Formal Security Analysis of Widevine through the W3C EME Standard

Stéphanie Delaune
Univ Rennes, CNRS, IRISA, France

Joseph Lallemand
Univ Rennes, CNRS, IRISA, France

Gwendal Patat
Fraunhofer SIT | ATHENE, Germany

Florian Roudot
Univ Rennes, CNRS, IRISA, France

Mohamed Sabt
Univ Rennes, CNRS, IRISA, France

Abstract

Streaming services such as Netflix, Amazon Prime Video, or Disney+ rely on the widespread EME standard to deliver their content to end users on all major web browsers. While providing an abstraction layer to the underlying DRM protocols of each device, the security of this API has never been formally studied. In this paper, we provide the first formal analysis of Widevine, the most deployed DRM instantiating EME.

We define security goals for EME, focusing on media protection and usage control. Then, relying on the TAMARIN prover, we conduct a detailed security analysis of these goals on some Widevine EME implementations, reverse-engineered by us for this study. Our investigation highlights a vulnerability that could allow for unlimited media consumption. Additionally, we present a patched protocol that is suitable for both mobile and desktop platforms, and that we formally proved secure using TAMARIN.

1 Introduction

The world of entertainment is evolving rapidly. Indeed, Over-the-Top (OTT) platforms, such as Netflix, Amazon Prime Video, and Disney+, have revolutionized the way we consume content. OTT streaming services are websites that consumers use to access their favorite movies. These services, valued at \$455 billion in 2022 [25], are accessible on almost all Internet-enabled devices, including smart TVs, computers, laptops, smartphones and tablets. The large distribution of media in user-controlled devices creates challenges for OTT platforms. The main challenge remains piracy; a recent study indicates that 80% of digital piracy is now due to streaming [40]. This is due to the fact that digital video is a set of binary information that represents video content. Bits can be easily copied, transferred, or stored. Therefore, digital video is intrinsically unprotected, *i.e.* digital video can easily be viewed by anybody and be infinitely duplicated. Thus, in some cases, it needs protection, namely access control, under some defined conditions. For instance, many OTT platforms enforce monthly subscription, meaning that customers pay a regular fee to access a catalog of premium video contents defined by the OTT.

Consequently, OTT platforms rely on DRM (Digital Rights Management), which is a technology that aims to protect media from piracy. Modern DRMs ship content in an encrypted form, and then control their decryption through authorized modules on users' devices. A prerequisite to decryption is to process the corresponding license. A DRM license describes the agreement between the OTT, or content providers, and the consumer. A license contains the decryption key, the associated usage rights and consumption policies that the DRM module is authorized to perform with the key. Given their sensitive nature, licenses are protected when delivered to the client's DRM module. The underlying protection mechanisms are proprietary and unique for each DRM system, thereby requiring Web OTT platforms to cumbersomely adapt their websites for each DRM using plugins. For instance, until December 2014, users needed to install some Wine-wrapped Microsoft Silverlight plugin to watch Netflix on Ubuntu [39].

Striving to make the Web an online environment to watch movies, the World Wide Web Consortium (W3C) published Encrypted Media Extensions (EME) as a W3C Recommendation or Web standard [14]. EME is an Application Programming Interface (API) that allows plugin-free playback of encrypted content seamlessly in all major Web browsers. Thus, Web developers no longer have to use proprietary tools required by external plugins. Roughly, the EME protocol abstracts the process of license acquisition and related actions, allowing OTT platforms to develop their Web applications once and to deploy them everywhere on the Web regardless of which DRM system is used. EME adoption was decried [21]; the opponents articulated security and privacy concerns about a "built-in DRM in the Web". Several privacy issues were pointed out by Patat *et al.* [34] that show how to leverage EME to construct unique fingerprints. As for security, we merely find some auto-congratulation messages of W3C members claiming that EME "*brings greater security to the Web*" [43]. This is unfortunate, since, despite being optional, EME is deployed on all major browsers [7]. Thus, any vulnerability on a given DRM system can leverage the EME promise: implement the exploit once (simply using Javascript) and compromise everywhere. This makes EME a lucrative attack target for piracy.

