SecureLease: Maintaining Execution Control in The Wild using Intel SGX

Sandeep Kumar
School of Information Technology
IIT Delhi
New Delhi, India
sandeep.kumar@cse.iitd.ac.in

Abhisek Panda
Department of Computer Science and Engineering, IIT Delhi
New Delhi, India
abhisek.panda@cse.iitd.ac.in

Smruti R. Sarangi
Department of Computer Science and Engineering, IIT Delhi
New Delhi, India
srsarangi@cse.iitd.ac.in

ABSTRACT

Modern software programs have dedicated license-check modules that restrict access to users, who possess valid credentials. They also have a large number of add-on pluggable modules that can be separately purchased and have their dedicated license managers. Sadly, recent work shows that regardless of their complexity, it is possible to break their security using a novel class of attacks known as *control flow bending attacks* (CFB attacks), where the program is run on a virtual CPU, unbeknownst to it.

In this paper, we propose *SecureLease* – a novel approach that efficiently solves this problem by running the license managers and other parts of the application in a trusted execution environment (TEE) (hardware managed sandbox). Since it is not enough to just move the license managers to the TEE because they can be circumvented using CFB attacks, we need to further handicap the attacker by also moving key parts of the application to the TEE subject to various runtime and software engineering constraints. Hence, we propose a novel application partitioning algorithm that is far superior to the state of the art scheme. Furthermore, we propose an extremely fine-grained yet scalable license management and license distribution scheme. The key insight is that we avoid remote calls and intelligently use some degree of local caching. Hence, our license validation phase is 66.34% faster than the nearest competing work.

CCS CONCEPTS

• **Security and privacy** → **Security services**; **Authorization**; **Access control**.

KEYWORDS

Execution control, Intel SGX, Licenses, Leases

ACM Reference Format:
Sandeep Kumar, Abhisek Panda, and Smruti R. Sarangi. 2022. SecureLease: Maintaining Execution Control in The Wild using Intel SGX. In *23rd International Middleware Conference (Middleware '22), November 7–11, 2022, Quebec, QC, Canada.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3528535.3531514

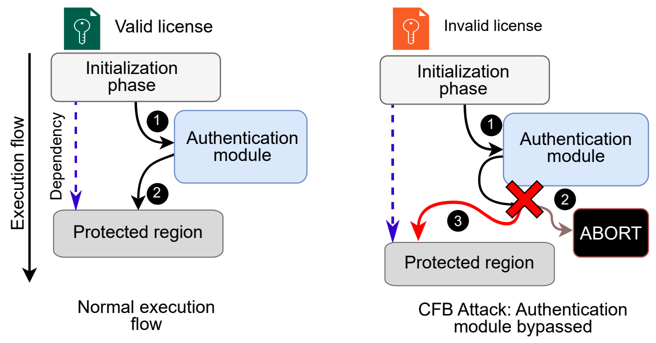


Figure 1: Figure showing execution flows in the case of a valid license file and during a *control flow bending* or CFB [32] attack to bypass the authentication module.

1 INTRODUCTION

Digital rights management (DRM) tools for protecting software are as old as commercial software itself. Today, it is almost unthinkable for a company to release a software without some form of DRM protection method such as a licensing or authentication mechanism. There is a need to restrict usage to only legitimate users who have purchased their rights. It is also not uncommon for a software bundle to have multiple DRM protection methods such as entering a license key and periodic online authentication. It is also possible to buy *add-ons* separately with restrictions on how they can be used. We shall say that any software (including an add-on) obtains an *ephemeral lease* before execution, with specific lease expiry conditions. The lease can be valid for either a fixed duration – *a time-based lease* – or for a fixed number of executions – *a count-based lease* [5, 10, 21]. We use the generic term *lease* for a license in this paper.

For natural reasons, DRM mechanisms are the prime targets of hackers and securing them is an important research area. Over the last 20 years, many influential works have been published [33, 35, 40, 44, 78] in the space of both attacks as well as countermeasures. One of the most powerful type of attacks in this space are *control flow bending* or CFB attacks [32, 56]. Here, the attacker forces a binary to take a certain execution path to execute a DRM "protected-logic", irrespective of the validity of the license provided. The insight is that a CFB attack tampers with the execution flow of an application by forcing it to take branches that would have been skipped in normal execution or forcing it to skip functions by running the application on a virtual CPU (like Intel Pin [62]) or by tampering with the binary (if possible). As shown in Figure 1, in the case of an invalid license file, the authentication module after verification will cause the execution to abort. However, the CFB attack forces the execution flow to proceed to the protected region (see Section 2.1.1 for more details).

F-LaaS [56] uses AI-based analyses to find the function within the execution that was responsible for the license check and then bypasses it – this approach was used to break many commercially used license managers [36, 76]. To secure a software against such attacks, the authors argue for a hardware-based countermeasure, yet fail to provide a practical and realizable implementation. The key idea that was proposed is to execute the license manager within a TEE like Intel SGX [19, 37]. This is not enough, because this part can be easily circumvented. The other solution is to run the entire binary within SGX. However, this is associated with high-performance overheads (see Section 2.3.2). A good solution is to identify key parts of the application and move them to SGX if it is possible to easily do so. This will severely handicap an attacker even if she manages to bypass the license check.

We leverage a growing body of work [34, 41, 42, 56, 60, 61, 63, 67, 68] that proposes to partition applications into secure and unsecure regions for various purposes. We show that a naive partitioning can be quite slow (2000× slowdown), and thus we propose an intelligent call graph clustering based partitioning scheme that outperforms state-of-the-art partitioning schemes.

The next step is to support a flexible, fine-grained license management scheme such that a "partitioned application" can seamlessly run on Intel SGX and use a large number of licenses — one for each add-on module. Most add-on modules today do not come with perpetual licenses. Instead, they use complex licenses that place limitations on the duration of use, number of uses, etc. We show in Section 4.3 that we can use a generic count-based lease (GCL) to implement all such kinds of licenses. The key idea here is to decrement the count based on the fulfillment of some condition. Once the counter reaches zero, the lease is deemed to have *expired*. We cannot make a call to the server to obtain such a lease every single time (large overheads). We thus propose an algorithm to efficiently locally cache leases in a *secure* manner.

The key idea is to predictively pre-distribute leases to client machines, which run applications that will require the leases in the future. This will eliminate the need for network communication. Hence, the last piece of this puzzle is to propose a novel lease distribution scheme that is sensitive to the demand on the client machine, its crash probability, the network conditions, the overall demand at the server, and the reserve that the server must keep.

Contributions: To summarize, the setting that we consider is as follows: a machine runs multiple complex software packages, where each package has a large number of add-on modules with separate licensing mechanisms. Each application is partitioned into secure and unsecure regions. All the licensing modules are in the secure region along with the code of a few *key* functions (functions intended to be protected by the license module, see Section 4.2.1). Lease acquisition is a frequent process where an add-on module obtains it before executing, and thus for efficiency we have secure local caching of leases as well as predictive distribution by a remote server to minimize costly network communication. The software developer assumes that the client machine may mount CFB attacks, but it will then be severely handicapped because it will not be able to run the key functions since they require a valid lease.

Our contributions can thus be summarized as follows:

- We present a secure, efficient, and scalable solution for running complex software packages in a manner that is immune to CFB attacks.
- We propose a novel application partitioning scheme that improves performance by 32.62% over the current state of the art solution, Glamdring [61].
- We propose a novel method to grant secure leases, even in the absence of a stable network connection. Subsequently, we propose a method to efficiently pre-distribute leases keeping various constraints in mind.
- We implement SecureLease on a real system and show that it outperforms the current state of the art solution [56] by an average of 66.34%.

The rest of the paper is organized as follows. We discuss the relevant background and motivation for the paper in Section 2.The related work is discussed in Section 3. This is followed by the design of SecureLease in Section 4 and its implementation in Section 5. Section 6 provides a detailed security analysis. Section 7 discusses the performance results. Finally, we conclude in Section 8.

2 BACKGROUND AND MOTIVATION

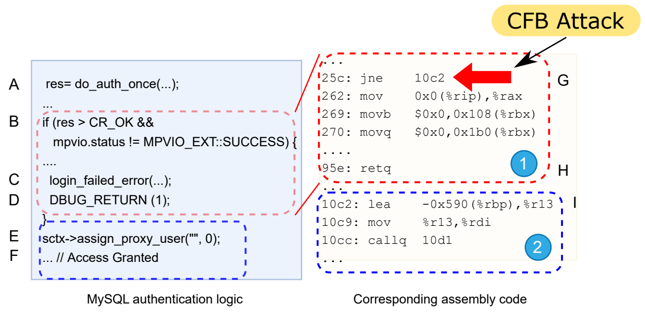
In this section, we discuss the relevant background required for the rest of the paper, and motivate the need for a better software authentication mechanism using Intel SGX.

2.1 Software-based Authentication Modules

Traditionally, the execution of an application (or a part of it) is guarded by an *authentication module* (or AM); it is also popularly called a license manager [36, 56, 76]. The authentication module comprises many different functions that are responsible for processing the input – a license file – from the user.

An authentication module typically acts as a black box with a single access point: *authentication function*. The user calls this function with the license file as its argument. The authentication module processes the license file, either locally or on a remote machine, and either returns *success* indicating a valid license file or *failure* indicating an invalid license file. The outcome is used to decide the path of the execution. As of today, this is the most widely used method to authorize the execution of an application [21, 36, 76]. The authentication module is generally not dependent on any other component of the application and vice-versa. We can have an AM for the full application or separate AMs for different add-on components [29, 30, 59].

