes in an enclave, Scone has to perform many system calls from the enclave: for the network operations and for the synchronizations (e.g., to acquire or release locks). Even if Scone optimizes the system calls with switchless calls, the large number of ocalls in Scone significantly slows down the execution of a request. This is not the case with Privagic because Privagic minimizes the code in the enclave. With Privagic, the code in the enclave accesses the data structure and only calls the operating system twice: one time to acquire a lock, and one time to release it.

As a result of the lower number of ocalls and of the faster transfer of control in Privagic, the throughput of Privagic is only 5 % to 20 % lower than the throughput of Unprotected for a small dataset. The throughput of Scone is between 6 to 10 times slower than Unprotected.

Large data set. We also observe that the latency and the throughput of Privagic degrades when the dataset size increases. In the worst case (dataset of 32 GiB), the throughput of Privagic remains, however, at least 2.3 times higher than the throughput of Scone.

The throughput of Privagic decreases with a large dataset because of cache effects. In detail, the central hashmap of memcached becomes larger with a larger dataset. Retrieving a key thus leads to more memory accesses,



(I) Throughput (II) Latency

Figure 7.1: Memcached with YCSB.

which translates into more last level cache (LLC) misses. This is the case for Unprotected, Privagic and Scone. For example, in Unprotected, we measured that increasing the dataset from 236 MiB to 32 GiB multiplies the ratio of LLC misses roughly by 3 (from 6.5% to 17.6% LLC misses).

For Privagic and Scone, the higher number of cache misses has an important effect because they happen while the processor is in enclave mode. As reported in [41], a LLC miss in enclave mode takes between 5.6 to 9.5 more time than in normal mode. As a result, while the higher number of LLC has a marginal effect for Unprotected, this is not the case for Privagic and Scone. For Privagic, the higher number of LLC misses translates into a worse latency, which itself translates into a degraded throughput. The throughput degradation with Scone also exists but is less visible because the latency of Scone is already high with a small dataset.

7.2.4 Takeaway

The experiment with memcached shows that (i) Privagic can scale to a large legacy application used in production, (ii) adapting a legacy application to use Privagic requires a modest engineering effort, (iii) the TCB of memcached with Privagic is much lower than the TCB of memcached with Scone, and (iv) Privagic is consistently more efficient than Scone.

7.3 Data structures

We now evaluate data structures in order to finely analyze the code generated by Privagic. We evaluate two data structures: a linked-list and a hashmap. We use the data structures as maps, i.e., they associate keys to values. We inject the load with our re-implementation in C of the YCSB benchmark [20] (see Figure 7.1). The benchmark directly accesses the map in the same thread without involving the network in order to emphasize the cost of using SGX.

We evaluate five configurations. Unprotected does not use SGX. Privagic-1 consists in coloring the whole hashmap. Privagic-2 consists in coloring the keys and the values with two different colors. Intel-sdk-1 exposes the interface of the maps in EDL (i.e., `put`, `get`). Intel-sdk-2 uses two enclaves in EDL: one for the keys and the other for the values. We tried to implement intel-sdk-2

as efficiently as we can by minimizing the number of ecalls.

In each configuration, we use keys of 8 bytes and values of 1024 bytes. We use machine A for this evaluation. For the experiments with a single color, we pre-initialize the map with 100 000 keys and then run the experiment. For the experiments with two colors, we pre-initialize the map with only 20 000 keys because the runs are much longer.

7.3.1 Engineering effort

For the engineering effort, we focus on the hashmap, which is representative of other data structures. For one color, using Privagic requires the modification of 5 lines of code: 1 line to color the hashmap, 2 lines to color two local variables, and 2 lines to declassify the result of a get. For two colors, we modified 6 lines in total: 2 lines to add the colors to the fields, 1 line to color a local variable, 2 lines to declassify the result of a get, and 1 line to declassify the result of a call to a hash function.