Problem Statement. Years after its wide integration, we fill this gap by taking a closer look at the security of EME as a protocol. In this paper, we present, to the best of our knowledge, the first in-depth security analysis of the EME API standard covering a rigorous formal analysis of both its workflow and security requirements. A significant challenge in conducting such an analysis is that the EME standard involves various *opaque* messages that depend on the underlying DRM system. It is important to note that the EME specifications do not introduce yet-another DRM system; rather, they abstract browser interaction with pre-existing DRM modules. Therefore, any complete analysis must be contingent on a particular instantiation of EME. We base our analysis on Widevine [16], which is a cross-platform (Windows, Android, Linux and macOS) and cross-browser (Firefox, Edge, Chrome, and all other Chromium-based browser) DRM owned by Google. The scope of other proprietary systems, such as FairPlay [1] and PlayReady [29], would have been more limited. More importantly, each DRM system introduces a plethora of proprietary technical details. We prioritized lucidity in our presentation and longitudinal analysis over completeness, particularly since our conclusions would have remained unaltered: the security of EME is quite brittle and requires formalization to accomplish the associated security goals.

In [22], Heilman introduces the concept of perfect DRM using game theory. A perfect DRM system is one such that content protected by it is not available for further copying; the option of piracy is unavailable to consumers. Therefore, we would like to formally prove the absence of piracy: *i.e.* that EME, when instantiated with a specific DRM system, namely Widevine, satisfies a desired security notion. At a methodological level, piracy can be hard to model because it involves security notions beyond the straightforward confidentiality and integrity of messages. In addition, it should scale well enough to allow us to model the complex EME workflow.

Contributions Summary. We present in this work four main contributions. First, we reverse engineer the EME messages as defined by Widevine. Second, examining the security mechanisms of Widevine, we discover a vulnerability that allows a malicious user to load arbitrary consumption policies for any license, given a plausible behavior of the Widevine client. We propose a mitigation to fix that flaw. Third, we define clear security goals capturing the notion of piracy. Finally, supported by our EME insights and reverse engineering, we formalize Widevine EME and its expected security goals, and perform formal analysis to prove the absence of piracy despite subtle behaviors of Widevine implementations. For our model, we leverage the TAMARIN prover [28], a symbolic protocol analysis tool, to analyze several non-trivial aspects of Widevine EME. One of the challenges in developing our model is that it requires finding middle ground between the EME protocol with its opaque messages, and the proprietary technical details of Widevine. Thus, we needed to identify the core theoretical concerns of EME, and apply them to how Widevine is implemented in practice. We notably identify how updating licenses in EME constitutes a central part causing sophisticated security issues. In fact, the vulnerability we identified was discovered through our formal analysis. We also formally study the mitigation we propose, and prove that it achieves our formalization of Heilman's perfect DRM security. Overall, our work shows that the W3C claims about "greater security" guaranteed by EME cannot be taken for granted; formal analysis is needed to verify them.

Related work. DRM systems have never ceased to be broken by attackers looking for pirated content. However, as breaking DRM is illegal in many countries, few hackers have ever published detailed technical descriptions of their attacks. Some remarkable exceptions are the "Beale Screamer" case [38], and more recently the L3-decryptor against Widevine [19]. Since 2010, Widevine has seen extensive adoption. Unfortunately, little literature was dedicated to the study of its security mechanisms. Patat *et al.* delve into the undocumented Widevine protocol, shedding light on its various cryptographic components on Android in [33]. They also explore the privacy implications of EME Widevine in [34]. Zhao [46] breaks the Trusted Execution Environment-based Android Widevine. These studies offer profound insight into the design of Widevine, yet the integration of Widevine with the W3C EME standard remained until now unstudied. Our work explores this area, examining how Widevine instantiates EME messages. Unlike previous attacks with sophisticated low-level exploits, ours could in principle be exploited by JavaScript code executed in a simple web page.

