Detecting piracy in standalone and network licensing systems

Master of Science in Technology Thesis
University of Turku
Department of Computing
Cyber security
May 2022
Iisakki Jaakkola

Supervisors:
Seppo Virtanen (University of Turku)
Tahir Mohammad (University of Turku)
Jarno Pato (Cadmatic)

UNIVERSITY OF TURKU
Department of Computing

Iisakki Jaakkola: Detecting piracy in standalone and network licensing systems

Master of Science in Technology Thesis, 67 p.
Cyber security
May 2022

Software has been traditionally sold as executable files which require an additional license to run. Licenses come in many forms but all of them aim at the single goal of preventing unauthorized use, which is also known as software piracy. Piracy can lead to big losses for software companies so it's important to study what can be done about it. Instead of trying to tackle the whole enterprise, I will focus on the first step of preventing piracy, that is, detecting it. This topic will be covered in the context of two traditional software licensing schemes: standalone licensing and network licensing. There are also modern cloud based licensing schemes like Software as a Service (SaaS) that don't need a license in the same sense, but it seems like the traditional models aren't going anywhere soon.

Standalone licensing means that a unique license is required for each installation of the application. Think of product keys on good old installation CDs. In network licensing the application requires a license only when it's actually running. This is achieved with the help of a dedicated license server which distributes licenses to client applications within the local network. I map out all notable attack vectors against both licensing systems, and suggest methods for detecting these attacks. Network verification is discovered to be an indispensable method for detecting piracy. Transport layer security (TLS) protocol will turn out to be the best line of defense against network based attacks. Even the often forgotten feature of client certificate authentication will turn out to be useful. Detection of local attacks, such as tampering and virtual machine duplication, is also covered. Overall this work is intended as a theoretical basis for implementing piracy detection systems in standalone and network licensing environments.

Keywords: software licensing, security, piracy detection, network licensing, standalone licensing

Contents

1 Introduction

In traditional software business, a software company pays programmers to write code, the code is compiled into an executable program, and then copies of this program are sold to customers. The users are forbidden to make copies of this executable and *especially forbidden* to further distribute these copies. For the software is protected by copyright. But this protection only applies in the legal realm, which lies beyond the scope of this thesis. In practice, if there were no countermeasures, it would be trivial to copy and distribute the software at will, regardless of what the law says. In the digital domain, this copyright infringement is commonly known as *piracy.*

It's easy to see that widespread piracy could pose problems to the described business model. Potentially a very large number of sales can be lost. In reality though, the implications of piracy are more complicated than that. There can even be positive effects. To give you a couple of examples, piracy undoubtedly increases the software's userbase. This can be beneficial if the software gains value from a large network of users. You might also prefer the customers pirating your software instead of buying a competing product. It's clear that the question *"What to do about piracy?"* is not an easy one to answer. Multiple strategies exist. The two extremes are preventing piracy altogether with technical measures (or trying anyway) or allowing it completely. A middle strategy could be to refrain from any technical prevention mechanisms, but instead attack selected big pirates with legal

means [1].

Regardless of the chosen strategy, it would be a good idea to *detect piracy*. Most prevention mechanisms rely on detection as the first step, and even in the absence of actual piracy prevention, detection can provide useful statistics. The question *"How to detect piracy?"* is therefore really important for many software companies, and it is also the main subject of this thesis.

A typical piracy detection solution begins by introducing the concept of software licenses. A license is like a digital key without which it should be impossible to use the software product.[1] A software license should be something unique that cannot be duplicated meaningfully. Copying software executables tends to be easy but this doesn't extend to the accompanying software license. This way it doesn't matter so much if the software itself gets copied, since the copies won't have valid licenses.

We are going to have a look how this works in practice. The focus is on two licensing schemes in particular: *standalone licensing* and *network licensing*. Standalone licenses are the traditional solution where a single installation requires a single valid license [2]. Unique activation keys in installation disks are a good example from the old days. Network licensing is a bit more complicated. These systems are used to implement *floating licenses*. In this scheme a license is required only when the application is running. When the license is not being used, it can *float* to some other user within the network [2]. With standalone licenses, the total amount of licenses in an organization equals the amount of installed products, whereas with floating licenses there can be fewer licenses. We are going to delve into the details of these two licensing schemes, and discus how to detect attacks against them. Much of the work will generalize to other settings as well.

This is a theoretical work, meaning that the presented methods are either cited from literary sources or backed up by logical arguments. The goal is to gather a

[1]From here on, the word *license* is going to refer to this technical concept, and not the more abstract legal licenses, unless explicitly stated otherwise.

comprehensive toolbox of piracy detection mechanisms and describe their working principles. This work should prove useful for anyone implementing piracy detection mechanisms in standalone or network licensing environments.

Regarding the level of technical details, a middle ground approach is adopted, meaning that I will not spend too much time explaining specific low-level technical details or code. Instead, the bulk of the text focuses on more abstract technical concepts. This way we can cover way more ground and get a better overall picture of the field and the available detection methods. How exactly the methods are implemented in practice is not important here.

The rest of this document is organized as follows. First in chapter 2 we will have a general look at the current situation in software licensing, and also educate ourselves on some necessary security concepts. In chapter 3 I'll give more detailed definitions for standalone and network licensing systems. Aspects of their technical implementations are also discussed. Here I'll also present some interesting modern alternatives. Once we have acquired a better understanding of how these systems work, we are going to see how they can be attacked in chapter 4. After trying our best to break the systems, we will switch on to the defending side in chapter 5. Here we will go through the attacks once more, but this time focusing on how to detect them. Finally, chapter 6 will contain the concluding remarks.

2 Background

2.1 Piracy statistics

Piracy is a difficult thing to measure because the pirates are mostly trying to stay hidden to avoid sanctions. Because of this, there are no comprehensive statistics about software piracy, but some surveys and studies still exist. Regarding the state of piracy in recent years, The Software Alliance's *Global Software Survey* is probably the best available report [3]. The most recent installation of this report estimates that the global rate of unlicensed software installations was 37% in 2017. That is to say that roughly every third software installed is a pirated copy. This was estimated to correspond to around 46,3 milliard USD worth of software sales, though the meaningfulness of this number is questionable. It's not like that much sales would actually be generated if piracy suddenly became impossible [1]. Nonetheless it's clear that piracy is quite common *on average*. According to the report, there are big regional swings. For example, in Asia-Pacific the piracy percentage is as big as 57%, whereas in North America it's only 16%.

The survey has been carried out a few times already. From 2011 to 2017 there seems to be a very slight decreasing trend in piracy. However, the paper notes that some of this is probably just due to decreases in PC sales. Overall, it seems like piracy isn't going anywhere anytime soon.

2.2 Existing solutions

While many software vendors opt to implement their own licensing solutions from scratch, there are also pre-existing solutions to choose from. On the proprietary side, some of the most popular products include FlexNet [4], LM-X [5] and the Reprise License Manager [6]. These products are likely quite expensive, but from the end-user's point of view, it's probably nicer to deal with somewhat standardized solutions. In the case of network licensing, this could mean that it's enough to install only one license server which deals all the different software licenses in the organization, as opposed to installing a separate server for each software vendor. We should also note that these are comprehensive licensing solutions. Piracy detection is only one of the features.

On the open source side we have most notably the Licensecc C++ licensing library by Open License Manger [7]. The project itself is completely free and it's released under BSD 3-Clause *legal license* [8]. If you're looking for code examples, this would be a good place to start. Some people might raise an eyebrow for using open source code to implement proprietary licensing. The argument being that it will be easier to crack if you get to read the original source code, as opposed to reverse-engineering from machine code. If you are worried about this, keep in mind that the BSD license permits you to modify the code as much as you like. So you could take the project as a baseline solution and then make some obfuscating changes to it.

Here I highlighted a few existing licensing solutions to showcase that the wheel has already been invented. There's no forcing need to start from scratch. However, there might be some advantages to it nonetheless. A self-made solution will surely be different from the other existing solutions. This means that if a vulnerability were to be discovered from FlexNet for example, your products would be unaffected. This *protection by esotericity* works even if your own solution would be objectively less

secure than the standard solutions. Though we should keep in mind that obscurity as such is a dubious source of security [9].

2.3 Properties of a license

We are going to have a more technical look into license implementations in chapter 3, but for now let's spend a moment to ponder about the more abstract requirements. What properties should *a software license* have?

The answer depends on the legal licensing terms set by the software vendor. The task of the technical license is to enforce these terms. Enforce *non-piracy*, so to speak. From this it follows that a license should have at least the following properties:

1. Difficult forge
2. Resistant to double spending
3. Easy to verify

Property 1 means that it should be difficult to fabricate a valid license out of thin air. If this is too easy, then it will be a tempting alternative for legitimately purchasing the product.

Property 2 might sound a bit funny at first. Double spending resistance is a concept more commonly encountered in the world of cryptocurrencies [10]. You can think of it like *"resistance to duplication"*, but as we are later going to see in section 3.1.2, licenses can be copied just the same as any other data. Therefore we should just give up on even trying to prevent copying, and focus our efforts at preventing double spending. That is, making sure the duplicate licenses are invalid.

Property 3 is about practicality. The sold software product should be able to verify the validity of a license with not too much effort. It would not be very useful

to have a license with absolute fool proof security, if it's verification boils down to solving a discrete logarithm problem.[1]

All in all, valid licenses should be easy to verify but hard to forge and duplicate.

2.3.1 Licensing system semantics

Assuming that these three properties are in check, it should be possible to build a licensing system that has the following simple semantics:

$$\text{Valid license} \iff \text{Application is allowed to run.}$$

The application requires a valid license to run, and conversely will refuse to run if the license is invalid. Based on this we can identify two kinds of faulty behaviour in a licensing system. A false negative, where the application won't start (because of the licensing system) even though the user has a valid license. Or a false positive, where the application will start even without a valid license. The latter is the case with piracy.

The point of licensing systems is to prevent false positives, but in doing so they will likely introduce a certain amount of false negatives as well. Before rushing into buying a ready made licensing solution or implementing your own, one should carefully weight the two failure types. An honest customer will likely be plenty annoyed if his application won't work because of a too strict licensing system. Personally I suggest that a vendor should try to reduce false positives only so long as it introduces very little false negatives.

[1] A computationally intensive task. The security of many cryptosystems relies on it's difficulty.

2.3.2 Free and reserved licenses

We say that a license is *free* when it's not in use. Once someone starts using it, it becomes *reserved*. Reserved licenses are no longer free for the taking. To give an example:

> Anna has *reserved* license A for herself. Barry doesn't know this and just tries to use the same license A. Barry's application will refuse to start because the license is already reserved for Anna. He complains to the IT-department, and a tired looking system admin comes down and gives him a new *free* license B. Barry then tries to start his application with license B and succeeds. In the process the license B becomes *reserved*.

This notion of free and reserved licenses will be used throughout the text.

2.3.3 Expiration

Licenses may be valid indefinitely or they might have an expiration date. Indefinitely valid licenses are known as *perpetual licenses.* Expiring licenses become *invalid* after the expiration date. This is a rather simple concept but it's an important one to keep in mind, as some of the attacks will try to extend the validity of an expired license.

2.4 Transport layer security

Transport layer security (TLS) protocol will play a pivotal role in the discussions to come. Many of the detection mechanisms rely on it's authentication component, while several attacks are trying to circumvent it. Therefore it's important to have a basic understanding of it's features. The following introduction should be enough. The exact version of the protocol isn't really important for us, but usually it's good to favor the most recent version, which at the time of writing is 1.3.

TLS is a cryptographic protocol for secure connections over the internet. It's widely adopted. Most notably it's used in HTTPS to secure connections to web servers. TLS provides confidentiality and authenticity. Here confidentiality refers to encryption. When a new TLS connection is formed, the parties will first perform a handshake where a cipher and an encryption key are agreed upon. After this all traffic will be encrypted and the connection can be considered secure. Authenticity means that the identities of the communicating parties can be verified. If this feature is used, the handshake will fail if identities cannot cannot be verified. In HTTPS usually only the web server is authenticated and anyone will be accepted as a client, but the protocol supports client authentication as well. [11], [12]

Identities are verified with *public key certificates*.[2] A certificate is basically just a document that contains information about it's owner. The most important pieces of information are the owner's public key and certificate's signature. If the signature is produced by the owner themselves, then it's known as a *self signed certificate.* This kind of certificate is only as trustworthy as it's owner. In order to a certificate more trustworthy, the owner can ask a widely trusted *certificate authority* to sign it. Before granting their signature, the authority verifies the identity of the certificate applicant. A web user can then trust this certificate just based on the valid signature from a reputable certificate authority, even though he has never even heard of the other person. This *chain of trust* certificate model is depicted in figure 2.1.

