F-LaaS: A Control-Flow-Attack Immune *License-as-a-Service* Model

Sandeep Kumar*, Diksha Moolchandani*, Takatsugu Ono§ and Smruti R. Sarangi*

*Department of Computer Science and Engineering
Indian Institute of Technology Delhi
New Delhi, India

§Department of Advanced Information Technology
Kyushu University
Fukuoka, Japan

*{sandeep.kumar, diksha.moolchandani, srsarangi}@cse.iitd.ac.in, §takatsugu.ono@cpc.ait.kyushu-u.ac.jp

***Abstract*—We use license servers to verify users' credentials and to restrict access to proprietary software. Due to logistical reasons, it is often economical to use third-party servers to manage licenses. Sadly, users on client machines can mount sophisticated attacks on the executables and try to circumvent the license check. This can be used to crack the software, and thus it is necessary for software writers to prevent such attacks, which include the use of additional code to check the integrity of the binary and the control flow. In spite of such techniques, modern *control flow bending*(CFB) techniques that rely on running instrumented binaries on virtual machines can circumvent such checks and change the behavior of branches and jumps at runtime. They are however extremely computationally inefficient. We propose an AI-based technique that is an order of magnitude faster than the state-of-the-art and show its efficacy by breaking three widely used license managers, and five popularly used software. Finally, we propose a new license management service, *F-LaaS*, which hides *key functions* in the binary. These functions are downloaded at runtime upon the successful verification of the license. We show that the mean performance overhead of *F-LaaS* is negligible: 0.26%.**

***Keywords*-License as a service, Control flow bending, Control flow attacks, Control Flow Graph, Control Flow Integrity**

I. Introduction

Licensing-as-a-service (LaaS) [1, 2] is the preferred model for distributing software via the internet today. Independent software vendors (ISVs) often use the services of third-party license servers that help them flexibly manage their licenses. Each client application contacts the license server (LS) at the time of starting an application, and obtains a license by supplying the proper credentials. It is the job of the LS to verify the credentials and enforce the lease (duration of validity of the credentials). Allowing third party license managers such as the Amazon AWS License Manager allows ISVs to exclusively focus on designing their application, and furthermore a few vendors can specialize in creating extremely secure license servers using the latest cryptographic technologies.

Even though this method is the default as of today, there are some major security loop holes. Unlike a web-based mechanism such as Ajax [3] where the code is downloaded at run time with the proviso that the correct credentials are presented to the server first, in a LaaS based model the full binary resides on the client's machine. Its security can be circumvented, and the license check can be bypassed. Even with the best cryptographic technologies, the weakest link in the chain is the fact that the binary on the client machine can be modified, or its execution can be altered.

In the last 15 years, there has been a lot of research in this area. Both attack as well as defense mechanisms have become increasingly sophisticated [4]. Starting from simple buffer overflow attacks, researchers have designed increasingly complex attacks such as code injection, and code reuse attacks [5]. In code injection attacks, we try to make the application run an alien code and thus cede control to an attacker, while in code reuse attacks we try to modify the control flow of an application such that the program counter jumps to another location in the binary. This new location contains a piece of code (known as a *gadget*) that is beneficial to the attacker. By executing a series of gadgets, we can implement any logic (shown to be Turing complete); this includes bypassing the license manager.

In response, we have a plethora of techniques to check the integrity of a binary by using methods to prevent unnatural jumps in a program by keeping a shadow return address stack [4], and methods to verify the correctness and integrity of call chains [6]. Sadly, all of these techniques rely on the fact that the processor itself is secure. This assumption fails to hold when we *run the program on a virtual processor* such as the Qemu virtual machine [7], or use binary instrumenters such as Intel PIN [8]. In these cases we have instruction level visibility of the program, and we can modify the execution of individual instructions notably *if* statements and function returns.

This is a very powerful technique. By just flipping the direction of a branch at run time, unbeknownst to the program, we can change its behavior. For example, we can just flip the direction of the *if* statement that checks for a valid license. This is a very powerful mechanism and is immune to all known techniques that check control flow integrity. Moreover, we can disable the license check, and all other methods that check the integrity of the control flow itself. These attacks are known as *control flow bending* (CFB) attacks [9].

Even though CFB attacks look ominous, they are however difficult to mount. This is because out of millions of instructions in a program, we need to find exactly that branch which checks if a license is valid or not. In modern obfuscated and stripped binaries, this process is computationally very inefficient, and is thus considered very time taking and seldom practical [10].

We make several important contributions in this paper.

1) We characterize the structure of programs that use license managers, and show that most programs that use license managers have some common features. These can be learnt and exploited to mount an effective CFB attack.
2) We design an AI-based algorithm to prune the search space of functions in a function call graph. Using our three pruning heuristics, we demonstrate an average speed-up of 20X over the nearest competing algorithm [9] for identifying the function that contains the instructions to invoke the license check routine.
3) We propose a new license management service that can be used to stop such CFB attacks. The key idea is that the client possesses a reduced version of the binary where these key functions are replaced with ineffectual and erroneous instructions. This prevents the attacker from getting any benefits even if she breaks the program. At runtime the code for these functions is supplied by the license management server via an encrypted channel if the user possesses valid credentials. The security checking code then patches the binary with these small code snippets.

The organization of this paper is as follows. In Section II, we discuss the relevant background followed by a formal definition of the threat and security model in Section III. In Section IV we provide several key insights into the execution of a binary. This forms the basis of the CFB algorithm to detect the key functions (contain code to invoke license checking routine) in a binary, which is described in Section V. Subsequently, we propose a method to defend against such attacks in Section VI, and discuss our experimental setup and results in Section VII. Finally, we conclude in Section VIII.

II. Background: License Managers and Attacks

In this section, we shall discuss the design and working of a license manager, and some of the popular attacks on license managers. Subsequently, we shall discuss control flow graphs (CFGs), and show their relevance with regards to this area of research.