2.1.1 Attack on Software-based Authentication Modules. As shown in related work [32, 56], a software-based authentication module is vulnerable to *control flow bending* (or CFB) attacks. A CFB attack



```
 res= do_auth_once(...);
...
if (res > CR_OK &&
   mpvio.status != MPVIO_EXT::SUCCESS) {
....
  login_failed_error(...);
  DBUG_RETURN (1);
}
sctx->assign_proxy_user("", 0);
... // Access Granted
```

```
...
25c: jne     10c2
262: mov     0x0(%rip),%rax
269: movb    $0x0,0x108(%rbx)
270: movq    $0x0,0x1b0(%rbx)
....
95e: retq
...
10c2: lea     -0x590(%rbp),%r13
10c9: mov     %r13,%rdi
10cc: callq   10d1
...
```

Figure 2: Authentication logic of MySQL 5.7 and its corresponding assembly code. The license check function `do_auth_once()` is guarding the access to the protected region.

aims to bypass the authentication module by allowing the execution of the "protected region" of the application even in the absence of a valid license file. The first step in a CFB attack is to identify the authentication function while the binary is executing. This can be done by analyzing the control flow graph (CFG) of the application's execution. We can follow a supervised approach where we compare the CFG of two executions – one with a valid license file and one without it. The other method is to use an unsupervised approach, where we try to guess the authentication function using multiple execution traces of the application [56].

Once the attacker identifies the authentication function, she can try to flip the branch instruction that makes the final decision, skip the function altogether, or skip a few related functions and possibly change the state of the program to reflect the fact that the license check has successfully passed. Given that we are running the program on a virtual CPU, we have full access to the internal state of the CPU, including the registers and memory, unbeknownst to the secure program. If it is possible to tamper with the binary, then we can also change the corresponding instructions and distribute it to others such that they can run the "cracked binary" on their CPU.

This attack is also feasible if we only execute the authentication module in SGX; note that SGX does not allow any other process to read or modify the code/data that resides in the secure region of main memory. Here, even though we cannot tamper with the execution flow of the authentication module, we always have the option of skipping the entire secure license-verification code by cracking the unsecure part of the application. For example, Figure 2 shows the authentication logic of MySQL 5.7[1]. The authentication module (`do_auth_once()`) is protecting the execution of the protected region (Line *E* onwards). Figure 2 also shows the corresponding assembly code. The `jne` instruction in Line *G* in Block 1 of the assembly code is controlling the access to the instructions in Block 1 and Block 2 . In the case of a valid license, the inequality condition for `jne` evaluates to `false`, and we execute the rest of the instructions in Block 1 If the license check fails, the inequality condition for `jne` evaluates to `true` and the branch is taken, resulting in the program to exit. The attack simply forces the `jne` instruction to not take the branch (even when the condition is `true`) [32, 56] resulting in the execution of the rest of the instructions in Block 1 . This can be easily achieved by running the application on a CPU emulator such as the Intel Pin tool [62]. This breaks the security of MySQL.

A CFB attack leverages the lack of dependency between the authentication module and the rest of the application. This is a key shortcoming of existing authentication mechanisms – once the AM's security is circumvented, the rest of the program can be executed seamlessly. This is why it is necessary to move other functions within the application to SGX to encumber a potential attacker.

[1]file: sql/auth/sql_authentication.cc

2.2 FaaS- and Plugin-based Applications

Modern distributed applications rely on many third-party components regardless of the specific architecture: client-server (CS) or peer-to-peer (P2P). Consider the case of CS applications first. Applications such as Matlab and Visual Studio use many third-party add-ons and extensions [14, 18, 22]. They rely on a multitude of third-party servers to provide these extensions. We can view them as plugin servers and the machine running the application as a client. Such an architecture is chosen because the vendor of the software typically does not have the expertise or resources to design all the add-ons. Hence, it is better to provide interfaces for third parties to develop add-ons; they also have the required expertise. For example, Visual Studio Code offers 30,000+ extensions [22] and Matlab offers over 10,000 functions bundled into 150+ toolboxes [18]. Furthermore, plugins in this space are beginning to support pay-per-use models [1, 8], and they thus require an elaborate license checking and access tracking mechanism. Now, let's consider tens of users running their jobs on such a machine (e.g., a university setting). We will need a very elaborate license-check mechanism that simultaneously supports 100s of license checks.

Now consider P2P systems. Modern applications are increasingly adopting a microservice (serverless design, FaaS) to manage and consolidate their services [38, 43, 52]. Such a design makes an application easy to manage, scale, and debug by splitting it into small services or functions, which are then deployed on different machines with frequent communication between them. For example, Netflix uses a serverless design where thousands of function calls [16] are made to handle different aspects of the platform such as video uploading, processing, distribution, and backup. Even a traditional company like Coca-Cola also uses a serverless design to collect information from thousands of vending machines and use this information to optimize the distribution mechanism [20]. Given that we have thousands of differentiated services provided by a host of parties, which are themselves running on a cloud architecture, each machine needs to track the usage of hundreds of such FaaS functions and account for them.

Given that distributed applications are scaling and the diversity of solutions is increasing, solutions for secure and scalable leasing mechanisms such as ours will continue to become more and more relevant.

2.3 Intel SGX

Intel Software Guard eXtension or SGX provides a secure way to execute an application on a remote, untrusted machine by creating a secure sandbox called an *enclave*. At boot time, SGX reserves a part of the main memory for its operation. This region is called the *Processor Reserved Memory* (or PRM) and is managed by the hardware. The size of the PRM is limited to 128 MB, and out of this, only ≈92 MB, called the *enclave page cache* (or EPC) is available for user applications. Intel SGX ensures the confidentiality, integrity, and freshness of the data stored in the EPC. Any application is by default unsecure; however, it can execute a few functions within SGX by running them within an enclave. Henceforth, the code and data in the enclave cannot be accessed or tampered with by any other process including the OS. Furthermore, it is possible to create a binary where some functions are encrypted and then they are decrypted within SGX (in the context of an enclave). This may require the services of other servers who first verify that the encrypted function is being executed in a valid enclave, and then they provide the key to decrypt it. If an application executing within SGX requires more memory than 92 MB, SGX transparently evicts pages from the EPC to the unsecure region of the main memory. SGX transparently handles a page fault for an evicted page and loads the page back into the EPC. Intel recently announced a scalable version of SGX where the EPC memory can be scaled up to 512 GB [50]. We discuss its pros, cons, and relevance of our work in its context in Section 7.5.

Local and Remote Attestation: Often there is a need for an enclave to communicate with another enclave on the same machine or on a different machine. They need to verify each other before initiating any communication. The developer needs to code the secure applications with identifiers about the other enclaves with which contact is to be established. For establishing contact locally (same machine), local attestation needs to be performed where digitally signed reports (mutually known information) are exchanged. Remote attestation (or RA) has a slightly different process where a hardware-generated report is sent to a third server that is trusted. The remote server uses the third server for establishing the bonafides of the client machine. As seen in our experiments, a single RA call takes 3-4 seconds. This is unsuitable for workloads that either have a short execution duration such as FaaS workloads (a few seconds [71]) or require a large number of license checks.

2.3.1 Execution control in SGX. By default, SGX only protects the integrity of the code, not its confidentiality. To enable confidentiality, SGX provides a feature called the *protected code loader* or PCL [49] that allows the execution of an encrypted enclave on a remote, untrusted machine. Here, the application contains encrypted functions that are only decrypted at runtime if the enclave has valid credentials. For this, we need the help of trusted servers. Once the application initiates a request to execute encrypted code in an enclave, a complicated chain of events starts. First, it is necessary to prove to a remote server that the enclave is a valid SGX enclave running on a trusted machine. Once that is done, we need to get a key to decrypt the encrypted code if the user holds valid credentials. The key itself is encrypted and is fetched over the network and handed over to the enclave, where the key is extracted by hardware and is used to decrypt the encrypted code within the enclave. The decrypted code is not visible to other processes including the unsecure part of the same process. The sad part is that this is a one-time activity and cannot be used to implement a lease because once an application has decrypted the secure code, it can continue use it. However, there is a solution to this problem, which is to embed the leasing logic within the secure code. The secure code will search for the existence of a lease and contact remote servers to verify the validity of leases and fetch more, if necessary. However, this results in a very high performance overhead (see Section 7).

2.3.2 SGX Performance Overheads. Intel SGX can be used to prevent CFB attacks against an application. An attacker cannot tamper with the execution flow of an application executing within SGX. However, executing a complete application in SGX can result in a slowdown of over 300× (HashJoin in Figure 9). The reasons for this overhead are a large number of EPC faults (a single fault can take up to 12,000 cycles [77] to service) and frequent OS interactions (involve TLB flushes). The SGX framework puts several restrictions on an application executing within it; the most notable being no direct access to the OS as the OS is not a part of the TCB [19]. To access an OS service, the application must do an `OCALL`. Similarly, to access a function inside an enclave, an `ECALL` is required. Weisse et al. [77] show that the cost of calling an enclave function (an `ECALL`) typically requires 17,000 cycles. Furthermore, porting a complete application to SGX is a non-trivial task mainly due to the limitations imposed by the SGX framework [45] and the concomitant software engineering challenges.

3 RELATED WORK

We envisage a system that runs a large application securely by partitioning the binary into *secure* and *unsecure* parts, where the secure regions are small and are executed within SGX. There is a plethora of research [34, 41, 42, 56, 60, 61, 63, 67, 68] on reducing the performance overheads of SGX by partitioning an application, and these ideas have been shown to be effective. In general, an application can be automatically partitioned using two methods: based on the sensitive data present in an application [61] (*data-based*) or based on some key events during execution [41, 56] (*code-based*).

In the data-based approach, developers annotate the source code by marking certain data structures as sensitive. Then, an information flow tracking mechanism is used to track the functions that can access these sensitive data structures. All of these functions are marked as secure and migrated to SGX at runtime. Lind et al. [61] use this method in their work *Glamdring*. However, any such approach employed to protect the authentication module will only migrate all the authentication module functions to SGX. This is because in modern applications, the authentication module is independent of other functions in the applications. The license provided by the user is processed only by the functions present in the authentication module and is not accessed by any other function in the application. However, a CFB attack can still be mounted in such cases where only the authentication module is executing within SGX— free from any tampering. In such cases, a CFB attack does not interfere with the working of the authentication module *but with its outcome.* For example, in MySQL the `do_auth_once()` will be migrated to SGX as a part of the authentication module (see Section 2.1.1). However, its output (stored in "res") is processed outside SGX, and the attack forces the execution to ignore its value by inserting a direct jump.