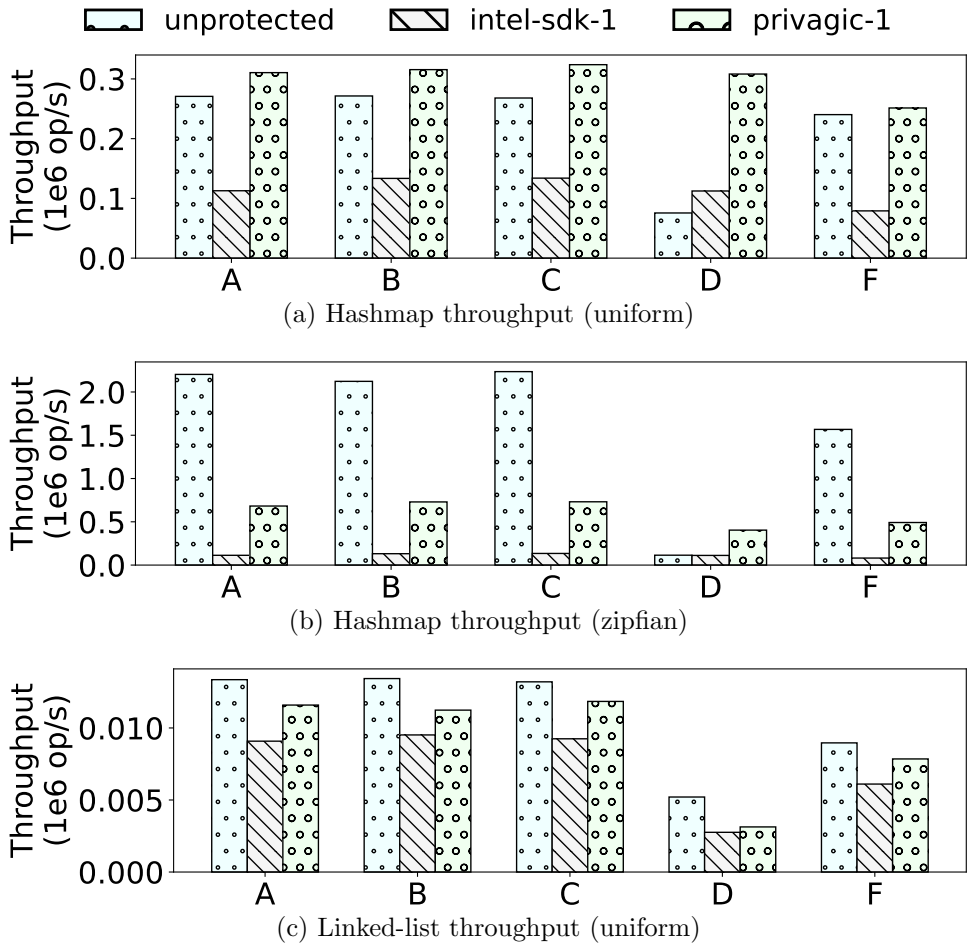
Manually porting the unprotected code to use one color with Intel SDK is relatively straightforward, but leads to 206 modified lines of code. Manually porting the unprotected code to use two colors with Intel SDK is more challenging and requires 6 ecalls. We have to handle the allocations of the keys and values in the different enclaves, and to expose several functions in order to exchange data between the enclaves.

Overall, we observe that using Privagic to protect a data structure requires a modest engineering effort. Manually using Intel SDK requires more work, and especially if the application uses multiple colors.

7.3.2 Performance analysis

Figure 7.2 and Figure 7.3 report the performance with one color, then Figure 7.4 with two colors.

Privagic versus Intel SDK with one color In Figure 7.2 and Figure 7.3, when we compare intel-sdk1 with privagic-1, we observe that Privagic divides the latency and multiplies the throughput by 2.4 to 6.1 for the hashmap, and by 1.1 to 1.2 for the linked-list. These results show that, by executing the code in parallel instead of using ecalls/ocalls, the code generated by Privagic is



(a) Hashmap throughput (uniform)

(b) Hashmap throughput (zipfian)

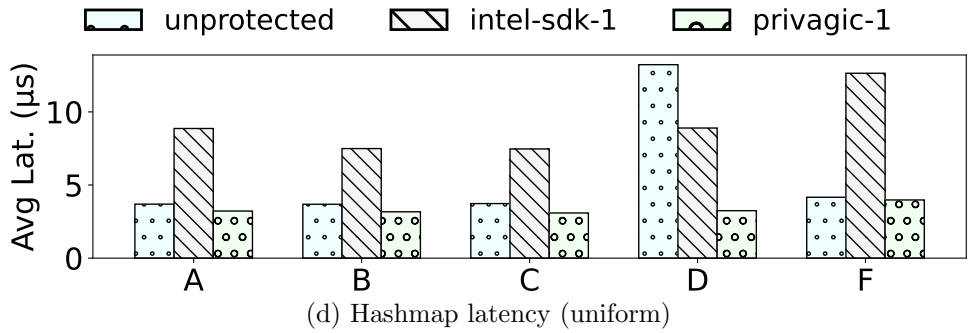
(c) Linked-list throughput (uniform)

Figure 7.2: Throughput of data structures with the YCSB benchmarks (1 color).

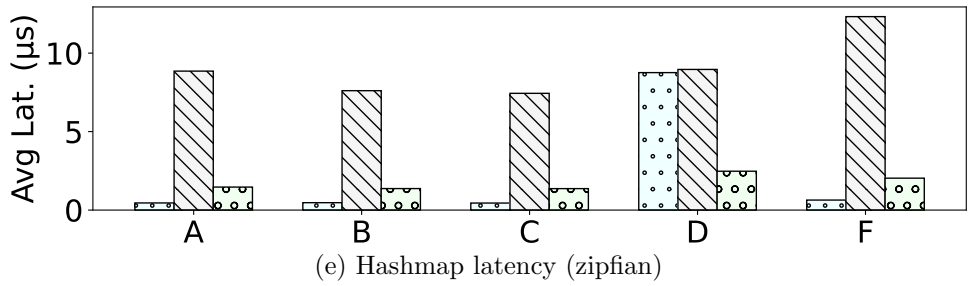
more efficient than a naive implementation with Intel SDK. The performance improvement brought by Privagic is more interesting with the hashmap than with the linked list because retrieving a key in a linked-list requires visiting sequentially many (key, value) couples (50 000 in average), which amortizes the cost of crossing an enclave boundary.