A. Design of a License Manager

A *license manager (LM)* is a dedicated module that runs locally along with the proprietary binary. Its job is to communicate the credentials to the license server (LS), and restrict the execution of the binary if required. As shown in Figure 1, a license file (alternatively called the password or *key*) is passed to the LM, which is responsible for validating the key in consultation with the LS running on a remote machine. However, the final decision to grant access to the secure region is still taken by the LM based on the LS's response. *This fact is exploited by the attackers* to gain access to the secure regions of the binary.

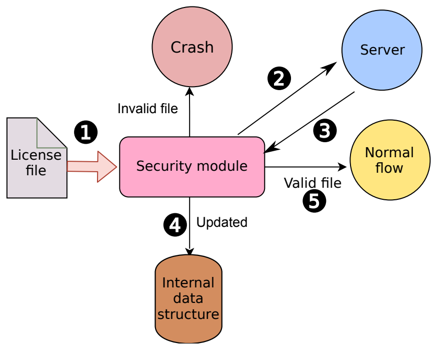

Figure 1: Control flow of a license manager.

B. Control Flow Bending Attacks

Control flow bending (CFB) [9] attacks aim to modify the control flow of a binary with a malicious intent. They hijack the control flow of the binary, which is then used to reveal the secrets of the binary or execute an unintended piece of code: a piece of code that should not be executed in the absence of a valid key. Based on prior analyses, the attacker figures out the execution paths that are beneficial to her, and *forces* the binary to follow those execution paths at runtime. Note that there is neither any new code being inserted nor any new path (in the control flow path) being carved out. Such CFB attacks are typically mounted with the help of either a runtime binary instrumentation engine [8] or a virtual machine that simulates a virtual processor such as Qemu [7]. Let us next describe two state of the art CFB attacks: CFDA and CGA.

C. CFDA and CGA

1) CFDA: Agarwal et al. [10] propose two novel techniques to break the security of the license check namely *Control flow deviation analysis* (CFDA) and *Call graph alteration* (CGA). Both the techniques are based on the CFB paradigm. They attack the license manager by identifying and locating those instructions in the binary that are responsible for enforcing the license checks.

Specifically, CFDA executes a binary twice: first with a valid key and then with an invalid key. The instruction traces (sequence of instructions executed by the binary) are collected for both the executions separately and are compared to find the point of *deviation*. Recording only the deviating branch instructions prunes the search space significantly. Note that there is always some deviation across runs of the same program because of non-determinism caused in libraries and memory allocation routines; hence, the algorithm identifies some additional instructions as well. Additionally, pruning the instructions corresponding to the dynamically linked libraries reduces the search space further by 100X. Subsequently, to break the license check the authors sequentially alter the behavior of the instructions found by their deviation analysis technique.

CFDA uses the longest common subsequence(LCS) algorithm to find the deviating branches. This has a runtime complexity equal to $\mathcal{O}(n^2)$, where n is the number of branch instructions left after pruning. This is prohibitive for large binaries. Additionally, this technique requires access to a valid license file, which might not be available all the time.

2) CGA: In contrast, CGA forces the binary to traverse all the paths that were skipped when it was presented with an invalid license key. This is done as follows. It first creates a mapping between the assembly code of a binary and an invalid execution trace. Using this mapping it creates a list of the paths that were *not* traversed by the binary, and then forces the binary to traverse them by changing the direction of *if* statements or by skipping functions. Eventually, it figures out the license checking function and skips the code that decides to exit the program if the license check fails. This is a very time intensive process.

The runtime of both the algorithms is very large and thus these techniques are very slow in practice. Considering the fact that modern binaries have millions of instructions [11] out of which $20 - 25\%$ are control flow instructions (call, jump and return instructions) [11], such methods are computationally infeasible.

D. Control Flow Graphs (CFGs)

A control flow graph or CFG of a binary is a graphical representation of a program's execution using a directed graph as shown in Figure 2. The nodes in the graph represent functions and a directed edge between any two nodes represents a function call from the source to the destination node. The function call patterns in an execution reveal several characteristics of the binary and are used by several defense methods to validate an execution [12].

III. Threat and Security Model

In this section we define the scope of the threat and the security model. We are focusing on *commercial-off-the-shelf* or COTS binaries whose life-cycle is as follows: A developer writes the application and ships it to the users in a single package. To ensure that only valid users are allowed to run the binary, the developer puts a license manager in the package. The license manager contacts the license server to validate the key at runtime.

A. Threat Model

We assume that the attacker has the complete package with her, which is typically the case for COTS binaries. The attacker, however, does not have access to the source code and relies only on the analysis of the binary and the license manager to get access to the protected parts. Furthermore, the attacker is free to execute the binary in any environment she desires, where the explicit aim is to bypass the security provided by the license manager. If we are running the binary on a virtual machine like Qemu, then we are assuming that the behavior of any subset of instructions can be altered.

As shown in related work [10, 9], the license manager check eventually boils down to a single conditional branch statement, or a single function call. If we can flip the branch statement or skip the function call, then we can effectively break the license. CFB attacks are ideal for such cases. We present an efficient version of CFB called *SmartCFB*, which can prune the search space using heuristics obtained from the control flow graph of the binary's execution.

B. Security Model

We aim to create an efficient license manager, *F-LaaS*, for COTS binaries that is immune to a binary analysis attack, such as CFB. We design the license manager as a service, which is deployed on a server with the following goals:

1) Low network bandwidth: The amount of data communicated between the license server (LS) and the application should be low.
2) Low storage: The license server should not be burdened with storing a large amount of data.
3) No access to the source code: This allows a LaaS provider to provide secure licensing services to multiple clients and vendors.

