Secure Software Licensing: Models, Constructions, and Proofs

Sergiu Costea
University POLITEHNICA of Bucharest
sergiu.costea@cs.pub.ro

Bogdan Warinschi
University of Bristol
bogdan@cs.bris.ac.uk

***Abstract*—The problem of secure software licensing is to enforce meaningful restrictions on how software is run on machines outside the control of the software author/vendor. The problem has been addressed through a variety of approaches from software obfuscation to hardware-based solutions, but existent solutions offer only heuristic guarantees which are often invalidated by attacks.**

This paper establishes foundations for secure software licensing in the form of rigorous models. We identify and formalize two key properties. Privacy demands that licensed software does not leak unwanted information, and integrity ensures that the use of licensed software is compliant with a license – the license is a parameter of our models. Our formal definitions and proposed constructions leverage the isolation/attestation capabilities of recently proposed trusted hardware like SGX which proves to be a key enabling technology for provably secure software licensing.

I. Introduction

Improvements in trusted hardware technologies like TrustZone and SGX [1] [2] enable new applications: restrict the access of programs to protected information [3] [4] [5] [6] or enable the remote execution of code while ensuring strong privacy and authenticity guarantees [7] [8]. Software vendors can use these applications to guard their intellectual property – code, keys and proprietary data – against unauthorized use or theft. This type of use falls within the broader problem of software licensing, where vendors want to ensure that executions of programs on remote machines outside their control comply to some well-defined rules.

When attacking software executed on their own machine, malicious parties have a variety of attack vectors at their disposal. For example, an attacker can glean information during transfer of the code, attempt to reverse engineer its execution, deduce information about program structure from its outputs under both normal and abnormal termination, listen on side-channels (e.g. pattern of memory accesses), etc. The complexity of possible attack scenarios easily leads to pitfalls; even ideas that seem to "obviously" work may unintentionally leak information.

In this paper we make the first steps towards rigorous models for secure software licensing. We leverage definitional ideas from the fields of secure obfuscation and modern cryptography. While mathematical models are not without their own pitfalls, they play a crucial role in clarifying security assumptions, adversarial powers and goals, thus serving as validators of system design.

Motivated by the realistic possibility that trusted hardware with advanced capabilities may soon be ubiquitous [9], we cast our models in a setting where machines are equipped with such technology. We propose constructions for these settings and demonstrate mathematically that they meet our desired requirements. Our possibility results show that certain applications – software-only signing dongles or authenticating DRM for commercial content – are both feasible and provably secure. We detail our results next.

System architecture and execution model. Our models and constructions are for a setting where it is not possible to store securely persistent state on the remote machine. This assumption is consistent with the design of Intel SGX, where on system reset the state of the running program is lost. Although SGX includes sealing, thus providing storage with integrity and confidentiality, it is not sufficient to provide offline licensing. This would require practical state continuity, where adversaries cannot force replays on local state. While hardware based solutions exist (ICE for SGX [6]), the current generation of SGX architectures does not provide secure storage with these properties.

We view a licensing scheme as a two-party protocol between a licensing server and a remote machine with access to trusted hardware. To avoid being tied down to a particular trusted hardware architecture, our model surfaces only two main capabilities of such hardware: their ability to completely isolate software execution from interference by the machine on which it resides and the ability to carry out some cryptographic operations.

Our licensing schemes are two-party protocols composed of two phases. In the first phase, the software vendor runs a protection algorithm, which takes as input the target program and the desired licensing restrictions. The algorithm outputs a protected version of the same program and a secret token, which are then shared with the client user through a secure communication channel. In the second phase, the user runs the program and uses the token to unlock the protection around the original code; depending on the licensing algorithm, this second phase can also involve communication between vendor and user.

System execution is captured via traces that record various events during the execution (e.g. the input-output at the interface of the trusted hardware, on which machine a certain

event occurs, etc.).

SECURITY MODELS. We identify three properties which should be satisfied by any good software licensing scheme. The first requirement is a correctness property and demands that the protected program still performs its function. The next two properties are security requirements: privacy of code demands that no information on the the source code or machine code is leaked; licensing compliance requires that the functionality of the program can only be unlocked under specific circumstances, dictated by the license. Code privacy requires that no information is leaked except the information obtained from program output. Our mathematical formulation is based on previous work on software obfuscation, namely the concept of virtual black-box and, informally, it says that an adversary with access to the licensed software does not glean any more information than if given access to the software in a black-box way. We remark that well-known impossibility results that show that this level of obfuscation is not possible in general [10] do not apply, as we consider solutions based on trusted hardware.

To define secure licensing, we consider a flexible definition parametrized by an arbitrary licensing predicate: secure licensing essentially imposes that with overwhelming probability execution traces satisfy this predicate. As we discuss later in the paper in more details, it is not possible to enforce arbitrary licensing predicates on arbitrary programs. For example, it is not possible to stop a user from executing a licensed program on the same input twice, even if it is on different hardware; once the user knows the behavior, they can reimplement it on their own and execute it as many times as desired. In turn, this indicates that a single generic construction that enforces arbitrary predicates is also not possible. Nonetheless, we specify three distinct licensing models and provide constructions that enforce them.

Licensed use is closest to the licensing model many programs use today. Once a user has purchased a program, he is allowed to run it an unlimited number of times and on an unlimited number of machines. Although legal restrictions that impose additional limitations might exist, they are not directly enforced by the licensing system.

Limited use imposes restrictions on the number of times a specific application can be run. This is commonly used in trial versions, where the application requires the user to purchase the complete version after a specific number of application starts have been counted. Having an application that can be executed at most 10 times can be used to rapidly implement trial versions, but it can also be used to sell cheaper rights of use for enterprise applications. For example, a casual user might be interested in processing a home made video using professional level tools, but finds the cost of a full purchase prohibitive. The application vendor might be interested in selling its processing tool for a discounted price, while limiting its number of uses. In a different scenario, one may also use this licensing model to delegate signing/decryption rights for a limited time.

Limited machines states that an application cannot be executed on more than a specified number of processors. Software vendors use this model for applications where limiting processing power is desirable, for example when licensing network services such as firewalls and instrusion detectors, or for commodity applications where the end-user pays on a per-installation basis.

CONSTRUCTIONS. We propose schemes that implement the three previous licensing models; for each scheme, we provide proofs of correctness and security, and identify the class of programs to which they apply. Our first scheme (Privacy Preserving Licensing) provides privacy of code, while allowing users to only execute the licensed application if they have a secret token received from the software vendor (typically following a purchase). Our second scheme (Run Count Licensing) builds upon the first one; in addition to privacy of code, it also allows the software vendor to restrict the number of program executions (and the total information users obtain about program functionality). We also show that some classes of programs cannot be licensed under the *limited use* licensing model. Our third scheme (Machine Count Licensing) provides privacy of code and allows software vendors to restrict the number of machines on which a program can run. This type of license is commonly used in virtualization, where vendors lock the number of processors that a virtual machine can run on in order to throttle performance.

II. RELATED WORK

Depending on the type of information, there are two categories of protection algorithms: data protection (e.g. books, music, video) and code protection (e.g. programs). Although some overlap exists between the two, technologies in the former typically only restrict access to static content; examples include Apple FairPlay [12], Microsoft Protected Media Path [13], and the AACS standard [14]. Our framework targets the latter application domain. Here, the protection mechanism — usually a secure licensing infrastructure — must allow the user to interact with the protected code. Most prior work is described in patents [15] [16] [17]. However, these solutions take a best effort approach, and do not provide any formal guarantees of security. Existing implementations that do not rely on special hardware on the client's part (e.g. licensing with online activation requirements or install keys) have shown that most licensing schemes are only a deterrent, and not a guarantee of security.

Program obfuscation has been studied in the literature as a means to protect code privacy. The notion essentially demands that no information about the original code can be obtained besides what one can deduce from its functionality. Barak et. al formalized obfuscators as virtual black-boxes [10] and proved that general obfuscation in this sense is impossible. Later contributions built obfuscators for specific classes of circuits, such as conjunctions [18] and point functions [19]. Our notion of privacy is essentially the one proposed by [10], and we bypass the impossibility result through the use of

trusted hardware. We remark that obfuscation goes only some way towards secure licensing (e.g. one can control the input output behavior of a program) but cannot prevent, for example, that the program is run on two different machines.

More recently, customizable hardware with cryptographic capabilities opened a path toward secure software licensing; as opposed to obfuscation, program code can be stored encrypted, and is only decrypted during execution. There are applications which rely on the Trusted Platform Modules (TPMs) to implement this idea. For example, TrInc [3] implements a secure monotonic counter, kept on a tamper-proof device; user programs can then request unique attestations from the device. Memoir [4] uses secure hardware to implement state continuity, while Pasture [5] proves whether a user has accessed a resource. These solutions require persistent state on client hardware, which is achieved by having access to secure non-volatile RAM (NVRAM) in custom hardware (thus increasing costs). Other limitations include: the small size of the NVRAM (which limits extensive use); only pieces of data or functionality are protected, not entire programs; applications and protocols need to be redesigned to use the new secure functions.