In the code-based approach, an application is partitioned based on important events such as specific function calls. Geneiatakis et al. [41] analyze the trace of a binary's execution to identify the call to the authentication function, and partition the application into *pre-authentication* and *post-authentication* phases. The authors argue that this partition can be used to implement different protection mechanisms in different regions based on the application's requirements. However, once the application is partitioned, it is unclear what should be migrated to SGX. Migrating the entire post-authentication phase functions might be riddled with the same issues of performance and porting.

Kumar et al. [56] propose to migrate functions that have a high out-degree, i.e., they make a large number of functions calls. The idea is that a function making many functions calls is orchestrating a complicated piece of logic, and migrating it inside SGX will render the application useless. However, they do not take into consideration the overheads due to `ECALL`s, `OCALL`s, and total EPC usage. We implemented their scheme on a real-SGX machine and found that this type of partitioning incurs an overhead of up to 2000×. Hence, we need a better way to partition the application.

3.1 Takeaways

The key takeaways are as follows:

(1) Software-only solutions are susceptible to CFB attacks and are thus not fully capable to protect modern applications.
(2) Intel SGX can prevent a CFB attack by executing a complete application within an enclave. However, doing so incurs a significant performance overhead (over 300×) along with concomitant porting issues.
(3) Current partitioning schemes for applications are ill suited to prevent CFB attacks.
(4) Remote attestations are costly and must be used as sparingly as possible.

Hence, there is a requirement for a secure, optimal, and scalable way of providing fine-grained control over the execution of an application on an untrusted system.

4 KEY DESIGN PRINCIPLES

Our *two key contributions* are a novel method of application partitioning that protects the intellectual property (IP) within an application, and more importantly a technique to mitigate the cost of costly remote verification of fine-grained leases.

4.1 Threat Model

We assume that the attacker has the complete binary of the application with her (the binary may have several encrypted functions). The execution of the application or certain parts of it are protected (encrypted and secured) by corresponding authentication modules and are only allowed to execute when the user possesses a valid license. The attacker is free to execute the application in any setting (a real/virtual machine or a CPU simulator.). SecureLease should be able to handle all such scenarios. As is standard with prior work in this domain, denial-of-service attacks or side-channel attacks are beyond the scope of this paper.

4.2 Dependency-based Partitioning

We propose a *dependency-based* partitioning algorithm, which can prevent CFB attacks while ensuring a minimal performance overhead. The key idea is to add a dependency between the authentication module and its corresponding protected region, such that if an attacker bypasses the authentication module, the binary will be rendered handicapped, i.e., not capable of producing any meaningful results. To do so, along with the authentication module, we also migrate a set of functions from the corresponding protected region to SGX. The access to those functions is only guaranteed if the user possesses a valid license file. If the user attempts to bypass the authentication module using a CFB attack, she will not have access to the other migrated functions and those functions will check for a valid lease when they are executing within an enclave.

We use the following observation to select the functions that are to be migrated to SGX while ensuring a minimal performance overhead. A modern application has a high-degree of modularity [11, 12, 53]. The submodules present in an application, including the authentication module show up as distinct clusters in its control-flow graph or CFG (Figure 7). In a CFG of an application, the nodes represent functions in the application and directed edges between the nodes represent a function call from the source node to the destination node.

Observation: The total number of intra-cluster function calls is much higher than the total number of inter-cluster function calls (see Figure 7). Moving a part of a cluster to SGX while the making the rest execute in the untrusted region will cause a high number of `ECALL`s and `OCALL`s. Hence, we migrate entire submodules to SGX.

4.2.1 Partitioning Algorithm: Let N be all the nodes (key functions) in the protected region of the authentication module (see Section 2.1.1). A "key" function is a function, which has a disproportionate amount of importance in implementing the logic of the program. These are typically identified by the developer. Many prior works [34, 41, 42, 61, 63, 67, 68] have made the same assumption, where they assume that the developer is aware of these functions and willing to annotate them. Furthermore, the code footprint of these functions is quite low as seen in our experiments and also others (20%-40% by Lind et al. [61]). If the developer annotations are not available, then it has been shown by Kumar et al. [56] that we can do a clustering-based analysis and treat functions that appear in a cluster as key functions based on the total number of calls made from and to the cluster. We have done something similar, albeit using a bespoke, SGX-specific approach.

We use the K-Means [51] clustering algorithm on the CFG of the application to identify k clusters from N nodes using the directed edges between the nodes. Let us say, C is the set of k identified clusters, with $c_i \in C, 0 \leq i < k$. To ensure optimal performance, we use the observation made by Hasan et al. [45]; we define two thresholds, m_t and r_t, which determine the amount of memory used by functions executing in SGX and the acceptable performance overhead, respectively. Hasan et al. [45] report that an application executing in SGX incurs negligible performance overheads if the memory footprint of the application is less than the EPC size i.e., 92 MB, and it performs a limited number of `ECALL`s and `OCALL`s.

After clustering, we sort C (all the clusters) in terms of the memory requirements of the individual clusters in increasing order. We use the standard `proc` interface to estimate the memory usage of a function [17, 57]. This value is used while "compiling" the SGX application as SGX requires the memory requirements to be stated upfront. We further fine tune the total amount of memory required by using the EMMT tool [7]. After sorting, we start adding clusters in increasing order (smallest first) till the total memory requirement of the set reaches m_t and simultaneously the performance overhead induced is less than r_t. After this step, all the functions in the set of clusters are migrated to SGX. Furthermore, we also move all the common data structures to the untrusted region of the application. This is required as functions that are not migrated to SGX cannot access data from the EPC. This is also in tandem with our goal of protecting the IP embedded within an application and not its data.

4.3 Modeling Lease Types

A typical leasing (or licensing) software supports different kinds of leases: perpetual lease, time-based lease, execution time-based lease, and count-based lease. A perpetual lease allows unrestricted access to the application. A time-based lease is valid for a fixed time, and a user can only execute the application during that period. An execution time-based lease puts a limitation on the amount of time an application can execute. Finally, a count-based lease restricts the execution of an application to a fixed number of times.

We argue that lease managers can use a *generalized counter-based lease (GCL)* to model all types of leases – here, we assume that a lease has an associated counter, which is modified based on certain criteria (as we describe next).

Consider a standard time-based lease that says that a software will run in the "evaluation mode" for the next 30 days and during this time the user needs to purchase a valid license. We can discretize the time into 1-day intervals, and increment the GCL counter once every day. Additional state information is required to note the times at which the last measurement was collected. If the system stays off for some time, then the GCL needs to be appropriately updated the next time the system turns on and the counter is updated. We can do something similar for a lease that is based on just the execution time. For a perpetual lease, the counter modification operation is vacuous and we can just have a binary value (software activated or not). Revoking a license will just involve setting the counter to 0. This abstraction is well correlated with the way that leases are actually implemented in popular license managers [5, 10, 21].

Needless to say, we need a secure way of accessing the GCL counter, maintaining its integrity, and securely storing it (along with associated state information) when the system shuts down. In Section 5, we shall explain how SecureLease ensures that all of these requirements are satisfied.

4.4 Lease Management

Figure 3 shows the high-level design of our novel solution. The design introduces three new components: *SL-Remote, SL-Manager, and SL-Local.* SL-Remote is a trusted remote server, SL-Manager is the authentication module added to the application, and SL-Local is a local service that handles the license check requests from the enclaves running on the same machine. The developer is responsible

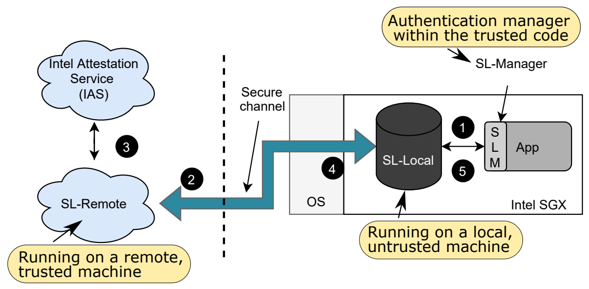

Authentication manager within the trusted code
SL-Manager
Intel Attestation Service (IAS)
Secure channel
3
2
4
1
5
SL-Remote
SL-Local
S L M
App
OS
Intel SGX
Running on a remote, trusted machine
Running on a local, untrusted machine

Figure 3: A high-level design of SecureLease.

for partitioning the applications. Furthermore, she also creates a GCL corresponding to every partition that needs to be separately accessed and leased.

The main idea here is that SL-Local manages a snapshot of *leases* on the local machine obtained from SL-Remote, which enforces the leases. SL-Local uses the leases to attest executions on the same machine, thus avoiding the costly remote attestation process. Here is the workflow:

❶ SL-Manager first performs a local attestation with SL-Local to ensure that the service is available. After that, it collects the license information from the user and forwards it to SL-Local for verification.

❷ SL-Local checks its local database to see if it has a valid GCL for the license. If it has, it updates its entries and sends a valid *token of execution* to SL-Manager (corresponding to the GCL). The token can be anything from a simple Boolean value to a data packet. Upon receiving the token, SL-Manager allows the execution to proceed.

❸ However, if SL-Local does not have any valid GCLs corresponding to the license information provided, it forwards the license file to SL-Remote and asks for a lease (or renewal). SL-Remote, after validating the license, may send a new/updated GCL to SL-Local. As a part of this process, SL-Remote also validates SL-Local. At any point in time, if the license information is not valid, then no further executions are allowed for that license file. This represents a situation where an attempt has been made to breach the security.

5 IMPLEMENTATION

In this section, we discuss the implementation of SecureLease. As discussed in Section 4, there are three main components of SecureLease: the remote server component (SL-Remote), the local component (SL-Local), and the SL-Manager, which is a part of an application (secure region).

5.1 SL-Remote & SL-Manager

SL-Remote is responsible for issuing licenses, and maintaining data for all the active SL-Locals. An SL-Manager instance is responsible for contacting SL-Local and presenting it with the license information provided by the user. If the license is valid, SL-Local will provide a *token of execution* that indicates that the execution can proceed. Note that SL-Manager is executed within SGX and is free from tampering. The developer is responsible for augmenting the secure component with an SL-Manager.