IV. Characteristics of a License Manager

In this section, we look at the characteristics of a license manager, and propose methods to efficiently identify a function that contains the license check functionality in a CFG out of the thousands of functions. We propose an approach based on clustering, and graph analysis to reduce the search space significantly. Let us first discuss the process of creating the CFG, and deriving insights from the generated clusters in this section. We will discuss the graph analysis based search space reduction technique in Section V.

A. Creation of the CFG

We use the Valgrind [13] tool to create the CFG for an execution with an invalid key. This tool instruments the binary to record when a function is invoked, caller-callee information, and the execution frequencies of functions. Each node in the CFG stores the following information: function name, address, in-degree (number of functions that have called it), and out-degree (number of functions it has called). We add an edge from a caller to a callee function. Each edge is annotated with the following additional information: source node, destination node, number of times the caller invoked the callee.

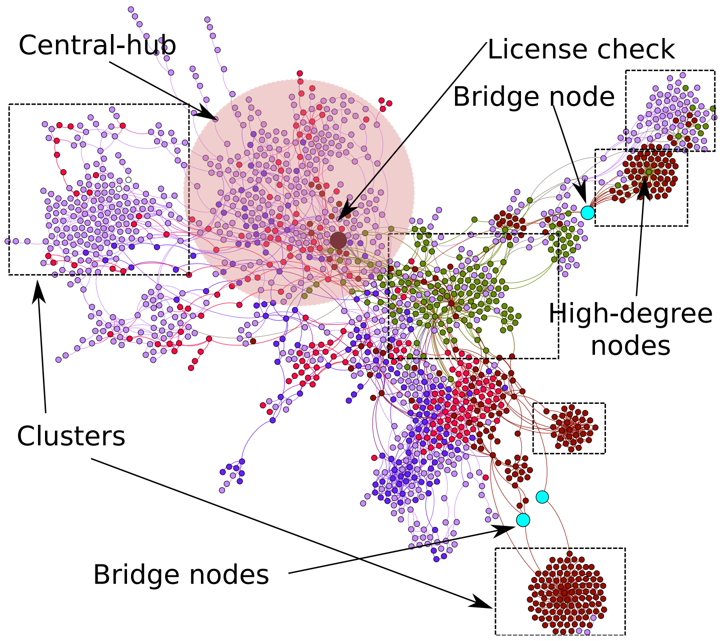

Figure 2: Call graph showing clusters, license check nodes, high-degree nodes and bridge nodes for the *MySQL* benchmark

B. *Clustering of the CFG*

A binary performs multiple tasks such as reading a file from the disk, accessing the network, solving complex mathematical equations, etc. Each of these tasks are extremely well defined tasks, and have a homogeneous character, where a set of functions cooperate to fulfill the task. To identify these cooperating nodes (functions), we need to devise a method to group functions based on the high level task that they are associated with. Our algorithm has two steps: map each node in the CFG ($V \rightarrow$ set of vertices, $E \rightarrow$ set of edges) to a point in a d-dimensional space, and then cluster the nodes in the graph such that nodes in the same cluster are related in terms of functionality.

To map the nodes to a d-dimensional coordinate space, we use the algorithm proposed by Grover and Leskovec [14]. The aim is to ensure that nodes that are close by in the CFG are also in each other's neighborhood in the mapped space. In each iteration of the algorithm, we perform a biased random walk in the CFG. For example, from node u, we traverse l nodes in the CFG with a randomly chosen traversal strategy (S) – the set of nodes are denoted by the set $N_s(u)$. Now, given a mapping strategy $f : V \rightarrow \mathcal{R}^d$, we wish to minimize the error given by:

$$\mathcal{E}(f) = -\sum_{u \in V} log\ [Pr(N_s(u)|f)]$$

$Pr(N_s(u)|f)$ is the probability that all the nodes in $N_s(u)$ are in the neighborhood of u in the d-dimensional space for a given mapping function, f. A simplifying assumption made in [14] is that $Pr(N_s(u)|f) = \prod_{v \in N_s(u)} Pr(v|f)$. Additionally, the authors propose an expression for $Pr(v|f)$ based on existing research on n-grams. This entire process is run iteratively, and in every iteration the authors apply the stochastic gradient descent algorithm to perturb f such that the error is minimized.

Subsequently, we cluster the graph (in the d-dimensional space) using the K-means algorithm. This algorithm minimizes the sum of the squares of distances of each point from its respective cluster center (centroid). For the purpose of illustration, we use the Principal Component Analysis (PCA) [15] method to project the points from a d-dimensional space to a 2-D space.

C. *Classification of Nodes and Clusters*

After clustering, the groups of functions responsible for specific tasks show up as distinct clusters of nodes as shown in Figure 2. The execution begins with the *main* function that is a part of a cluster. All the functions in this cluster are typically invoked by the *main* function and provide some auxiliary functionality. For performing larger tasks we call other functions that initiate these tasks. The nodes associated with these tasks form their own distinct clusters as shown in Figure 2. We refer to such nodes that initiate the execution in a cluster of nodes as *bridge nodes*.

D. *Key Assumptions*

Based on the results of clustering, we present our assumptions regarding the position of the license check function with respect to the *central hub* (cluster that contains the *main function* in the CFG). We shall provide some intuitive reasons here. Subsequently, we shall verify these assumptions (2,3, and 4) in Section VII-D.

1) **Assumption 1**: The license check function or its wrapper is contained in vicinity of the *central hub*

primarily because it needs to be called before we call most of the other functions. This was empirically observed.

2) **Assumption 2**: The Euclidean distance of the nodes from their respective cluster centers closely follows a normal distribution [16]: $\mathcal{N}(\mu, \sigma^2)$, where μ is the mean and σ^2 is the variance of the distribution. The Euclidean distance between the license check node and the center of its cluster is typically more than the standard deviation (σ).
3) **Assumption 3**: The license check node is situated at the periphery of multiple clusters. We define a score function (see Section V-B) that captures the distance of the nodes from multiple clusters.
4) **Assumption 4**: The call to the license check function is typically made from a bridge node. Such bridge nodes have a lower degree (sum of in-degree and out-degree) as compared to other nodes in their respective clusters.