Privagic versus unprotected with one color We first analyze the performance while ignoring workload D, which we discuss separately at the end of the section because the results are more unexpected.

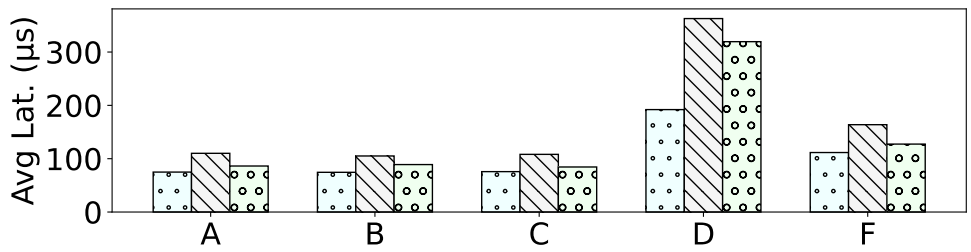
In Figure 7.2 and Figure 7.3, we observe that Privagic optimizes the



(d) Hashmap latency (uniform)



(e) Hashmap latency (zipfian)



(f) Linked-list latency (uniform)

Figure 7.3: Latency of data structures with the YCSB benchmarks (1 color).

performance of the hashmap as compared to unprotected for a uniform distribution. In detail, if we ignore the workload D, Privagic decreases latency and increases throughput from 4% to 21%. The better performance of Privagic as compared to unprotected comes from a better parallelism. Privagic generates a parallel code by leveraging the fact that we can execute in parallel the U and the colored instructions, which allows Privagic to improve performance while also enforcing security.

For the hashmap, with a zipfian access pattern, privagic-1 is, however, less efficient than unprotected. If we ignore workload D, privagic-1 multiplies the latency and divides the throughput by 2.9 to 3.2 times. With a zipfian access pattern, we measured that the number of L1 cache misses drops from 30% to

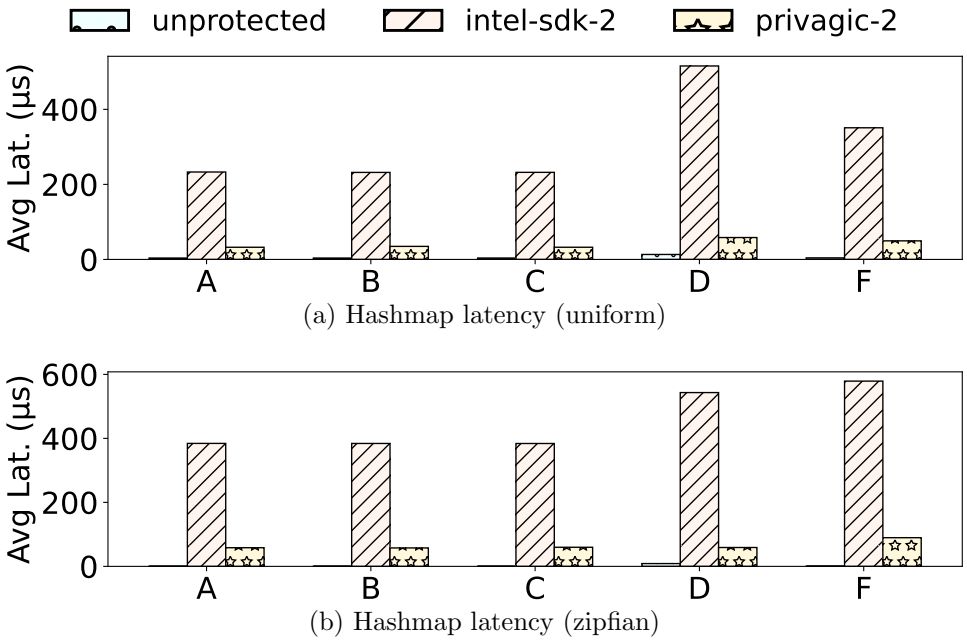
20%. This better L1 locality explains why the zipfian access pattern significantly improves the performance of unprotected as compared to the uniform access pattern. The performance improvement brought by the better locality is less important for Privagic, because the time spent in communication between the enclave and the untrusted part of the application dominates. As a result, despite better parallelism, Privagic is less efficient than unprotected with a zipfian access pattern.

For the linked list, privagic-1 is also less efficient than unprotected (degradation of 13% to 28% when we ignore workload D). In this experiment, each request leads to many accesses to the linked list, and thus many L3 cache misses. These L3 cache misses are more costly with Privagic because of the cost of checking the tree of hashes when a cache line is loaded from memory.