Other work has formally studied the security of Web standards. Besides EME, the W3C develops Web standards for various security applications, including notably the Web Authentication API [23], the Web Cryptography API [45], and the Web Payment API [24]. Formal methods are generally not used during the standardization process (though it has been suggested *e.g.* by [20]). They have however been applied to several already existing W3C standards [6, 13, 17]. These analyses all uncovered severe, previously unknown attacks, illustrating the power of formal verification techniques. Because of its opaque messages, EME remained without analysis, which our work addresses. Unlike other work, we do not use specialized formal models designed for the Web, such as WebSpi [3] (based on PROVERIF) or the Web Infrastructure Model (WIM) [15]. Indeed, the threat model we consider is ill-adapted to these models, since we suppose that the attacker controls the browser.

Reproducibility and Responsible Disclosure. All our formal models written in TAMARIN are available online [11], for reproducibility purposes, and to assist future work on this topic. Moreover, we have communicated our draft to the editors of the latest EME draft. On February 13th 2024, they discussed our technical findings in the media WG meeting. The meeting minutes can be found in [44], where our findings were acknowledged. In further discussions, editors suggested a Widevine contact, which we did reach on March 25th, but we have not got any answer. This is unfortunate, since our results concern the particular instantiation of EME by Widevine, even though EME editors mentioned that they would follow on the suggested patch and modify the standard accordingly whenever Widevine reply.

2 Background on DRM

DRM is a set of tools used to protect content from piracy. Today, most large streaming platforms use either Widevine [16] or PlayReady [29] to ensure that only authorized viewers can access protected content.

2.1 DRM Common Architecture

DRM systems are complex, and their design differs depending on the context in which they are used. Here, we only consider DRMs protecting content delivered over the Internet in unicast mode. In this family, all DRMs share at least five common components [36]: (1) a content packager to encrypt content, (2) a content server that delivers the protected media, (3) a content provider that manages subscriptions, (4) a license server that delivers licenses under the control of the content provider, and (5) a client application that enables content consumption. The first two elements are often mapped to a single entity called *Content Delivery Network* (CDN), whereas the content provider and the license server are mapped to an entity called *Over-the-Top* (OTT) media service platform.

We now describe the transactional model of DRM, which details what happens once a subscribed client selects some content to consume. First, the associated CDN delivers the protected content to the client's device. To play back the received content, the DRM client needs the corresponding license. Thus, it requests a valid license from the license server operated by the content provider. A license holds the content's symmetric encryption key, as well as the usage rights granted to the client. Once the license server has verified that the requester fulfills some mandatory conditions, it builds the corresponding license, and cryptographically binds it to the targeted DRM client. When receiving the license, the DRM client is now ready to decrypt the protected content.

2.2 DRM Common Security Model

In [12], Diehl presents a layered security model for DRM based on four main features: content protection, rights enforcement, rights management, and trust management. The *content protection* layer securely seals the content (using *e.g.* symmetric encryption), so that nobody may access it unless explicitly authorized. On the client side, this layer decrypts the content, for the client application, if it receives the associated secret key from its upper neighbor, namely the *rights enforcement* layer. That layer ensures that the content will be used under the conditions indicated by the usage rights, as they were defined by the *rights management* layer. This third layer, on the client side, forwards its decision to its lower counterpart to prevent any unauthorized use. On top, the *trust management* layer ensures that only trustful principals interact, and behave as expected. Trust is enforced via authentication and certificates.

We can map the different DRM elements to this four-layer model. The content packager and content server, namely the CDN, implement the content protection layer. The license server implements the rights enforcement layer, whereas the content provider implements the rights management layer.

2.3 W3C EME Standard

As mentioned in the introduction, in response to the fragmented landscape of DRM solutions, recent technical standards have been defined to solve the interoperability issue. Of particular interest, we focus on the W3C EME standard. It was established to offer a uniform API, allowing web pages to seamlessly interact with browser-supported DRM systems. The primary objective of EME is to enable content providers to work consistently across various browsers, by abstracting a common workflow for license acquisition and usage rights. Although optional, EME has gained support from major web browsers, including Edge, Firefox, Chrome, Safari, Opera, as well as their mobile counterparts [7]. We present below the different components involved in EME, and we detail its API.