More generally, in order to pass authentication in TLS, one needs to have a certificate that the other party trusts together with the private key associated with that certificate. The verifying party is free to choose which self signed root certificates (certificate authorities) he trusts. [11]

[2]It's assumed that the reader has basic understanding of public key cryptography. I.e. public and private keys, and how there can be used for digital signatures.

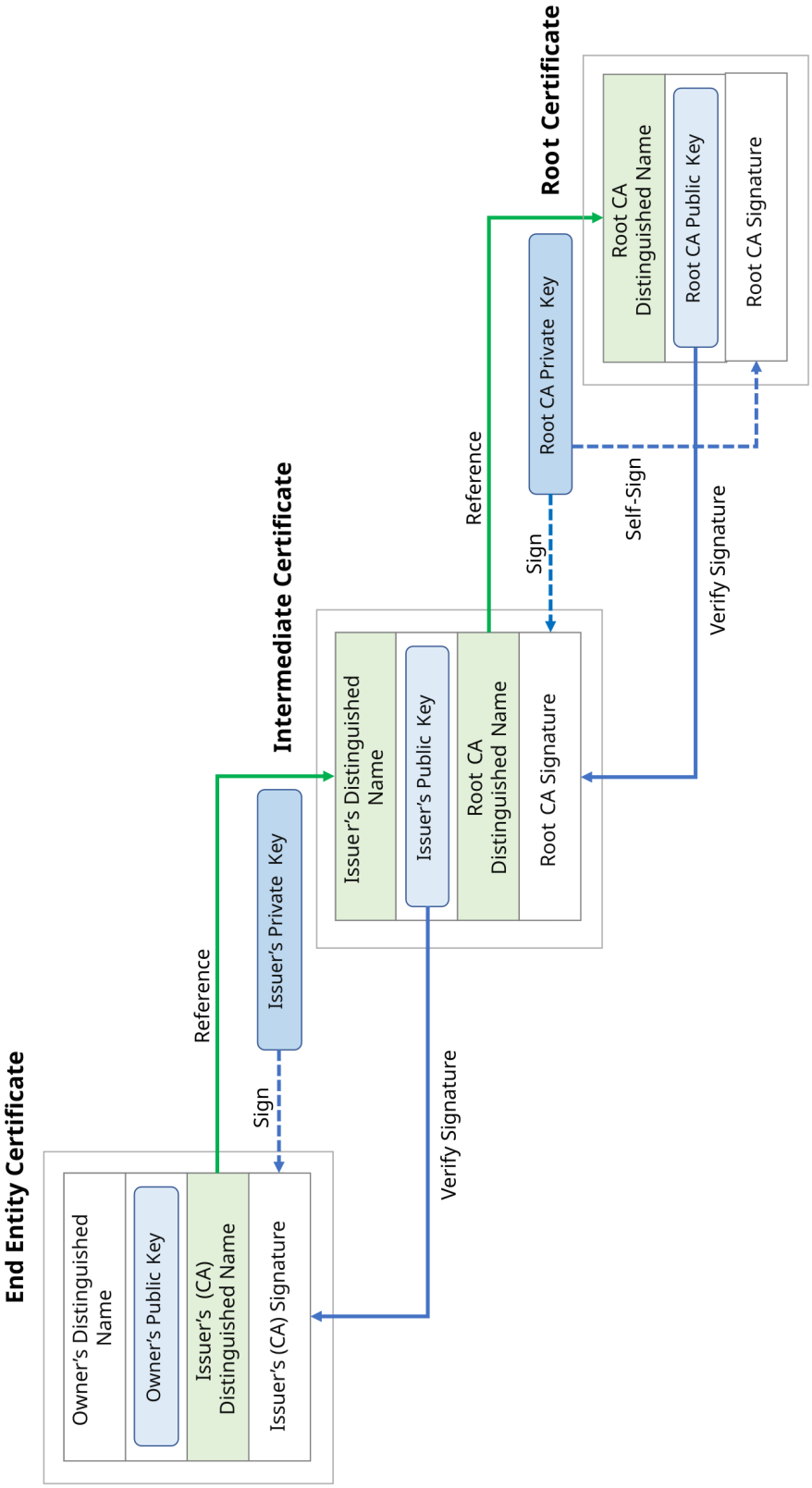

Figure 2.1: Chain Of Trust by Yuhkih, licensed under CC-BY-SA-4.0 [13].

3 Licensing schemes

In this chapter I present the two licensing schemes in more detail. Their motivations and relative merits are discussed. I will also provide speculations on how these licensing schemes are typically implemented, or at the very least, how they can be implemented. We will see that the seemingly different systems have actually quite a lot in common. Finally, we will have a look at some modern developments in the field of software licensing.

3.1 Standalone licenses

Standalone license are probably what most people have in mind when they hear the word *software license*. This is not surprising because standalone licensing is commonly used in consumer software. Products keys on the boxes of installation CDs used to be common back in the days when we still had installation CDs. Standalone licenses are also known as *packaged licenses* (the license comes with the installation package) and *node locked licenses* (the license is tied to a specific network node). [2]

The semantics of standalone licenses are quite simple. One standalone license is needed for one installation of the software. In other words, one license per one computer (node). This corresponds roughly to one one license per one user. Standalone license is considered reserved even when the application is not running. For example, during the night when the computer is powered down. [2]

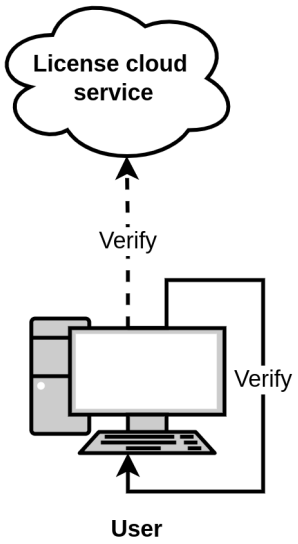

Figure 3.1: Standalone licenses are verified locally on the user's machine. When there's a network connection they could be verified from an online service as well.

3.1.1 Internet connection

Standalone licenses come in two flavors: Those that are verified locally on the user's machine (true standalone) and those that use internet connection. Both operating models are visualized in figure 3.1. The online check could be either primary or additional way of verifying the license. If it's the primary means of verifying the license, then an internet connection is mandatory. This requirement can be problematic. It will render the software useless in non-connected environments. For example, it wouldn't be possible to use the software on an airplane where there's no connection. And then on the business side a lot of work is being done in isolated *secure networks*, where there's no connection to the outside internet for security reasons.

We see that the requirement for internet connection poses a dilemma. Requiring the connection could be used to make the license system more secure (two checks is surely better than one), but on the other hand it will rule out potential customers and can introduce false negative failures. A middle way solution would be to use online checking when a connection is available. This should make the system more

secure in online environments, while still allowing use in offline environments as well. Unfortunately the hybrid model is quite easily circumvented by just blocking the internet connection. I will return to this issue in 4.1.1 and 5.2.1.

3.1.2 Implementation

Here I'll present some commonly used techniques for implementing standalone licenses. There aren't too much sources to cite, as pretty much everyone wants to keep their licensing implementations secret. The Open License Manager [7] is a delightful exception. It implements standalone licensing so I will be using it as an example. I can only speculate about the inner details of proprietary systems, but we should be able to get a decent idea by just looking at the public user interfaces.

Activation key

The familiar activation codes have been used as a software license for a long time. On the surface, the system appears quite simple. When a user purchases the software, she gets a unique activation key. During the installation—or maybe the first time she starts the application—the software asks for the activation key. The application should refuse to start if she doesn't provide a valid key.

This part of the system is not difficult to implement. One could for example just encrypt the software with the activation key. You can be confident that it's practically impossible to run the program without the correct key. While this does prevent license forgery, it does absolutely nothing about copying. A pirate could just buy one valid license and distribute copies of the software along with the key. The copies would work just as well on anyone's machine.

The encryption scheme might still be a good start but it clearly requires additional measures to prevent double spending. There seems to be only one good way to do this. That is, to use an online service to register program activations. In this

system, the application will check from an online service whether this license has already been activated [2]. If it is, the seemingly valid activation key is rejected. This method is depicted in figure 3.1 as the *License cloud service*.

Activation keys can be used together with other methods as well.

Dongle

Software licensing dongles are usually small USB devices that look just like ordinary pen drives, but they are used more like physical keys. The application is tied to a particular dongle and it will refuse to run if the dongle is not connected to the PC [2]. Basically it is just a tamper resistant hardware activation key. The idea is that it's much harder to duplicate the physical device than just plain data. In so far as this is true, dongles indeed solve the double spending problem that was present in plain activation keys. Unlike activation keys, dongles can be secure even without network access.

The obvious downside of dongles is the requirement for a physical device. They aren't free to manufacture and they require proprietary drivers to be installed in operating systems. Also like all physical devices, they eventually break. Customers of the modern age expect to be able to purchase software directly from the internet without needing to wait any physical contraptions to arrive in mail after 3 to 5 working days.

Despite their inconveniences, dongles have been quite popular in the past [9], and they are still used here and there. Sentinel's dongles are probably among the most commonly used [14].

Hardware fingerprint

Hardware fingerprinting attempts to solve the problems with physical dongles. In this system the license is tied to a particular computer with a unique fingerprint of

that computer. When the application starts, it first computes a fingerprint of the device and compares it to the expected fingerprint [2]. If the computed fingerprint differs from the expected one, the application will refuse to run. This implies that the customer will have to inform the vendor of his hardware fingerprint before making the purchase. The vendor will typically provide a free tool for this.

Apart from the minor inconvenience of providing the hardware fingerprint beforehand, this system really appears to solve the problems in the earlier systems. So how is it done in practice?

It's possible for a software to read details about the computer's hardware components. This information can include things like component model names and serial numbers. Depending on the operating system and the execution environment, the amount of information available can vary. Nevertheless, there should be enough to uniquely identify the computer. The gathered information is then turned into a fingerprint. This is likely done by just hashing together some subset of the available data. The resulting fingerprint will appear like a random string of characters, but it is still linked to the data that was used in the generation. Fingerprints generated like this from different computers will differ substantially.

Open License Manager can generate hardware identifiers based on disk identifiers and network addresses [7]. Disk identifiers seem like a natural choice, since it's precisely the disk where the software product will be installed. It makes sense to tie it to the storage medium rather than some other component that has nothing to do with the program. This way there won't be any problems, if for example the graphics card is upgraded into a newer model. In general though, there's no telling what information is used in computing the hardware fingerprint.

Hardware virtualization is also a concern. A virtual machine (or a container for that matter) might not have a stable fingerprint because they aren't inherently tied down to hardware. The same virtual machine could be started on different

physical hardware each time. The open license manager uses MAC addresses in some virtualization environments, but even these can change. Moreover, it might actually be possible to configure the virtual machine in such a way that it *does have a specific stable fingerprint.* Sounds great right? Not so fast. The problem here is that the virtual machine itself can be duplicated at will, which defeats the whole purpose of hardware fingerprints. We'll return to this issue in 4.3.2.

Hardware fingerprinting solves many of the dongle's problems without still requiring an active network connection, but it's clear that it's not a suitable solution for virtualized environments. Then again, neither is dongle.

License file

A license file is not a comprehensive license implementation like the previous techniques were. Rather, it's convenience tool used together with the other methods. The idea is simply to encapsulate the license into a file that's easy to move around. When using license files, the application only asks the user for a license file and nothing else (if the file wasn't already found automatically). This sounds quite convenient and they indeed appear to be widely utilized in the industry.

A simple example of a license file would simply be a text file that contains the activation key. This doesn't improve security but it makes the product activation a more fluent experience for the user.

License files can be used with hardware fingerprints and dongles as well. We simply write the hardware fingerprint or the dongle's ID to the license file, along with any additional licensing information. This additional information can include things like *"What software is this license for?"*, *"What software version is this license for?"* and *"How long is this license valid?"* A single license file can also contain multiple licenses. Typically this means that one of them is the main application and others are add-ons. For example the Open License Manager supports all these features [7].

You might be wondering that can't we just change the content of the license file and acquire infinite licenses this way? Well, yes indeed, if it were just a plaintext file it would be possible to forge licenses this way. Luckily there are measures against this. One way is to obfuscate the file, making it incomprehensible for a human reader. The most obvious way of doing this would be to just encrypt the file. The decryption key of course has be embedded into the software, so that it's able to read the license file. While this certainly makes editing the file difficult, there's actually no need go this far. The same level of security can be reached with a human readable license file as well.

This can be achieved with cryptographic signatures. Before giving the license file to the customer, the software vendor signs the license file with his private key. Anyone can still read the file content's, but any modifications will invalidate the signature. The application will then verify this signature to make sure that the license file has not been tampered with, and will only then trust the contents. For the verification to work, the public key of the vendor has to be embedded in the software. Forging the signature requires similar computational effort as breaking the encryption, and in both cases we had to embed one cryptographic key into the executable. The security between encrypted and signed license files is indeed the same. Users however will likely find the human readable format much more pleasant than encrypted binary nonsense. For example the Open License Manager uses this technique [7].