The results with workload D are unexpected. Even intel-sdk-1 have often better performance than unprotected with this workload. After investigation, we found that this result is not significant. In detail, workload D allocates many objects in the enclave with privagic-1 and intel-sdk-1. Since workload D never frees an object in an enclave, allocation in the enclave is especially fast, since it only consists in bumping a bump pointer. In unprotected, the allocations to add the elements to the map interleave with the allocations/frees of the load injector. The memory allocator has thus to traverse a linked list of free chunks, which slows down execution.

Two colors In Figure 7.4, we observe that privagic-2 is significantly faster than intel-sdk-2. Privagic-2 divides the latency and multiplies the throughput from 6.4 to 9.2 times as compared to intel-sdk-2. Despite our optimizations in intel-sdk-2, the ecalls and ocalls of intel-sdk-2 significantly degrade performance.

We also observe that privagic-2 also significantly degrades performance as compared to unprotected. With two colors, for each request, the enclaves intensively communicate, which annihilates the benefit of executing the code in parallel. This result shows that using two colors has a nonnegligible cost, even with a compiler that optimizes the generated code.



(a) Hashmap latency (uniform)

(b) Hashmap latency (zipfian)

Figure 7.4: Hashmap with the YCSB benchmarks (2 colors).

7.3.3 Takeaway

This set of experiments with data structures shows that (i) adding colors to classic data structures is easy, (ii) Privagic is consistently more efficient than a code manually ported to SGX with Intel SDK, (iii) for some workloads, Privagic is more efficient than unprotected with one color, (iv) with two colors, the code generated by Privagic remains significantly less efficient than an unprotected code.

Chapter 8

Limitations and perspectives

Contents

Privagic has its own set of limitations that open doors to new perspectives. In this chapter, we first discuss the limitations and prospects linked with message-passing technique of our runtime FastSGX in §8.1, furthermore we reflect on ways to solve CPU consumption problem due to extra threads in §8.2, following that we address the limitation in using mutli-color structure in §8.3, and finally, we conclude by reflecting on the future perspective to adapt Privagic for multiple languages in §8.4.

8.1 Message passing

Privagic's runtime FastSGX relies on a message-passing mechanism. With Privagic, an attacker may try to attack an enclave by generating `cont` and `spawn` messages. Because a `cont` message simply unblocks an enclave without changing its execution flow, an attacker cannot temper the execution flow of an enclave with `cont` messages. In hardened mode, a `cont` message cannot carry a F value. This ensures that the enclave executes exactly the code that

it is supposed to execute, even if an attacker sends unexpected `cont` messages. In relaxed mode, when a `cont` message carries a F value, as for any value that comes from the untrusted part of the application, the developer may have to check its integrity.

An attacker can temper the execution flow of the application by sending unexpected `spawn` messages. Privagic does not implement a protection against this attack vector, but techniques that rely on message authentication, as proposed in Glamdring [34] or Scone [6], should solve the issue.

8.2 Number of threads

Another limitation of our prototype comes from the number of worker threads. Currently, for each thread of the application, Privagic runs one worker thread per enclave, which multiplies the number of threads by the number of colors plus one. We did not optimize this part of the prototype, but techniques such as configless switchless calls [62] should allow Privagic to adapt the number of worker threads to the actual workload.

8.3 Multi-color structure

The last limitation of our prototype comes from the impossibility of using multi-color structures in hardened mode. Because of the indirection introduced between the structure located in unsafe memory and the colored fields (see §6.4), using a multi-color structure requires the use of cross-enclave pointers, which is unsafe and thus prohibited in hardened mode. Currently, in hardened mode, a developer can thus only use multiple colors if they do not belong to the same data structure. Using multi-color structures in hardened mode requires the use of authenticated pointers, which we let as future work.

8.4 Multi-language

As described in Figure 5.1, Privagic is a part of LLVM (the backend of the compiler). Privagic now works for applications written in C language.

Frontends exist for multiple languages like C++, rust, GO, that provide LLVM Intermediate Representation (LLVM IR) for the backend to work with. In order to adapt Privagic for those languages, some language-specific modifications should be added to Privagic. A major part of the Privagic code could be kept intact to integrate other languages' characteristics. As the runtime FastSGX is completely independent of Privagic, only the Privagic part should be modified for applications written in other languages, with a frontend for LLVM. We save this as a future project.

Chapter 9

Conclusion

To summarize, this thesis challenges the conventional approach to partitioning application code between secure and non-secure memory zones. While existing methods grapple with pointers, multi-threading complexities, and a two-partition limitation, our novel approach introduces secure typing as a more accurate technique. In detail, the thesis proposes two contributions.

FastSGX. Deploying legacy applications on Trusted Execution Environments (TEEs) for confidential computing appears to burden developers. Another drawback of using the Intel SGX TEE is the high cost associated with entering or leaving an enclave. Despite the effectiveness of switchless calls in addressing this issue, efficiently utilizing this technique remains challenging, mainly due to three issues: static worker thread configuration in the Intel SGX runtime leading to inefficient CPU resource usage, the call semantic causing unnecessary CPU wastage, and the current SGX runtime design hindering direct enclave-to-enclave calls, introducing additional control transfer overhead.