V. Algorithm to Prune the CFG

In this Section, we provide an algorithm to prune the search space such that we can efficiently search for the function that invokes the license manager. The algorithm is based on the assumptions listed in Section IV-D.

Figure 4 shows a flowchart of the entire process. The mapping and clustering phases were explained in Section IV. The pruning phase consists of three sub-phases: *radius-based pruning, score-based pruning,* and *degree-based pruning*. After the three levels of pruning, we apply the *CGA* technique (see Section II) on the pruned graph and try to bypass the license check. If the license check node is found, the algorithm terminates. On the contrary, if the license check node is not found in the pruned graph, the entire pruning algorithm is repeated with a slight perturbation in the hyper-parameters, which are used at each level of the pruning sub-phases. We show in Section VII that the average number of iterations for convergence is approximately 80.

A. Radius-based pruning

Based on Assumption 2 in Section IV-D, we propose a radius-based pruning scheme that takes the centers (μ^j) of the clusters ($\mathcal{C}_j$) as input and removes the nodes within a $k*\sigma$ radius around the cluster center. Here, k is a tunable hyper-parameter depending on the number of unwanted nodes we want to remove. This is mathematically expressed as follows.

$$\mathcal{G}(V,E) \xrightarrow[pruning]{radius-based} \mathcal{G}'(V',E') \tag{1}$$

$$\begin{aligned} V' &= \Big\{ v \in V \mid \big[\Sigma_{l=1}^{d} (v_l - \mu_l^j)^2\big] > (k*\sigma)^2 \Big\} \\ E' &= \big\{ (v_i, v_j) \in E \mid (v_i \in V') \wedge (v_j \in V') \big\} \end{aligned} \tag{2}$$

B. Score-based pruning

After radius based pruning, we prune the search space based on Assumption 3 listed in Section IV-D. In this phase, we assign a score to each of the nodes on the basis of their proximity to each cluster center. We define the score function ($\mathcal{S}(i) : \mathcal{R}^+ \to \mathcal{R}^+$) for node i as:

$$\mathcal{S}(i) = \prod_{j=1}^{num\_clusters} Pr\big(dist(i, \mu_j) \sim \mathcal{N}(0, \sigma_j^2)\big) \tag{3}$$

Here, $Pr\big(dist(i, \mu_j) \sim \mathcal{N}(0, \sigma_j^2)\big)$ is the value of the *pdf* function of the normal distribution with mean 0, and standard deviation, σ_j, at the point $dist(i, \mu_j)$. The $dist(i, \mu_j)$ function yields the Euclidean distance between node i, and the center of cluster j (μ_j). Note that the score will be higher for nodes that are close to all the cluster centers, as compared to nodes that are located at the periphery of the CFG. Moreover, based on our assumption in Section IV-D, this value will be higher for the license check node. Thus, we define a threshold ζ (tunable hyper-parameter) to prune the nodes with a score less than ζ (see Figure 3 for an example). The score-based pruning of the graph is given by:

$$\mathcal{G}'(V',E') \xrightarrow[pruning]{score-based} \mathcal{G}''(V'',E'') \tag{4}$$

$$\begin{aligned} V'' &= \{ v' \in V' \mid \mathcal{S}(v') > \zeta \} \\ E'' &= \big\{ (v_i', v_j') \in E' \mid (v_i' \in V'') \wedge (v_j' \in V'') \big\} \end{aligned} \tag{5}$$

C. Degree-based pruning

The input to this phase of the algorithm is a pruned graph from the earlier phases. Since we emphasize that the only crucial nodes in the CFG are those that are in the set V_{bridge} (where V_{bridge} denotes the set of all the bridge nodes), we want the set of nodes (V) in the pruned graph (from the earlier phases) to be a superset of the set V_{bridge}. We observe (Assumption 4 in Section IV-D) that all the nodes in the set V_{bridge} are characterized by a significantly lower degree (say $< \gamma$) as compared to the degree of the other nodes in the graph. Thus, we filter out the nodes with a degree greater than γ. The transformation of the graph after degree-based pruning is given by –

$$\mathcal{G}''(V'',E'') \xrightarrow[pruning]{degree-based} \mathcal{G}'''(V''',E''') \tag{6}$$

$$\begin{aligned} V''' &= \{ v'' \in V'' \mid \deg(v'') < \gamma \} \\ E''' &= \big\{ (v_i'', v_j'') \in E'' \mid (v_i'' \in V''') \wedge (v_j'' \in V''') \big\} \end{aligned} \tag{7}$$

Here the *deg* (degree) function is the sum of the in-degree and the out-degree of a node.

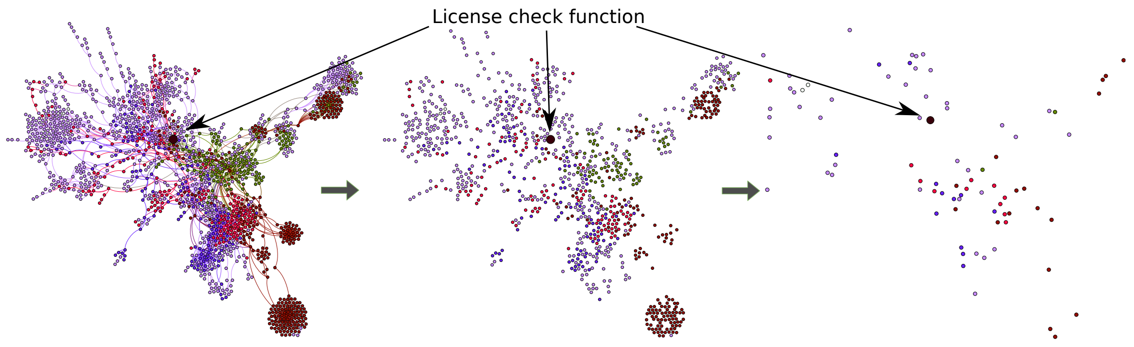

Figure 3: Visualization of the CFG across the pruning sub-phases for MySQL. (a) After clustering of the graph based on the nodes' properties, (b) after radius based pruning, and (c) after score and degree based pruning. The color of a node represents the cluster to which it belongs, and the black node represents the license check function.