3.2 Network licenses

Floating network licenses are quite different from ordinary standalone licenses. In this licensing scheme, a dedicated *license server* is installed in the customer's private

network.[1] When a customer then purchases licenses, they are delivered to the license server instead of individual users' machines. From there the license server will deal out the licenses to users within the network. In practice, when a user starts up her software, it will first ask the license server for a license. If there are free licenses left, the server will respond with a permission to start. If all the licenses on the server are reserved, a negative reply is sent. [2]

The license server could be further connected to the internet and make queries a vendor's license cloud service. These queries could be additional verifications much like the ones in standalone licenses but there are other uses as well. For example, the connection could be used to deliver newly purchased licenses to the local license server. Otherwise an administrator will have to somehow update the licenses manually.

The operation of network licenses is summarized in figure 3.2.

3.2.1 Motivation

Just by looking at the figures, it's clear that the network licensing system is more complex than standalone licensing. Unnecessary complexity tends to make systems more prone to errors, so *what is the point of network licensing to begin with?* The motivation comes from big organizations where there may be thousands of software licenses in use. Not everyone is going to require a license for every program in the house, and the users' needs will also vary over time. Furthermore, in a big organization there will be a constant flow of people coming and going. All of this flux makes it near impossible to manage software licenses efficiently. After all the organization would like to avoid purchasing unnecessary licenses. [2]

Network licenses are a solution to this problem. The burden of manually tracking

[1] If the customer is a private user, she can just install the license server on her local computer. From the user's perspective, the setup works just as if a normal standalone license was being used. Though in reality, the system is needlessly complex for this kind of use.

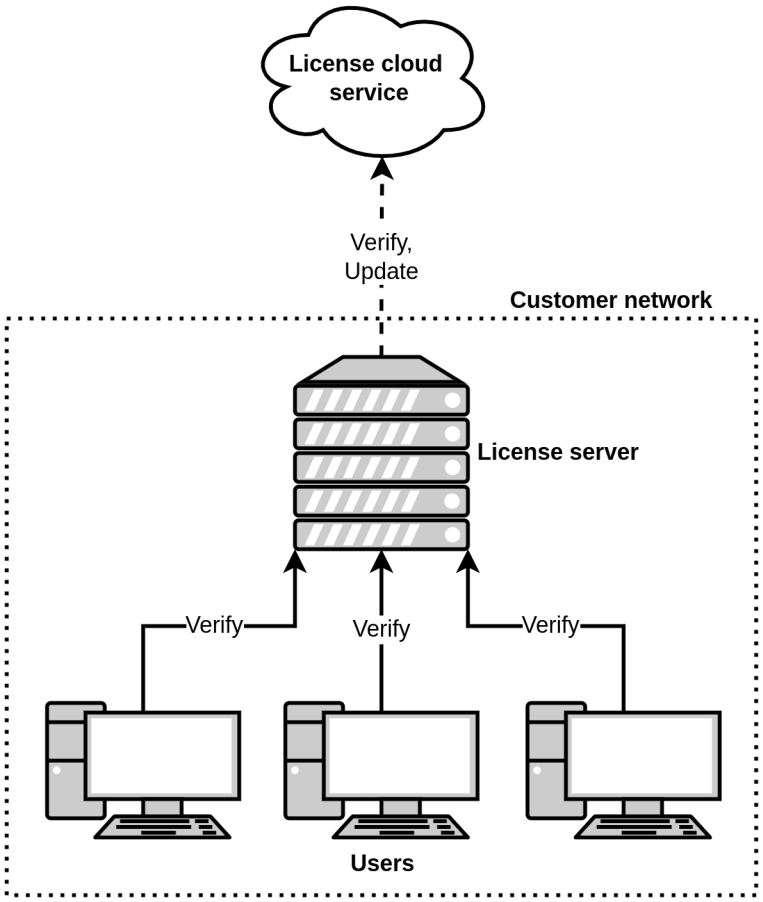

Figure 3.2: Floating network licenses are verified from a license server which resides within the customer's network. If the license server has a connection to the internet, it could further verify the licenses from an online service. The internet connection could have other uses as well, such as updating licenses.

and managing individual licenses is transferred to the license server which performs these tasks automatically [2]. This frees up administrators' time and also guarantees that no unnecessary licenses will be left lying around in end-user's computers. In big organizations the gains are clear enough, but in smaller organizations the effect might not be so positive. The license server will still require some administration and you may not get away with just one. Different vendors' software might require different license servers since there's no commonly accepted standard for software licensing [2].

3.2.2 Implementation

Once again, there's not much technical documentation available about network licensing systems, so this section will be somewhat speculative in nature. Even the Open License Manager doesn't support floating licenses yet so I can no longer use it as an example [7]. But let's not get disheartened, it was never our goal to delve into nitty gritty technical details.

The implementation of network licenses relies heavily on the license server. Of course the applications on users' computers have a role to play too, but here it's a far lesser role than in standalone licenses. The applications don't have to worry about verifying license files or hardware fingerprints, they are merely clients in a more or less standard client-server architecture.

While we don't know exactly what packages get transmitted between these two parties, it seems obvious that in some part of the communication the client application has to send a `license request`, and the server has to send a `response`, which contains a boolean permission for starting the application. Prior to this, it's likely that some packages are exchanged in order to authenticate the parties and establish a secure encrypted channel. Standard TLS would work just fine. Other information might be transmitted as well but I don't really have a way to know what that

might be. Regardless, it seems safe to assume that this additional information is not crucial from a security standpoint.

License servers can be divided into two groups based on the frequency of communication between the server and the client software. This interval defines the maximum duration for how long the server will lease it's licenses to clients. Long leases are known as *license borrowing* and I will dub the shorter periods *continuously verified network licenses.*

Continuously verified network licenses

In this model the client application keeps verifying it's license from the server with short regular time intervals [2]. In essence, the server and the client maintain communication for as long as the application is running. When the application stops, the server can free the license just based on the observation that the client went silent. Similarly, should the license server suddenly go dark, the client would no longer receive responses to it's license verifications and it could easily deduce that something is amiss, and proceed to shut down.

This model clearly requires a stable connection between the license server and the client. This shouldn't surprise us since we are after all talking about *network licenses.* Still, there are downsides as well. In a big organization there could be thousands of clients in constant communication with the license server. Especially in the past when network bandwidth was scarce, the license server traffic could cause congestions in the local area network [2]. I don't think that this is really much of a problem anymore. Continuous verification model seems like a natural choice today.

However there is one problem with this model stat still persists. It's not possible to use the software when roaming outside the organization's network. This problem can be solved by extending the network with a VPN *(Virtual Private Network)*, but this is a bit dubious from the software vendor's perspective. It would allow

sharing licenses to practically anywhere over the internet. I expect that at least some vendors prohibit this kind of use. And even if it was allowed, it still requires a good enough internet connection, which still cannot be taken for granted.

License borrowing

The networking issues of continuously verified licenses were fixed by introducing *license borrowing*. In this variant, the client and the license server do not need to be in constant connection. The connection can be closed after the client application receives a positive response to it's license request. The client can now unplug from all networks and travel freely, while continuing to use the application with the *borrowed license*. The server will keep that license reserved until the client connects again and frees the license, or a borrowing timeout is reached. Likewise, the client application will continue to function until the borrowed license expires.

This is a more complex (and therefore more error prone) system. Both the client and the server will have to keep track of the borrowed license, possibly for a considerable duration. This state will also have to persist through reboots. This means that it's not enough to keep it in memory, but it will have to stored on disk. From the client's perspective, this starts to sound awfully reminiscent of standalone licensing. Likely the borrowed license will be stored as a license file similar to the ones discussed in 3.1.2.

A new security concern presents itself here. In the case of standalone licensing, the license file was issued (and signed/encrypted) by the software vendor, but now in the case of borrowed licenses, the file is issued by the on-premise license server. This is a problem because now the key that is used to sign/encrypt the borrowed license file has to be stored in the license server, which is physically accessible for potential attackers. The key could be uncovered from the license server's executable and used for license forgery.

The situation could be improved if the on-premise license server is further connected to the vendor's cloud service (figure 3.2). Then it would be possible to have the license cloud service sign/encrypt the borrowed license file, before sending it to the client application. This way the key will stay securely in the vendor's premises. This scheme would bring the security back on par with usual standalone licensing, if we disregard the added complexity.

An analogous problem arises in the license server's end as well. It is difficult to keep track of borrowed licenses in a secure manner. For example, say that the server saves *"a negative license file"* on disk to account for the borrowed license. Then the borrowed license can be released back to the pool by simply removing this file, effectively allowing easy double spending. Fixing this issue convincingly will likely require communication with the vendor's cloud service as well.

It seems clear that the borrowing scheme is quite a bit more complex than it's continuous alternative. Both the client and the server will require extra features. Especially the client will need to be promoted from a simple network client to a full blown standalone license implementation. Nonetheless, when implemented with care, license borrowing does indeed solve the network issues that were nagging the continuously verified network licenses. Essentially, license borrowing is a hybrid model between standalone licenses and continuously verified network licenses. *"Does it combine the best or the worst of both worlds?"*, is another question entirely. Personally, I would avoid going down this path if at all possible. The rest of the thesis will focus on continuously verified network licenses.

The license server itself

Let's have a brief look at the implementation of the server itself. In so far as it's a server that serves clients over network, there probably isn't anything special about it. It will contain a pool of licenses which it will then distribute to clients when

requests arrive. But where does this pool of licenses come from? It sounds absurd that the licenses would be hard coded into the executable, which leaves us with two reasonable options. The first one is they come from *the internet*, that is, the server obtains the list of licenses directly from the vendor's cloud service each time it's booted. Some sort of authentication is required of course. The second possibility is the already familiar *license file*. Remember when I told you in section 3.1.2 that a single license file could contain multiple licenses? Well this would be the perfect use case for this feature.

In fact, everything I said about the activation of standalone licensed applications in section 3.1.2 applies to the activation of the license server as well. The server's licenses are likely tied down to a particular server machine, just like standalone installations are tied down to the user's machine. If it were not so, it would be possible to have multiple license servers distributing the same set of licenses, which would be double spending. The possible implementation mechanics are the same: dongles, hardware IDs and cloud services.

So in the end, network licensing turned out to have a lot of things in common with standalone licensing. A hint of this symmetry can be grasped by comparing the respective figures 3.1 and 3.2. Starting from the cloud services at the top, the two graphs are actually quite similar. In network licensing the added intermediary license server has just assumed the license verification duties that used to belong to the standalone licensed application.

3.3 Modern models & technologies

If you find yourself thinking *"But what about the modern models and technologies?"*, this section is for you. If you're just interested in the traditional schemes I already covered, you're welcome to skip this part.

3.3.1 Software as a service

One prominent competing model is the so called *Software as a Service (SaaS)*. In SaaS, there's usually no executable at all on the user's computer. Instead the whole application is hosted in the cloud, and it's accessed with a normal web browser. Alternatively there might be a thin client software installed locally, which is used to access the cloud resources. Typically SaaS products are sold on subscription basis, that is, you pay a monthly fee and in exchange you get to use the most recent version of the cloud hosted software, until you eventually stop paying. [15]

We can estimate the popularity of SaaS by looking at the Golden research engine data. The research engine finds about 24 500 companies in software industry, and about 21 500 companies in SaaS industry [16]. From this we can estimate that roughly 88% of software companies are using SaaS. Though I should hasten to add that this doesn't mean that the same companies wouldn't be involved in traditional software business as well. Regardless, it seems clear that SaaS is quite popular.

The upcoming discussions about security in standalone and network licensing models bears very little significance for SaaS. This is simply because the model is so profoundly different. There isn't much for pirates to target their attacks locally. There is of course the cloud service itself, but that is a question of overall web security and not so much about licensing technology anymore. Web security is an important topic but it's better covered elsewhere. In this work I just assume the security of cloud services.

So the upcoming discussion is not going to be relevant for SaaS, but we should keep in mind that there are other ways of implementing subscription based business models too. SaaS is just one of many. The traditional standalone and network licensing schemes are well equipped for implementing subscription models as well. One way of doing this would be to issue a license file with one month expiration time, and sending a new one whenever a monthly payment is received. If the system

is connected to the internet, this license delivery could easily be automated. The user doesn't need to be aware of such technicalities.

In the end, SaaS is really just another technical licensing scheme that is capable of implementing different business strategies, just like standalone and network licensing. The concepts *licensing scheme* and *business model* are largely orthogonal. It might be that new software products tend to favor SaaS, but I doubt that the traditional schemes are going to disappear anytime soon. Plus, there's always the special case of disconnected secure networks, where SaaS can never work.