EME Components. We list below the main entities as introduced in the standard.

- **CDM**: The Content Decryption Module refers to the DRM system specified in EME. It is responsible for protecting licenses, and enforcing usage rules when decrypting media content on the user's device.
- **User-Agent**: The user-agent, *e.g.* a browser, refers to the entity implementing the EME workflow and API. The EME API is invoked through JavaScript to communicate with a specific CDM.
- **License Server**: License requests generated by the CDM are transmitted to the corresponding license server associated with the specific CDM. This server manages licenses and usage rights for protected content.

The EME design splits the client application, as described in Rosenblatt *et al.*'s architecture [36], into the user-agent (mostly open-source) and the CDM (mostly proprietary).

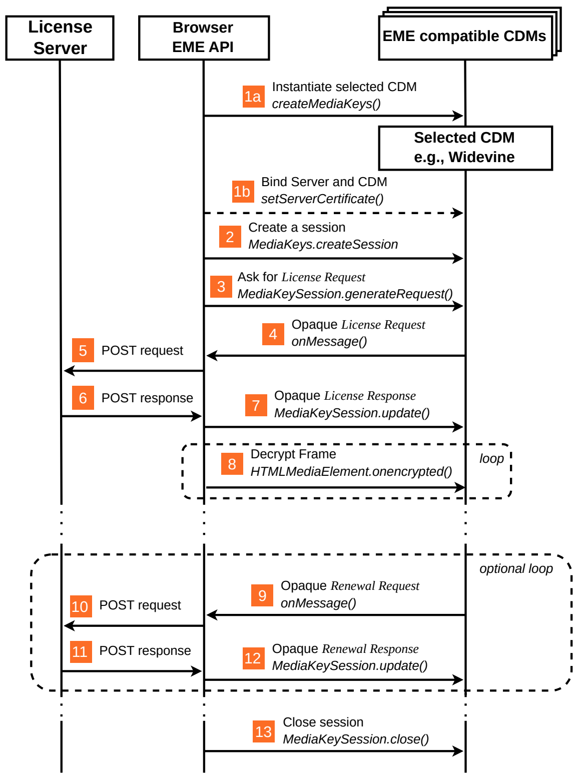
License Server
Browser EME API
EME compatible CDMs
1a Instantiate selected CDM createMediaKeys()
Selected CDM e.g., Widevine
1b Bind Server and CDM setServerCertificate()
2 Create a session MediaKeys.createSession
3 Ask for License Request MediaKeySession.generateRequest()
4 Opaque License Request onMessage()
5 POST request
6 POST response
7 Opaque License Response MediaKeySession.update()
8 Decrypt Frame HTMLMediaElement.onencrypted()
loop
optional loop
9 Opaque Renewal Request onMessage()
10 POST request
11 POST response
12 Opaque Renewal Response MediaKeySession.update()
13 Close session MediaKeySession.close()

Figure 1: EME Workflow – License Acquisition and Renewal.

Protocol Workflow. EME establishes a standardized API that revolves around acquiring licenses, as in the rights enforcement layer, while abstracting the specificities of the underlying protection techniques. Consequently, the EME API depends on various objects and events to facilitate the implementation of proprietary DRM systems. Note that user authentication by the content provider falls outside the scope of EME. An overview of the protocol is given in Figure 1. We assume that the EME user-agent has already obtained all necessary details about the encrypted media from the CDN, including supported codecs and which DRM system to use.

The execution of the EME workflow begins at step $\boxed{1a}$, after gaining access to a KeySystem using `requestMediaKeySystemAccess` method with, as inputs, the requested DRM and configurations for decrypting and decoding the media, an instantiation of a CDM is performed using `createMediaKeys`. At this particular stage, the CDM is accessible and represented by a `MediaKeys` object; however, it is unable to receive any license. Optionally in $\boxed{1b}$, the `setServerCertificate` method can be invoked to bind the `MediaKeys` object to a given license server certificate. Then, in step $\boxed{2}$, a session is established within the CDM, represented by a `MediaKeySession` object, and is identified by a unique sessionID. The EME standard mandates that sessions be associated with separate key wallets and isolated cryptographic environments. They are used to exchange data, mostly licenses, between the license server and the CDM. When a session is created, its key wallet is empty, and it cannot decrypt any content.