More recent hardware such as TrustZone [1] or SGX [2], offer improved capabilities [20] that can be used to create an all-around solution. Haven [8] is built on top of SGX and protects the runtime of entire legacy applications without changes to the code base. Our constructions are similar to VC3 [7], which allows users to run encrypted MapReduce functions over untrusted clouds running trusted hardware. As opposed to obfuscation, such hardware allows the implementation of virtual black-boxes for general purpose applications. Our constructions share with VC3 the basic idea to protect programs by encryption which is then only removed inside an isolated environment. Our overall goal is to enforce restricted use for these programs.

Our models and constructions rely on trusted hardware which we view as an abstract machine with strong guarantees, and our proofs rely on the validity of this abstraction. Side-channel attacks and controlled-channel attacks [21] against code executed inside trusted environments provide additional information to the attacker. However, measuring their impact and providing countermeasures is orthogonal to our problem, as these additional attack vectors do not invalidate our definitions and constructions.

III. Notation and Definitions

We use calligraphic uppercase letters (e.g. $\mathcal{M}$, $\mathcal{K}$) for unordered sets, and notations such as $\overrightarrow{M}$ or uppercase greek letters to refer to vectors or associative arrays, respectively. Specific elements can be accessed either via integer indices ($\overrightarrow{M}[2]$) or key lookup in the case of associative arrays ($\Lambda[M_1]$). The notation $a \leftarrow b$ represents assignment, while $a \xleftarrow{\$} \mathcal{K}$ where $\mathcal{K}$ is a set represents random sampling from the elements of the set according to the uniform distribution. We write $\mathsf{Exp} \Rightarrow x$ or $A \Rightarrow x$ when an experiment or adversary outputs value x. We say that two circuits C_1 and C_2 are equivalent, if $\forall x$, $C_1(x) = C_2(x)$.

For procedure calls where the behavior is influenced by some variable x not passed as a parameter, we explicitly specify the dependency within square brackets: $\mathsf{Procedure}[x](a, b)$.

Given a stateful algorithm A, the notation st_A always refers to the current state of the algorithm. If an algorithm is marked as stateful within the text, then the state is implicitly updated during algorithm execution and persists through iterations of the same algorithm. When referring to variable x contained in the pseudocode of algorithm A from outside the pseudocode, we write $st_A.x$.

Basic cryptographic primitives. For a public encryption scheme pke we write $\mathcal{K}^{pke}$, Enc^{pke} and Dec^{pke} to mean the key generation, encryption and decryption algorithms respectively. The notation K^+ is used for public keys, while K^- is used for private keys. We write $c \leftarrow \mathsf{Enc}^{pke}_{K^+}(m)$ for the process of encrypting message m under public key K^+ and obtaining ciphertext c. We write $m \leftarrow \mathsf{Dec}^{pke}_{K^-}(c)$ for obtaining m as the result of decrypting c using decryption key K^-. We require that pke satisfy $\mathsf{Dec}^{pke}_{K^-}(\mathsf{Enc}^{pke}_{K^+}(m)) = m$ for any message m. We use similar notation for symmetric encryption, with K for the shared secret key. For both symmetric and asymmetric schemes we use (and assume familiarity with) the standard notion of indistinguishability under chosen-ciphertext attacks ($\mathsf{IND\text{-}CCA}$ security) [22].

IV. Machines, programs, and processes

Machines. Our machine model is heavily inspired from [23]. We define a *machine* as a computing abstraction with memory, central processing unit and input and output devices, capable of running interactive programs. The term *program* refers to a well-formed collection of code detailing the behavior of some algorithm.

Programs communicate with the outside world using the following set of instructions:

- $\mathsf{Send}(x)$. On $\mathsf{Send}(x)$, the value x is transmitted on the network. Execution continues until a Receive or Return instruction is received.
- $x \leftarrow \mathsf{Receive}()$. On $\mathsf{Receive}()$, the process blocks waiting for input. When input is received, it is parsed and stored in x.
- $\mathsf{Return}\ x$. On Return, the process immediately terminates with output x.

We write $P = P_1 \| P_2 \| \ldots \| P_k$ for a program composed of sections P_1 through P_k. We describe sections either through their raw representation (when dealing with data or code) or through high level pseudocode descriptions (only when the section contains plaintext code). External code can also call a specific section with a set of arguments and obtain the output (similarly to a procedure).

We use the term *process* to refer to one specific instance of program execution. Once a process starts, we assume the code is loaded into memory in its entirety, and thus consider it to be a part of process state. Processes are identified via handles

— unique global identifiers. Handles are generated at process creation, and are used to both differentiate between processes on the same machine and on different machines.

ISOLATED PROCESSES AND TEEs. We require that machines be able to create and run processes in complete isolation. An isolated process does not allow read or write access to its internal state, either by outside observers or other processes running on the same machine. Isolated processes — or, more precisely, parts of processes — can be implemented in practice using Trusted Execution Environments (TEEs); secure hardware enforces integrity and confidentiality of state [2]. When refering to TEEs, we refer strictly to the process running within them.

MACHINE INTERFACE. Similarly to [23], we define a small machine interface consisting of three calls:

- $\mathsf{Init}(1^\lambda)$ is the machine constructor. It takes as input some security parameters and outputs a vector of machines $\overrightarrow{M}$. This procedure models the production of TEE capable machines; cryptographic keys are stored on each constructed machine (similarly to how a processor vendor burns keys on a chip). These keys are assumed secret and programs gain access to the keys through a restricted set of privileged instructions to hardware components. Public parameters are assumed to be available to all parties. To simplify notation we do not show these parameters and simply write $\overrightarrow{M} \leftarrow \mathsf{Init}(1^\lambda)$. We assume that each machine is identified by some unique integer ID.
- $M.\mathsf{Load}(P)$ takes as input a program P, and outputs a handle δ. The handle δ is a reference to a newly created process on machine M, running inside a TEE. $M.\mathsf{Load}(P)$ is an abstraction of all steps executed by the hardware and operating system of a physical machine M to create a new process for program P. We write $\delta \leftarrow M.\mathsf{Load}(P)$ for the action of initializing a new process on machine M (with process handle δ) running an instance of program P.
- $M.\mathsf{Run}(\delta, i)$ takes as input a process handle δ and input i and models sending the input i to the process identified by δ. The process runs until it encounters Receive or Return; the output of Run consists of the sequence of values sent on the network using Send and to the user using Return.

PRIVILEGED INSTRUCTIONS. We allow for processes runing inside TEEs to make a set of privileged calls to gain access to secret keying material, stored on machine hardware during Init. Although the procedures are based on the features provided by SGX, our definitions are compatible with any hardware platform that implements calls satisfying the security guarantees outlined at the end of this section. Only TEEs are given access to these calls; external users are unable to call them directly.

This restricted set contains the following instructions:

- $\mathsf{MSign}[P](m)$: given some bit string message m, the machine uses secret keying material stored within it to return a signature on m. Signatures can be verified using MVerify, with the property that if $\sigma \leftarrow \mathsf{MSign}[P](m)$, then $\mathsf{MVerify}[P](m, \sigma) = 1$. As opposed to MSign, MVerify is a public algorithm and can be executed anywhere, with the public keys used by the verification assumed to be known.
- $\mathsf{MKeyGen}[P](m)$: deterministically computes a secret key for use by the calling process, depending on P, m, and the secret keys stored within the machine. The deterministic nature gives processes sharing the same code (i.e. instances of the same program) access to the same key, thus enabling a program to store encrypted data for future runs. We require that MKeyGen operate in a similar way to pseudorandom functions.
- $\mathsf{MRand}()$: outputs any requested number of random bits. When a program needs access to randomness to sample value x from distribution $\mathcal{D}$, we write $x \xleftarrow{\mathsf{MRand}} \mathcal{D}$.

P represents the code of the entire program, received by Load when the process was created. The value is used as an internal seed within MSign and MKeyGen; if a forged program P' attempts to sign the same message, the verification procedure for it would fail, since the verifier checks the signature $\mathsf{MSign}[P'](m)$ against the expected program P.

Currently, randomized programs are not modeled[1]. However, the model can be easily extended to support such operations by defining a random tape on each machine; programs would receive data from both the random tape and standard input. Since the policy enforcing constructions do not look at input and output, they are compatible with this extension.

We assume that i. no adversary can produce a signature on behalf of TEE's that holds some program, unless this signature has been obtained from an instance of such a TEE, ii. no adversary can distinguish keys generated via MKeyGen and bitstrings generated via MRand from uniformly random strings (of appropriate size and structure). Formalizations for these properties are standard [24].

V. LICENSING SCHEMES

We define licensing as an interaction between two parties: the software vendor and the software user. The former intends to publish a piece of code which enforces restrictions on how it can be executed, and which encodes some specific functionality. The latter purchases tokens from the vendor, thus unlocking the functionality of the software. We model the parties as two efficient and stateful algorithms: Vendor and User.