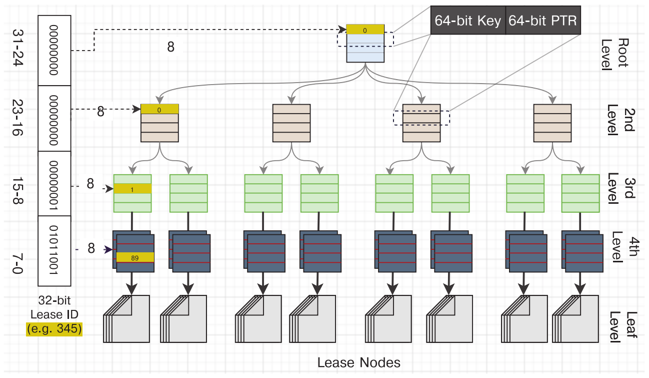


Figure 4: Organization of leases. We index into different levels of the trees using the bits in the lease ID.

5.2 SL-Local

SL-Local is a local service running within Intel SGX that attests the execution requests (license check requests) from applications running on the same machine. It maintains a set of leases, which are obtained from SL-Remote.

5.2.1 Design of SL-Local. A single server in a data center can have multiple VMs and containers that can host thousands of applications (though only a few are active at the same time) [29, 30, 47, 59]. Furthermore, each of those applications can have multiple regions that need to be protected by different leases (for example, an application using a large number of "add-ons"). Hence, in a typical scenario, at a given time, a single system might require to hold 1000-5000 leases (similar numbers reported in [29, 30, 59]).

There are many different ways to organize the lease data (and the corresponding metadata). The techniques could be array-based, hash table-based, or tree-based. They all have their pros and cons in terms of operational efficiency, simplicity, and memory utilization. We implement a paging-like mechanism where unused leases can be offloaded to the untrusted region to save the precious EPC memory. To enable this, the metadata entry for each lease contains: a 64-bit encryption key (used for encrypting leases before offloading) and a 64-bit pointer (points to a dynamically allocated lease, `NULL` if offloaded). This remains the same across all the techniques. To decide what will be the best scheme, we first list down the key requirements (for SL-Local):

(1) Efficient handling of concurrent attestation requests for the same or a different lease. Hence, it should have an efficient mechanism to find a lease.
(2) All the SL-Local data is stored in the EPC. However, as the size of the EPC is limited and is shared across enclaves, SL-Local should efficiently use this memory.
(3) SL-Local should support an efficient "shutdown" and "re-initialization" procedure while ensuring confidentiality, integrity, and freshness of the leases.

5.2.2 A Tree-Based Design. Inspired by how an OS maintains a page table, we organize the leases present in SL-Local in a 4-level tree structure (see Figure 4). We call it a *lease tree*. Here, all the nodes in the tree are 4 KB each (page size). Each node contains 256 entries of size 16 B each. Each entry consists of a 64-bit key and a 64-bit pointer that points to a node in the next level (`NULL` if the node has not been allocated yet). Each lease is assigned a 32-bit unique ID. Its bits are used to index into the lease tree, just like a page table. As we have 256 bits in every node, we need 8 bits to index into it ($2^8 = 256$). Each of the lease data structures (at the leaf level) contains a single GCL and is pointed to by an entry present in its parent node in the fourth level of the tree. The size of a lease is 312 B. It contains a 32-bit lock, 64-bit hash, and 300 B for the lease data [36, 76].

Running example: We explain how we locate a lease in the tree using an example (say accessing a lease with ID 345). Figure 4 shows how we use the bits of the lease ID to locate the lease. The first 8 MSB bits of the ID are used to index into the root level node (0^{th} entry in the example) and fetch the pointer to the node in the second level. Subsequently, the next 8 bits in the ID are used to index into the previously fetched node (again 0^{th} entry in the example) and fetch a node in the third level. We repeat this process. The pointer in the indexed entry of the last node will give us access to the desired lease data structure.

Memory efficiency: Since the lease tree is stored in the EPC, we want to ensure that its memory requirement is as little as possible. A tree-based implementation provides a natural solution for our limited memory problem. We only create internal nodes if they are required. Furthermore, a subtree can be offloaded to the untrusted region if it is not in use (like data pages in a page table [57]) after "committing" it (see Section 5.6). The ACIF (authenticity, confidentiality, integrity and freshness) properties of the evicted data are ensured by always keeping a root-of-trust (a trusted entity [39]), i.e., the root node in the EPC. Furthermore, the leases within an application are allocated in such a manner that they exhibit spatial locality, i.e., all the leases of an application can be within the same 4^{th} level node (if the total number of leases required is less than 256).

5.2.3 Performance of the Tree-Based Design. We analyzed the `find()` performance for a tree-based SL-Local and a hash-based SL-Local. We implemented two variants of a hash table using MurmurHash (used to implement the unordered map in C++ STL) [13] and SHA-256 [65]. The latency of the `find()` operation in (μs) is shown in Table 1.

Table 1: Lookup peformance for different schemes.

	Lease Ops			
Technique	10	100	1,000	5,000
Murmur Hash	40 μs	52 μs	144 μs	440 μs
SHA-256	149 μs	182 μs	742 μs	1,803 μs
Tree	26 μs	33 μs	61 μs	184 μs

As can be seen, for 5000 operations, a tree-based implementation outperforms a hash table-based approach based on MurmurHash and SHA-256 by 58% and 89%, respectively. This is due to the time taken for computing the hashing function. Once a lease is found, the update operations remain the same across different schemes. Furthermore, a tree-based design outperforms an array-based or hash table-based design by up to 94% in terms of the memory footprint since we can also offload metadata nodes in a tree structure while doing so is a non-trivial operation in an array or hash table

Table 2: Terminology

Notations	
L	License to be renewed
TG	Total #GCLs corresponding to a license.
g	Allocated GCLs.
C	Number of concurrent requests for L
α_i	Weight of node i, s.t. $\sum_{i=0}^{C-1} \alpha_i = 1$
n	Network reliability $\in [0, 1]$. 0: Dead Network, 1: Stable Network
h	Node Health$\in [0, 1]$ 0: High Crash Prob., 1: Low Crash Prob.
β	Per-license scale down factor to bound the expected losses $\in [0, 1]$
τ	Maximum total expected GCL loss
D	Lease scaling factor

structure [58]. Hence, we opted for a tree-based design for implementing SL-Local.

5.2.4 SL-Local Initialization. During initialization, SL-Local performs two key operations: establishing a secure connection with SL-Remote and restoration of any previously stored state on the untrusted client machine. For the first operation, SL-Local reads its unique id (SLID) from a plaintext file (NULL if this is the first initialization). A SLID is a unique id assigned to each SL-Local by SL-Remote to uniquely identify it. SL-Local then initiates an `init()` procedure with SL-Remote. SL-Remote performs a remote attestation to ensure that SL-Local is genuine.

For the next operation, after successful validation, SL-Local receives its SLID (if it was `NULL` before, otherwise it remains unchanged) and a 64-bit key, which we call the old backup-key ($\mathcal{OB}_{\mathcal{K}}$). This key is then used to restore any locally saved state (discussed in Section 5.6).

5.3 Adaptive GCL Renewal

Once the initialization of an SL-Local is done, it needs to fetch GCLs from SL-Remote before it can handle attestation requests from an application locally. SecureLease imposes strict criteria that say that if a client system or SL-Local crashes for any reason, all the valid GCLs currently stored on the client machine are lost. This is required to prevent replay attacks on an instance of SL-Local (more details in Section 5.7). Hence, SL-Remote employs heuristics while distributing GCLs to SL-Locals such that the total expected loss of the GCLs is bounded ($< \tau$).

We now explain the adaptive lease allocation mechanism. Table 2 explains the terminology. While issuing a GCL to an SL-Local, SL-Remote considers different factors such as the network delay (n), stability of the client (h), and the number of concurrent requests (C) for the same lease (see Table 2). The complete algorithm for lease renewal is shown in Algorithm 1. We assume that a developer wants to restrict the execution of the application corresponding to a particular license to a total of TG times. Here, the assumption is that all the client machines that request leases are part of a multi-party group and share a set of licenses.

Concurrent requests: SL-Remote first gets the total number of concurrent requests for the license. If there is more than one node accessing the license, SL-Remote computes the share of the requesting node i, (G_i) (Line 3). SL-Remote ensures that $\sum_{i=0}^{C-1} G_i \leq TG$ is satisfied. Here, C is the total number of concurrent instances of the application.

Sub-GCL: Now, if SL-Remote assigns the full G_i to the node i, the loss will be G_i in case the node goes down immediately. Hence, to limit the losses, SL-Remote first applies a default reduction policy (Line 4). Doing so ensures that it can handle subsequent requests for the same license from other nodes and also restricts the loss suffered in case a node crashes. SL-Remote employs a configurable policy (D) that controls the number of GCLs given to a node. The scaled-down G_i is henceforth called a sub-GCL (g_i). A sub-GCL allows only g_i executions of the applications to be handled locally on node i by SL-Local before renewing the lease again. Intuitively, $g_i = G_i/D$.

Node and Network health: SL-Remote imposes a penalty if the node health (h) is low (Line 5). However, SL-Remote does not impose any penalty for network connection issues; in fact, it allocates more leases to a node that is suffering from constant losses in connectivity to ensure that it has sufficient resources to handle any local allocation requests. This is only done for the nodes with health greater than a threshold (T_H), subject to a limitation that it cannot be more than G_i (Line 7).

Restricting losses: Furthermore, each license is associated with an upper bound of the expected loss (τ). SL-Remote ensures that the expected loss of the license is always less than τ on every node it is assigned to. In order to do so, SecureLease uses a global per-license scale down factor (β). Expected loss (ExpLoss) for a license L can be calculated as follows:

$$ExpLoss(L) = \sum_{i=0}^{C-1} g_i * (1 - h_i) \tag{1}$$

where, $(1 - h_i)$ is the crash probability for the node i, and g_i is the sub-GCL assigned to the same node. g_i can be calculated as follows:

$$g_i = \begin{cases} (\frac{TG}{D*C}) * \frac{\alpha_i * h_i}{n_i}, & \text{if } g_i < G_i \\ G_i, & \text{otherwise} \end{cases} \tag{2}$$

We also scale up the number of licenses if the expected loss is low (Line 16 in Algorithm 1).