Acquire Licenses. In order to obtain media decryption keys and corresponding usage rights, also known as licenses, media-specific `Initialization Data` is sent to the CDM in step $\boxed{3}$. This data is used by the CDM's `generateRequest` method to generate a license request. The specific content of the generated request is entirely dependent on the DRM system being used, as indicated in step $\boxed{4}$. It is treated as an opaque message within the EME framework. Upon receiving the opaque license request, the user-agent forwards it to the license server in step $\boxed{5}$. The server answers with an opaque license response in step $\boxed{6}$. Using the EME API call `update` in step $\boxed{7}$, the response is transmitted to the CDM session, and subsequently handled by DRM-specific mechanisms. According to EME, a successful call to `update` implies that the decryption key is now in the CDM memory, and may only be used according to its usage rights. Now that the CDM session possesses at least one license, it is ready to play the encrypted media in the browser through the HTML5 video element in step $\boxed{8}$. At this stage, the session can either be closed in step $\boxed{13}$, or undergo the renewal process.

Renew Licenses. Following their reception, licenses may or may not be renewed, depending on their usage rights. If permitted, the CDM fires an event to deliver a renewal request to the user-agent. It does so using the `MediaKeyMessageEvent` interface, with its type attribute set to `license-renewal` in step $\boxed{9}$. Similar to the initial license acquisition process, this request is forwarded to a license server in step $\boxed{10}$. Once the renewal response is received in step $\boxed{11}$, the `update` method is invoked in step $\boxed{12}$, updating proprietary metadata as needed.

3 Widevine EME

Widevine is a closed-source proprietary DRM developed by Google [16]. Over the years, Widevine has grown into one of the most widely adopted DRM technologies, supporting various formats such as MPEG-DASH, HLS, or CMAF [26,27,31], and is now leveraged by leading OTT platforms, such as Netflix, and Disney+. Widevine is supported on a wide range of devices, including Android devices, smart TVs, game consoles, and desktop browsers on all major operating systems, including Windows, Linux, and macOS. Since its integration as EME CDM in the Chromium project [8] in February 2013 (three months before the release of the EME first public draft), Widevine has been steadily deployed in various browsers. Note that most browsers provide support for only one or two CDM systems, including Widevine in most cases.

The Widevine cross-platform technology had to adapt its CDM implementation, especially its callable API, with respect to the host framework and the required security level.

This adaptation does not come without risk, since Patat *et al.* [33] leverage the abstraction layer defined for Android to break Widevine's security. One might reasonably wonder whether the EME abstraction can also be used somehow to bypass Widevine protection. In other words, is the Widevine instantiation of EME secure enough for streaming services?

To answer this question, we first dissect the different opaque messages instantiated by Widevine during the EME workflow. We also identify the different elements being present in the CDM internal memory during an opened session. Then, we present an attack given the target CDM not enforcing some additional security verification.

3.1 Experimentation Settings

Operating Systems and Browsers. In order to have a broad view of Widevine's EME instantiation, we used Android mobile devices, being the most used OS in the world [41], and desktops implementation of the CDM. For Android, we used Nexus 5/5X and Pixel 1, 3, 4, and 6 from Android versions 6 to 12. On desktop, we used Windows 10/11, macOS Big Sur, and Ubuntu 22.04. Each time, we performed our analysis on major up-to-date browsers with Firefox up to version 122, Chrome version 121, and Edge version 120 [42].

Widevine Ecosystem. For each device and browser, we used the up-to-date CDMs of Widevine, ranging from version 3.1.0 to 16.1.0 for Android, and the desktop version 4.10.2557.0. As for the license server, we set up our environment to communicate with the integration test license server provided by Widevine, to stay as close to the intended server behavior as possible.[1]

Research Methodology. Our approach to analyzing the Widevine EME protocol involves the observation of EME messages and API calls, using reverse engineering techniques and the monitoring of the Widevine CDM activity. To this end, we leveraged various tools, including Frida [35], Ghidra [30], Widevine-dedicated monitoring utilities such as WideXtractor [2], and custom scripts to inspect and manipulate EME interactions. We performed static, dynamic and API reverse engineering techniques to analyze Widevine on both Desktop and Android platforms.