Our first contribution is FastSGX, a message-based runtime for SGX that overcomes the previously listed limitations. FastSGX not only provides dynamic adjustment of worker threads based on workload but also enables parallel code execution in the sender while a receiver processes a message. Furthermore, it allows control transfer from one enclave to another. Our evaluation shows that FastSGX consistently outperforms the Intel SGX-SDK.

Privagic. We propose in this thesis to ease the use of TEEs in legacy applications by using explicit secure typing. A secure type is a type enriched with a color, indicating in which enclave a value resides. We implemented

a compiler named Privagic, which takes as input an application with secure types and automatically partitions it for Intel SGX. Privagic generates parallel code by leveraging the fact that two instructions accessing two different colors do not have data dependency. We adapted FastSGX to build the Privagic runtime, facilitating the deployment of partitioned code on Intel SGX.

We evaluate Privagic with both micro- and macro-benchmarks. Overall, our evaluation shows that (i) using Privagic in legacy application requires a modest engineering effort, (ii) Privagic minimizes the TCB as compared to solutions that embed a complete application such as Scone, (iii) the code generated by Privagic is more efficient than a manually written code that relies on Intel SGX, and than an application fully embedded in an enclave with Scone.

Bibliography

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA, 2006.

[2] Frances E. Allen. Control flow analysis. *SIGPLAN Not.*, 5(7):1–19, jul 1970.

[3] Julien Amacher and Valerio Schiavoni. On the Performance of ARM TrustZone. In José Pereira and Laura Ricci, editors, *19th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS)*, volume LNCS-11534 of *Distributed Applications and Interoperable Systems*, pages 133–151, Kongens Lyngby, Denmark, June 2019. Springer International Publishing.

[4] Amazon. Aws nitro system, 2022.

[5] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, Johns Hopkins University, 1994.

[6] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. SCONE: Secure linux containers with intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 689–703, Savannah, GA, November 2016. USENIX Association.

[7] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick

McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, page 259–269, New York, NY, USA, 2014. Association for Computing Machinery.

[8] Godmar Back and Wilson Hsieh. The kaffeos java runtime system. *ACM Trans. Program. Lang. Syst.*, 27:583–630, 07 2005.

[9] Mike Barnett, Manuel Fahndrich, Francesco Logozzo, and Diego Garbervetsky. Annotations for (more) Precise Points-to Analysis. In *International Workshop on Aliasing, Confinement and Ownership in object-oriented programming (IWACO 2007)*, Berlin, Germany, 2007.

[10] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding applications from an untrusted cloud with haven. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 267–283, Broomfield, CO, October 2014. USENIX Association.

[11] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility safety and performance in the SPIN operating system. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, page 267–283. Association for Computing Machinery, 1995.

[12] Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Christof Fetzer, Peter Pietzuch, and Rüdiger Kapitza. Securekeeper: Confidential zookeeper using intel sgx. In *Proceedings of the 17th International Middleware Conference*, Middleware '16, New York, NY, USA, 2016. Association for Computing Machinery.

[13] David Brumley and Dawn Song. Privtrans: Automatically partitioning programs for privilege separation. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*, SSYM'04, page 5, USA, 2004. USENIX Association.

[14] David Bühler, Pascal Cuoq, Boris Yakobowski, Matthieu Lemerre, André Maroneze, Valentin Perelle, and Virgile Prevosto. Eva - the evolved value analysis plug-in.

[15] Stephen Checkoway and Hovav Shacham. Iago attacks: Why the system call api is a bad untrusted rpc interface. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, page 253–264, New York, NY, USA, 2013. Association for Computing Machinery.

[16] Lixia Chen, Jian Li, Ruhui Ma, Haibing Guan, and Hans-Arno Jacobsen. EnclaveCache: A Secure and Scalable Key-Value Cache in Multi-Tenant Clouds Using Intel SGX. In *Proceedings of the 20th International Middleware Conference*, Middleware '19, page 14–27, New York, NY, USA, 2019. Association for Computing Machinery.

[17] Stephen Chong, Jed Liu, Andrew C. Myers, Xin Qi, K. Vikram, Lantian Zheng, and Xin Zheng. Secure web applications via automatic partitioning. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, page 31–44, New York, NY, USA, 2007. Association for Computing Machinery.

[18] Google Cloud. Confidential computing, 2022.

[19] Confidential Computing Consortium. Confidential computing - open source community, 2022.

[20] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, page 143–154, New York, NY, USA, 2010. Association for Computing Machinery.

[21] Loïc Correnson, Pascal Cuoq, Florent Kirchner, André Maroneze, Virgile Prevosto, Armand Puccetti, Julien Signoles, and Boris Yakobowski. Frama-c user manual.

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

[23] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN*

Symposium on Principles of Programming Languages, POPL '77, page 238–252, New York, NY, USA, 1977. Association for Computing Machinery.

[24] Manuvir Das. Unification-based pointer analysis with directional assignments. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, page 35–46, New York, NY, USA, 2000. Association for Computing Machinery.