3.3.2 Blockchains

You must have heard about blockchains, right? Well in case you haven't, they're a trendy way of storing data in blocks which are linked together with cryptographic hashes [10]. When such blockchains are distributed publicly in the internet, it's possible to use them to implement immutable public ledgers. Cryptocurrencies, such as Bitcoin and Ethereum, are common examples of this. Such immutability sounds tempting for licensing use as well. After all, we don't want that our licenses can be tampered with.

It would certainly be possible to store licensing data in a blockchain, but it's also easy to go wrong. The most obvious blunder would be to set up your own blockchain and making yourself the king of the system. A blockchain with centralized authorship is only ever as trustworthy as the author, which means that there are no security gains compared to traditional cryptographic signatures. This would just add unnecessary complexity. The reason why cryptocurrencies manage to establish trustworthy immutability, is their distributed architecture, and the consensus mechanisms built upon it, such as *proof of work* and *proof of stake.* [17]

To actually gain security in licensing from a blockchain, you would need to relinquish authorship and embrace democracy. One way this could play out is that

software companies would decide to set up a common distributed blockchain where everyone distributes licenses for their software. This way the customer wouldn't have to place all her trust in any particular software vendor, but instead trust whole network of vendors, along with the system itself. In a way, vendors would be keeping each others in check. Maybe it would even be possible to incentivize software users into maintaining the blockchain network by giving them free products for their troubles.[2]

An easier way to leverage a functional distributed *trustworthy* blockchain would be to just tag along to one of the well established cryptocurrencies. Many of them support arbitrary tokens and the so called *smart contracts* which can be used to implement your own systems on the blockchain. Ethereum is probably the best example of this [18].

Then how exactly would this blockchain be used in licensing? In practice the blockchain would take the role of the cloud service. After all, it is really just a data storage that can be accessed with an internet connection. A simple way of utilizing this immutable online storage for licensing would be to just publish to the blockchain the data that's normally contained in the license file. The client software can then search the blockchain for blocks that contain licenses addressed for that particular client. There would need to be some sort of identification for this. Once the client finds a suitable license, it can mark the license reserved, so that other's won't be able to use it. There should be no risk of this data being tampered with, given it's all recorded in a public blockchain.

There are problems with this setup though. First off, there's very little privacy since the blockchain is public.[3] Secondly, this really doesn't make the pirate's life

[2]Public blockchain networks are all about incentives. Why would an individual waste computer resources on maintaining a blockchain if he's not getting anything in return? In cryptocurrencies this typically means that they're getting a little bit of the currency itself.

[3]There are privacy oriented blockchains, such as the Monero cryptocurrency [19], but I'm not convinced that they are suitable for implementing the features that we need.

any harder. In the next chapter I will be covering attacks against the two traditional licensing schemes. Any attacks targeted at the license cloud service could be targeted against the blockchain network as well, namely fake cloud service and man in the middle attacks. This means that in the end the blockchain solution didn't really give us any additional security, even when implemented properly. The one thing that it did give, is that the customer no longer has to place so much trust in any particular software vendor. Once the license is sent to the public blockchain, she can be sure that it won't be modified.

Given these considerations, it seems unlikely that software vendors should implement this approach. It's in their best interest to keep power over their own products.

4 Attacks against licensing systems

Here I will list and describe possible attack types against licensing systems. Later on in chapter 5 I will be looking at how to go about detecting these attacks. Remember that this list is *not exhaustive*. When it comes to security, one should never assume that you know what your opponent is going to do. Novelties are to be expected.

Here the attacks are neatly divided into separate categories, but we should keep in mind that this isn't very realistic. In reality the attacks could be performed in tandem in almost any combination imaginable. The attacker will use all tools in her disposal as she sees fit, including the ones we don't know about.

This chapter is divided into three sections. Section 4.1 covers attacks against standalone licenses, section 4.2 against network licenses and finally section 4.3 covers attacks that are not restricted to either of scheme in particular.

4.1 Attacks against standalone licenses

Figure 4.1 sums up the main types of attacks against standalone licenses. I will cover them in more detail in the coming subsections. For now we can observe that if the licensing system doesn't use internet connection at all, there actually isn't anything left apart from local cracks. They are the only attack type that will always be possible which makes them an important class of attacks. I will cover them later in the general attack section 4.3.1, as they affect network licenses as well. This section will mostly center around the network based attacks.

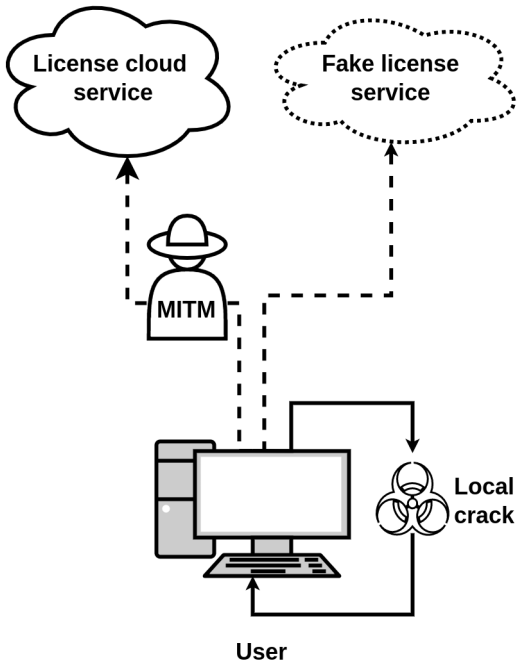

Figure 4.1: Attack vectors against standalone licenses

4.1.1 Cutting the internet connection

A simple way to subvert standalone license checks that rely on internet connection, is to just cut the connection. This could be done just by unplugging the internet cable, or less disruptively by editing firewall rules to block only the particular software. It might even be possible to block only those packages that are used to verify the software license, while letting the other traffic pass.

The effectiveness of this attack depends on the vendors policy on internet connection (see 3.1.1). If the software unconditionally requires an active connection, it won't do any good to cut it. However, if the software works offline too, then disabling the connection will disable the online license verifications too. This essentially undoes the online checks completely. One would still need to bypass offline license verifications somehow, so this attack will not be sufficient on it's own.

The more advanced versions of this attack involving firewall rules might work even if the application seemingly requires internet connectivity. It might be that

a few missing network packages aren't enough ring alarm bells in the software. Speaking of which, the detection of blocked internet connections will be discussed in more detail in section 5.2.1.

4.1.2 Fake license cloud service

If the licensing implementation relies on a vendor's cloud service, one attack you could try is to create a fake license service of your own. This fake service would then answer positively to any license requests. It would probably take a decent amount of protocol reverse engineering to figure out how exactly the the license service is supposed to answer to requests and the task is made even harder by the fact the traffic is most likely encrypted. Then again you don't have to accurately reverse engineer the whole service, only the part that verifies the license against double spending. A working fake service is likely orders of magnitudes simpler than the real thing.

It's likely that the encryption is the biggest obstacle. There are plenty of protocol reverse engineering tools available, but they will be useful only after the communication has been decrypted [20]. So how feasible the decryption really is? We can say with confidence that *it's always possible*. This follows from the fact that the certificates and keys used to encrypt the traffic are always stored on the local computer. In principle, it's possible to read them from memory or storage. In practice though, this task is far from trivial. Nevertheless, once you have uncovered the keys, the traffic can be decrypted, and the rest of the work should prove relatively easy.

I'd like to draw your attention to two aspects of this attack. First, I never specified the encryption scheme used. Most likely it's standard TLS, but the same principle will work against any encryption scheme that's based on keys. Secondly, we never had to resort to brute force or directly crack the cryptosystem. This is really the core idea that makes this attack possible to begin with. Extracting the

keys might be very hard, but I doubt that it's even nearly as hard as cracking the cryptosystem head on. Man in the middle approaches can also be used to decrypt the traffic [21], but more on that in 4.1.3.

Once you manage to reverse engineer the protocol and implement your own fake version of it, you need to redirect the traffic intended for the real license service to the fake service. There's nothing special special about this step. Standard networking tools will do. However, there might be still one more bridge to cross. The application might not trust your fake license service. Assuming that TLS is used for encryption and authentication, this would mean that the real license service has an identifying certificate, which your fake service does not [12]. The authentication step of the connection fails.

Assuming that online security is sound in the vendor's official license service, stealing the real certificate can be considered infeasible. Physical access would be required. But stealing the official certificate isn't the only way. It could also be possible to fool the local application to trust a fake certificate. The application has to have the public part of the official certificate stored somewhere. There are basically two options for this. First, the vendor's certificate is signed by a public certificate authority, and the root certificate of this authority is stored somewhere in the operating system. This means that it's the *operating system* that trusts the real license service (and distrusts your fake). This case should be fairly easy to circumvent. Just generate a new self signed certificate and add it to the the operating system's trusted certificates. After this, the application should trust your fake license service without problems.

The other possibility is that the vendor's certificate is embedded into the application executable. This variation is a bit more resilient. You will need to find the binary certificate and replace it with your own self signed certificate. Once done, the application should once again trust the fake license service without any hiccups.

Additionally in this version it actually ceases to trust the real thing, which lowers the chances that it would successfully *"phone home"* about the incident. This last part was an example of *tampering*, which will be covered in more detail in 4.3.1.

The detection of fake license cloud services themselves will be covered in 5.2.2.

4.1.3 Man in the middle attack on license cloud service

Instead of completely replacing the vendor's license service with a fake, it would be possible to conduct a *man in the middle attack (MITM)*. This essentially means that you transparently monitor the traffic between the application and the license service, and only make selective changes. Namely here the interesting change would be to change a negative license response into a positive one, making the application think that it has a valid license.

This attack will also require decrypting the traffic, but man in the middle is a special case in this regard. The attacker can sneak into the connection at it's very beginning. This shouldn't be a challenge as she can simply restart the application to reset the connection. She will then perform valid handshakes (TLS or other) with both parties of the connection, and continues to forward traffic between them, like it was a direct connection. Since she has performed valid handshakes with both parties, she also has valid decryption keys for both connections, and can therefore decrypt the passing traffic with ease. It should then be possible to reverse engineer the license verification protocol, and intervene at the critical moment. [22]

In reality it likely isn't quite that simple. We run into the same problem as last time: *certificates*. The application can detect the man in the middle attacker because she doesn't have the license service's certificate. The cure for this problem is the same as last time. We need to make the application trust our self signed certificate, which might require tampering.

In a man in the middle attack there's an additional certificate problem as well.

The license service might also authenticate the client application with a certificate. If so, then the server won't trust the packages forwarded from the man in the middle attack because the attacker doesn't have the client application's certificate. Luckily for us attackers, we know that the certificate—along with it's private key—has to be embedded somewhere in the application executable. We just have to extract it and install it into our machine that's conducting the MITM attack. Again, I'm not saying that it's easy, but it should indeed be possible.

After the certificate obstacles have been solved, the attacker gets to observe and even modify the communication undetected. From here it shouldn't be too hard to fool the application into double spending a license.

I will cover the detection of this man in the middle attack in section 5.2.3.

4.2 Attacks against network licenses

Attack types against network licensing systems are illustrated in figure 4.2. Compared to the corresponding graph about standalone licenses (figure 4.1), there are quite a lot of possible attacks. This just follows from the fact that network licensing systems are more complicated to begin with. There are more elements and therefore also more possible points of failure.

I will cover the different attacks in the coming subsections, except for the two local cracks, which will be covered in 4.3.1.

4.2.1 Attacks against the vendor's license cloud service

The license server is installed in organization premises, but it could still communicate with an outside license cloud service. If this is the case, then the fake license service (section 4.1.2) and man in the middle attacks (section 4.1.3) can be carried out on this connection just as in the case of standalone licensing.