- *Static Analysis:* Using Ghidra, we decompiled Widevine binaries to gain a structural understanding of the implementation and to identify key functions involved during the protocol, such as license acquisition or renewal.

- *Dynamic Analysis:* We leveraged Frida for Android and WideXtractor for EME monitoring on Desktops. These tools enabled us to write hooks for the identified functions and monitor the data processed during typical CDM-Server communications.

- *API analysis:* We manipulated the arguments of identified function calls, by introducing errors, redundancies, and forged parameters, to note all resulting behaviors.

This setup allowed us to intercept all messages between our test devices and the Widevine integration test server, during the decryption and playback of protected media content using the CDM. We documented all message exchanges and API invocations for subsequent analysis, enabling a comparative study across different operating systems and CDM versions. Through static analysis, we gained insights into the structure of opaque messages within EME. This analysis allowed us to better understand Widevine EME, including license acquisition, renewal processes, and cryptographic elements based on prior works [33, 34], but also on our own reverse engineering efforts for Widevine EME message format, security validation, temporal checks, and the management of data within the CDM memory. We also study the CDM behavior when errors occur during an open session.

3.2 CDM Internals

We present the different stages and operations of the Widevine CDM, from the initial state setup to license acquisition and renewal process. A summary of message contents and memory internals are presented in Table 2 and Table 3 of Appendix A.

3.2.1 Initial State

The Widevine CDM is provisioned with a pair of RSA keys, called the Device RSA Key, either in a hardcoded form like in desktop implementations, or on the fly through a provisioning mechanism [33]. With it, the CDM composes a Client ID from the Device RSA Key public part combined with device metadata [34]. Additionally, it possesses a service certificate public key obtained from the remote license server (via `setServerCertificate`), which holds the corresponding private key. Once the EME workflow has started and the necessary information has been retrieved from the license server, a session within the CDM can be initiated, setting the stage for license acquisition.

3.2.2 Acquire Licenses

Client Request. Following a call to `generateRequest`, the CDM creates a nonce in its memory to serve as an anti-replay measure. Using the public key within the service certificate from the license server, the CDM encrypts a freshly generated privacy key, which is used to encrypt, for privacy purposes (as shown in [34]), the Client ID containing the public Device RSA Key. Alongside these elements, the request also includes a randomly generated Request ID, the desired Key ID (KID for short) extracted from the media metadata and used by the license server to retrieve the corresponding key, and the request time T1 which is stored in the CDM memory. The

[1] `https://proxy.staging.widevine.com/proxy`

whole request is signed using the Device RSA private key of the device and then sent to the license server. Three elements are stored in a session memory of the CDM: the nonce, T1, and the whole request. Our assumptions on server-side request processing are discussed in Section 3.2.4.

License Format. The license consists of two main parts. The first part contains the encryption by the Device RSA public key of a fresh Session Key. Three keys are derived from that Session Key: the Asset Key, the MAC Server Key and the MAC Client Key. The derivation process, detailed in [33], involves some constant values as well as the client request. The derived keys are used to protect the second part of the license, in which we find the previous Request ID, the request time, KID and the license usage rights called policies. Among others, the policies define the license duration Δ_1. Also included in this part are the decryption key, called Content Key, and its KCB (Key Control Block). The KCB contains the request nonce, and is encrypted by the Content Key, which is encrypted by the Asset Key. The integrity of the second part is protected with an HMAC using the MAC Server Key.

License Processing. Upon receiving the license response, the CDM acts in two phases. In the first phase, it decrypts the Session Key using its Device RSA private key, derives the three keys from this Session Key, and stores them in its memory. In the second phase, it verifies the HMAC of the received license and checks the equality of the stored T_1 and the one received in the response. The CDM first retrieves the Content Key, and then decrypts the KCB to extract the nonce. It then checks the nonce, and removes it from its memory. After successful verification, the CDM computes the license expiration time T1 + Δ_1. The CDM loads the Content Key in its memory for use. During our tests, we observed that any failed attempt of a future load does not invalidate the current Content Key in memory; media decryption can still happen.