[25] Jérémie Decouchant, David Kozhaya, Vincent Rahli, and Jiangshan Yu. Damysus: Streamlined bft consensus leveraging trusted components. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 1–16, New York, NY, USA, 2022. Association for Computing Machinery.

[26] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, jul 1987.

[27] Nicolas Geoffray, Gaël Thomas, Gilles Muller, Pierre Parrend, Stéphane Frénot, and Bertil Folliot. I-jvm: a java virtual machine for component isolation in osgi. In *Proceedings of the international conference on Dependable Systems and Networks, DSN'09*, pages 544–553. IEEE Computer Society, 2009.

[28] Adrien Ghosn, James R. Larus, and Edouard Bugnion. Secured routines: Language-based construction of trusted execution environments. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 571–586, Renton, WA, July 2019. USENIX Association.

[29] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming, Revised Reprint*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2012.

[30] Galen Hunt and Jim Larus. Singularity: Rethinking the software stack. *ACM SIGOPS Operating Systems Review*, 41(2):37–49, April 2007.

[31] Jianyu Jiang, Xusheng Chen, TszOn Li, Cheng Wang, Tianxiang Shen, Shixiong Zhao, Heming Cui, Cho-Li Wang, and Fengwei Zhang. Uranus:

Simple, efficient SGX programming and its applications. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security (ASIACCS 2020)*, pages 826–840, Taipei, Taiwan, 2020.

[32] David Kaplan, Jeremy Powell, and Tom Woller. Amd memory encryption - whitepaper v9. Technical report, AMD, 10 2021.

[33] Taehoon Kim, Joongun Park, Jaewook Woo, Seungheun Jeon, and Jaehyuk Huh. Shieldstore: Shielded in-memory key-value storage with sgx. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, New York, NY, USA, 2019. Association for Computing Machinery.

[34] Joshua Lind, Christian Priebe, Divya Muthukumaran, Dan O'Keeffe, Pierre-Louis Aublin, Florian Kelbert, Tobias Reiher, David Goltzsche, David Eyers, Rüdiger Kapitza, Christof Fetzer, and Peter Pietzuch. Glamdring: Automatic application partitioning for intel SGX. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 285–298, Santa Clara, CA, July 2017. USENIX Association.

[35] Shen Liu, Gang Tan, and Trent Jaeger. Ptrsplit: Supporting general pointers in automatic program partitioning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 2359–2371, New York, NY, USA, 2017. Association for Computing Machinery.

[36] 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 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, page 1607–1619, New York, NY, USA, 2015. Association for Computing Machinery.

[37] Andrea Mambretti, Kaan Onarlioglu, Collin Mulliner, William Robertson, Engin Kirda, Federico Maggi, and Stefano Zanero. Trellis: Privilege separation for multi-user applications made easy. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, pages 437–456.

[38] Jämes Ménétrey, Marcelo Pasin, Pascal Felber, and Valerio Schiavoni. Twine: An embedded trusted runtime for webassembly. In *37th IEEE*

International Conference on Data Engineering, ICDE 2021, Chania, Greece, April 19-22, 2021, pages 205–216. IEEE, 2021.

[39] Maged M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. Parallel Distrib. Syst.*, 15(6):491–504, jun 2004.

[40] Microsoft. Microsoft azure confidential computing, 2022.

[41] Meni Orenbach, Pavel Lifshits, Marina Minkin, and Mark Silberstein. Eleos: Exitless os services for sgx enclaves. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, page 238–253, 2017.

[42] Sandro Pinto and Nuno Santos. Demystifying arm trustzone: A comprehensive survey. *ACM Computing Surveys*, 51:1–36, 01 2019.

[43] Christian Priebe, Divya Muthukumaran, Joshua Lind, Huanzhou Zhu, Shujie Cui, Vasily A Sartakov, and Peter Pietzuch. SGX-LKL: Securing the host OS interface for trusted execution. *arXiv preprint arXiv:1908.11143*, 2019.

[44] Atanas Rountev, Ana Milanova, and Barbara G. Ryder. Points-to analysis for java using annotated constraints. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '01, page 43–55, New York, NY, USA, 2001. Association for Computing Machinery.

[45] Konstantin Rubinov, Lucia Rosculete, Tulika Mitra, and Abhik Roychoudhury. Automated partitioning of android applications for trusted execution environments. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, page 923–934, New York, NY, USA, 2016. Association for Computing Machinery.

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

[47] Vasily A. Sartakov, Stefan Brenner, Sonia Ben Mokhtar, Sara Bouchenak, Gaël Thomas, and Rüdiger Kapitza. Eactors: Fast and

flexible trusted computing using sgx. In *Proceedings of the 19th International Middleware Conference*, Middleware '18, page 187–200, New York, NY, USA, 2018. Association for Computing Machinery.

[48] 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 *2015 IEEE Symposium on Security and Privacy (SSP 15)*, pages 38–54, San Jose, CA, USA, 2015. IEEE.