Vendor runs within the trusted zone of the software vendor (i.e. on some hardware within the vendor's network) and does not need access to machines with TEE capabilities. Vendor receives a program (modeled as a circuit[2]) and outputs a protected version which is subject to some desired restrictions. The algorithm executes indefinitely (similar to a server

[1]Programs have access to randomness for the purpose of network exchanges; however, the final output of the program must be deterministic.

[2]We use circuits as computational abstraction for programs to stay close to existing obfuscation literature on which we build upon. Any other formalism such as Turing machines, RAM machines, or even a detailed machine model may be used.

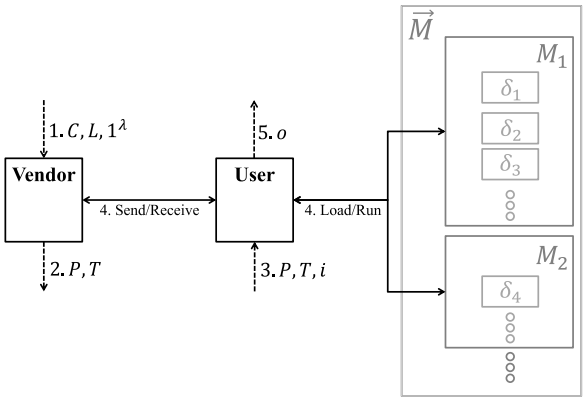

Fig. 1. Overview of interactions between Vendor and User. Vendor takes as input circuit C, license specification L and outputs protected program P and use-token T. User takes P, T and some input i and computes $o = C(i)$, if the license permits.

listening for requests), and maintains a persistent internal state which tracks licensing related data.

User runs on remote hardware and has access to machines capable of initializing TEEs through the interface in Section IV. User is stateful, and updates state object st_{User} with each execution. Future iterations of the algorithm use the state to make correct licensing related decisions (for example, to maintain a list of usable machines).

The scheme is used to license circuits that belong to some arbitrary set of circuits $\mathcal{D}$.

The interaction between User and Vendor is shown in Fig. 1. The complete syntax is defined as follows:

- $P, T \leftarrow \mathsf{Vendor}(C, L, 1^\lambda)$. Vendor takes as input a circuit to protect C, a licensing compliance predicate L and security parameter 1^λ and outputs a protected version of the program P and use-token T. The token is used to gain access to the functionality of C. Licensing compliance predicates are defined in the following section.
- $o \leftarrow \mathsf{User}^M(P, T, i)$. User receives protected program P, license token T and input value i, and is given access to machine $M \in \overrightarrow{M}$, capable of running TEEs. User outputs value o, where o is either the evaluation of C over input i (and correctly recovering functionality) or is $\perp$ (signifying that all valid executions have already been used). User is stateful, and the state is stored internally in st_{User}.

LICENSE COMPLIANCE PREDICATES. Licensing compliance is a desirable property of licensing schemes, which states that users can never obtain more information about the behavior of a licensed program than the amount allowed by the license. For example, if a software vendor licenses a program for one-time use, then the user can never obtain knowledge about valid outputs for two different inputs. Licensing restrictions may refer to the number of machines on which the program is executed, or the number of times the user should be able to execute the licensed program.

We define licensing compliance using predicates over execution traces. For each machine, we record the traces of local processes together with a unique integer timestamp describing the time of execution. The structure containing the traces for all machines forms the *execution transcript*, which we formally define as the map $\tau : ID \rightarrow (\mathbb{N} \times \mathcal{I} \times \mathcal{O})^*$, where $\mathcal{I}$ is the set of valid inputs and $\mathcal{O}$ is the set of valid outputs. The function maps machine identifiers to lists of tuples, each pair containing the timestamp and input and output trace of a single process.

We define $\mathcal{T}$ to be the set of all possible transcripts. We write $(i, o) \in \tau(id)$ if there exists some machine identifier id such that $\tau(id)$ contains the tuple (t, i, o), for some value of t. We also write $(i, o) \in \tau$ if we know at least one machine's trace contains some tuple (t, i, o).

We model licensing compliance using licensing predicate L, where $L : \mathcal{D} \times \mathcal{T} \rightarrow \{0, 1\}$. For example, consider a licensing restriction where the licensed program can only run once. L in this case is defined as:

$$L(C, \tau) = \begin{cases} 1, \text{ if } |\{(i, o) | i \in \mathcal{I}, o \in \mathcal{O}, \\ \qquad (i, o) \in \tau, C(i) = o\}| = 1 \\ 0, \text{ otherwise} \end{cases} \tag{1}$$

In this example, if the adversary could run the program twice, then they would obtain a transcript containing two valid pairs, and the licensing predicate would evaluate to 0. Note that the adversary also has other ways of breaking the license; if they could somehow obtain the code (the description of C), or guess the behavior of C, they could write an identically behaving program themselves, obtain multiple valid input and output pairs, and thus create a transcript on which L evaluates to 0. This highlights the fact that our transcript definition also accounts for any code extraction or guessing capability the computationally bounded adversary might possess; licensing schemes that are compliant with a predicate in our model are guaranteed to protect their behavior against all such adversaries.

CORRECTNESS. Licensing schemes take as input the program of the software provider, formally modeled as a circuit. Regardless of any transformation the circuit suffers, a licensing scheme must offer a means to compute the same function as the initial circuit, for all valid inputs. If this is the case, we say that the scheme *preserves the functionality* of the program.

Let u be this predicate, where $u : \Sigma_U \times \Sigma_V \times \mathcal{M} \rightarrow \{0, 1\}$. The predicate equals 1 while the scheme still allows the user to correctly execute the program (and obtain valid output), and equals 0 otherwise. The sets Σ_U and Σ_V consist of the possible states of User and Vendor, respectively, while $\mathcal{M}$ is the set of machines the user has access to.

Using u, we now define the property of functionality preservation:

Definition 1 (Functionality preservation)**.** Let λ be a security parameter, L a licensing compliance predicate and $\mathcal{S} = (\mathsf{User}, \mathsf{Vendor}, u_L)$ a licensing scheme with algorithm states st_{User} and st_{Vendor}, both initially empty. $\mathcal{S}$ *preserves functionality* with respect to L if for any circuit $C \in \mathcal{D}$, any transcript $\tau \in \mathcal{T}$, any input value i, any machine $M \in \mathcal{M}$ and for:

$$P, T \leftarrow \mathsf{Vendor}(C, L, 1^\lambda) \tag{2}$$

then it holds that $\mathsf{User}^M(P, T, i) = C(i)$ if and only if $u_L(st_{\mathsf{User}}, st_{\mathsf{Vendor}}, M) = 1$.

The circuit to license is C, which is first passed to the vendor algorithm Vendor, thus obtaining the protected version P. The definition states that if the license permits more executions on some machine M, then the user algorithm User is able to recover the initial functionality of C. Otherwise, the behavior is undefined. User is stateful, and its future iterations are able to use the information obtained from previous ones. When Vendor executes, the state of User is considered empty (the user does not have any information before the protected version of C is constructed).

VI. Security for licensing schemes

ADVERSARIAL MODEL. We assume the communication channel between the users and the vendor provides confidentiality, authentication and integrity (for example, using TLS tunnels). The licensing scheme is a two party protocol, where one of the parties is untrustworthy; Vendor runs in a trusted environment, while the user is malicious and might deviate from its normal operation described in User. For example, the user might attempt to bypass licensing restrictions by exploiting faults in the licensing scheme implementation. To distinguish from the fair user algorithm User, we assume a computationally bounded adversary A directly interacts with Vendor instead of User. The adversary has acceess to an arbitrarily large set of machines that support TEE, and may run the licensed program on any number of these machines.

Secure licensing schemes guarantee that licensed program code remains secret, and that program functionality can only be recovered if the license allows it. We formalize these concerns in two definitions: *circuit privacy* and *licensing compliance*.

A. Circuit privacy

Notice that even if executed under license, some information about the program (e.g. its behavior on some inputs) is unavoidably leaked. Our first notion demands that a good licensing scheme should only leak whatever information an adversary can obtain by observing the input/output behavior of the licensed software, and nothing more. In particular, it follows that a good scheme should protect the program in transit to the user's machine and should ensure that when executed on the remote machine information like the control path followed by the execution is kept private. Our definition is based on virtual black-boxes (VBB), as introduced in obfuscation literature. First we give the formal definition and then we discuss it.

Definition 2 (Circuit privacy)**.** Let λ be a security parameter and L a licensing predicate. Licensing scheme $\mathcal{S} = (\mathsf{User}, \mathsf{Vendor}, u)$ provides *circuit privacy* if for all efficient adversaries A, there exists an efficient algorithm S s.t. for all $C \in \mathcal{D}$ and for all functions π, $\pi : \mathcal{D} \to \{0,1\}^*$, then there exists a negligible function f in λ s.t.:

$$\left| Pr[A^{\mathcal{M}}(P, T) \Rightarrow \pi(C)] - Pr[S^C(|C|) \Rightarrow \pi(C)] \right| \leq f(\lambda) \tag{3}$$

where the first probability is taken over the choice $(P, T) \leftarrow \mathsf{Vendor}(C, L, 1^\lambda)$, the internal coin tosses of Vendor, A and machines in $\mathcal{M}$, and the second probability is taken over the internal coin tosses of S.