Algorithm 1 GCL Renewal

```
 1: function RenewLease($L$, $C$, $n$, $h$)
 2:     $TG \leftarrow$ GetTotalGCL($L$)                                ▷Max GCL
 3:     $G_i \leftarrow (\alpha_i * TG/C)$                              ▷Max GCL per node
 4:     $g_i \leftarrow G_i/D$                                          ▷Apply the default Policy
 5:     $g_i \leftarrow g_i * h_i$                                      ▷Crash penalty
 6:     if $h_i > T_h$ then
 7:         $g_i \leftarrow \min(G_i, g_i * (1/n_i))$   ▷Network benefit if the client's health is good.
 8:     end if
 9:     $\beta \leftarrow$ FetchBeta()
10:     if ExpLoss(L) > $\tau$ then
11:         while ExpLoss(L) > $\tau$ do                                ▷Expected loss
12:             $\beta \leftarrow \beta * \frac{(ExpLoss(L)-\tau)}{ExpLoss(L)}$     ▷Scaling down
13:             $g_i \leftarrow \beta * g_i$
14:         end while
15:     else
16:         $\beta \leftarrow \frac{(\tau-ExpLoss(L))}{\tau}$ , $g_i \leftarrow \beta * g_i$    ▷Scaling Up
17:     end if
18:     return $g_i$
19: end function
```

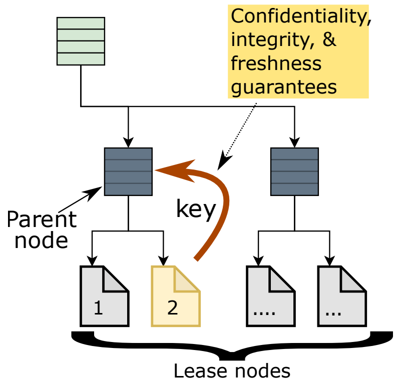


Figure 5: Procedure for committing a lease.

5.4 Issuing a Lease

Prior to issuing a lease, SL-Manager and SL-Local validate each other using the local attestation feature of SGX [37]. Once the validation is completed, SL-Local takes the license from the corresponding SL-Manager and checks the lease corresponding to the license provided. If it has a valid GCL (a positive value), it is decremented, and a token of execution is sent to the SL-Manager, indicating that the execution can proceed. To ensure correctness in the event of multiple requests for the same license arriving at the same time, we use the lock associated with the lease (locked using `sgx_spin_lock()` [72]).

5.5 Committing a Lease

The *lease tree* stays in the trusted region of the memory and hence, cannot be tampered with without detection. However, once the application requesting a particular set of leases quits, there is no need to keep the corresponding leases in the EPC memory since it might affect the performance of other enclaves due to memory pressure. Hence, we introduce a "commit" operation for a lease, after which it can be evicted to the untrusted region.

Algorithm 2 Protect Data

```
1: function PROTECT(Data)
2:     $H' \leftarrow$ HASH($Data$)
3:     $key \leftarrow$ RANDOMKEYGEN()
4:     $C \leftarrow$ AES($Data||H, key$)
5:     return $< C, key >$
6: end function
```

Algorithm 3 Validate

```
1: function VALIDATE(C,key)
2:     $D||H \leftarrow$ DEC($C, key$)
3:     $H' \leftarrow$ HASH($D$)
4:     if $H == H'$ then return $D$
5:     end if
6:     return NULL
7: end function
```

Committing a single lease file, say L, requires first locking the lease. We then *protect* L, which involves calculating a hash (H) of L, generating a random 64-bit key (k) for it, encrypting $L||H$ (lease file concatenated with its hash value) using k, and sending the encrypted payload to the untrusted region (see Algorithm 2). The key (k) is stored in the corresponding entry in its parent node (see Figure 5). Note that the key changes every time the lease is committed and thus replay attacks are not possible.

5.6 SL-Local Exit and Re-Initialization

When SL-Local legally shuts down, SecureLease ensures that its state can be securely restored during its next instantiation. This prevents the costly step of repopulating all the leases. However, we cannot store the lease-tree data in the untrusted region as SGX does not provide any freshness guarantees for this. To ensure freshness, SL-Local first stops servicing all the attestation requests, then commits all the nodes in the tree (except for the root node) to the untrusted region. Finally, while committing the root node, the generated 64-bit random key (key_R) is sent to SL-Remote. This marks the shutdown procedure as complete.

During its next initialization, SL-Local will get key_R as the old-backup key ($\mathcal{OB_K}$) from SL-Remote. It will read the encrypted root node from the untrusted region, decrypt it using $\mathcal{OB_K}$, and validate it within SGX (see Algorithm 3). We then proceed to populate the subsequent levels (with keys stored in their corresponding entry in the parent node).

5.7 Replay Attacks on SL-Local

Now, consider an event where SL-Local crashes due to either a bug or a wilful attack. The key question is that if SL-Local had valid leases with it, but could not complete graceful shutdown, then what should happen to those leases when SL-Local comes back up at a later point in time? In an optimistic approach, SL-Remote can trust that the SL-Local instance went down because of a genuine reason (and not because of an attack) and can restore the lost leases to it. However, this approach is susceptible to a replay attack.

Replay attack: A replay attack on SL-Local can be mounted like this: Assume a lease allows N executions of an application. The attacker starts the application, gets the token to execute, and immediately crashes SL-Local running on her system. Here, the lease update (allowed executions from N to $N - 1$) did not persist in the crash. In the next instantiation of SL-Local, SL-Remote will trust that it crashed due to a genuine reason and restores the lease with the previous limit (N). Hence, the attacker can run the application any number of times by repeating this process.

Hence, SecureLease follows a pessimistic approach, where in the event of an SL-Local crash, all the leases present in SL-Local are deemed to be used. Consequently, we have designed our lease renewal scheme in such a manner that it restricts the total expected loss corresponding to a license (see Algorithm 1).

5.8 Design Benefits

There are many benefits of this design:

(1) The total number of remote attestations required is substantially reduced ($\approx$ 99% as shown in Section 7.4).
(2) SL-Local does the heavy lifting; this significantly reduces the load on the server and allows SL-Remote to scale.
(3) After SL-Local gets a lease from SL-Remote, there is no need for network communication till the issued sub-lease (g_i) is valid.

6 SECURITY ANALYSIS

Here, we analyze the security guarantees of SecureLease.

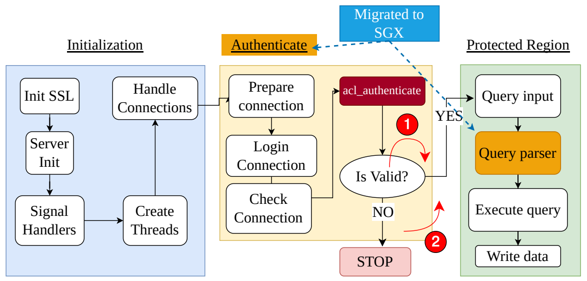


Figure 6: A high-level working of MySQL [15], two CFB attacks on it, and the partitioning done by SecureLease.

6.1 Control Flow Bending (CFB) Attacks

SecureLease migrates certain key functions of an application to SGX to prevent CFB attacks. In order to identify these key functions, SecureLease relies on the developer. CFB attacks can always be mounted on the unsecure portion of the application. It may be possible to bypass a license check for instance and also fix some local state to make the program believe that the license check has passed. This is possible to do because we are assuming a virtual CPU in our threat model. However, in such cases, the attacker will not have access to the key functions executing inside SGX, resulting in an incomplete execution. To explain further, we use a mock-up attack on MySQL and show how SecureLease protects it.

Figure 6 shows a high-level overview of steps for (1) initializing a MySQL server, (2) authenticating a query request (AM), (3) and processing the query. A CFB attack (❶ in Figure 6) attacks the function `acl_authenticate` within the AM and makes it return *success* even when the license is not valid (discussed in Section 2.1.1). To stop the attack, a developer may choose to migrate only the AM to SGX. However, now, a branch can be forced by changing the output of the authentication function that is processed outside the authentication logic (❷ in Figure 6).

To prevent these attacks, SecureLease creates a dependency between the authentication function and its corresponding protected region by migrating the AM and a *key function* to SGX. MySQL has many key functions such as query parsing, query dispatch, and executing the query. SecureLease picks the query parsing logic as its key function. If an attacker bypasses the authentication check by a CFB attack, she will not have access to the query parsing system, rendering the complete application useless. Moving additional key functions (such as the query executor) inside SGX may improve the security slightly but will definitely add to the performance overhead.

6.2 Replay Attacks

SL-Local controls the execution of applications and manages their leases. An attacker cannot tamper with the code of SL-Local since the code is secure and decrypted at runtime within an enclave.

SecureLease guarantees that if an SL-Local instance exits gracefully, its changes will persist across multiple executions. If the attacker attempts to replay an old version of the lease tree (LT') then the validation of the root node at the time of `init()` will fail (Line 4 in Algorithm 3) since the root node of LT' would have been encrypted by another key. Now assume that the root is the same but a node's value is tampered with or replayed. We can detect this because the key to encrypt it lies with its parent, and this key changes every time the data is persisted. Given that the hash is also a part of the bundle, we will be able to easily detect if there has been a tampering or replay attack.

Table 3: System configuration

Hardware Settings		
Core i7-10700 CPU, 2.9 GHz	Disk: 256 GB (SSD)	
CPUs: 1 Socket, 8 Cores, 2 HT		
DRAM: 16 GB	L1: 256 KB, L2: 2 MB, L3: 16 MB	
System Settings		
Linux kernel: 5.9	ASLR: Off	GCC: 9.3.0
DVFS: performance	Transparent Huge Pages: never	
SGX Settings		
PRM: 128 MB	Driver: 2.11	SDK version: 2.13

Table 4: Description of the workloads in SecureLease along with the specific settings used in the paper.