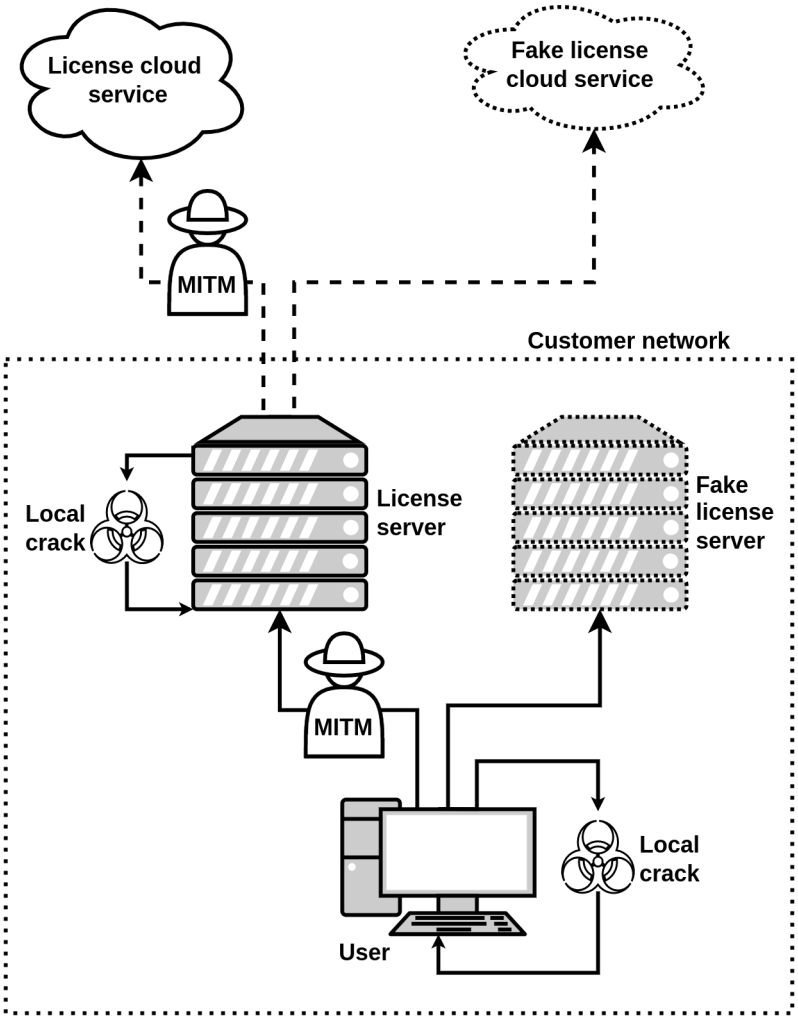

Figure 4.2: Attack vectors against network licenses

Additionally, one might try to just block the connection to the vendor's license cloud service altogether. The license server could function just fine without the connection. After all, what's the point of implementing an on-premise license server if it will anyway verify all licenses from an online service? It would seem like the license server is simply an unnecessary point of failure in this case. The floating license semantics could be implemented in the vendor's cloud as well.

One thing to note is that while these attacks may be carried out in the same manner as with standalone licenses, the results might differ. The attacks might not actually get us any closer to license double spending or forgery. This is because the connection between the on-premise license server and the vendor's cloud service might not be used for verifying license requests like in standalone licensing. It could be used just for delivering new license files for example. Intercepting these isn't really useful from a pirate's perspective. The license file is likely cryptographically protected, so it cannot be easily tampered with in this stage.

The detection of these attacks will be covered in 5.3.1.

4.2.2 Fake license server

On to the new component, the on-premise license server. An analogous fake server replacement attack is possible here too. This means implementing your own unofficial license server which will grant any license requests from the client applications. The attack shares a lot in common with the fake license cloud service attacks we've been discussing. You need to reverse engineer the protocol used between the client applications and the license server, at least to a certain degree. Once again, encryption and authentication are likely to form an obstacle, but only to a lesser degree this time. This is because, unlike the cloud service, the license server is within physical reach of the attacker. He might be able to extract keys or tamper with the certificates embedded into the license server's executable. In the cloud service's

case this wasn't possible, and the only remaining option was tampering in the client application's end. Of course this is still an option, but it's the only one anymore. For example, now it's a possibility to extract the license server's valid certificate and simply use that one in the fake server. From the client's perspective, it would look like a trustworthy license server. No tampering was necessarily involved either.

In this setting where both ends of the communication are closely at hand, the attacker really has a plethora of options at his disposal. It highlights the greater security offered by license services in could, far outside the attacker's reach.

I'll discuss the detection of fake license servers in section 5.3.2.

4.2.3 Man in the middle attack against a license server

Network licensing offers the attacker another opportunity to attempt a man in the middle attack. This attack is performed between the license server and the client application. The idea is to just passively forward the traffic except for the server's response to a license request. This response should be changed to a positive response. It would once again make the client application believe that it has a valid permission to run.

Here too the attack is made easier by the fact that the server is readily accessible. The certificate authentication methods usually used to prevent man in the middle attacks, are not so effective here. In the case of a man in the middle attack against the license cloud service, we had to resort self signed certificates and tampering with the client application. While this is still very much a possibility, another opportunity has presented itself. Just like in the fake license server attack, it's theoretically possible to *"steal"* the valid certificate from the server's executable. This certificate can be then used in the machine conducting the man in the middle attack to perfectly impersonate the license server. It might also be necessary to steal a corresponding certificate from the client application, if the server authenticates it's

clients. Either way, the end result is that you get to monitor the decrypted traffic between the license server and the client application undetected. Then just pick a right moment to intervene, and enjoy your free license.

Of course in practice it likely won't be quite this fun and easy, but with determination, it should be possible, and it's definitely easier than attacks against cloud services. Let's also note that while I've been talking of certificates and implicitly referring to standard TLS encryption and authentication, the attacks don't rely on this. A similar attack is possible in more obscure crypto systems as well, and it's anyway easy to blow the whole security if you're writing your own crypto. I'd advice to sticking with standards, even after seeing that they can be cracked in these white box scenarios.

The detection of man in the middle attacks between client applications and the license server is discussed in 5.3.3.

4.3 General attacks

In this section I'll go through attacks that are not restricted to either licensing scheme in particular.

4.3.1 Tampering

Tampering means directly modifying the application's executable file. In figures 4.1 and 4.2 it refers to the *local cracks*. We've seen that it can happen in both the user's computer and in the license server. It's assumed that the clouds are under the vendor's control and therefore cannot be tampered from the outside. That's why there aren't any crack-marks in the clouds.

The executable—whether the application itself or the license server—is in the end just a a file of machine instructions and some data. Actually this isn't quite accurate

as some languages are interpreted as is (e.g. Python) or compiled to intermediary byte code (e.g. Java), but even those "executables" contain the program definition, just in a bit more abstract format. And most importantly for our present purposes, they can be tampered with just the same. The key difference between regular programming and tampering is that in programming the original source code and documentation are available, whereas in tampering this usually isn't the case. This makes tampering generally significantly harder than programming.

To leverage tampering for piracy, the pirate wants to modify the program in such a way that it works normally even without a valid license. In practice this means finding the program location where licenses are checked, and changing them to something that doesn't check the license. In the easiest case there is only one such place in the program, but it could also be way more complicated than that. This kind of tampering is also known as *cracking*.

To give you another example, in section 3.1.2 I told you that when using license files, a cryptographic key needs to embedded in the software. Otherwise the signature in the license file could not be verified. You could try to look for this key in the executable and once you find it, you can try to replace it with a key of your own. With the key under your control, it would be possible to just write your own license file, and application would accept it as the signature seems sound. To make matters worse, it's often easy to find cryptographic keys amid other data, because high quality keys tend to have far greater entropy than any other type of data [23]. Figure 4.3 illustrates this quite clearly.

In the case of a license server, the pirate would probably want to tamper it in such a way that it will respond positively to all license requests. This way a perfectly benign user application would get a "valid" license from the server, even thought there might be no free licenses actually left. One cracked license server could potentially allow an unlimited amount of software products to be used without

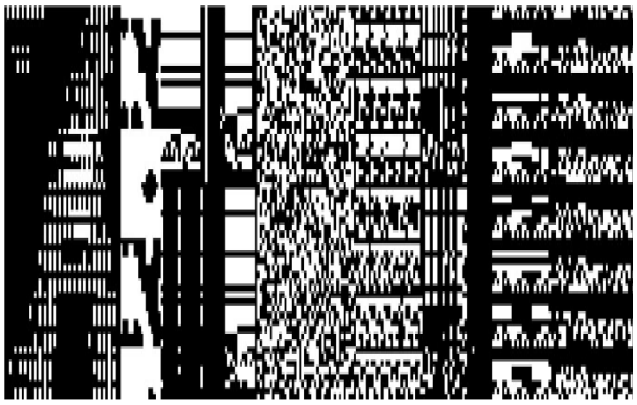

Figure 4.3: A memory dump from a cryptographic algorithm. Zeros are black. It's easy to visually distinguish the high entropy band in the middle, which contains a cryptographic key. Other parts have clear regularities. [23]

licenses. This sounds bad, but we should remember that any cracked software can be copied endlessly so a cracked server isn't actually quite as dire as it initially seems. There is one real advantage to cracking license servers though. The same server potentially serves licenses for multiple different client programs, so cracking the server saves the trouble of cracking each of the client programs individually. So, tampering is more serious in the case of license servers but the order of magnitude remains the same.

I will return to tampering related subjects in the next chapter. Tamper-protection is reviewed briefly in section 5.1, and tamper detection methods are discussed in 5.4.2.

Reverse engineering

Tampering is closely related to *software reverse engineering*, which refers to the act of uncovering program logic without access to the source code [9]. It's easy to see how this could come in handy when trying to crack executables (which in this context are almost always closed source). Understanding the program is almost mandatory for making useful changes in the program.

We won't be looking at specific *static* reverse engineering or tampering methods,

but suffice to say that there are rather advanced tools available [24]. We are mainly interested in detecting the tampering attempts, and for this it really doesn't matter how the adversary came to make the specific modifications. Detecting them will anyway be impossible when the application is not running.

Debugging

It's also possible to do reverse engineering dynamically when the program is running. This is commonly known as *debugging*. There's really nothing special about debuggers, as they are a standard tool used in everyday program development. They allow programmers to run the program one step at a time, and observe how the program behaves in practice. Usually debugging is done with the program source code available, but it's possible even with just machine instructions. Seeing the program execution playing out in slow motion before your eyes can be really helpful in understanding the software. Therefore debuggers are a common tool in reverse engineers' toolboxes. [25]

For pirates it comes with a downside. Because debugging happens at runtime, we as the original program authors now have a change of detecting the presence of a debugger. More on that in section 5.4.3

4.3.2 Virtual machine duplication

Virtual machines have gotten more and more popular over the years. It's only natural that customer's should want to run their applications in virtual environments. This however poses some problems for the licensing systems. It's a difficult task to duplicate a physical computer in such detail that the software couldn't tell the difference, but for virtual machines this might be as easy as hitting `Ctrl + C` followed by `Ctrl + V`. That is, just copying and pasting.

This is problematic for both of our licensing schemes. First, lets say you have

installed an application with a valid standalone license to a virtual machine. You could just take a snapshot of the whole virtual machine and pass it to someone else. It would be really hard for the installed application to tell that it's in fact a copy. An online license service is likely the only solution that can prevent this.

Most desktop users probably aren't running hypervisors quite yet, but for servers it's commonplace. Therefore it should be expected that even a law obedient system administrator has a high change of installing his organization's license server into a virtual machine. Denying this would be really backwards from the vendor's side.

So, a license server is installed into a virtual machine, and a bunch of licenses is bought for it to distribute. A malicious system administrator could try to double the license count by duplicating the virtual machine that holds the license server. Half of the user's would be directed to the duplicate server while the other half still get their license from the official server. The client application shouldn't have any quarrels with this. From their perspective the duplicate is just as valid as the original, and to be fair, they really are identical apart from different network addresses. Clearly this is an instance of license double spending. One could also argue that this cloning of license servers is more convenient than cloning workstations containing standalone licensed software. This is an important point because the overall point of licensing systems is to make piracy *inconvenient*.

As was the case with standalone licenses, here too it seems like the only possible remedy is to require some sort of cloud service verification. This is in conflict with what I previously said about license server's internet connection in 4.2.1. I stand behind both views, and instead I suggest that this contradiction rises from deeper problems in the network licensing architecture.

I will return to the topic of detecting virtual machine attacks in section 5.4.4.

4.3.3 Turning back time

Since licenses are often issued for a certain period of time, one obvious way to circumvent the system would be to *go back in time*. While generally impossible, in computers this can be achieved by modifying the system clock. In principle this should work. If you really can convince the licensing system that it's living sometime in the past, it should consider already expired licenses valid. This is another type of license double spending.

In most systems, it shouldn't be very hard to adjust the system clock to one's liking. However, I don't think that any modern system is fooled quite so easily. Programs can query time from the internet as well [25]. It would be necessary to intercept these queries as well. Furthermore, turning back the system clock could leave suspicious looking files that appear to have dates in the future. Files like this can give away your attack. The best way of making the time jump would be to revert the whole system to an earlier state so that no relics of the actual time remain. Of course you could also install a completely new system and never introduce it to the correct time.

This attack is more difficult with network licenses. Here you would potentially have to fool the license server *and* all the client applications of your mendacious time. All parties could also query time from the internet.

I cover the detection of these clock attacks in section 5.4.1.

5 Detecting the attacks

In this chapter I suggest methods for detecting the different types of attacks introduced in the previous chapter. The chapter is organized similarly to the last one. In section 5.2 I cover the detection of attacks against standalone licenses, in section 5.3 attacks against network licenses, and finally in section 5.4 we look at general attacks not tied to either licensing scheme in particular.