The definition states that the operation of licensing scheme $\mathcal{S}$ leaks no information about circuit C, with the exception of information leaked by its black-box behavior. First, Vendor is called to build P, the protected version of the circuit. The protected version ensures confidentiality of the code during transfer to the user. The scheme gives active adversary A access to the existing runtimes, and allows the adversary to take an active part in the network communication performed by the scheme. Function π represents some property of the input circuit C, expressed as a sequence of bits of finite length. The value $\pi(C)$ quantifies the knowledge that an algorithm is able to obtain about the circuit, and the definition states that the adversary would not be able to obtain signifcantly more information about C than a simulator with only black box access.

We consider the length of the circuit to be unimportant information, and leak it to the simulator. If the length is critical, an alternative definition could be considered which would only include circuits up to some predefined length; for shorter circuits, schemes could pad the circuit's description up to maximum length and thus hide the true value.

We note that the above definition does not reflect and is not concerned at all with any limitations that licensing may impose on how the adversary accesses P (in particular the simulator has no constraints on how it accesses C). In this sense, the definition provides an upper bound on the information learned about the program.

VBB might seem too strong in the context of licensing (because it does not account for access controls that licensing may impose), but this notion is very close to what SGX-like hardware provides and we can comfortably demand it from a good licensing scheme.

A possible alternative to the VBB definition is an indistinguishability based one; in this variant, an adversary chooses two different circuits and submits both to an oracle implementation of the protocol. Similarly to a cryptographic indistinguishability game, the oracle chooses one of the circuits at random and proceeds to execute the rest of the protocol accordingly. The adversary can then attempt to execute the

protected circuit on any input for which both circuits would obtain the same output (as otherwise it would be trivial to distinguish them). Such a definition provides similar security guarantees to our VBB version – indeed, an indistinguishability game is used in the proof of Theorem 1 to construct a simulator for the black box.

Choosing between the VBB style definition and the indistinguishability one becomes similar to choosing between semantic security [25] and indistinguishability for encryption [26]. We settled on the VBB-style definition due to its simplicity and because it captures the intuition that the adversary does not learn anything except what the functionality of the program allows him to learn. Future work could investigate in more depth the relationship between the two possible definitions, to evaluate their influence on the final security guarantees.

Circuit privacy is weak on its own; however, circuit privacy for licensing schemes is always paired with an additional licensing compliance property. The VBB functionality captures circuit privacy, while a separate, trace-based definition, provides access control to the licensed program. Licensing compliance models the progression of adversarial knowledge throughout license usage, and at the same time enforces access restrictions to impose a restriction on the knowledge gained by the adversary. It is possible to merge the two; however, this leads to a definition that is unnecessarily complex and unintuitive, as it is no longer clear when we are protecting code and when we are protecting functionality.

B. License compliance

From the perspective of software vendors, *licensing compliance* provides a rigorous model of the additional security restrictions that a scheme supports. For example, schemes where the licensed program's functionality can only be unlocked a limited number of times, or where the licensed program can only run on a limited number of machines. The restrictions are expressed using licensing predicates. A scheme satisfies *secure license compliance* with respect to a licensing predicate L if it satisfies L-Compliant licensing, which we define as:

Definition 3 (L-Compliant licensing)**.** Let L be a licensing predicate. Licensing scheme $\mathcal{S} = (\mathsf{User}, \mathsf{Vendor}, u)$ provides L-compliant licensing for circuits in $\mathcal{D}$ if for all efficient adversaries A there exists a negligible function f in λ s.t. the following statement holds:

$$Pr[A^{\mathcal{M}}(P,T) \text{ s.t. } L(C,\tau) = 0] \leq f(\lambda) \tag{4}$$

where the probability is over $C \xleftarrow{\$} \mathcal{D}$, $P, T \leftarrow \mathsf{Vendor}(C, L, 1^{\lambda})$, the internal coin tosses of A, Vendor, and machines in $\mathcal{M}$.

A remark that explains our definition is that it is moot to securely license programs that are known to the adversary. Our definition models the idea that the adversary may not have all of the information about programs by selecting the program at random from some class $\mathcal{D}$ of programs.

For some licensing scenarios, proving compliance with a single licensing predicate L is not sufficient. Examples include licensing restrictions which depend on additional parameters, such as running a specific program at most n times. In these cases, we expand the definitions to a set of licensing predicates, with one specific predicate for each value of n. To simplify notation, we use a single predicate in our definitions and proofs, with the mention that the algorithms we describe can be easily expanded to support classes of predicates.

Depending on their needs, software vendors might be interested in various licensing models. Restrictions are contained within the licensing compliance predicates defined in Section V. We define three predicates, designed to meet common licensing requirements: *licensed use*, *limited use* and *limited machines*.

Licensed use states that, in addition to circuit privacy, no other guarantees are provided by the scheme. The circuit privacy aspect is not included in the predicate; instead, it is included in the VBB definition, which each scheme must satisfy separately.

Definition 4 (Licensed use compliance)**.** Consider the notations and initial setup from Definition 3. Scheme $\mathcal{S}$ satisfies *licensed use compliance* for circuits in $\mathcal{D}$ if it provides L-compliant licensing for circuits in $\mathcal{D}$, with L defined as:

$$\forall \tau \in \mathcal{T} \text{ and } \forall C \in \mathcal{D}\text{: } L(C,\tau) := 1 \tag{5}$$

Limited use guarantees that there is an upper bound on the number of valid circuit input-output pairs that can be obtained; this licensing model can be used when a limited number of executions is desirable, e.g. trial versions of applications.

Definition 5 (Limited use compliance)**.** Consider the notations and initial setup from Definition 3. Scheme $\mathcal{S}$ satisfies *limited use compliance* for circuits in $\mathcal{D}$ if it provides L_n-compliant licensing for circuits in $\mathcal{D}$, for all L_n, $n \in \mathbb{Z}_{\geq 1}$, where L_n is defined as:

$$L_n(C,\tau) := |\mathcal{Y}| \leq n, \quad \mathcal{Y} = \{(i, C(i)) | (i, C(i)) \in \tau\} \tag{6}$$

The predicate states that an adversary attacking a licensing scheme which allows at most n executions is unable to create an input/output transcript τ containing more than n valid elements.

Limited machines guarantees that there is an upper bound on the number of machines that can reveal new information about the functionality of a licensed program.

We define a function $First$ that takes as input a transcript τ and an input/output pair (i, o), and outputs a machine M such that $(i, o) \in \tau[M]$ with timestamp t and there exists no other machine M' where $(i, o) \in \tau[M']$ with a timestamp of lower value than t. In other words, the function returns the first machine that ran a process with trace (i, o). If no machine exists, the function returns ϕ.

Definition 6 (Limited machines compliance)**.** Consider the notations and initial setup from Definition 3. Scheme $\mathcal{S}$

satisfies *limited machines compliance* for circuits in $\mathcal{D}$ if it provides L_n-compliant licensing for circuits in $\mathcal{D}$, for all L_n, $n \in \mathbb{Z}_{\geq 1}$, where L_n is defined as:

$$L_n(C,\tau) := |\mathcal{Y}| \leq n,$$
$$\mathcal{Y} = \{M | \exists (i, C(i)) \in \tau : First(\tau, (i, C(i))) = M\} \quad (7)$$

In machine oriented licensing, once a specific processor is licensed it can be used any number of times to process data. Once the licensed program is executed on an input, its behavior is no longer secret and can be emulated on any number of other machines (e.g. by writing a new program implementing the same function for that exact input). In other words, only the first run is interesting (by revealing new information) and thus can be restricted. *Limited machines* states that at most n machines can be used to reveal this new information.

We define the template for a valid licensing as a scheme $\mathcal{S}$ simultaneously satisfying:

- Preserve functionality (Definition 1)
- Circuit privacy (Definition 2)
- L-compliant licensing (Definition 3), for some class of licensing predicates

VII. Constructions

A. Privacy Preserving Licensing

We introduce a licensing scheme that satisfies functionality preservation (Definition 1), circuit privacy (Definition 2) and licensed use compliance (Definition 4). Such a scheme is of interest to vendors that wish to give their users unlimited access to the software, as long as they purchase a secure use-token first.

Construction. Let $\mathcal{S}_{ppl} = (\mathsf{User}, \mathsf{Vendor}, u)$ be our scheme, where User and Vendor conform to the syntax defined in Section V and u is the utility predicate defined for correctness purposes.

Vendor takes input C and outputs a program P for the user. Let $P = P_1 \| P_2$ be the format of this program. P_1 contains the implementation of various licensing-related tasks, such as communicating with the server or generating keys. P_2 is the encrypted circuit we intend to protect and is subject to execution restrictions. The pseudocode description of P_1 is listed in Fig. 2.