3.2.3 Renew Licenses

Client Request. If the loaded policies allow the CDM to update its Content Key, an event of type license-renewal is triggered by the CDM. On renewal request creation, a newly generated privacy key is used to encrypt the Client ID as in the original request. Alongside them, the initial Request ID and request time T1 are used with a renewal request time T2, a counter C set to 0, or to the current counter in memory if it is not the first renewal request. The whole request is protected by an HMAC using the MAC Client Key. The CDM may generate multiple queries for the same renewal event identified by the same counter, but with different time. Indeed, EME events and renewal load might be asynchronous, and therefore multiple renewal queries may have been emitted before any successful load. Different queries may point to the same event if they all include the same identifier, *i.e.* counter.

License Format. The renewal license includes the Request ID, T1 and T2, an incremented counter $C = C + 1$, and the updated policies with the new duration time Δ_2. The license is protected with an HMAC using the MAC Server Key.

License Processing. The CDM can now update its Content Key. If the HMAC validation is successful, the new counter value replaces the existing one in memory, only if it is strictly greater than the former value; otherwise, the key update is unsuccessful. The expiration time of the Content Key is updated to T2 + Δ_2 + (T2' - T2). Here, T2' denotes the time indicated in the license response. Since multiple renewal queries with varying T2 values may have been sent, the received T2 might not correspond to the most recent one stored in the CDM.

3.2.4 Other Considerations

Different CDM Instances. Widevine provides different CDM implementations. Here, we highlight one in particular. In Android, during renewal, the KCB of the loaded Content Key is sent through the Client Request. Thus, a nonce is generated by the CDM, that checks its presence in the response as in the original request.

Minimum Viable License Server. In our study, we did not reverse engineer license servers, because we did not have any direct access to them. Indeed, we only experimented with the Widevine test license server to avoid legal issues. Here, we describe what we expect from a license server regarding the different EME stages, in conformance with what was observed from the test server. We assume that all values expected to be random are generated thusly. In addition, during license acquisition, the license server verifies the signature of the client request, and verifies that the Client ID was issued by some trusted certification authority controlled by Widevine. It also stores stateful information about the received license request: Request ID and derived session keys. As for license renewal, the license server verifies the HMAC of the client request, constructs the response, and does not store any state about it. For both cases, we suppose that the license server always replies to a request with a valid format without considering the semantics of the different fields. For instance, we assume that the license server would reply to valid queries with a valid license, even if the request contains times in the future, or a series of renewal requests that do not start with counter 0. This allows us to expand the scope of our work by analyzing the Widevine CDM implementation regardless of any additional non-standardized security mechanisms enforced by proprietary license servers.

3.3 Potential Attack

During our formal analysis of the Widevine EME (see Sections 5 and 6), we found one critical vulnerability which we describe in the following (see Figure 2), along with suggestions on how to fix it.

Attack Requirement. The attack exploits an understated detail while processing licenses of type license-request. Recall

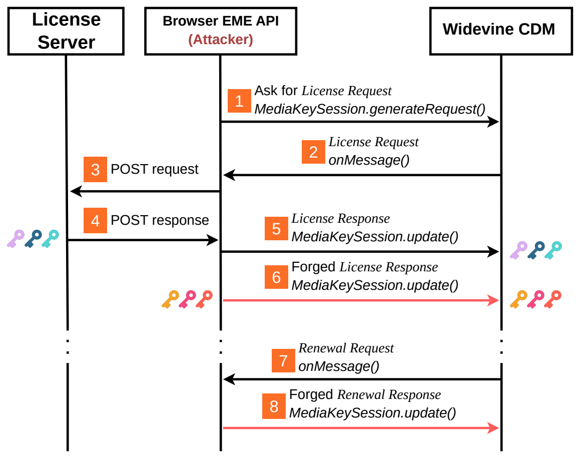
License Server
Browser EME API (Attacker)
Widevine CDM
1 Ask for License Request MediaKeySession.generateRequest()
2 License Request onMessage()
3 POST request
4 POST response
5 License Response MediaKeySession.update()
6 Forged License Response MediaKeySession.update()
7 Renewal Request onMessage()
8 Forged Renewal Response MediaKeySession.update()