But first before we dive into it, I need to emphasize that this section *is not perfect.* In fact, it could never be perfect, as we are going to see in section 5.1. No silver bullets are offered here. The idea is more to just map out the most relevant mechanisms out there, fallible as they may be.

5.1 Perfect piracy detection is impossible

Even though chapter this chapter contains plenty of methods for piracy detection, we should mind our expectations. It's always going to be possible to crack the system one way or another. We saw in chapter 4 that there are multiple potential points of failures in licensing systems, but here we need to concern ourselves with just the local executable. After all, if we manage to crack the local file, we might as well do it in such fashion that the cracked executable just ignores the rest of the system.

Regardless of the detection methods used, it's always going to boil down to this: In our program there are `if` statements that decide whether piracy has been detected

or not. One branch registers the detection, while the other just continues execution as usual. Any piracy detection method can be nullified by modifying these critical `if` statements. [26]

In practice this may not be so easy to do. There are numerous branching points in programs. Detecting the ones that correspond to piracy detection measures is likely going to be difficult. Furthermore, methods exist for making this even harder. Code obfuscation tries to hide the program logic by somehow mixing the executable without actually modifying the program behaviour. There are numerous methods to choose from [27]. Obfuscation will definitely make the cracking harder [28], but not impossible. The underlying program logic is still there, even if scrambled to the extreme. In fact, perfect obfuscation has been shown to be impossible [24], [29].

Then maybe we can prevent tampering with the executable in the first place? There are indeed methods that try to achieve this. Software based tamper-protection refers to building the program in such a way that the program itself will detect if it has been modified. Again, this will surely make it harder to crack the software but once again not impossible [30]. The earlier if statement argument applies to tamper protection too. Additionally, M. H. Jakubowski *et al.* present the following clever argument against perfect tamper-protection:

> Informally, if we assume the existence of a generic tool to tamper-protect any program, we can simply feed such a tool's output to itself. Since any tamper protection involves some form of program transformations, this would essentially force the tool to break its own tamper protection in order to modify the tamper-resistant program. Thus, such a tool cannot exist. ([30])

That was tamper-protection from within the application, but it's possible take measures from the outside too. Namely by using *code signing*. The idea is a simple

one: The executable is cryptographically signed with the software vendor's certificate, and the operating system checks this signature before starting the program. Windows for example supports this feature [31], [32]. However, the feature is geared more towards malware than piracy. And even if it was intended specifically to stop piracy, it would at best transform the tampering problem from the user space application to the operating system itself, which is just another program that can be tampered with.

We've now exhausted all options for protecting our piracy detection `if`s from tampering, and thus shown that perfect piracy detection is impossible. For a determined team of pirates, it's always going to be possible to remove our piracy detection mechanisms, given enough time. A special emphasis should be given to this *time factor*. In the absence of perfectness, a good piracy detection mechanism is such that it takes sufficiently long time to break, and the time used to implement the it is at least an order of magnitude smaller than the breakage time [30]. This way the vendor has decent odds to stay ahead in the cat and mouse game of piracy.

Tamper resistance is a widely researched field but we won't spend any more time on it here. I hope that this brief introduction managed to establish some of the basic ideas, and clarify the overall nature of piracy detection. It is indeed a cat and mouse game against crackers, and there are no silver bullets. The idea isn't trying to build perfect systems, merely systems that are sufficiently inconvenient to break while not being too inconvenient to build either. Given enough time and effort, even the hardest systems will be cracked. The video game industry is a great example of this.[1] We need to mind our expectations. Now with our heads humbled down, let's see what we actually *can do* to detect piracy.

[1] As usual, there aren't any real statistics about piracy, but one can nonetheless get a feel for this by browsing the CrackWatch subreddit [33]. It keeps track of newly cracked video games. In many cases, a cracked version is available quite soon after the game's official launch.

5.2 Standalone licenses

5.2.1 Detecting a blocked network connection

This rather primitive attack was described in 4.1.1. Detecting it shouldn't be much of problem. A straight forward method would be to make a test connection or two. For example, ping the vendor's cloud service or some well established server such as Cloudflare's public DNS resolver at IPv4 address `1.1.1.1` [34]. If the request is rejected or times out, it's a dead giveaway that the application has no internet connection.

This however doesn't really account for detecting piracy. It's perfectly reasonable for an honest user to boot up his computer without a network connection every now and then. On the other hand the more nuanced version of this attack where a firewall is used to block only a specific application or packages, can already constitute as partial evidence for piracy. Unfortunately it's not possible to detect this kind of network block with pings alone because they will time out just the same. The application has to make further inquiries to distinguish this more advanced case of network blocking. Most notably you could query the operating system for connectivity information. For example, the Windows Win32 API has method `INetworkListManager::GetConnectivity` which can be used for this [35]. If the system says that it has a connection, but your application doesn't appear to have it, then it's an indication that the application's connection is specifically blocked. This kind of detection is already wort recording, but on it's own even this isn't enough evidence for piracy. It could just be that the system's firewall has a *block by default* policy enabled. This kind of configurations would trigger false alarms on new installs before the user adds the application to firewall whitelist.

I chose to cover this attack as mainly affecting standalone licenses, but the same principles can be used in network licensing systems as well. If they happen to require

external connectivity that is.

5.2.2 Detecting a fake license cloud service

This attack was described in section 4.1.2. Differentiating between the real license cloud service and a good fake can be tricky. Lucky for us, authenticating web servers is actually a very well researched topic. Standard web protocols support authentication with certificates [12]. You probably know this feature by the little lock icon that browsers display in their address bars [36]. It means that the connection is protected and that the web server in question has a valid trusted certificate. Anyone could set up a fake web server but you'd be able to tell it apart based on the lock icon, or more technically, based on an invalid certificate.

We can use the same technology for detecting fake cloud services created for the purpose of using our application for free. The real cloud service should be setup to use TLS, and it needs a certificate. It wouldn't hurt to have it signed by a widely recognized certificate authority, but this isn't strictly necessary for our purposes as there won't be any lock icons when our application makes contact. The important thing is that the public key of the server's certificate is embedded into the application.[2] Then we can let standard networking libraries verify the server's certificate when the application initiates a connection.

This is the de facto way of authenticating servers. When describing the fake license cloud service attack, I also proposed ways of breaking the detection system we just built by means of tampering, so it's by no means perfect. However I don't think that we can do much better at this time. We will just have to try to detect those tampering attacks as well but more on that later.

One additional thing you could try is to introduce some obscurities to the messaging protocol and then detect fake server's based on deviations from these ob-

[2] In 4.1.2 we noted that it would also be possible to let the operating system handle the certificates, but this would be easier to circumvent. Embedding is a better option.

scurities. For instance, each message could contain an otherwise meaningless "#" character in the end with a 50% change. It could be that the fake server does not have this special behaviour, but of course this is just a matter of reverse engineering. I don't believe that spending time on this kind of detection schemes is fruitful.

5.2.3 Detecting a man in the middle attack on license cloud service

This attack was covered in 4.1.3. Detecting it is quite similar to detecting a fake license cloud service, as the man in the middle attacker is really a special kind of fake server. However there are some differences too.

So first off, authenticating the server's certificate works here just the same as it did with a fake cloud service. However, here the methods based on obscure communication protocols do not work at all, because the application is in fact communicating with a valid license cloud service, even though there's someone snooping the connection in between. This means that the obscurity requirements of the communication are always met, and no detections can be made based on them. This is also an argument against such methods to begin with.

The real difference comes with the observation that in a man in the middle attack, the cloud service is still communicating with the application *indirectly*, as opposed to not communicating at all in the pure fake cloud case. This opens up new piracy detection possibilities. Now it's possible to make detections in the server's end too. Methods based on obscurity won't work here either for the same reasons as before, but certificate authentication certainly does. As we remember from section 2.4, in TLS certificate authentication is not restricted to servers. Client certificates can be authenticated just the same. This feature is less widely known because it's not commonly used in the web, but the idea is just the same. The client has a certificate which the server checks for validity and trust.

In our present case this means that we can embed a client certificate to the applications executable, and verify it in the cloud service. The vendor needs to become a certificate authority for it's applications, because the client certificates need to be signed, and it's not feasible nor even sensible to have them all signed by a real certificate authority. The vendor's self signed root certificate will do just fine. This root certificate can then be added as a trusted certificate in the license cloud service. Now the server can be set up to verify the client certificate when the applications connect to it. A naive man in the middle attacker will not have a valid client certificate, and will be detected. When discussing the attack in 4.1.3, I said that a more advanced attacker might actually get the valid client certificate from the application's executable. Against this there's nothing to be done.

One final note about the client certificates. It's possible to use the same certificate in every copy of the application or alternatively use individual certificates. The choice has zero effect on the attacker's job for cracking a single application, but individual certificates would force the attacker to go through the same process on each application instance. This is because it would easy for the license cloud service to notice that the same certificate is used simultaneously from multiple locations.

Leveraging this, there's also a possibility of getting information of which customer is doing the attacking. For this the vendor would have to keep up a database of client certificates and their owners. If the same certificate is observed making $n > 1$ simultaneous connections, it means that $n - 1$ of those clients are pirated. From the database it can be checked who originally bought the application with this certificate. This will essentially detect all attacks where the attacker succeeds in creating a man in the middle client but then just lazily copies it over and over again.

If the attacker has not bothered with extracting the certificates, then this will be detected, but we cannot tell who it is. This is because the certificate will be just fabricated in this case and won't match anything in the vendor's database.

On the other hand, if the attacker is advanced enough to automate the certificate extraction process, then we won't at first detect it, because the certificates are sound. On a longer time scale it might become possible though. If a particular client whose license is reserved, keeps on resiliently checking the license day after day, it's a sign that something may be amiss. Surely a normal human user would quickly realize that the application doesn't have valid license and do something about it. If however a man in the middle attacker changes the responses from negatives to positives, this situation could very well arise. In this case it's also possible to identify the user. Detecting cases like this is a question of data analysis. Possible methods range from traditional statistics to modern machine learning models. Of course, this type of detections don't make any sense if the application is such that it would make sense to start it automatically, for example on each boot. Then the owner might just not have noticed the license problem, and we would be making false positive piracy detections.

To sum up, man in the middle attacks between a standalone licensed application and a cloud license service can be detected most of the time.

5.3 Network licenses

5.3.1 Detecting attacks against license cloud service

These attacks were covered in 4.2.1. Basically they're the same attacks as in standalone licenses but this time targeted at the on-premise network license server. Likewise, the detection methods are the same as they were for standalone licenses. However, there is one crucial difference that we can potentially use to our advantage here. The on-premise license server has local clients connected to it.[3]

[3]If it doesn't have any clients, then it doesn't make much sense to attack it. In order to *"steal"* licenses from an unmodified license server, you eventually need to connect to it with a client application.

The server could ask some of it's client programs to supply information that can be used to potentially detect attacks. This could be as simple as asking the client that does it have an internet connection. If it does, and the server does not, then this implies that someone has blocked the server's network access.

Using this kind of *delegation to clients* is not quite as simple when trying to detect a fake server or a man in the middle attacker. What you would have to do, is to ask a client to make a request directly to the vendor's license cloud service. The client would do this and deliver the response to the local license server, which can then potentially use this information to infer whether the contacted cloud service is the same one that is in direct contact with the license server. Fake license cloud services and man in the middle attacks could be detected like this. It's obvious that this detection scheme is starting to get quite complicated and therefore it's also prone to errors. The delegated checks are also easy to circumvent. Simply redirect all the traffic from the local network to the fake/MITM server instead of the real license service. This kind of check has only potential to detect sloppy attacks where the redirection is made only for the machine running the on-premise license server. Therefore I suggest that these kinds of checks contribute mostly to just unnecessary complexity, and should be avoided. The delegated network connectivity check mentioned earlier is an exception to the rule. The only reasonable way to circumvent it is to just cut the whole local network out of internet, which can be considered a sufficient inconvenience.

5.3.2 Detecting a fake license server

Fake license server attacks were covered in 4.2.2. Their detection is going to be similar to the methods used in detecting fake license cloud services. Only this time the detection happens in the client application, which actually makes the situation very similar to standalone licenses. For standard network protocols, it really doesn't

make much difference whether the faked server is on another continent or in the same building. The principle remains the same. It's a question of *authenticating* the server, cloud or local.

It's possible to use some tricks based on obscurity, but the standard way of doing this is TLS with server certificate authentication [12]. A certificate is embedded to the server's executable, and the public key part is embedded into client applications. Then, when the clients connect to the on-premise server, they can authenticate the server based on the certificate. If the attacker has managed to copy the certificate to her fake license server, then there's not much to be done.