The high level descriptions for User and Vendor can be found in Fig. 3. The scheme operates as follows. First, the software vendor algorithm takes the circuit to protect C, and obtains its ciphertext using symmetric key encryption under some randomly generated secret key K_C. Vendor uses a compiler to obtain a runnable program P, which consists of licensing scheme related code P_1, listed in Fig. 2, and the ciphertext. Afterwards, the vendor algorithm blocks to wait for licensing authorization requests received from users.

User takes input P and the secret use-token T. The user then executes $\mathsf{Load}(P)$ on machine M and starts the execution of P_1. The licensing authorization procedures in P_1 run inside the

Procedure $\mathsf{P}_1\,(P)$
  $i, T \leftarrow \mathsf{Receive}\,()$
  $(K_E^+, K_E^-) \xleftarrow{\mathsf{MRand}} \mathcal{K}^{pke}(1^\lambda)$
  $\mathsf{Send}\,(\mathsf{MSign}[P](T \| K_E^+), T, K_E^+)$
  $\mu \leftarrow \mathsf{Receive}\,()$
  if $\mu \neq \perp$ **then**
    $K_C \leftarrow \mathsf{Dec}^{pke}_{K_E^-}(\mu)$
    $C \leftarrow \mathsf{Dec}^{ske}_{K_C}(P2)$
    return $C(i)$
  else
    return $\perp$

Fig. 2. Pseudocode for licensing-related code within programs protected by scheme $\mathcal{S}_{ppl}$. P_1 is the first code that executes when protected programs start, and is responsible for communicating with the licensing server.

Procedure $\mathsf{User}^M\,(P,\ T,\ i)$
  $\delta \leftarrow M.\mathsf{Load}(P)$
  $\mathsf{Send}\,(M.\mathsf{Run}(\delta, T, i))$
  $o \leftarrow M.\mathsf{Run}(\mathsf{Receive}\,())$
  return o
Procedure $\mathsf{Vendor}\,(C,\ L,\ 1^\lambda)$
  $K_C \xleftarrow{\$} \mathcal{K}^{ske}(1^\lambda)$; $T \xleftarrow{\$} \{0,1\}^\lambda$
  $P_2 \leftarrow \mathsf{Enc}^{ske}_{K_C}(C)$; $P \leftarrow P_1 \| P_2$
  $\mathsf{Send}\,(P,\ T)$
  while 1 **do**
    $K_E^+, \sigma \leftarrow \mathsf{Receive}\,()$
    if $\mathsf{MVerify}[P](T \| K_E^+, \sigma) = 1$ **then**
      $\mathsf{Send}\,(\mathsf{Enc}^{pke}_{K_E^+}(K_C))$
    else
      $\mathsf{Send}\,(\perp)$

Fig. 3. Pseudocode for User and Vendor from scheme $\mathcal{S}_{ppl}$. Vendor generates the protected program and waits for licensing requests, while User acts as an intermediary between the processes and the server. Predicate L is never used because it always evaluates to 1 for *licensed use compliance*

TEE environment built on the machine, using the arguments received on the first call to Run (with P, T and circuit input i). On execution start, P_1 generates a pair of public-key cryptography keys (K_E^+ and K_E^-) using MRand for the necessary randomness. Then, MSign is called to create a secure signature on the newly generated key K_E^+ and the authorization token T. P_1 runs inside a TEE, thus keeping value K_E^- secret. P_1 outputs K_E^+ and the signature, which User takes and forwards to Vendor.

Once Vendor receives a public key K_E^+ together with the signature, it checks whether the signature is valid using MVerify. The verification is possible because MVerify uses only public knowledge internally and Vendor also knows P and T since it created them earlier. If the check is successful, then Vendor uses K_E^+ to encrypt the key K_C it used to create P_2 and sends the result back to User. User forwards the encrypted K_C to the running process, which recovers K_C by

decrypting the received message using K_E^-. The recovered K_C is used to decrypt P_2, thus obtaining the code of C. Execution is then passed to C, together with input i, thus obtaining $C(i)$. Finally, User takes the output of the terminated process and returns it.

For the purpose of functionality preservation, the utility predicate u required by Definition 1 always returns 1. In other words, by executing User with token T, the initial functionality of C can always be recovered.

All functionality and security definitions state that circuits belong to a predefined set $\mathcal{D}$. To prove that schemes satisfy these definitions, we first define the set and afterwards build the actual proof for the circuits within. Let $\mathcal{D}_{ppl}$ be the set of acceptable circuits for $\mathcal{S}_{ppl}$. We define $\mathcal{D}_{ppl}$ as the set of deterministic circuits of size up to k, where k is some polynomial function of scheme security parameters.

We summarize the guarantees provided by the implementation in Theorem 1.

Theorem 1. *Let* $ske = (\mathcal{K}^{ske}, \mathsf{Enc}^{ske}, \mathsf{Dec}^{ske})$ *and* $pke = (\mathcal{K}^{pke}, \mathsf{Enc}^{pke}, \mathsf{Dec}^{pke})$ *be the schemes used to implement* $\mathcal{S}_{ppl}$*. If* ske *and* pke *are* IND-CCA *secure, then scheme* $\mathcal{S}_{ppl}$ *satisfies* functionality preservation*,* circuit privacy *and* licensed use compliance *for circuits in* $\mathcal{D}_{ppl}$*.*

PROOF SKETCH. Since u always evaluates to 1, the proof for *functionality preservation* is trivial, as it simply follows the explanation of scheme operation above.

For *circuit privacy*, we build simulator S and show that for any adversary A, S can simulate the entire scheme given only black box access to C, for any $C \in \mathcal{D}_{ppl}$. In the simulated world, S operates both the machines A has access to and Vendor. The adversary assumes the role of the User, and receives a protected program P from the simulator. We prove that the adversary cannot distinguish between the real world and the simulated world.

The simulator randomly chooses a circuit C^* from $\mathcal{D}_{ppl}$ and creates its own encrypted program P^*, which it supplies to the adversary instead of P. When the adversary reaches the point where C^* is evaluated in i, the simulator queries black box C. The adversary sees the same behavior in the real world and the simulated world. The adversary can only detect that P^* is fake by finding some inconsistency between P^* and the behavior of black box C. We compute a bound on the probability of distinguishing.

We use a second indistinguishability game, set in the simulated world, to show that A cannot find a contradiction for the first one. In this second game, adversary A_2 chooses two possible values for C^*: C_0^* and C_1^*. The simulator chooses to use C_b^* to build P^* depending on some secret preselected random bit b. The adversary must guess the value of bit b. Like in the first game, correctly following the protocol leads to queries to C, so this does not actually leak information about C_0^* and C_1^* to help A_2 decide.

Program P^* operates in a simulated TEE, which keeps its internal state secret from the adversary. The simulated TEE operates in exactly the same way as the real one. The process associated with the program uses the true randomness output by MRand to run the key generation algorithm $\mathcal{K}^{pke}$, thus creating the pair of keys K_E^- and K_E^+. From the security assumptions regarding MSign and MVerify, the signature over T and K_E^+ is accepted by Vendor only if it was created by P^*, thus guaranteeing that the associated private key K_E^- is secret. Based on the secrecy of the pke private key, unforgeability of the signature and IND-CCA property of pke, we deduce that A_2 cannot extract any information about K_C. From this and the IND-CCA property of ske, we obtain that A_2 cannot extract any bit from P^*. Additionally, due to the non-maleability of ske, A_2 cannot force the execution of some $\widetilde{C}$ where there exists some relationship between C^* and $\widetilde{C}$. Therefore, we conclude that A_2 cannot determine b with reliable probability. From this, it immediately follows that A cannot extract any meaningful information about C^* to distinguish between the real world and the simulated world.

The proof for *licensed use compliance* is immediately evident, as the associated compliance predicate is always true.

B. Run Count Licensing

The second licensing scheme we describe satisfies functionality preservation (Definition 1), circuit privacy (Definition 2) and limited use compliance (Definition 5). This type of scheme operates in a similar way to the previous one, except the use-token no longer has unlimited uses. Each time the user executes the software, a licensing server is contacted which tracks the number of times the token has been used. Vendors can use the scheme to provide limited access evaluation versions of the application or pay-per-use products to users.

CONSTRUCTION. Let $\mathcal{S}_{rcl} = (\mathsf{User}, \mathsf{Vendor}, u)$ be the scheme we describe in this section. As opposed to $\mathcal{S}_{ppl}$, in this case the protected circuit can only be executed a limited number of times. The restriction is implemented by changing the behavior of Vendor, with User and P_1 remaining the same as in Fig. 2 and Fig. 3. The new version of Vendor is included in Fig. 4.

We now give an overview of the differences introduced in the $\mathcal{S}_{rcl}$ version of Vendor. Vendor defines an internal counter $\overrightarrow{\gamma_A}[T]$ which tracks the number of times use-token T has been successfully used to perform licensing. When Vendor is executed it takes input L, where L is an object containing parameters required to evaluate the predicates defined by limited use compliance (specifically, use limit n). As opposed to the $\mathcal{S}_{ppl}$ version which on successful token validations always outputs the encryption of K_C, $\mathcal{S}$ checks whether the counter for the token is less than the limit authorizations $L.n$. Depending on the result of the comparison, either the counter is incremented and the encryption of K_C is sent, or a failure message is returned.