Workloads	Description	Input
BFS [73]	Traverse graphs generated by web crawlers. Use breadth first search.	Nodes: 1 M, Edges: 23 M
B-Tree [27]	Create a B-Tree and perform lookup operations on it.	Elements: 3 M
HashJoin [28]	Probe a hash-table (used to implement equi-join in DBs)	Data Table Size: 1.22 GB
OpenSSL [66]	Encryption-decryption library.	File Size: 151 MB
PageRank [73]	Assign ranks to pages based on popularity (used by search engines).	Nodes: 10 K, Edges: 50 M
Blockchain [64]	A distributed ledger that stores data, the hash of the content, and the previous block's hash in a block.	Chain length: 1000
SVM [46]	Popular ML algorithm (application: text and hypertext categorization)	Data: 4000, Features: 128
MapReduce [54] (FaaS)	Count the occurrences of a word in a set of files	Data: 19 MB, Map:5, Reduce:2
Key-Value [74] (FaaS)	Read and write operations on a key-value store.	70 MB, 500 K elements
JSONParser [55] (FaaS)	Parse JSON strings	Size: 1 KB, Count: 10 K
Mat. Mult. [75] (FaaS)	Perform matrix multiplication	Dimension: 2000 × 2000

7 EVALUATION

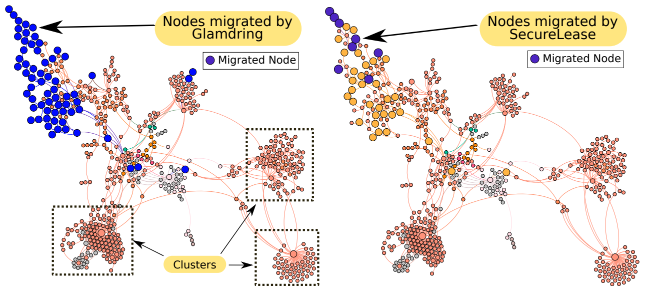
In this section, we report the performance numbers of SecureLease and compare it with F-LaaS [56] and Glamdring [61].

7.1 Experimental Setup

The details of our evaluated system are shown in Table 3. We use the standard RDTSC instruction [25, 69] to measure the latency of ECALLs. Furthermore, we modified the Intel SGX driver code [48] to collect statistics about SGX-related events, such as the total number of page evictions, page allocations, and page load backs. Table 4 shows the workloads used for experiments in the paper. These workloads have been used by the SGX community across different domains such as machine learning [46], cryptography [66], blockchain [64], key value storage [74], and different processing paradigms [28, 31, 54]. We also evaluated SecureLease for 4 different FaaS workloads that make a large number of license check calls (10 K in JSONParser to 500 K in Key-Value in less than a minute). Furthermore, these workloads can scale up to several GBs – suitable

Table 5: Table showing the static and dynamic coverage for Glamdring (Glam.) and SecureLease (SLease), functions migrated by SecureLease, memory-related statistics for Glamdring and SecureLease, and the performance improvement in SecureLease.

Workload	Functions Migrated	Static Coverage		Dynamic Coverage		Memory (EPC Evicts)		Perf. Impr.
		Glam.	SLease(vs Glam.)	Glam.	SLease (vs Glam.)	Glamdring	SLease	
BFS	`update()`	36.2 K	10 K (27.76%)	11.53 B	10.88 B (94.39%)	200 MB (147 K)	4 MB (0)	43.39%
B-Tree	`find(), leaf(), create()`	23.9 K	23.4 K (97.94%)	29.6 B	23.5 B (79.24%)	280 MB (1,430 K)	4 MB (0)	35.99%
HashJoin	`probe()`	22.9 K	10.3 K (45.09%)	33 B	30.2 B (91.39%)	130 MB (7,909 K)	4 MB (0)	84.14%
OpenSSL	`decrypt()`	815.3 K	811.9 K (99.58%)	189.1 B	181 B (95.71%)	310 MB (3,539 K)	4 MB (0)	74.83%
PageRank	`map(), reduce(), set_rank()`	23.3 K	10.5 K (45.28%)	8.9 B	8.82 B (99.09%)	1,360 MB (2,234 K)	4 MB (0)	84.93%
Blockchain	`insert(), hash()`	32.9 K	11.2 K (34.23%)	133.5 B	129.6 B (97.03%)	4 MB (0)	4 MB (0)	3.30%
SVM	`predict()`	12.52 K	11.58 K (92.50%)	295 B	293.5 B (99.35%)	110 MB (50 K)	85 MB (0)	14.11%
MapReduce	`tokenize(), word_count()`	104 K	103 K(98.86%)	14 B	12.9 B(92.53%)	82MB(0)	66MB(0)	35.65%
Key-Value	`set()`	118 K	118 K (99.9%)	13 B	10 B (78.21%)	162 MB (59 K)	4 MB (0)	68.80%
JSONParser	`parse()`	580 K	566 K (97.58%)	12.5 B	12.34 B (98.82%)	34 MB (0)	4 MB (0)	8.88%
Mat. Mult.	`multiply()`	122 K	101 K (82.5%)	192.7 B	192.5 B (99.85%)	320 MB (147.5 K)	81 MB (0)	52.53%
Geo. Mean	-	–	67.80%	–	92.93%	Mean:280,MB (1,410 K)	24 MB (0)	32.62%



(a) Glamdring **(b) SecureLease**

Figure 7: Figure showing the functions migrated by Glamdring and SecureLease for a representative benchmark OpenSSL.

for the new scalable SGX [50]. We use Intel VTune [9] to collect statistics for applications executing within Intel SGX. We report the performance overheads normalized to a vanilla setting (no SGX and no attestations).

7.2 Application Partitioning Performance

Here, we compare the performance of our partitioning scheme (no attestations) with that of Glamdring. We report the key functions migrated by SecureLease, the static and dynamic coverage [79], the memory usage of both Glamdring and SecureLease, and the performance improvement of SecureLease over Glamdring in Table 5.

Compared to Glamdring [61], SecureLease reduces the total size of the code that is to be executed in SGX by an average of 67.80% while maintaining a dynamic coverage of 92.93%. SecureLease incurs an average slowdown down of 41.82% over the vanilla mode (no SGX) but outperforms Glamdring by 32.62% because of reduced operations within SGX and efficient memory utilization that leads to no EPC faults in SecureLease. For example, in OpenSSL, SecureLease reduces the total memory stall cycles by 65.85%, and the total number of dTLB-misses comes down by $\approx$ 98% (not shown) when compared with Glamdring [61]. Figure 7 shows the functions migrated by Glamdring and SecureLease for the workload OpenSSL.

Table 6: Memory usage of SecureLease with and without eviction.

# Total leases	1K	5K	10 K	50 K
No-Evict	332 KB	1.6 MB	3.2 MB	15.6 MB
SecureLease	332 KB	1.6 MB	**1.6 MB**	**1.6 MB**

Summary: SecureLease provides similar levels of dynamic coverage but migrates far fewer functions, and since the functions are better chosen, the performance numbers are better.

7.3 SL-Local Performance

The design of SecureLease allows SL-Local to service multiple applications on the same machine. Here, we measure the performance of SL-Local in the presence of concurrent lease allocation requests using a micro-benchmark. Each concurrent instance of the micro-benchmark executes for 10 seconds and records the total number of successful lease allocation requests. Figure 8 shows the performance of SL-Local in two modes: when concurrent applications are trying to access the same lease and when they are accessing different leases. As already mentioned, we perform a local attestation before servicing the lease allocation request. However, a local attestation is a costly operation compared to updating the lease value. Hence, the cost of doing a local attestation dominates the total procedure (98%). To optimize this, an application can choose to obtain multiple tokens of executions with a single local attestation. Note that this is application-dependent. We configure our micro-benchmark and SL-Local such that 10 tokens are granted with a single local attestation. This leads to an average performance improvement of 10× (see Figure 8).

SecureLease memory footprint: As shown in Table 6, SecureLease is capable of servicing a large number of leases while maintaining approximately the same memory footprint of 1.6MB by evicting unused or cold leases (see Section 5.6).

7.4 Complete Performance Evaluation

Here, we measure the total performance overhead of the workloads while executing with SecureLease (partitioning and lease allocation)

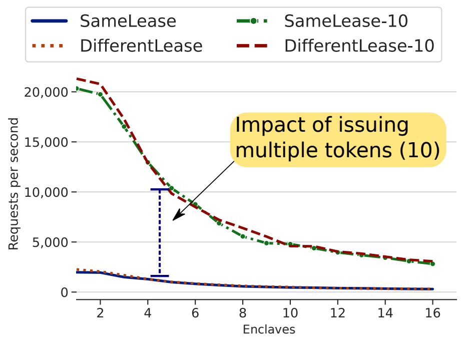


Figure 8: Attestation performance for single and concurrent requests.

and compare it with F-LaaS [56] and Glamdring [61]. We enable lease allocations in Glamdring using the same lease-based method as in SecureLease. To ensure a fair comparison with F-LaaS, we use the same set of migrated functions for both F-LaaS and SecureLease; only the lease allocation logic is different. This is done this way because the partitioning scheme in F-LaaS results in a very high-performance overhead (up to 2000×). Based on our evaluation setup, we use a value of 25% for the parameter D (g_i = 25% of G_i) as it offers a balance between the performance of an application and crash-based attacks on SL-Local (as seen in our experiments). As ours is a local setup, we use a high value for T_H (0.9). We assign a default value of 0.01 for β which is shown to be a good estimate in similar scenarios [70]. For τ, we set its value as 10% of the total GCL, a lower value results in frequent remote attestations as it will severely restrict the number of leases (g_i) allocated to SL-Local.

Figure 9 shows the performance overhead of F-LaaS, Glamdring, and SecureLease w.r.t. the vanilla setting. SecureLease outperforms F-LaaS by 66.34% due to fewer remote attestation calls (by ≈ 99%), and Glamdring by 19.55%. Adding attestation to SecureLease reduces its benefit over Glamdring (32.62% ⟶ 19.55%) due to fewer ECALLs made by Glamdring (by 8%) to get leases. This is because Glamdring migrates almost the complete application to SGX, whereas SecureLease breaks the protected region and requires more ECALLs and leases to execute the logic. However, in this case also SecureLease outperforms Glamdring mainly due to far fewer EPC faults.