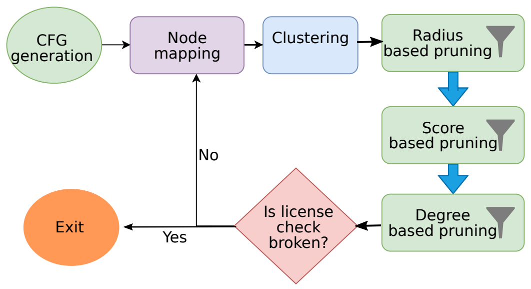

Figure 4: Flowchart of the entire process

VI. Defense Method: *F-LaaS*: Flexible- Licensing as a Service

We have created an algorithm to prune the CFG significantly, and consequently reduce the effort in finding the function that contains the license check. We shall show in Section VII that the time it takes to find the license check function using our *SmartCFB* technique reduces by 20X as compared to other state-of-the-art algorithms. This attack mechanism can prove to be really effective in such scenarios, and thus there is a need to propose a defense mechanism to thwart such attacks.

We propose *F-LaaS*, a novel licensing service that is immune to our *SmartCFB* approach, and all the attacks of a similar genre. The basic idea is shown in Figure 5. We distribute a *reduced binary* where some important functions of the original binary are replaced with *nop* instructions, thereby making them ineffectual. Let us refer to them as O-functions (omitted functions). The aim here is that even if the attacker is able to circumvent the license check and gain access to the restricted parts of the application, she will still not be able to run it the O-functions of the code will be missing. Furthermore, even if she attempts reverse engineering, we are limiting the amount of information that she can gather.

The normal flow of operations is as follows. If the correct license is presented, then the license server sends the missing parts of the binary over a secure encrypted channel, and the license manager (LM) dynamically patches the code after remote attestation (verification). Most modern Intel processors have the SGX [17] trusted execution environment that provides secure execution and remote attestation services. To ensure that the LM does not leak the patching information it gets from the server, we run the LM in a secure environment provided by Intel SGX [18]. SGX is a secure container that does not allow the code and data to be modified at runtime. In addition, it ensures that a given process runs in an isolated secure container and all the traffic between the processor and memory is encrypted. Even a malicious OS cannot breach the security provided by an SGX container.

We have several options here. Either we run the entire application including the LM in the SGX container. This provides the highest level of security. However, the overheads of SGX including attestation have been measured to be at least 40% [19]. This is prohibitive in most cases; thus, a better option is to only run the LM and the O-functions in the secure container after attestation. We shall evaluate all of these options in Section VII.

The aims are thus three-fold: Handicap the user without a valid license as far as possible, and ensure that a user with a valid license can execute the binary seamlessly, however, she cannot store the code (O-functions) received from the license server. The third point needs to be ensured, otherwise the user can patch the binary herself, break the license check, and get access to the entire functionality.

A. Selection of the O-Functions

To ensure an efficient implementation of *F-LaaS*, the O-functions removed from the binary should be small in size but at the same time should be disproportionately important in the execution of the binary. To determine these functions, we start out by considering those functions that are executed only after the license check is validated. Additionally, we discard all boilerplate code.

To determine the importance of a function in this set, we use a standard software engineering metric called *cohesion* to identify the functions that are critical to understanding a program [20]. Let f represent a function, R_{in} be its in-

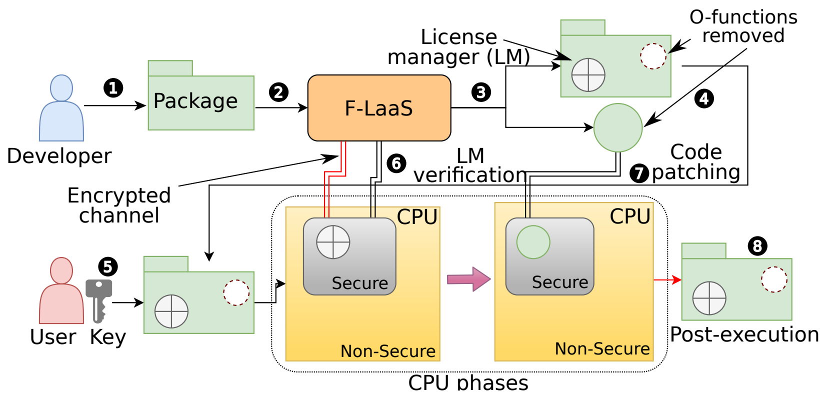

Figure 5: Overview of F-LaaS (shows the working when the license is valid)

degree, and let R_{out} be its out-degree. The cohesion $\nu(f)$ is equal to $R_{in}/(R_{in}+R_{out})$.

O-functions have a low cohesion value, which means that for them R_{out}/R_{in} is high. In other words, we want functions that are invoked relatively infrequently, yet they invoke a lot of other functions. This means that they have complex logic embedded in them, and thus if those functions are removed, it will be hard for an attacker to guess their functionality, and patch them.