[49] Shweta Shinde, Dat Le Tien, Shruti Tople, and Prateek Saxena. Panoply: Low-TCB linux applications with SGX enclaves. In *24th Annual Network and Distributed System Security Symposium (NDSS 17)*, San Diego, CA, USA, 2017. The Internet Society.

[50] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, page 32–41, New York, NY, USA, 1996. Association for Computing Machinery.

[51] Meysam Taassori, Ali Shafiee, and Rajeev Balasubramonian. Vault: Reducing paging overheads in sgx with efficient integrity verification structures. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, page 665–678, New York, NY, USA, 2018. Association for Computing Machinery.

[52] Hongliang Tian, Qiong Zhang, Shoumeng Yan, Alex Rudnitsky, Liron Shacham, Ron Yariv, and Noam Milshten. Switchless calls made practical in intel sgx. In *Proceedings of the 3rd Workshop on System Software for Trusted Execution*, SysTEX '18, page 22–27, New York, NY, USA, 2018. Association for Computing Machinery.

[53] Chia-Che Tsai, Donald E Porter, and Mona Vij. Graphene-SGX: A practical library OS for unmodified applications on SGX. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 645–658, 2017.

[54] Chia-Che Tsai, Jeongseok Son, Bhushan Jain, John McAvey, Raluca Ada Popa, and Donald E. Porter. Civet: An efficient Java

partitioning framework for hardware enclaves. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 505–522, Online, August 2020. USENIX Association.

[55] Nico Weichbrodt, Pierre-Louis Aublin, and Rüdiger Kapitza. Sgx-perf: A performance analysis tool for intel sgx enclaves. In *Proceedings of the 19th International Middleware Conference*, Middleware '18, page 201–213, New York, NY, USA, 2018. Association for Computing Machinery.

[56] Ofir Weisse, Valeria Bertacco, and Todd Austin. Regaining lost cycles with hotcalls: A fast interface for sgx secure enclaves. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, page 81–93, New York, NY, USA, 2017. Association for Computing Machinery.

[57] Christian Wimmer, Codrut Stancu, Peter Hofer, Vojin Jovanovic, Paul Wögerer, Peter B. Kessler, Oleg Pliss, and Thomas Würthinger. Initialize once, start fast: Application initialization at build time. *Proc. ACM Program. Lang.*, 3(OOPSLA), oct 2019.

[58] Yongzheng Wu, Jun Sun, Yang Liu, and Jin Song Dong. Automatically partition software into least privilege components using dynamic data dependency analysis. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, ASE '13, page 323–333. IEEE Press, 2013.

[59] P. Yuhala, P. Felber, V. Schiavoni, and A. Tchana. Plinius: Secure and persistent machine learning model training. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 52–62, Los Alamitos, CA, USA, jun 2021. IEEE Computer Society.

[60] Peterson Yuhala, Hugo Guiroux, Jean-Pierre Lozi, Pascal Felber, Valerio Schiavoni, Alain Tchana, and Gaël Thomas. Secv: Secure code partitioning via multi-language secure values. In *Proceedings of the 24nd International Middleware Conference*, Middleware '23. Association for Computing Machinery, 2023.

[61] Peterson Yuhala, Jämes Ménétrey, Pascal Felber, Valerio Schiavoni, Alain Tchana, Gaël Thomas, Hugo Guiroux, and Jean-Pierre Lozi. Montsalvat: Intel sgx shielding for graalvm native images. In *Proceedings of the 22nd International Middleware Conference*, Middleware '21, page 352–364, New York, NY, USA, 2021. Association for Computing Machinery.

[62] Peterson Yuhala, Michael Paper, Timothée Zerbib, Pascal Felber, Valerio Schiavoni, and Alain Tchana. SGX switchless calls made configless. In *53rd Annual IEEE/IFIP International Conference on Dependable Systems and Network, DSN 2023, Porto, Portugal, June 27-30, 2023*, pages 229–238. IEEE, 2023.

[63] Peterson Yuhala, Michael Paper, Timothée Zerbib, Pascal Felber, Valerio Schiavoni, and Alain Tchana. SGX Switchless Calls Made Configless (PER). In *Proceedings of the international conference on Dependable Systems and Networks, DSN'23*. IEEE Computer Society, 2023.

[64] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. Secure program partitioning. *ACM Trans. Comput. Syst.*, 20(3):283–328, aug 2002.

[65] Lantian Zheng, Stephen Chong, Andrew C. Myers, and Steve Zdancewic. Using replication and partitioning to build secure distributed systems. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy*, SP '03, page 236, USA, 2003. IEEE Computer Society.

[66] Wenting Zheng, Ankur Dave, Jethro G Beekman, Raluca Ada Popa, Joseph E Gonzalez, and Ion Stoica. Opaque: An oblivious and encrypted distributed analytics platform. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 283–298, Boston, MA, USA, 2017.