7.5 Impact of Scalable SGX

Intel recently announced a scalable version of SGX which supports an EPC size of up to 512 GB [50]. However, SecureLease is agnostic to this. Let us elaborate on the relevance of SecureLease.

The new version of SGX does not provide important security guarantees such as protection against integrity violations and replay attacks (Table 3 in [50]). The onus of providing these security guarantees is transferred to a trusted firmware, which will have access to the secure memory as per Intel's latest documentation [50]. The firmware developer can choose the features that need to be implemented. However, in order to ensure complete protection, the firmware must provide integrity and freshness guarantees (much like the current version of SGX). This will restrict the secure memory footprint of an application due to performance constraints.

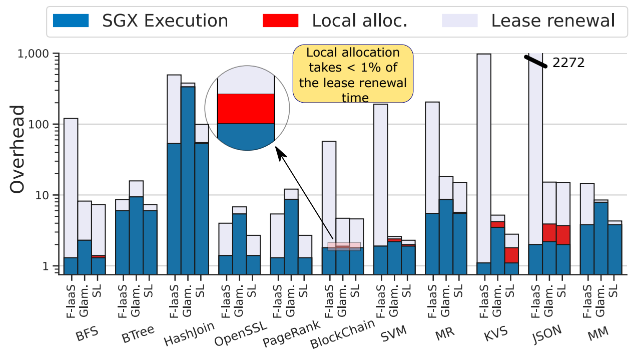


Figure 9: Performance overhead comparison for F-LaaS, Glamdring (Glam.), and SecureLease (SL) due to SGX, allocation requests with SL-Local (Local alloc.), and lease renewal.

Hence, a partitioned binary that exclusively tries to reduce the secure memory footprint will continue to be extremely relevant.

Even when we consider a scenario where an application is completely utilizing the EPC of size 512 GB, it still cannot make system calls and invoke privileged instructions within SGX. Within SGX, we also cannot isolate add-ons from each other as they share their address space. This allows a malicious add-on to steal data from the application or other add-ons [2–4, 6, 23, 24] (CVE-2018-[20031, 20032, 20033, 20034]). Hence, there is a need to protect an application from itself or one plug-in from another [26]. Our partitioning algorithm isolates the add-ons, and also seriously handicaps the attacker even if she manages to mount a few successful CFB attacks.

8 CONCLUSION

In this paper, we propose a hardware-based, secure, efficient, and scalable method to securely authorize the execution of an application on a remote, untrusted server. We leverage Intel SGX to ensure a controlled, secure, and tamper-free environment on a remote server for the license managers and large parts of the execution (without which the program cannot successfully complete). This severely limits the utility that an attacker can derive by mounting CFB attacks on the application. Moreover, our design mitigates several costly Intel SGX operations, such as the costly remote attestation step. Finally, we propose an effective distributed lease management algorithm. SecureLease incurs a performance overhead of 41.82% over the vanilla setting as compared to 72.08% for the nearest competing solution [61]. Our proposed partitioning algorithm and lease management mechanisms have a generic scope and can be repurposed for other application areas as well.

ACKNOWLEDGMENTS

We thank our anonymous reviewers for their insightful comments. This work has been supported in part by the Semiconductor Research Corporation (SRC).

REFERENCES

[1] [n.d.]. Amazon Cognito - Simple and Secure User Sign Up & Sign In | Amazon Web Services (AWS). https://aws.amazon.com/cognito/.. (Accessed on 02/28/2022).
[2] [n.d.]. Bug Reports. https://in.mathworks.com/support/bugreports/details/2285242. (Accessed on 02/19/2022).
[3] [n.d.]. Bug Reports. https://in.mathworks.com/support/bugreports/details/2566624. (Accessed on 02/19/2022).
[4] [n.d.]. Bug Reports. https://in.mathworks.com/support/bugreports/details/1902249. (CVE-2018-20033, Accessed on 02/19/2022).
[5] [n.d.]. Creating Secure, Effective And Hassle-Free Trial License Strategies For Your Software. https://www.ssware.com/articles/creating-secure-effective-and-hassle-free-trial-license-strategies-for-your-software.htm. (Accessed on 11/03/2021).
[6] [n.d.]. Deep dive into Visual Studio Code extension security vulnerabilities | Snyk. https://snyk.io/blog/visual-studio-code-extension-security-vulnerabilities-deep-dive/. (Accessed on 02/19/2022).
[7] [n.d.]. enclave-measurement-tool-intel-sgx-737361.pdf. https://www.intel.com/content/dam/develop/external/us/en/documents/enclave-measurement-tool-intel-sgx-737361.pdf. (Accessed on 02/23/2022).
[8] [n.d.]. Firebase Pricing. https://firebase.google.com/pricing. (Accessed on 02/28/2022).
[9] [n.d.]. Fix Performance Bottlenecks with Intel® VTune™ Profiler. https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html. (Accessed on 11/19/2021).
[10] [n.d.]. LICENSE4J License Manager User Guide. https://www.license4j.com/documents/LICENSE4J-License-Manager-User-Guide.pdf. (Accessed on 11/03/2021).
[11] [n.d.]. Linux kernel coding style — The Linux Kernel documentation. https://www.kernel.org/doc/html/v4.10/process/coding-style.html.
[12] [n.d.]. Modularity (networks) - Wikipedia. https://en.wikipedia.org/wiki/Modularity_(networks). (Accessed on 11/17/2021).
[13] [n.d.]. MurmurHash - Wikipedia. https://en.wikipedia.org/wiki/MurmurHash. (Accessed on 11/18/2021).
[14] [n.d.]. MySQL :: MySQL 5.6 Reference Manual :: 5.5 MySQL Server Plugins. https://dev.mysql.com/doc/refman/5.6/en/server-plugins.html. (Accessed on 02/28/2022).
[15] [n.d.]. MySQL :: MySQL Internals Manual :: 1 A Guided Tour Of The MySQL Source Code. https://dev.mysql.com/doc/internals/en/guided-tour.html. (Accessed on 02/25/2022).
[16] [n.d.]. Netflix AWS: The Netflix Serverless Case Study | Dashbird. https://dashbird.io/blog/serverless-case-study-netflix/. (Accessed on 03/08/2022).
[17] [n.d.]. /proc/*/status Vm… fields. https://ewx.livejournal.com/579283.html. (Accessed on 02/19/2022).
[18] [n.d.]. Products and Services - MATLAB & Simulink. https://in.mathworks.com/products/index.html?s_tid=hp_fp_viewall. (Accessed on 02/19/2022).
[19] [n.d.]. SDK | Intel® Software Guard Extensions | Intel® Software. https://software.intel.com/en-us/sgx/sdk.
[20] [n.d.]. Serverless Framework: The Coca-Cola Case Study | Dashbird. https://dashbird.io/blog/serverless-case-study-coca-cola/. (Accessed on 03/08/2022).
[21] [n.d.]. Software Protection, Software Licensing, Software Virtualization. https://enigmaprotector.com/en/help/manual/405997dd4213981eab804ab38693ffc8. (Accessed on 11/03/2021).
[22] [n.d.]. Top 40+ VSCode Extensions for Developers in 2022 - Tabnine Blog. https://www.tabnine.com/blog/top-vscode-extensions/. (Accessed on 02/19/2022).
[23] [n.d.]. Vulnerabilities in Visual Studio Code Extensions Expose Developers to Attacks. https://www.securityweek.com/vulnerabilities-visual-studio-code-extensions-expose-developers-attack. (Accessed on 02/19/2022).
[24] [n.d.]. Zoom : Security vulnerabilities. https://www.cvedetails.com/vulnerability-list/vendor_id-2159/Zoom.html. (Accessed on 03/03/2022).
[25] 2020. RDTSC — Read Time-Stamp Counter. https://www.felixcloutier.com/x86/rdtsc.
[26] 2022. Mistrust Plugins You Must: A Large-Scale Study Of Malicious Plugins In WordPress Marketplaces. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA. https://www.usenix.org/conference/usenixsecurity22/presentation/kasturi
[27] Reto Achermann. 2020. mitosis-project/mitosis-workload-btree: The BTree workload used for evaluation. https://github.com/mitosis-project/mitosis-workload-btree.
[28] Reto Achermann. 2020. mitosis-project/mitosis-workload-hashjoin: The HashJoin workload used for evaluation. https://github.com/mitosis-project/mitosis-workload-hashjoin.
[29] Robin Ankele and Andrew C. Simpson. 2017. On the Performance of a Trustworthy Remote Entity in Comparison to Secure Multi-party Computation. *2017 IEEE Trustcom/BigDataSE/ICESS* (2017), 1115–1122.
[30] Raad Bahmani, Manuel Barbosa, Ferdinand Brasser, Bernardo Portela, Ahmad-Reza Sadeghi, Guillaume Scerri, and Bogdan Warinschi. 2016. Secure Multiparty Computation from SGX. *IACR Cryptol. ePrint Arch.* 2016 (2016), 1057.
[31] S. Beamer, K. Asanovic, and D. Patterson. 2015. The GAP Benchmark Suite. *ArXiv* abs/1508.03619 (2015).
[32] Nicolas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R. Gross. [n.d.]. Control-Flow Bending: On the Effectiveness of Control-Flow Integrity *(SEC'15)*.
[33] Nikolaos Chalkiadakis, Dimitris Deyannis, Dimitris Karnikis, Giorgos Vasiliadis, and Sotiris Ioannidis. 2020. The Million Dollar Handshake: Secure and Attested Communications in the Cloud. *2020 IEEE 13th International Conference on Cloud Computing (CLOUD)* (2020), 63–70.
[34] Chia che Tsai, Jeongseok Son, Bhushan Jain, John McAvey, Raluca A. Popa, and Donald E. Porter. 2020. Civet: An Efficient Java Partitioning Framework for Hardware Enclaves. In *USENIX Security Symposium*.
[35] Huili Chen, Bita Darvish Rouhani, Cheng Fu, Jishen Zhao, and Farinaz Koushanfar. 2019. DeepMarks: A Secure Fingerprinting Framework for Digital Rights Management of Deep Learning Models. *Proceedings of the 2019 on International Conference on Multimedia Retrieval* (2019).
[36] Gabriele Contini. [n.d.]. open-license-manager/open-license-manager: Software licensing, copy protection SDK in C++. https://github.com/open-license-manager/open-license-manager.
[37] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. *IACR Cryptol. ePrint Arch.* 2016 (2016), 86.
[38] Lorenzo De Lauretis. 2019. From Monolithic Architecture to Microservices Architecture. In *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. 93–96. https://doi.org/10.1109/ISSREW.2019.00050
[39] Ivan De Oliveira Nunes, Xuhua Ding, and Gene Tsudik. 2021. On the Root of Trust Identification Problem. In *Proceedings of the 20th International Conference on Information Processing in Sensor Networks (Co-Located with CPS-IoT Week 2021)* (Nashville, TN, USA) *(IPSN '21)*. Association for Computing Machinery, New York, NY, USA, 315–327. https://doi.org/10.1145/3412382.3458274
[40] Ashutosh Dhar Dwivedi. 2019. A Scalable Blockchain Based Digital Rights Management System. *IACR Cryptol. ePrint Arch.* 2019 (2019), 1217.
[41] Dimitris Geneiatakis, Georgios Portokalidis, Vasileios P. Kemerlis, and Angelos D. Keromytis. [n.d.]. Adaptive Defenses for Commodity Software through Virtual Application Partitioning *(CCS '12)*.
[42] Adrien Ghosn, James R. Larus, and Edouard Bugnion. 2019. Secured Routines: Language-based Construction of Trusted Execution Environments. In *USENIX Annual Technical Conference*.
[43] Dong Guo, Wei Wang, Guosun Zeng, and Zerong Wei. 2016. Microservices Architecture Based Cloudware Deployment Platform for Service Computing. In *2016 IEEE Symposium on Service-Oriented System Engineering (SOSE)*. 358–363. https://doi.org/10.1109/SOSE.2016.22
[44] Junqi Guo, Chuyan Li, Guangzhi Zhang, Yunchuan Sun, and Rongfang Bie. 2019. Blockchain-enabled digital rights management for multimedia resources of online education. *Multimedia Tools and Applications* 79 (2019), 9735–9755.
[45] Aisha Hasan, Ryan Riley, and Dmitry Ponomarev. 2020. Port or Shim? Stress Testing Application Performance on Intel SGX. In *2020 IEEE International Symposium on Workload Characterization (IISWC)*. 123–133. https://doi.org/10.1109/IISWC50251.2020.00021
[46] Marti A. Hearst. 1998. Support Vector Machines. *IEEE Intelligent Systems* 13, 4 (July 1998), 18–28. https://doi.org/10.1109/5254.708428
[47] IBM. 2021. POANK8YE. https://www.ibm.com/downloads/cas/POANK8YE. (Accessed on 11/09/2021).
[48] Intel. 2019. Intel SGX Linux* Driver. https://github.com/intel/linux-sgx-driver.
[49] Intel. 2019. Intel(R) Software Guard Extensions Protected Code Loader for Linux* OS. https://github.com/intel/linux-sgx-pcl.
[50] Simon Johnson, Raghunandan Makaram, Amy Santoni, and Vinnie Scarlata. 2021. Supporting Intel SGX on Multi-Socket Platforms. (2021). https://www.intel.com/content/www/us/en/architecture-and-technology/software-guard-extensions/supporting-sgx-on-multi-socket-platforms.html
[51] T. Kanungo, D.M. Mount, N.S. Netanyahu, C.D. Piatko, R. Silverman, and A.Y. Wu. 2002. An efficient k-means clustering algorithm: analysis and implementation. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 24, 7 (2002), 881–892. https://doi.org/10.1109/TPAMI.2002.1017616
[52] Justas Kazanavičius and Dalius Mažeika. 2019. Migrating Legacy Software to Microservices Architecture. In *2019 Open Conference of Electrical, Electronic and Information Sciences (eStream)*. 1–5. https://doi.org/10.1109/eStream.2019.8732170
[53] Brian W. Kernighan and Rob Pike. 1999. *The Practice of Programming*. Addison-Wesley Longman Publishing Co., Inc., USA.
[54] Jeongchul Kim and Kyungyong Lee. 2019. Practical Cloud Workloads for Serverless FaaS. In *Proceedings of the ACM Symposium on Cloud Computing* (Santa Cruz, CA, USA) *(SoCC '19)*. Association for Computing Machinery, New York, NY, USA, 477. https://doi.org/10.1145/3357223.3365439
[55] Jeongchul Kim and Kyungyong Lee. 2019. Practical cloud workloads for serverless faas. In *Proceedings of the ACM Symposium on Cloud Computing*. 477–477.
[56] Sandeep Kumar, Diksha Moolchandani, Takatsugu Ono, and Smruti R. Sarangi. 2019. F-LaaS: A Control-Flow-Attack Immune License-as-a-Service Model. In *2019 IEEE International Conference on Services Computing (SCC)*. 80–89. https://doi.org/10.1109/SCC.2019.00025