VII. Evaluation

A. Benchmark Details

Table III shows the benchmarks used for the evaluation. These include four open source license managers, two versions of MySQL (code differs significantly between them), Nginx (secure web server), and a commercial Linux based application that does binary analysis (name cannot be disclosed). For the license managers, we bundle them with synthetic applications that have network, file, and disk based system calls. The four applications have in-built license managers. For MySQL, we capture the CFG between issuing a command and its response.

B. Experimental Setup

We execute our benchmarks on an Ubuntu Linux 18.04 machine, with 32 GB of memory, 8 Intel i7 cores, and 1TB of disk space. We use Intel PIN version 3.7 [8] and Valgrind (Callgrind) (version 3.4.1) [13] to get the instruction trace of the benchmarks. Intel Pin was also used to break the security check in the binary by inserting a direct jump instruction at the relevant place. Java applications were converted to an x86 binary using the GNU compiler for Java (*gcj*) [21]. We disabled ASLR during the experimentation. Some recent work [22] can be leveraged to figure out the random offsets that ASLR adds to addresses; this is orthogonal to our technique.

C. Hyper-parameter Tuning

In this Section, we explain the tuning of the hyper-parameters required in our pruning algorithm.

Hyper-parameter	Description
d	Dimensions of the space to which nodes are projected
C	Number of clusters for K-means
k	Hyper-parameter for the radius-based pruning phase
ζ	Hyper-parameter for the score-based pruning phase
γ	Hyper-parameter for the degree-based pruning phase

Table I: Description of the hyper-parameters used in the *SmartCFB* algorithm

As explained in Section V, the five hyper-parameters (see table I) control the size of the final pruned CFG. Our aim is to find the values of these hyper-parameters such that we can prune the size of the search space as much as possible.

As we decrease k or ζ, the size of the pruned CFG goes up (see Section V). In comparison the number of nodes in the CFG increases as we increase γ. We start with small values for d, C, $-k$, $-\zeta$ and γ such as 2, 2, -10, -10 and 1 respectively. This will result in a very small pruned CFG (without the license check node). Our aim is to increase the size of the pruned CFG, conservatively, till we find the license check node inside the pruned CFG (validated using CGA [10]). To eliminate redundant work, we never check the presence of the license check instructions in the same function twice.

Using these initial values of the hyper-parameters, we increase each of them by adding small randomly generated values to each hyper-parameter in every iteration. This will result in an increase in the size of the final pruned CFG. Eventually, the pruned CFG will contain the license check node. It typically takes 20-35 iterations to find the license check node.

A brief description of this process is shown in Algorithm 1.

The final values of hyper-parameters (d, C, k, ζ, and γ) are shown in Table II. We observe that pruning gives reasonable results when the nodes are mapped to a d-dimensional space, with d being in the range of 80 to 125. A higher dimension value allows for better flexibility in the representation of the nodes. We also note that the number of clusters for MySQL benchmarks are higher compared to

Algorithm 1

```
1: Input: Let the initial set of hyper-parameters be $\theta_0$.
   Instead of $k$ and $\zeta$ we store $-k$ and $-\zeta$ in $\theta_0$.
2: Output: $\theta_n$ is the value of the hyper-parameters in the
   $n^{th}$ iteration.
3: for $n = 1, 2 \ldots, N$ do
4:     Generate a random perturbation vector $\Delta_n \in \mathcal{R}^{+5}$.
5:     Update the values of the hyper-parameters:
6:     $\theta_{n+1} = \theta_n + \Delta_n$
7:     If SmartCFB is successful then break ;
8: end for
9: return $\theta_{N+1}$
```

others. This is because of the higher (1.8 X) number of nodes and edges in its CFG.

D. Validation of Assumptions

In this section we empirically validate the assumptions made in Section IV-D.

Validation of Assumption 2:
To establish our assumption 2 in Section IV-D, that the license check node is roughly 1-1.5 standard deviations apart from its cluster-center (in most cases), we report the distance of the license check node from its cluster-center in terms of the number of standard deviations in Table II. The distance (measured in terms of standard deviations) is in the range of 1 to 3.53, with the exception of MySQL 5.7 (with a value of 0.59). This is because of the large number of auxiliary functions.

Benchmarks	Dimension (D)	Clusters (C)	k (License check node distance)	Score (ζ) (percentile)
OpenLicense [23]	100	5	1.6 (1.62)	4.32 (98)
LicenseManager [24]	80	15	1.4 (1.41)	1.4 (84)
License3j [25]	100	10	3.5 (3.53)	4.14 (96)
License4j [26]	100	5	0.9 (1.1)	1.4 (64)
MySQL 5.7 [27]	100	17	0.58 (0.59)	6.9 (52)
MySQL 8.0 [27]	125	25	0.97 (1)	0.02 (72)
Nginx [28, 29]	100	7	1.08 (1.1)	1.66 (73)
Linux SW	100	15	1.2 (1.3)	2.7(85)

Table II: Table showing the final values of the hyper-parameters. Percentile of the license check node is shown along with the score threshold ζ. The value of γ for degree based filtering was set to 10 for all the benchmarks.

Validation of Assumption 3:
To verify our assumption, that the license check node is situated at the periphery of multiple clusters, we report the percentile of the score (ζ) assigned to the license check node in Table II. A high value of ζ for a node indicates that the node is closer to a large number of clusters. We observe that the percentile of the score for the license check node is high (64-98%) for most of the benchmarks, which indicates that it is much closer to the cluster-center than the other nodes in the CFG. MySQL 5.7 is an exception, where even though the score is relatively higher (6.9), the percentile value is low (52%) owing to the large number of auxiliary functions in MySQL.

Validation of Assumption 4:
To validate our assumption that the bridge nodes have a lower degree (sum of in-degree and out-degree) as compared to other nodes in their respective clusters, we show the relative ranking (in terms of percentiles) of the degree of the license check node in Table III . The value of the percentile is low (3.5-6.2%), which indicates that roughly 95% of the nodes in the CFG have a greater degree, which validates our assumption.