Given states st_{User} and st_{Vendor}, the utility predicate for some machine $M \in \mathcal{M}$ is defined as:

Procedure Vendor $(C,\ L,\ 1^\lambda)$
  $K_C \xleftarrow{\$} \mathcal{K}^{ske}(1^\lambda)$; $T \xleftarrow{\$} \{0,1\}^\lambda$
  $P_2 \leftarrow \mathsf{Enc}^{ske}_{K_C}(C)$; $P \leftarrow P_1 \| P_2$
  $\overrightarrow{\gamma_A}[T] \leftarrow 0$
  Send $(P,\ T)$
  while 1 **do**
    $K_E^+, \sigma \leftarrow$ Receive $()$
    if $\mathsf{MVerify}[P](T \| K_E^+, \sigma) = 1$ **then**
      if $\overrightarrow{\gamma_A}[T] < L.n$ **then**
        $\overrightarrow{\gamma_A}[T] \leftarrow \overrightarrow{\gamma_A}[T] + 1$
        Send $(\mathsf{Enc}^{pke}_{K_E^+}(K_C))$
      else
        Send $(\perp)$
    else
      Send $(\perp)$

Fig. 4. Algorithm Vendor for $\mathcal{S}_{rcl}$ limits the number of times a program can be executed. It is similar to the algorithm in $\mathcal{S}_{ppl}$, except it only provides K_C on requests if the token has uses left.

$$u(st_{\mathsf{User}}, st_{\mathsf{Vendor}}, M) := \begin{cases} 1 \text{ if } st_{\mathsf{Vendor}}.\overrightarrow{\gamma_A}[T] < st_{\mathsf{Vendor}}.n \\ 0 \text{ otherwise} \end{cases} \quad (8)$$

Let $\mathcal{D}_{rcl}$ be the set of acceptable circuits for $\mathcal{S}_{rcl}$. While functionality preservation and circuit privacy can be satisfied for any deterministic circuit, limited use compliance requires introducing some restrictions on the set.

Limited use compliance states that an adversary with access to a black box that can be queried a limited number of times can gain no more information about the circuit within, other than the information explicitly output by the black box. In other words, the adversary cannot guess anything about the circuit, which includes guessing unseen output values for new inputs. Building practical licensing implementations on this model guarantees to the vendor that users are not able to reverse engineer their work, and must unlock functionality by requesting access each time.

For certain circuits it is impossible to prove limited use compliance (e.g. for the circuit implementing the identity function). We therefore restrict $\mathcal{D}_{rcl}$ to only a subset of $\mathcal{D}_{ppl}$.

We build $\mathcal{D}_{rcl}$ by looking at the hardness of guessing input and output pairs, which we formalize in Definition 7.

Definition 7 (One-more hardness)**.** Consider the one-more hard game in Fig. 5. Set $\mathcal{F}$ is *one-more hard* (OMH) if for any efficient adversary A, there exists negligible function g in λ s.t.:

$$\mathsf{Pr}[\mathsf{Exp}^{\mathsf{omh}}_{A,\mathcal{F}} \Rightarrow 1] \leq g(\lambda) \quad (9)$$

We let $\mathcal{D}_{rcl}$ be the set of circuits implementing functions from sets that are one-more hard according to Definition 7.

Procedure Compute (i)
  $n_c \leftarrow n_c + 1$
  return $f(i)$
Experiment $\mathsf{Exp}^{\mathsf{omh}}_{A,\mathcal{F}}$
  $f \xleftarrow{\$} \mathcal{F}$
  $\mathcal{O} \leftarrow A^{\mathsf{Compute}()}$
  return $|\mathcal{O}| > n_c \wedge \forall (i,o) \in \mathcal{O}, f(i) = o$

Fig. 5. One-more hard game for $\mathcal{F}$.

The restriction to *one-more hard* functions is necessary to satisfy restricted executions, as we show in the following theorem.

Theorem 2 (Hardness assumption necessity)**.** *Let $\mathcal{F}$ be a set of functions. If there exists an efficient adversary A_1 that breaks the one-more hardness game for $\mathcal{F}$, then there exists an adversary A_2 that breaks **limited use compliance** for circuits implementing functions from $\mathcal{F}$.*

PROOF SKETCH. Let A_1 be the adversary against the one-more hardness property of $\mathcal{F}$. The experiment samples a random function f from $\mathcal{F}$. For any integer n_c, A_1 wins the game by receiving n_c input and output pairs for f, and independently guessing a new pair with non-negligible probability p. Adversary A_2 against *Limited use compliance* license L_n (for some value of n) receives a protected program P, a licensing token T and access to a set of secure machines $\mathcal{M}$. *Limited use* licenses state that A_2 can use the secure machines to create a transcript containing at most n valid input and output pairs. A_2 runs the program legitimately n times, and feeds the input and output to A_1, which eventually outputs a new pair (i, o). A_2 implements a program that takes input i and outputs o, and runs the program on a secure machine, thus creating a transcript that violates the license. The probability that A_2 breaks the license is lower bounded by the probability p of A_1 succeeding. □

The above theorem shows that if $\mathcal{F}$ is not one-more hard, then it is impossible to prove that usage restrictions, built according to our formal definitions, can be enforced. Although the restriction to *one-more hard* functions may seem prohibitive, it still allows for interesting applications. For example the class $\mathcal{F}$ may consist of signing algorithms (with signing keys hardwired), and secure licensing would allow delegating signing rights for some fixed number of times. Security of the digital signature scheme would essentially mean that the class of functions is one-more hard.

We summarize the properties of $\mathcal{S}_{rcl}$ in Theorem 3.

Theorem 3. *Let $ske = (\mathcal{K}^{ske}, \mathsf{Enc}^{ske}, \mathsf{Dec}^{ske})$ and $pke = (\mathcal{K}^{pke}, \mathsf{Enc}^{pke}, \mathsf{Dec}^{pke})$ be the schemes used to implement $\mathcal{S}_{rcl}$. If ske and pke are* IND-CCA *secure, then scheme $\mathcal{S}_{rcl}$ satisfies* functionality preservation*,* circuit privacy *and* limited use compliance *for circuits in $\mathcal{D}_{rcl}$.*

PROOF SKETCH. *Functionality preservation* is easily proven by first looking at the state of the system when $u = 1$, and following the implementation of the scheme to see that C is evaluated correctly. For the reverse direction of the equivalence, we follow the implementation when $u = 0$. In this case, C is no longer evaluated, and the equivalence immediately follows.

The additional behavior coded into Vendor only serves to restrict the number of executions, which has no impact on privacy. Thus, the proof for *circuit privacy* remains unchanged from $\mathcal{S}_{ppl}$.

Let A be our adversary for *limited use compliance*. We prove that for any circuit C randomly selected from $\mathcal{D}_{rcl}$ and for any predicate L_n specified in Definition 5, L_n applied on the transcript created by running the scheme evaluates to 1. We use the simulator from our proof for *circuit privacy*, and first note that each access to the functionality of circuit C is tied to the simulator issuing a query to the black box. The proof for *functionality preservation* states that functionality is preserved a maximum of n times, where n is specified by L_n. Therefore, the simulator needs to perform at most n queries to C, and the transcript contains at most n pairs of input and output for C. The predicate evaluates to 0 if the transcript contains more than n valid input/output pairs, and A must create a new pair without access to the black box. This is exactly the one-more hardness game from Fig. 5. Since the function is randomly sampled from $\mathcal{D}_{rcl}$ which is a union of sets satisfying Definition 7, it follows that the adversary cannot find a pair with sufficient probability. Thus the predicate evaluates to 1 with very high probability and the proof is concluded.

C. Machine Count Licensing

The final licensing scheme we present satisfies functionality preservation (Definition 1), circuit privacy (Definition 2) and limited machines compliance (Definition 6). The scheme is built on top of the previous one, but instead of counting each execution it only counts the first one for each machine. This allows vendors to sell software on a per-processor payment model, with the user purchasing use-tokens in order to activate the application on a limited set of machines.

CONSTRUCTION. Let $\mathcal{S}_{mcl} = (\mathsf{User}, \mathsf{Vendor}, u)$ be a scheme satisfying the above. This scheme only allows the protected circuit to be executed on a limited number of machines.

$\mathcal{S}_{mcl}$ implements this restriction by using an approach similar to $\mathcal{S}_{rcl}$ to limit the number of executions of a reencryption function. This function first decrypts the protected circuit the same way the previous schemes do, but instead of evaluating the circuit on some input, it returns its reencryption under a different key. In an actual implementation, the output is then saved on persistent storage for future use. The reencryption takes place in P_1, outlined in Fig. 6.

Depending on the input (specifically, argument $\widetilde{P}$), P_1 performs either of two operations.