ECOLE
DOCTORALE

Titre : Privagic : informatique confidentielle rendue pratique grâce au typage sécurisé

Mots clés : SGX, LLVM, sécurité et partitionnement de code

Résumé : L'informatique confidentielle consiste à protéger les données des utilisateurs lorsqu'elles sont traitées dans un système non fiable tel qu'une infrastructure cloud. Au niveau matériel, l'informatique confidentielle repose sur un environnement d'exécution fiable (TEE) (SGX d'Intel, SEV d'AMD ou TrustZone d'ARM). Un TEE est un environnement matériel qui isole une zone de mémoire, appelée enclave, d'un système d'exploitation ou d'un hyperviseur potentiellement compromis. Étant donné qu'il est difficile de partitionner manuellement une application entre une enclave et la mémoire non sécurisée, de nombreux outils de partitionnement automatique ont été proposés. Avec ces outils, le développeur annote certaines valeurs sensibles et l'outil analyse ensuite le code pour trouver les emplacements de mémoire dans lesquels les valeurs sensibles propagent. La plupart de ces outils se comportent incorrectement en présence de pointeurs. Lorsqu'ils sont corrects, ils ne parviennent pas à gérer les threads en raison de la difficulté à suivre les pointeurs dans une application multi-thread. Les outils actuels sont également incapables de diviser une application en plus de deux partitions. Cela est causé par une surapproximation, qui conduit à des emplacements mémoire faussement partagés entre les deux partitions.

Nous proposons de partir d'un autre point de l'espace de conception, en laissant le développeur annoter explicitement tous les emplacements de mémoire qui contiennent des valeurs sensibles. Puisque le développeur annote explicitement tous les emplacements de mémoire sensibles, il n'est pas nécessaire d'analyser le code. Nous évitons ainsi par construction tout risque d'erreur d'analyse dans une application multithread. Afin de permettre au développeur d'indiquer les emplacements de mémoire qui contiennent des valeurs sensibles, nous introduisons une nouvelle construction du langage appelée « type sécurisé ». Un type sécurisé est un type enrichi d'un identifiant d'enclave, que nous appelons une couleur.

Le typage sécurisé indique comment partitionner le code. En lui-même, le typage sécurisé ne fournit aucune garantie de sécurité. Pour renforcer la sécurité, nous proposons donc de compléter le typage sécurisé par des règles de typage. Le typage explicite de chaque emplacement de mémoire susceptible de contenir une valeur sensible rend possible le partitionnement d'une application multithread. L'ajout d'un type sécurisé à chaque emplacement mémoire sensible peut prendre beaucoup de temps au développeur. Pour cette raison, nous proposons de faciliter l'utilisation du typage sécurisé avec une forme simple d'inférence de type. Nous déduissons le type d'une variable locale non colorée, mais seulement si le code ne crée pas de pointeur sur la variable. Dans ce cas, la variable ne s'échappe pas de la portée d'une seule fonction, ce qui évite l'analyse inter procédurale. De plus, comme la variable ne s'échappe pas de la portée de sa fonction, elle ne peut pas être accédée par un autre thread. Avec cette restriction, la déduction d'un type sécurisé nécessite une simple analyse de la chaîne use-def, et le type déduit est correct même dans les applications multithread. Nous avons mis en œuvre notre principe de typage sécurisé dans le cadre Privagic pour Intel SGX et le langage C.

Le compilateur Privagic s'appuie sur le compilateur LLVM, ce qui signifie qu'il ne s'appuie pas sur la sémantique C : il considère une représentation intermédiaire de bas niveau du code avec des types sécurisés ajoutés aux variables, aux arguments et aux champs des structures de données. Notre évaluation avec des micro- et macro-applications montre que (i) le typage sécurisé peut gérer les pointeurs, le multi-threads et plus de deux partitions, (ii) l'ajout de types sécurisés dans une application existante est facile, (iii) le typage sécurisé réduit la base de confiance et est plus efficace que l'intégration complète d'une application dans une enclave.

Title : Privagic: confidential computing made practical with secure typing

Keywords : SGX, LLVM, security and code partitioning

Abstract : For more than twenty years, several tools have been proposed to automatically partition an application between a secure memory zone and a non-secure memory zone. These tools analyze the data flow of the application in order to identify the memory locations that may contain sensitive values. Most of these tools behave incorrectly in the presence of pointers. When they are correct, they are unable to handle threads because of the difficulty to track pointers in a multi-threaded application. The current tools are also unable to split an application in more than two partitions. This is caused by over-approximation, which leads to memory locations falsely shared between the two partitions.

In this thesis, instead of starting from data flow analysis, we propose to start from a more accurate technique: language typing. We introduce secure typing, which consists in embedding a partition identifier in the type system of a language. Based on secure typing, we designed a language-agnostic compiler based on LLVM. The compiler takes a legacy application enriched with secure types as input, and generates multiple partitions for Intel SGX. Our evaluation with micro- and macro-applications show that (i) secure typing can handle pointers, multiple threads and more than two partitions, (ii) adding secure types in a legacy application is easy, (iii) secure typing reduces the trusted computing base, and is more efficient than embedding a full application inside an enclave.

Institut Polytechnique de Paris
91120 Palaiseau, France