Benchmarks	Rank of the degree of the license check node (percentile)	Cohesion of O-function (Cohesion of cluster)
OpenLicense [23]	3.5	0.33 (0.63)
LicenseManager [24]	4.2	0.1 (0.52)
License3j [25]	3.7	0.11 (.53)
License4j [26]	5.6	0.32 (.7)
MySQL 5.7 [27]	3.9	0.09 (0.6)
MySQL 8.0 [27]	4.3	0.2 (0.7)
Nginx [28, 29]	6.2	0.22 (0.43)
Linux SW	5.7	0.3 (0.73)

Table III: Table showing the rank of the degree of the license check function (percentile), and mean cohesion values of O-functions.

E. Results for SmartCFB

SmartCFB reduces the search space (measured in terms of the number of functions we try to break) by an average of 22X (range:3-75X) as shown in Figure 6. The baseline uses CGA (see Section II).

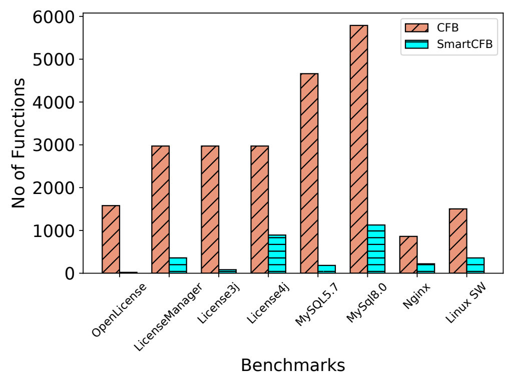

Figure 6: Performance comparison of CFB vs SmartCFB (mean reduction in the number of functions is 22.3X)

As can be seen in Figure 6, we do the most pruning for OpenLicense, and MySQL 5.7: 75X and 25X respectively. This result can be inferred from Table II. In OpenLicense manager, the score-based pruning removes the highest number of nodes, since the score of the license check node is at the 98th percentile.

For MySQL 5.7, a higher pruning is achieved despite the low values of k (radius-based filtering) and ζ(score-based filtering) because the number of high-degree nodes constitute a major fraction of its CFG and they are filtered out in the degree-based filtering phase. In contrast, we achieve only a 3X reduction for License4j because of low values of k, and ζ, which result in a lesser degree of pruning. Also, a high value of the degree percentile means that there are many lower degree nodes in its CFG, thereby resulting in a lesser amount of pruning.

F. Selection of the O-function(s)

As explained in Section VI-A, we use a metric called cohesion to decide the function(s) that we need to remove to create a *reduced* version of the binary. A tradeoff exists between the number of functions we decide to remove, the performance overheads, and the amount of information we remove from the binary. This is a hard problem.

In the software engineering community, researchers have been working on program understanding for the last 40 years. We use the classic results by Soloway and Erhlich [30], which say that most programmers (novice and expert) understand programs (or reverse engineer) by primarily looking at functions that show the overall flow of actions. This is referred to as a *plan*. Their paper and later work on top-down program understanding advocates the view that the functions that determine the flow of the program (sequence of work that needs to be done) carry the maximum amount of information.

We thus need to verify if the O-functions that we remove as per our low-cohesion metric satisfy the criteria laid down by Soloway and Erhlich. This requires manual analysis of the source code. Our criteria for choosing an O-function is that it should have the lowest value of cohesion in its cluster, and the total number of instructions in its forward slice should at least be 5000. We then analyzed the source code of the O-functions that we obtained, and in every case we found that the code significantly determined the subsequent plan of execution.

G. Evaluation of the Performance of F-LaaS

In this section, we discuss the performance overhead of a binary executing with *F-LaaS*. We execute the license check code within a secure SGX enclave. In addition, we also execute the code to fetch O-functions, and patch them within the enclave. The baseline system does not use SGX. To calculate the performance overhead, we simulate the benchmarks on a cycle accurate architectural simulator, Tejas [31]. We ignore the network overheads: time it takes to fetch the O-functions (their encrypted versions can be locally cached).

Figure 7 shows the performance overheads of *F-LaaS* when we are running the license check function in the SGX enclave. The overhead is defined as the fraction of the additional execution time of *F-LaaS* as compared to the baseline implementation. This is dependent on the slowdown induced by running parts of the code in SGX (roughly 20-30% in most cases), and the size of the license manager and O-functions. We observe from Figure 7 that the average performance overhead is between 0-40% for the license manager by itself, and the slowdown induced due to the O-functions is just 0.26% on an average.

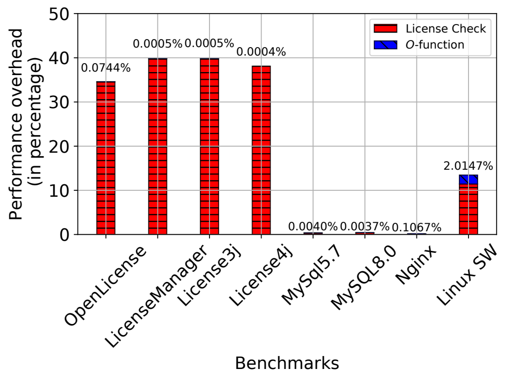

Figure 7: Comparison of performance overheads while running O-functions within an SGX enclave

This can be understood better from Table IV. We execute the benchmarks with Valgrind to get the number of instructions executed by the license check module and the O-functions. The size of our O-functions is roughly between 5000 and 25,000 instructions, which is minimal as compared to even the size of the license manager that can execute up to 2 billion instructions.

Since running the license manager on SGX is not a necessary feature in our design, we only consider the overhead of patching O-functions as the real overhead, which is 0.26%.