[57] Sandeep Kumar, Aravinda Prasad, Smruti R. Sarangi, and Sreenivas Subramoney. 2021. Radiant: Efficient Page Table Management for Tiered Memory Systems. In *Proceedings of the 2021 ACM SIGPLAN International Symposium on Memory Management* (Virtual, Canada) *(ISMM 2021)*. Association for Computing Machinery, New York, NY, USA, 66–79. https://doi.org/10.1145/3459898.3463907

[58] Sandeep Kumar and Smruti R. Sarangi. 2021. SecureFS: A Secure File System for Intel SGX *(RAID '21)*. Association for Computing Machinery, New York, NY, USA, 91–102. https://doi.org/10.1145/3471621.3471840

[59] Kubilay Ahmet Küçük, Andrew J. Paverd, Andrew C. Martin, N. Asokan, Andrew C. Simpson, and Robin Ankele. 2016. Exploring the use of Intel SGX for Secure Many-Party Applications. *Proceedings of the 1st Workshop on System Software for Trusted Execution* (2016).

[60] Titouan Lazard, Johannes Götzfried, Tilo Müller, Gianni Santinelli, and Vincent Lefebvre. 2018. TEEshift: Protecting Code Confidentiality by Selectively Shifting Functions into TEEs. In *Proceedings of the 3rd Workshop on System Software for Trusted Execution*. Association for Computing Machinery. https://doi.org/10.1145/3268935.3268938

[61] Joshua Lind, Christian Priebe, Divya Muthukumaran, et al. [n.d.]. Glamdring: Automatic Application Partitioning for Intel SGX. In *ATC'17*.

[62] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Chicago, IL, USA) *(PLDI '05)*. ACM, New York, NY, USA, 190–200.

[63] M. Melara, M. Freedman, and M. Bowman. 2019. EnclaveDom: Privilege Separation for Large-TCB Applications in Trusted Execution Environments. *ArXiv* abs/1907.13245 (2019).

[64] Saurav Mohapatra. 2019. mohaps/libcatena: a simple toy blockchain written in C++ for learning purposes. https://github.com/mohaps/libcatena.

[65] U.S. Department of Commerce, National Institute of Standards, and Technology. 2012. *Secure Hash Standard - SHS: Federal Information Processing Standards Publication 180-4*. CreateSpace Independent Publishing Platform, USA.

[66] OpenSSL. 2019. OpenSSL. https://www.openssl.org/.

[67] Tapti Palit, Fabian Monrose, and Michalis Polychronakis. 2019. Mitigating data leakage by protecting memory-resident sensitive data. *Proceedings of the 35th Annual Computer Security Applications Conference* (2019).

[68] Christian Priebe, Kapil Vaswani, and Manuel Costa. [n.d.]. EnclaveDB: A Secure Database Using SGX. *Proceedings - IEEE Symposium on Security and Privacy* 2018-May ([n. d.]), 264–278.

[69] Colin Robertson. [n.d.]. __rdtsc | Microsoft Docs. https://docs.microsoft.com/en-us/cpp/intrinsics/rdtsc?view=msvc-160.

[70] Addison Sears-Collins. [n.d.]. How to Choose an Optimal Learning Rate for Gradient Descent – Automatic Addison. https://automaticaddison.com/how-to-choose-an-optimal-learning-rate-for-gradient-descent. (Accessed on 11/20/2021).

[71] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 205–218. https://www.usenix.org/conference/atc20/presentation/shahrad

[72] Shweta Shinde, Jinhua Cui, Satyaki Sen, Pinghai Yuan, and Prateek Saxena. 2020. Binary Compatibility For SGX Enclaves. *ArXiv* abs/2009.01144 (2020).

[73] Julian Shun and Guy E. Blelloch. 2013. Ligra: A Lightweight Graph Processing Framework for Shared Memory. *SIGPLAN Not.* 48, 8 (Feb. 2013), 135–146. https://doi.org/10.1145/2517327.2442530

[74] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Joseph E. Gonzalez, Joseph M. Hellerstein, and Alexey Tumanov. 2020. Cloudburst: Stateful Functions-as-a-Service. *Proc. VLDB Endow.* 13, 12 (jul 2020), 2438–2452. https://doi.org/10.14778/3407790.3407836

[75] Bohdan Trach, Oleksii Oleksenko, Franz Gregor, Pramod Bhatotia, and Christof Fetzer. 2019. Clemmys: Towards secure remote execution in FaaS. In *Proceedings of the 12th ACM International Conference on Systems and Storage*. 44–54.

[76] Peter Verhas. 2019. License3j: Free Licence Management Library. https://github.com/verhas/License3j.

[77] Ofir Weisse, Valeria Bertacco, and Todd Austin. 2017. Regaining Lost Cycles with HotCalls: A Fast Interface for SGX Secure Enclaves. *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, 81–93.

[78] Yun Zhang, Zhi Tang, Jinke Huang, Yue Ding, Hao He, Xiaosheng Xia, and Chunhua North Haven Li. 2020. A Decentralized Model for Spatial Data Digital Rights Management. *ISPRS Int. J. Geo Inf.* 9 (2020), 84.

[79] Jianyi Zhou and Dan Hao. 2017. Impact of Static and Dynamic Coverage on Test-Case Prioritization: An Empirical Study. In *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. 392–394. https://doi.org/10.1109/ICSTW.2017.74