The first operation runs when $\widetilde{P}$ is equal to 0, and consists of the reencryption step. P_1 initially communicates with the licensing server to recover the key to decrypt P_2 and obtain C, and then uses $\mathsf{MKeyGen}$ to generate a secret symmetric key. The generated key is then used to reencrypt C. Finally, the new encryption of the circuit is returned.

Procedure $\mathsf{P}_1\,(P)$
- $K_S \leftarrow \mathsf{MKeyGen}[P]()$
- $T, i, \widetilde{P} \leftarrow \mathsf{Receive}\,()$
- **if** $\widetilde{P} = 0$ **then**
  - $(K_E^+, K_E^-) \xleftarrow{\mathsf{MRand}} \mathcal{K}^{pke}(1^\lambda)$
  - $\mathsf{Send}\,(\mathsf{MSign}[P](T \| K_E^+), T, K_E^+)$
  - $\psi \leftarrow \mathsf{Receive}\,()$
  - **if** $\psi = \perp$ **then**
    - **return** $\perp$
  - $K_C \leftarrow \mathsf{Dec}^{pke}_{K_E^-}(\psi)$
  - $C \leftarrow \mathsf{Dec}^{ske}_{K_C}(P_2)$
  - **return** $\mathsf{Enc}^{ske}_{K_S}(C)$
- **else**
  - $C \leftarrow \mathsf{Dec}^{ske}_{K_S}(\widetilde{P})$
  - **return** $C(i)$

Fig. 6. In the case of $\mathcal{S}_{mcl}$, depending on argument $\widetilde{P}$, P_1 either creates a reencryption of the licensed circuit, usable only on the current machine, or executes the circuit contained in the reencryption on some input i.

The second operation runs when $\widetilde{P}$ is different from 0. In this case, P_1 expects the argument to contain an encrypted program to execute. Because program P is actually the same, $\mathsf{MKeyGen}$ returns the same key it generated during the reencryption operation. This allows P_1 to correctly decrypt $\widetilde{P}$ and obtain circuit C. Finally, the circuit is evaluated in i and the result is returned.

Note that in order to run C, P_1 did not contact the server once it had the correct value for $\widetilde{P}$. Another aspect which differs from previous licensing schemes is that if reencryption took place on a certain machine, then it can only be opened by running P again on that exact machine. Users cannot migrate the encrypted code to another processor, as $\mathsf{MKeyGen}()$ would generate a different decryption key.

The implementation of User and Vendor for $\mathcal{S}_{mcl}$ is described in Fig. 7. Similarly to the previous schemes, Vendor initially builds P containing stub licensing code P_1 and the encrypted circuit P_2, and then blocks while waiting for messages from users. The behavior of User differs slightly. User is stateful, and keeps track of information across multiple calls. Specifically, User tracks a set of encrypted circuits in Λ, indexed using the numerical ID of existing machines. Initially, User has no knowledge of any valid ciphertexts, and Λ contains null values for each machine.

When User is called to run the protected circuit on a specific machine M, it first checks whether it knows the proper ciphertext. If the check fails, User executes program P on M without specifying any extra arguments. In this case, P behaves similarly to $\mathcal{S}_{rcl}$, and performs licensing while communicating with the server. The difference occurs

Procedure User^M $(P,\ T,\ i)$
  if $\Lambda[M] = 0$ **then**
    $\delta_1 \leftarrow M.\mathsf{Load}(P)$
    $\mathsf{Send}\,(M.\mathsf{Run}(\delta_1, T, 0, 0))$
    $\Lambda[M] \leftarrow M.\mathsf{Run}(\mathsf{Receive}\,())$
  $\delta_2 \leftarrow M.\mathsf{Load}(P)$
  $o \leftarrow M.\mathsf{Run}(\delta_2, T, i, \Lambda[M])$
  return o
Procedure $\mathsf{Vendor}\,(C,\ L,\ 1^\lambda)$
  $K_C \xleftarrow{\$} \mathcal{K}^{ske}(1^\lambda)$; $T \xleftarrow{\$} \{0,1\}^\lambda$
  $P_2 \leftarrow \mathsf{Enc}^{ske}_{K_C}(C)$; $P \leftarrow P_1 \| P_2$
  $\overrightarrow{\gamma_A}[T] \leftarrow 0$
  $\mathsf{Send}\,(P,\ T)$
  while 1 **do**
    $K_E^+, \sigma \leftarrow \mathsf{Receive}\,()$
    if $\mathsf{Verify}[P](T \| K_E^+, \sigma) = 1$ **then**
      if $\overrightarrow{\gamma_A}[T] < L.n$ **then**
        $\overrightarrow{\gamma_A}[T] \leftarrow \overrightarrow{\gamma_A}[T] + 1$
        $\mathsf{Send}\,(\mathsf{Enc}^{pke}_{K_E^+}(K_C))$
      else
        $\mathsf{Send}\,(\perp)$
    else
      $\mathsf{Send}\,(\perp)$

Fig. 7. Implementations for the $\mathcal{S}_{mcl}$ version of User and Vendor. User stores Λ containing encryptions of the licensed circuit for specific machines. Once the encryption is known, the circuit can be run any number of times on that machine only.

when returning the final value, where instead of outputting $C(i)$, it outputs the encryption of C under key K_S. User receives this encryption, and stores it in $\Lambda[M]$. If the check for $\Lambda[M]$ succeeds, User immediately proceeds to the final circuit evaluation phase.

Circuit evaluation takes place after $\Lambda[M]$ is known. The user algorithm runs the same program P, providing the actual circuit input i and $\Lambda[M]$ as arguments. In this case, P generates the secret key K_S, uses it to retrieve the circuit by decrypting $\Lambda[M]$ received as argument, and finally evaluates the circuit contained within on input i.

For $\mathcal{S}_{mcl}$, u is defined as:

$$u(st_{\mathsf{User}}, st_{\mathsf{Vendor}}, M) := \begin{cases} 1 \text{ if } st_{\mathsf{Vendor}}.\overrightarrow{\gamma_A}[T] < st_{\mathsf{Vendor}}.n \vee st_{\mathsf{User}}.\Lambda[M] \neq \phi \\ 0 \text{ otherwise} \end{cases} \tag{10}$$

Let $\mathcal{D}_{mcl}$ be the set of circuits accepted by the scheme. The hardness requirements of Definition 7 also apply here. Therefore, we define $\mathcal{D}_{mcl}$ to be equal to $\mathcal{D}_{rcl}$.

Theorem 4 below states the guarantees of $\mathcal{S}_{mcl}$.

Theorem 4. *Let* $ske = (\mathcal{K}^{ske}, \mathsf{Enc}^{ske}, \mathsf{Dec}^{ske})$ *and* $pke = (\mathcal{K}^{pke}, \mathsf{Enc}^{pke}, \mathsf{Dec}^{pke})$ *be the schemes used to implement* $\mathcal{S}_{ppl}$*. If* ske *and* pke *are* IND-CCA *secure, then scheme* $\mathcal{S}_{mcl}$ *satisfies* functionality preservation*,* circuit privacy *and* limited machines compliance *for circuits in* $\mathcal{D}_{mcl}$*.*

PROOF SKETCH. The equivalence relation in *functionality preservation* immediately follows from the implementation, as circuit functionality is recovered when $u = 1$, and when $u = 0$ this is no longer the case.

We prove *circuit privacy* by starting from the simulator proof for $\mathcal{S}_{rcl}$. The simulator again implements the whole world, including machines and Vendor, in a similar way to the previous proofs. The simulator is given access to a black box implementation of C and chooses a circuit C^* at random to forge P^*. We consider an adversary A_2 that submits values C_0^* and C_1^* to the simulator, and needs to decide whether the simulator chose the former or the latter. The proof from the previous sections holds up to the reencryption step in exactly the same way, with A_2 unable to decide based on the information up to that point.

We look at the implementation after, and see that P_1 performs an encryption of C_b^* under a key generated by MKeyGen. From our assumptions regarding machine operation, we know that MKeyGen generates a key that cannot be guessed. Therefore, A_2 has no knowledge about the key, and from the IND-CCA property of ske it follows that A_2 cannot distinguish based on the output of the reencryption. We end the proof by looking at how P_1 operates when receiving the reencryption of C_b^*. In this case, if the simulator decides that A_2 followed the entire algorithm correctly, it queries the black box it has access to and provides the actual functionality of C. If A_2 submitted something else, it simply decrypts and evaluates the result on the input received from the adversary. Since ske provides non-maleability, the output is not related to either C_1^* or C_2^*. In both cases, A_2 does not gain sufficient information to decide, thus implying that the initial adversary is also unable to differentiate between the real world and the simulated world.

For *limited machines compliance*, we base the proof on the compliance proof for $\mathcal{S}_{rcl}$. First, note that the simulator in the *circuit privacy* proof above only queries black box C when the adversary follows the protocol accordingly. From the *functionality preservation* proof for $\mathcal{S}_{mcl}$ we obtain that when following the protocol we only recover functionality while $u = 1$. It can be easily seen by inspecting the implementation and the definition of u that an adversary cannot infringe on *limited machines compliance* by not deviating from the protocol. If the adversary chooses to deviate, the simulator for *circuit privacy* shows that no information about the circuit can be extracted.