Benchmark	License check #instructions	O-Functions #instructions
OpenLicense	3,938,155	8,481
LicenseManager	1,256,847,478	15,194
License3j	1,640,145,369	19,135
License4j	1,759,762,658	20,673
MySQL5.7	1,735,905	15,446
MySQL8	2,642,487	24,757
Nginx	13,427	5,158
Linux SW	41,626	7,331

Table IV: Table showing instructions executed by the license check function and the O-functions selected by *F-LaaS*.

VIII. Conclusion and Future Work

In this paper, we show that software-based defense methods used to control access to the protected region of a binary are not sufficient and can be broken with the *SmartCFB* technique. We reduce the search space and hence improve the performance by an average of 22X in breaking 8 widely used programs: open source and proprietary. To protect against such attacks we need hardware support to ensure tamper-proof execution. We have proposed to use the Intel SGX [18] trusted execution environment in this paper. However, an inherent drawback of such technologies is that they are limited in terms of the memory size of the protected region. This causes a significant slowdown in the performance of the application as shown in Section VII. To solve this problem, we propose *F-LaaS*, an efficient License-as-a-service mechanism, which identifies important functions, fetches (or decrypts) their code at runtime and executes them in a secure container with a performance overhead limited to 0.26 %.

In this paper we proposed the concepts underlying *F-LaaS* and showed a proof-of-concept. In the future, we would like to do a more in depth study of the design space of binary patching mechanisms that run in secure execution environments, and create a version of *F-LaaS* that is significantly more flexible, performance and power efficient.

Acknowledgment

The authors would like to thank the anonymous reviewers for their valuable comments. This work has been partially supported by the Semiconductor Research Corporation, and the Ministry of Science and Technology (Govt. of India) via the grant DST/INT/JST/P-30/2016.

References

[1] W. Ziegler, H. Rasheed, and K. Catewicz, "Leveraging use of software-license-protected applications in clouds," in *CLOSER*, 2016.

[2] W. Ziegler, "A framework for managing quality of service in cloud computing through service level agreements," 2017.

[3] L. D. Paulson, "Building rich web applications with ajax," *IEEE Computer*, vol. 38, pp. 14–17, 2005.

[4] N. Burow, X. Zhang, and M. Payer, "Shining light on shadow stacks," *CoRR*.

[5] J. Dagit, S. Winwood, G. Ritter, J. Berkes, and A. Wick, "Code re-use attacks and their mitigation," Galois, Inc., Tech. Rep., 2017.

[6] X. Ge, W. Cui, and T. Jaeger, "Griffin: Guarding control flows using intel processor trace," in *ASPLOS*, 2017.

[7] F. Bellard, "Qemu, a fast and portable dynamic translator," in *USENIX ATC, FREENIX Track*, 2005.

[8] "Pin - a dynamic binary instrumentation tool," https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool.

[9] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross, "Control-flow bending: On the effectiveness of control-flow integrity." in *USENIX Security Symposium*, 2015.

[10] K. Agarwal, P. Kallurkar, S. Krishna, and S. R. Sarangi, "Ethical hacking of license managers," 2015.

[11] "Overview of the spec benchmarks," http://jimgray.azurewebsites.net/benchmarkhandbook/chapter9.pdf.

[12] L. Davi, M. Hanreich, D. Paul, A.-R. Sadeghi, P. Koeberl, D. Sullivan, O. Arias, and Y. Jin, "Hafix: Hardware-assisted flow integrity extension," ser. DAC, 2015.

[13] "Valgrind home," http://valgrind.org/.

[14] A. Grover and J. Leskovec, "node2vec: Scalable feature learning for networks," in *SIGKDD*. ACM, 2016, pp. 855–864.

[15] H. Abdi and L. J. Williams, "Principal component analysis," *WIREs Comput. Stat.*

[16] D. Arthur and S. Vassilvitskii, "K-means++: the advantages of careful seeding," in *SODA*, 2007.

[17] "hiie-report-s16-17.pdf," https://courses.cs.ut.ee/MTAT.07.022/2017_spring/uploads/Main/hiie-report-s16-17.pdf.

[18] V. Costan and S. Devadas, "Intel sgx explained," *IACR Cryptology ePrint Archive*, 2016.

[19] O. Weisse, V. Bertacco, and T. Austin, "Regaining lost cycles with hotcalls: A fast interface for sgx secure enclaves," in *ACM SIGARCH Computer Architecture News*, vol. 45, no. 2. ACM, 2017, pp. 81–93.

[20] A. Lakhotia and J.-C. Deprez, "Restructuring functions with low cohesion," in *WCRE*. IEEE, 1999.

[21] "GCJ," https://gcc.gnu.org/wiki/GCJ.

[22] D. Evtyushkin, D. V. Ponomarev, and N. B. Abu-Ghazaleh, "Jump over aslr: Attacking branch predictors to bypass aslr," *MICRO*, 2016.

[23] "An open license manager written in c++," https://github.com/open-license-manager/open-license-manager.

[24] "Java licensing," https://mvnrepository.com/artifact/net.nicholaswilliams.java.licensing.

[25] "verhas/license3j: Free licence management library," https://github.com/verhas/License3j.

[26] "License4j - license manager, java software licensing solutions," https://www.license4j.com/.

[27] "Mysql :: Mysql 8.0 reference manual," https://dev.mysql.com/doc/refman/8.0/en/.

[28] "Nginx — high performance load balancer, web server, & reverse proxy," https://www.nginx.com/.

[29] "Digest authentication for nginx," https://github.com/atomx/nginx-http-auth-digest.

[30] E. Soloway and K. Ehrlich, "Empirical studies of programming knowledge," *IEEE Transactions on software engineering*, no. 5, pp. 595–609, 1984.

[31] S. R. Sarangi, R. Kalayappan, P. Kallurkar, S. Goel, and E. Peter, "Tejas: A java based versatile micro-architectural simulator," in *PATMOS*. IEEE, 2015.