The remaining line of defense would be to also send some sort of check to a license cloud service, but then what would be the point of using the network licensing scheme to begin with? If we are going to check every license from a cloud service anyway, then we might as well be using the simpler standalone licensing scheme.

5.3.3 Detecting a man in the middle attack between a client application and the license server

This man in the middle attack was presented in 4.2.3. From the client application's point of view, the attack is identical to the fake license server attack. Therefore the client side detection remains the same, namely authenticating the the server's certificate. I bet you're already getting a hang of how this certificate authentication works, and fear not, for it's going to be save us once more. Or at least try it's best.

The difference to a fake server is that in this scenario the on-premise license server is still operational because the man in the middle attacker doesn't *replace* it like a fake server does. From this it follows that the license server also has a shot at detecting the attack. The way to do this is analogous to the method that was used to detect man in the middle attacks in the license cloud service's end (see 5.2.3).

From the license server's point of view, the client connections will not be com-

ing from actual clients but from the man in the middle attacker(s). This can be detected if the server authenticates it's clients. In TLS terms, we are going to use client certificates to authenticate the client applications when they request licenses from the license server. In practice, a client certificate is embedded into all client executables. These certificates should be signed by the vendor, and the corresponding public key should be embedded into the server's executable. There's no need to involve certificate authorities. With this, the server can verify the authenticity of it's clients, provided that the attacker hasn't managed to extract the client certificates or tampered with the license server.

Here it would be useful to use unique certificates for all client applications.[4] This way it would not be enough to extract only one of them, but the work would have to be carried out individually on each application. The attacker could try to use the same certificate on multiple man in the middle clients, but this would be easily detectable by simply observing that the same client appears to make more than one connection simultaneously. The server doesn't need to have list of all existing certificates to do this.

Similarly to standalone licenses, if the application is such that it wouldn't make sense to autostart it, then even the more advanced attacks could potentially be detected by paying attention to repeated license requests that receive negative responses. It could be that someone flips these responses to positives and that's why the client is happy to keep requesting permissions to start. However, this would require some kind of bookkeeping in the on-premise license server, and these records could be potentially tampered with. This nullification of detection was not possible in standalone licensing where the bookkeeping was done off-premises in the vendor's cloud.

[4]This doesn't imply that the server would need to store a massive amount of client certificate public keys. All the different client certificates can still be signed with one and the same root certificate, which is controlled by the vendor. It's enough for the license server to store only this public key.

5.4 General

5.4.1 Detecting clock attacks

This attack was discussed in 4.3.3. The idea was to extend the validity of expired licenses by rewinding the system clock. There are several mechanisms that can be used to detect this kinds of attacks.

The obvious one is to verify the time from online, if there's an internet connection that is. There are several potential places for querying time. Public network time servers (NTP) are a natural choice, but those aren't know for their security. The attacker could for example use man in the middle methods to change the returned time to his liking [37]. Network time security (NTS) is a hardened version of the standard NTP time protocol [38]. It should be the preferred choice in critical applications. The vendor could setup his own NTS server or use public ones. However, there isn't necessarily any need to use these timing standards at all, since accuracy isn't really a concern for us. We are interested in detecting offsets of several days, as it wouldn't make much sense to pirate an application for only 15 minutes. This means that the vendor could just implement a simple API for querying time from the cloud service. Using TLS with client and server authentication should be plenty secure. You don't even need to care about time zones if you don't mind someone possibly pirating the application for a whopping 12 hours.

Network license servers have the additional option of querying time from their clients. It might be reasonable to spoof time in the single machine running the server, but spoofing it in all the clients would already be quite an undertaking.

There are also other methods that don't rely on network. These methods are not aimed at discovering what time it really is, but merely detect if it has been altered. This could be done for example by saving the current time periodically to a file. Then it's possible to compare the saved time to the one reported by the system.

If there's a big difference, it means that someone has altered the clock. Of course it wouldn't be very hard for the attacker to observe this saving action and thereby locate the time save file. Then the attacker could simply replace that file too and avoid detection.

A more resilient way would be to store the time in a file that the application needs for some other purpose, for example a normal save file of a game. Sure the attacker could still replace this file with a backup that he took earlier, thereby allowing himself to rewind the clock to the moment of that backup, but it wouldn't be very useful as this would also rewind the game progress to this point. The only way to benefit from this kind of piracy would be to always start a new game or reverse engineer the game save file completely.

One more possibility would be to just rely on files that are completely unrelated to the application itself. Windows log files could be an example. Just check the file metadata for the time when it was last modified. If this time is too far from the apparent present time, then it implies that the system clock has been tampered with. However, it shouldn't be too hard to observe what files the application is reading to catch this sorta check. Furthermore, I would imagine that anyone seriously trying this kind of attack, would anyway rewind the whole system to an earlier snapshot. In this case there wouldn't be any discrepancies in metadata dates either. In my opinion using files that contain essential, changing, data for the application is the best option. Of course it could be that the application in question just doesn't use files like this, in which case the other ways should be adopted.

Finally, in all of this we should remember that there are also perfectly normal reasons for shifting the system clock. Namely, summertime and traveling to a different timezone. Probably the easiest way to deal with these is just to ignore infrequent hops of time that are no more than a day in size. Surely a more sophisticated algorithm could be developed but there doesn't seem to much value to be gained from

it, merely an increase in false positive piracy detections.

5.4.2 Tamper detection

Tampering attacks were covered in 4.3.1. This is probably the most common tool for pirates, so it would be good to detect it reliably. Unfortunately, in 5.1 we already saw that this is impossible. We are inherently at a disadvantage but let's just detect what we can, and hope that it's enough.

Because of it's importance in piracy detection, tamper detection is a widely researched area and there's probably an equally large amount proprietary methods hiding in executables. I won't be able to cover the whole field here, but instead I'll focus on the basics and highlight a couple prominent methods.

Checksums

Tamper detection is essentially *self-checking*. The application tries to check whether it has been modified. The most obvious way of checking whether a piece of data has changed—in this case the application's executable—is using a checksum. That is, you calculate a cryptographic hash of some parts of the executable, and package it with the application. Once started, the application can calculate the same hash and compare it to the original one. They will differ if the executable has been tampered with. This is quite an insecure way of doing this, but it serves as a good starting point.

Oblivious hashing is one improvement on this idea. Instead of calculating a static hash of the executable and hiding it somewhere, it calculates a running trace of the program's execution. This is achieved by cumulatively hashing parts of the runtime program and comparing these to precomputed values. The main advantage of this method is that the attacker cannot compute these values from executable itself. The program has to be executed, or at least simulated, in order to precompute new

values. [39]

Verifying the side effects of a program comes with some limitations. Not all program runs are the same. This could lead to inconsistent hash trails, and therefore also to false positive piracy detections. *Short range oblivious hashing* is an improved method that attempts to solve this problem by calculating the cumulative hashes for all possible branches of execution. This way all possible program runs are covered. This sounds computationally expensive, but that's why it's called *short range*. This method is only feasible on restricted parts of the program. [40]

Redundancy

Redundancy is more commonly used in environments where computation needs to be fault-tolerant. An important computation can be performed multiple times (possibly in parallel) and then the most frequent result is selected as the actual final result. A similar idea can be used for tamper detection as well. After all, tampering can be viewed as a kind of computation error. If the individual results differ, tampering has occurred.

Jakubowski *et al.* have developed a system of software *tamper tolerance* based on this idea [30]. Tamper tolerance means that the faults caused by tampering are just silently corrected. The system works along the lines of what I described in the earlier paragraph. It's illustrated in figure 5.1. The missing piece of the puzzle is *individualization*. An individualized program or code block is one whose functionality remains the same as the original but the code is different. Individualization is needed so that tampering one individualized version wouldn't affect the others. Thanks to individualization, there's no need for separate processors either. The redundant modules can be executed normally on a single core. There are numerous ways of individualizing programs automatically [27].

Tamper tolerance can be viewed as consisting from two distinct steps: detection

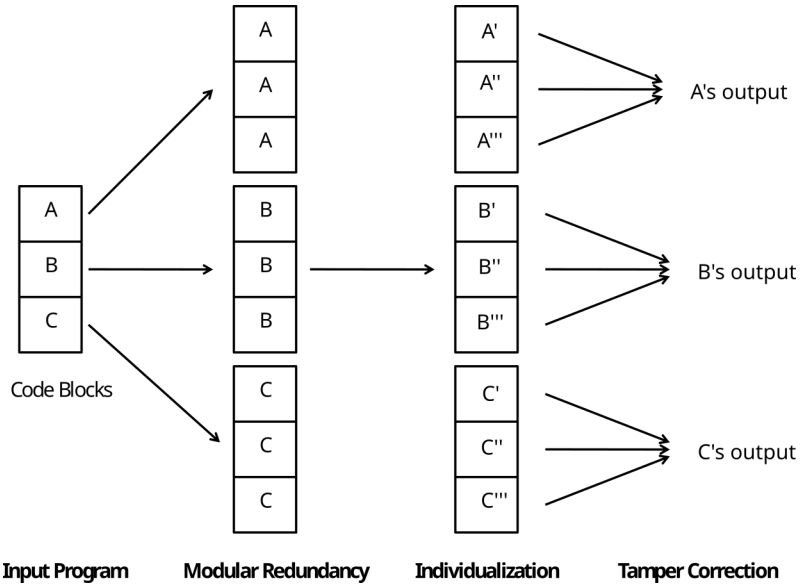

Figure 5.1: A high level view of redundant tamper-tolerant software. [30]

and fixing. Fixing is a reaction to a positive tamper detection. For our purposes of just detecting tampering, the fixing step can be omitted. This also allows for smaller executables since two individualized copies are enough for detection whereas fixing requires a minimum of three.

5.4.3 Debugger detection

Earlier in section 4.3.1 I noted that debugging can be a powerful tool in reverse engineering and tampering. Furthermore, there really shouldn't be any good reasons for the user to debug the application. If it misbehaves, the correct course of action is to contact the vendor, and file a bug report. Therefore detecting a debugger likely suggest that something shady is going on. EXCEPT, when it's one of the vendor's employees doing the debugging. This is a perfectly normal thing in a programmer's daily work, and no obstacles should be built to prevent it. Detecting and reporting a debugger is still fine, but I usher to use caution on more radical responses. Penalizing debuggers could easily backfire.

The core feature of debuggers is that they can pause the program execution. A

simple way of detecting this would be to monitor execution times of critical code block with timers. Most functions will probably execute within a fraction of a second on a modern computer, so when some function suddenly takes minutes to finish, it's a strong sign that a debugger is present. [25]

Debuggers might implement some of their features by adding special instructions to the executable [25]. This is essentially the same as tampering which means that methods used to detect tampering can be used to detect debuggers as well.

The strength of these two methods is that they're universal. More advanced systems no doubt exist, but they require detailed knowledge of how particular debuggers work. Targeting specific debuggers will require you to manually cover all popular debuggers (for your particular technology) one by one, and keeping up with their updates. It can be a lot of work. My recommendation is sticking with general purpose methods.

5.4.4 Detecting virtual machine duplication

This attack was described in section 4.3.2. In principle, the detection is a two phase process. First we have to detect whether the application is running in a virtual machine to begin with. Second we have to decide whether that virtual machine is unique or has it been duplicated.

Lets start from the second point because it's going to set the bounds. This is because there's in fact no consistent offline way of detecting whether the virtual machine is unique. It could be possible in some hypervisors but it doesn't really help as there are numerous hypervisors and emulators, each with their own sets of behaviours [41]. Most of the time, the duplicate VMs are going to appear identical to us. Basically we are left with online cloud service as our only detection mechanism. The service can easily note if two instances of the same license are connecting simultaneously. It's interesting to note that this detection mechanism is actually

exactly the same as it was in the other similar cases. Whether or not the application is running in a virtual machine doesn't actually bear any significance for this check. Virtual machine duplication is duplication all the same.

So the first step of detecting a virtual machine turned out to be useless after all. It could just be skipped altogether. This is a totally reasonable thing to do, but maybe there can be some indirect value in detecting virtual machines nonetheless. It's true that we cannot make any definitive piracy detections based on this info, but it may serve as confirming evidence for some other detection mechanisms. After all, it is generally easier to tamper with the application in a virtual environment [41]. So lets have a brief look into virtual machine detection anyway.