The adversary up to this point only has access to outputs obtained while following the protocol correctly. Additionally, similarly to *limited runs compliance* we can deduce that the adversary knows the reencryption of C for at most n machines, where n is defined by the licensing predicate. Let $\mathcal{M}^*$ be this set of machines. First, note that due to the IND-CCA property of ske and the security guarantees of MKeyGen, the adversary cannot derive a valid encryption of C for a different machine

outside $\mathcal{M}^*$. To infringe on *limited machines compliance* as stated in Definition 6, the adversary needs to discover a new input/output pair, without obtaining it by running the circuit on machines in $\mathcal{M}^*$. This is actually the one-more hardness game from Definition 7 which we know the adversary cannot win, thus concluding the proof.

VIII. Extensions

Our formal model is inspired by the current version of SGX, which does not provide secure client side storage. Implementations have existed for some time [3] [4], and others have been proposed that specifically address the case of SGX [6]. It may be possible that secure hardware will eventually be enhanced with additional features.

For the area of secure licensing, the presence of secure storage on the client side could be used to simplify the communication protocol between client and server. For example, communication costs can be reduced by authorizing executions in bulk; the licensing server authorizes a large number of executions in a single communication session, with usage statistics tracked on secure storage. However, this only improves the constructions — which are already intuitive — and does not impact the formal definitions and caveats. Care must still be taken to ensure executing code works as a virtual black box, and the licensed functions must remain one-more hard to prevent unlicensed retrieval of functionality. Our formal model can easily include these extensions by reflecting the additional capabilities of the trusted hardware in the interface exposed to licensing schemes.

Theorem 2 shows that the restriction of licensing scheme to one-more hard functions is necessary, for reasonably realistic licensing models. This restriction is satisfied by some cryptographic software (e.g. signatures or authenticated encryption) and can already serve as a bridge towards licensing the more typical commercial applications[3]. A more direct bridge towards licensing more common commercial applications would require changes that invalidate the hypothesis of that theorem. For example, an interesting extension to our framework is to identify weaker assumptions that model that (much of) the functionality of the licensed program is expected to stay hidden. Conversely, an extension could also consider the possibility that some partial information about the program may be leaked.

One licenseable aspect of programs which is not covered by functionality or code privacy is efficiency. For example, a vendor might design an algorithm that performs some known function faster than its competitors. In this case, the output can be computed by the adversary using publicly known but slower algorithms. Although this is a perfectly legitimate licensing scenario, it is not currently covered by our proposed predicates. A possible extension of the model could include efficiency as a desirable metric, and describe it in a formal manner.

[3]E.g., by only executing this software if accompanied by a signature with a time-stamp.

An aspect not currently covered in our model is the issue of side-channels. At a minimum, the proposed schemes leak information about running time and an upper bound on circuit size. Additionally, depending on architectural features, timing attacks or controlled channel attacks [21] might leak information. Our definition of secure licensing (and our constructions) relies on a model for machines with access to SGX-like secure hardware. While this model does not account for side-channels in SGX, our definitional framework could easily include it in a modular way, by enhancing the attacker's knowledge of the program. Our constructions would then need to be adapted to include defences against relevant leaks.

IX. Conclusion

In this paper we initiate a rigorous study of secure software licensing. We formalize security guarantees for protecting both the confidentiality of the code base and its behavior, define three models that match current licensing use cases, and show that some classes of functions cannot be licensed.

We leverage recent advances in secure hardware platforms to design three practical licensing protocols, answering various software vendor needs such as limiting the number of executions for a particular application, or restricting the number of machines on which it can run. For each scheme, we proved compliance with the security definitions in our models.

Our contribution provides a foundation for future work such as expanding the model to account for other sources of information leakage such as side-channels, or exploring the extensions discussed in Section VIII.

Acknowledgment

We thank Manuel Barbosa, Cédric Fournet, Guillaume Scerri, Martijn Stam, and the anonymous reviewers for the helpful suggestions and discussions regarding this paper. The work was funded in part by the Sectoral Operational Programme Human Resources Development 2007-2013 of the Romanian Ministry of European Funds through the Financial Agreement POSDRU/159/1.5/S/134398. This work was supported in part by European Union Seventh Framework Programme (FP7/2007-2013) grant agreement 609611 (PRACTICE), and ERC Advanced Grant ERC-2010AdG-267188-CRIPTO.

References

[1] Tiago Alves and Don Felton. Trustzone: Integrated hardware and software security. *ARM white paper*, 3(4):18–24, 2004.

[2] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. Innovative technology for cpu based attestation and sealing. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, page 10, 2013.

[3] Dave Levin, John R Douceur, Jacob R Lorch, and Thomas Moscibroda. Trinc: Small trusted hardware for large distributed systems. In *NSDI*, volume 9, pages 1–14, 2009.

[4] Bryan Parno, Jacob R Lorch, John R Douceur, James Mickens, and Jonathan M McCune. Memoir: Practical state continuity for protected modules. In *Security and Privacy (SP), 2011 IEEE Symposium on*, pages 379–394. IEEE, 2011.

[5] Ramakrishna Kotla, Tom Rodeheffer, Indrajit Roy, Patrick Stuedi, and Benjamin Wester. Pasture: Secure offline data access using commodity trusted hardware. In *OSDI*, pages 321–334, 2012.

[6] Raoul Strackx, Bart Jacobs, and Frank Piessens. Ice: A passive, high-speed, state-continuity scheme. In *Proceedings of the 30th Annual Computer Security Applications Conference*, pages 106–115. ACM, 2014.

[7] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. Vc 3: Trustworthy data analytics in the cloud. In *Proceedings of the 36th IEEE Symposium on Security and Privacy, S&P*, volume 15, 2014.

[8] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding applications from an untrusted cloud with haven. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[9] Intel Corporation. Intel 64 and IA-32 architectures software developer's manual, Sep 2015. Reference no. 325462-056US.

[10] Boaz Barak, Oded Goldreich, Rusell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, and Ke Yang. On the (im) possibility of obfuscating programs. In *Advances in cryptologyCRYPTO 2001*, pages 1–18. Springer, 2001.

[11] Secure software licensing: Models, constructions, and proofs (full version). https://goo.gl/29sBVv.

[12] FairPlay. https://developer.apple.com/streaming/fps. [Online; accessed 10-February-2016].

[13] Protected Media Path. https://msdn.microsoft.com/en-us/library/windows/desktop/aa376846%28v=vs.85%29.aspx. [Online; accessed 10-February-2016].

[14] Advanced Access Content System. http://www.aacsla.com/home. [Online; accessed 10-February-2016].

[15] Gregory Robert, David Chase, and Ronald Schaefer. Software licensing management system, June 26 1990. US Patent 4,937,863.

[16] Pradyumna K Misra, Bradley J Graziadio, and Terence R Spies. System and method for software licensing, February 13 2001. US Patent 6,189,146.

[17] Yeu Liu, Ravi Pandya, Lazar Ivanov, Muthukrishnan Paramasivam, Caglar Gunyakti, Dongmei Gui, and Scott WP Hsu. Supplementary trust model for software licensing/commercial digital distribution policy, January 3 2012. US Patent 8,091,142.

[18] Hoeteck Wee. On obfuscating point functions. In *Proceedings of the thirty-seventh annual ACM symposium on Theory of computing*, pages 523–532. ACM, 2005.

[19] Zvika Brakerski and Guy N Rothblum. Obfuscating conjunctions. In *Advances in Cryptology–CRYPTO 2013*, pages 416–434. Springer, 2013.

[20] Matthew Hoekstra, Reshma Lal, Pradeep Pappachan, Vinay Phegade, and Juan Del Cuvillo. Using innovative instructions to create trustworthy software solutions. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, page 11. ACM, 2013.

[21] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. 2015.

[22] Charles Rackoff and Daniel R Simon. Non-interactive zero-knowledge proof of knowledge and chosen ciphertext attack. In *Advances in CryptologyCRYPTO91*, pages 433–444. Springer, 1992.

[23] Manuel Barbosa, Bernardo Portela, Guillaume Scerri, and Bogdan Warinschi. Foundations of hardware-based attested computation and applications to SGX. https://eprint.iacr.org/2016/014.pdf.

[24] Shafi Goldwasser, Silvio Micali, and Ronald L Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal on Computing*, 17(2):281–308, 1988.

[25] Shafi Goldwasser and Silvio Micali. Probabilistic encryption. *Journal of computer and system sciences*, 28(2):270–299, 1984.

[26] Mihir Bellare, Anand Desai, Eron Jokipii, and Phillip Rogaway. A concrete security treatment of symmetric encryption. In *Foundations of Computer Science, 1997. Proceedings., 38th Annual Symposium on*, pages 394–403. IEEE, 1997.