No single method will be able to detect all hypervisors. The individual detection mechanism are based on the specific technical details of the particular hypervisor in question, and as a result, they will have to be addressed one by one. Known bugs in hypervisors can be leveraged in detection too. Suffice to say that creating a comprehensive virtual machine detection system from scratch is going to be a big undertaking, and it needs to be updated every time one of the hypervisors updates it's behaviour. Here I'm just going to give a couple examples for the more popular hypervisors. [41]

VMware is a proprietary, closed source hypervisor that relies on hardware support for virtualization. There are several ways of detecting it. For example, if the operating system within the virtual machine (guest) is Windows, then several Windows registry keys will have VMware specific values. VMware also supports communication between the guest and the real outer operating system (host). An application can leverage this to detect VMware. If the host communication method appears to work, even by just returning an error, then this implies that the application is running inside VMware. [41]

QEMU is an open source hypervisor that typically uses software for it's virtu-

alization. This means that most hardware based detection mechanisms wont work here. Qemu also doesn't support communication between the host and the quest operating systems. Still there have been some subtle differences compared to normal hardware based computers. For example, the `CPUID` instruction returns a different value on an AMD processor. It's supposed to return the processor's name but on Qemu it returns *"QEMU Virtual CPU version x..x..x"* instead. Qemu also doesn't support double fault exceptions. Normally the processor would rise one when an exception occurs while it's still in the middle of handling another exception. You could try to artificially cause this situation and then observe how it's handled. [41]

5.5 Reaction after detection

What to do after the application has detected piracy? This is an off-topic question, but we'll have short look at it anyway because it bears some indirect significance for our detection mechanisms.

The traditional answer is to halt the application, or at least degrade it in some regard [2]. Initially this seems like a natural choice. The vendor would like the user to pay for the application so it makes sense to block unlicensed usage as soon as it's detected. But in fact this might not be so productive. By observing where the application stops, the attacker can obtain information about the location of the piracy detection mechanism in the executable [30]. This will make it easier for the attacker to tamper away these mechanisms.

It's harder to locate detection mechanisms that don't immediately lead to observable consequences. Therefore it could be a good idea to delay the response [30], or choose a less disruptive response to begin with, such as merely *phoning home*. This would simply mean that the application sends a piracy notification to the vendor. The vendor can then choose whether it would like to do something about it, and at least the notification will constitute to valuable statistics. Of course this notification

could be prevented too, but there's less incentive of doing so since the pirate can already use the application normally at this point.

There's a lot more to different piracy response strategies but I won't go any further here. For our purposes the key point is that the chosen response strategy will affect the detection mechanisms.

6 Conclusions

I covered the main attack types against standalone and network licensing systems. Usually when people talk about piracy, they think only about the local crack attacks that involve tampering with the executable, but here we discovered that there are several network based attacks too. Some of these were pure network attacks, while others served only as supplements to traditional tampering attacks. All attack types were covered in considerable detail.

Network licensing was discovered to be the more complex of the two systems. This was observed to result in a larger amount of attacks against network licenses. While it's true that network licensing systems also had some additional detection mechanisms at their disposal, it's safe to conclude that network licensing is less secure than standalone licensing.

I presented possible detection mechanisms for each attack type. The importance of vendor controlled cloud services was discovered several times. Without them, it's practically impossible to prevent license double spending. TLS authentication mechanism for servers *and clients* proved out to be a powerful tool for detecting attacks on all network connections. Especially good results were achieved for man in the middle attacks against cloud services, where even the successful attacks could be eventually detected via bookkeeping. Overall, the list of detection mechanisms was not exhaustive, but the most important methods were covered. I didn't dive deep into unnecessary technical details of these methods, but instead focused on

their key principles. Many of these methods are discussed in more detail in the cited papers, while some of them are still waiting for further research.

I talked about the overall goal of piracy detection. There's no point in aiming for perfectness, as it was shown to be impossible. In this sense the attacker always has the advantage. Rather, the goal is to make our detection systems only sufficiently difficult to attack, so that most users won't bother trying, or at least will give up before succeeding. Too ambitious detection mechanisms will likely just lead to false positive piracy detections.

I offered some considerations about what should done after piracy is indeed detected. There didn't seem to be obvious right answers, but it was clear that the chosen strategy would have consequences on our detection mechanisms. The question is game theoretic and complex in nature, but it's also rather well studied. These studies would be natural follow up readings after this thesis. Regardless of the chosen response strategy, the detection step covered here will remain important.

This study should provide enough pointers and ideas to develop a competent piracy detection system in standalone and network licensing environments. The attacks and detection mechanisms are summarized in the tables on the next page.

Table 6.1: Summarisation of attacks against standalone licenses and their detection mechanisms.

Attack	**Detection**
Cutting the internet	Ping, Ask the operating system
Fake license cloud service	TLS server authentication
MITM on cloud service	TLS server & client authentication, Bookkeeping

Table 6.2: Summarisation of attacks against network licenses and their detection mechanisms.

Attack	**Detection**
Cutting the internet	Ping, Ask the OS, Delegate to clients
Fake license cloud service	TLS server authentication
MITM on cloud service	TLS server & client authentication, Bookkeeping
Fake license server	TLS server authentication
MITM on license server	TLS server & client authentication

Table 6.3: Summarisation of generic attacks and their detection mechanisms.

Attack	**Detection**
Clock alterations	Time servers, Save files, Timestamps on other files
Tampering	Checksums, Oblivious hashing, Redundancy, ...
Debuggers	Timers, Detect special behaviour of specific debuggers
VM duplication	License cloud service, Detect specific hypervisors

References

[1] K. Reavis Conner and R. P. Rumelt, 'Software piracy: An analysis of protection strategies', *Management science*, vol. 37, no. 2, pp. 125–139, 1991.

[2] D. Ferrante, 'Software licensing models: What's out there?', *IT Professional*, vol. 8, no. 6, pp. 24–29, 2006.

[3] BSA, 'Global software survey', The Software Alliance, Tech. Rep., Jun. 2018. [Online]. Available: `https://gss.bsa.org/`.

[4] Revenera. 'FlexNet publisher', [Online]. Available: `https://www.revenera.com/software-monetization/products/software-licensing/flexnet-licensing` (visited on 15/02/2022).

[5] X-Formation. 'LM-X license manager', [Online]. Available: `https://www.x-formation.com/lm-x-license-manager/` (visited on 15/02/2022).

[6] Reprise Software. 'Reprise license manager', [Online]. Available: `https://www.reprisesoftware.com/products/software-license-management.php` (visited on 15/02/2022).

[7] Open License Manager. 'Licensecc: A C++ software license manager', [Online]. Available: `https://open-license-manager.github.io/licensecc/index.html` (visited on 22/02/2022).

[8] Open Source Initiative. 'The 3-clause BSD license', [Online]. Available: `https://opensource.org/licenses/BSD-3-Clause` (visited on 16/02/2022).

[9] P. C. v. Oorschot, 'Revisiting software protection', in *International Conference on Information Security*, Springer, 2003, pp. 1–13.

[10] D. Yaga, P. Mell, N. Roby and K. Scarfone, 'Nistir 8202: Blockchain technology overview', Oct. 2018.

[11] S. Chen, S. Jero, M. Jagielski, A. Boldyreva and C. Nita-Rotaru, 'Secure communication channel establishment: TLS 1.3 (over TCP fast open) vs. QUIC', in *European Symposium on Research in Computer Security*, Springer, 2019, pp. 404–426.

[12] E. Rescorla, 'The transport layer security (TLS) protocol version 1.3', RFC Editor, RFC 8446, Aug. 2018.

[13] Yuhkih. 'File:chain of trust.svg'. (17th Sep. 2020), [Online]. Available: `https://commons.wikimedia.org/wiki/File:Chain_Of_Trust.svg` (visited on 15/03/2022).

[14] Sentinel. 'Sentinel dongles', [Online]. Available: `https://sentineldongle.com/` (visited on 22/02/2022).

[15] D. Ma, 'The business model of "Software-As-A-Service"', in *Ieee international conference on services computing (scc 2007)*, IEEE, 2007, pp. 701–702.

[16] Golden. 'Golden query tool', [Online]. Available: `https://golden.com/query` (visited on 04/05/2022).

[17] I. Jaakkola, 'Proof-of-work ja proof-of-stake hajautetuissa kryptovaluutoissa', Translated title: Proof-of-Work and Proof-of-Stake in distributed cryptocurrencies, University of Turku, Jan. 2021.

[18] 'Welcome to Ethereum', [Online]. Available: `https://ethereum.org/en/` (visited on 05/05/2022).

[19] 'Monero - secure, private, untraceable', [Online]. Available: `https://www.getmonero.org/` (visited on 05/05/2022).

[20] J. Duchene, C. Le Guernic, E. Alata, V. Nicomette and M. Kaâniche, 'State of the art of network protocol reverse engineering tools', *Journal of Computer Virology and Hacking Techniques*, vol. 14, no. 1, pp. 53–68, 2018.

[21] T. Radivilova, L. Kirichenko, D. Ageyev, M. Tawalbeh and V. Bulakh, 'Decrypting ssl/tls traffic for hidden threats detection', in *2018 IEEE 9th International Conference on Dependable Systems, Services and Technologies (DESSERT)*, IEEE, 2018, pp. 143–146.

[22] A. Mallik, 'Man-in-the-middle-attack: Understanding in simple words', *Cyberspace: Jurnal Pendidikan Teknologi Informasi*, vol. 2, no. 2, pp. 109–134, 2019.

[23] K. Bhatia and S. Som, 'Study on white-box cryptography: Key whitening and entropy attacks', in *2016 5th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions)(ICRITO)*, IEEE, 2016, pp. 323–327.

[24] G. Canfora, M. Di Penta and L. Cerulo, 'Achievements and challenges in software reverse engineering', *Communications of the ACM*, vol. 54, no. 4, pp. 142–151, 2011.

[25] M. N. Gagnon, S. Taylor and A. K. Ghosh, 'Software protection through anti-debugging', *IEEE Security & Privacy*, vol. 5, no. 3, pp. 82–84, 2007.

[26] M. Dalheimer and F.-J. Pfreundt, 'Genlm: License management for grid and cloud computing environments', in *2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, IEEE, 2009, pp. 132–139.

[27] S. Hosseinzadeh, S. Rauti, S. Laurén *et al.*, 'Diversification and obfuscation techniques for software security: A systematic literature review', *Information and Software Technology*, vol. 104, pp. 72–93, 2018.

[28] D. Pavlovic, 'Gaming security by obscurity', in *Proceedings of the 2011 new security paradigms workshop*, 2011, pp. 125–140.

[29] B. Barak, O. Goldreich, R. Impagliazzo *et al.*, 'On the (im) possibility of obfuscating programs', *Journal of the ACM (JACM)*, vol. 59, no. 2, pp. 1–48, 2012.

[30] M. H. Jakubowski, C. W. N. Saw and R. Venkatesan, 'Tamper-tolerant software: Modeling and implementation', in *International Workshop on Security*, Springer, 2009, pp. 125–139.

[31] Microsoft. 'Authenticode'. (15th Aug. 2017), [Online]. Available: `https://docs.microsoft.com/en-us/previous-versions/windows/internet-explorer/ie-developer/platform-apis/ms537359(v=vs.85)?redirectedfrom=MSDN` (visited on 16/02/2022).

[32] Microsoft. 'MSIX package signing overview'. (12th Jan. 2022), [Online]. Available: `https://docs.microsoft.com/en-us/windows/msix/package/signing-package-overview` (visited on 16/02/2022).

[33] Reddit. 'r/CrackWatch', [Online]. Available: `https://www.reddit.com/r/CrackWatch/` (visited on 17/02/2022).

[34] Cloudflare. 'Cloudflare 1.1.1.1', [Online]. Available: `https://developers.cloudflare.com/1.1.1.1/` (visited on 28/02/2022).

[35] Microsoft. 'INetworkListManager::GetConnectivity method (netlistmgr.h)', [Online]. Available: `https://docs.microsoft.com/en-us/windows/win32/api/netlistmgr/nf-netlistmgr-inetworklistmanager-getconnectivity` (visited on 28/02/2022).

[36] A. P. Felt, R. W. Reeder, A. Ainslie *et al.*, 'Rethinking connection security indicators', in *Twelfth Symposium on Usable Privacy and Security (SOUPS 2016)*, 2016, pp. 1–14.

[37] J. Selvi, 'Bypassing http strict transport security', 2014.

[38] D. Franke, D. Sibold, K. Teichel, M. Dansarie and R. Sundblad, 'Network time security for the network time protocol', RFC Editor, RFC 8915, Sep. 2020.

[39] Y. Chen, R. Venkatesan, M. Cary, R. Pang, S. Sinha and M. H. Jakubowski, 'Oblivious hashing: A stealthy software integrity verification primitive', in *International Workshop on Information Hiding*, Springer, 2002, pp. 400–414.

[40] M. Ahmadvand, A. Hayrapetyan, S. Banescu and A. Pretschner, 'Practical integrity protection with oblivious hashing', in *Proceedings of the 34th Annual Computer Security Applications Conference*, 2018, pp. 40–52.

[41] P. Ferrie, 'Attacks on more virtual machine emulators', *Symantec Technology Exchange*, vol. 55, p. 369, 2007.