Figure 2: Widevine EME Key Derivation Attack.

that the CDM acts in two phases to load the Content Key and its associated policies. The attack requires that all modifications to the CDM memory during the first phase persist even if the second phase fails. For instance, if a license contains an invalid HMAC tag, and a valid session key encrypted with a valid Device RSA public key, then the derivation process (first phase) succeeds and the newly derived keys are stored in the memory. However, the second phase aborts as the HMAC verification does not succeed. It is important to note that Widevine CDM implementations may behave otherwise. Our hypothesis is supported by the interface of the Android CDM, where, according to [33], the load process involves two different calls: `DeriveKeysFromSessionKey`, followed by `LoadKeys`. However, we did not implement the attack to avoid any legal issues.

Attack Description. The objective of this attack is to allow the attacker, that is a subscribed user, to get more than what they pay for from the content provider, which is the victim. In technical terms, the attacker would like to load arbitrary policies, and hence an indefinitely long expiration time, for any Content Key that can be renewed at least once. To do this, the attacker does not need more than intercepting and making EME calls in the browser (via a plugin for example), on the attacker's device.

The attack scenario consists of three stages. The first stage is to observe the EME inputs and outputs during a successful license acquisition. Of particular interest, the attacker needs the client request, the Request ID and the Device RSA public key. This corresponds to steps $\boxed{1}$ to $\boxed{5}$ in Figure 2. In the second stage, the attacker forges a new license that attempts to load as a license of type license-request. This license contains a valid encryption of an attacker-generated session key, and some gibberish other values. As shown in step $\boxed{6}$, the attacker calls the `update` API, and thus overwrites the Asset and MAC keys of the opened session with other keys that they control. The third stage is to wait for a renewal request that the attacker stops in step $\boxed{7}$. Then, in step $\boxed{8}$, the attacker forges a new license of type license-renewal leveraging the freshly modified keys. The attacker can, for instance, include any expiration time. The CDM would accept this license, since the attacker knows the derived keys. A similar attack is found even when renewal licenses contain encrypted KCBs.

Attack Mitigation. There are several ways to address this vulnerability, *e.g.* one could force all CDM implementations to commit their memory only when all the license acquisition process succeeds. This solution might not be straightforward to implement, especially when this may involve several API calls for modularity reasons. More broadly, we did not opt for any solution that requires subtle modifications in the behavior of Widevine CDMs. Instead, we suggest the following fix including two modifications. The first one concerns the EME specifications; we propose that an opened session is bound to a read-only server certificate that cannot be changed. Any subsequent call to `setServerCertificate` only impacts future sessions. The second modification concerns the format of license of type license-request. We propose that the license be signed by the private key of the server certificate. Thus, the CDM solely accepts licenses from the same entity for which it has generated the license request. Note that our mitigation suggests making use of the same RSA key pair for both signing and decryption in order to reduce storage requirements and costs for key certification. This practice does not come without risk, and can be a source of insecurity depending on the used signature/encryption schemes [10]. The Widevine CDM uses OAEP for encryption and PSS for signature, as found by Patat *et al.* [33]. Haber and Pinkas have shown that these two schemes are jointly secure; in other words, they can be securely combined using the same key without compromising their security [18].

4 EME Security Requirements

Despite not affecting all implementations of the Widevine CDM, our vulnerability demonstrates how brittle the security of a popular EME instantiation can be. This may be partly explained by the fact that the security objectives of EME are not well understood, or even clearly stated. The EME standard [43] is actually rather brief when it comes to security. It shortly introduces three potential attack vectors: input validation, CDM update frequency, and network. Further security specifications are omitted on the ground that *requirements for security cannot be met without the knowledge of the security and privacy properties of the Key System and its implementation(s)* [43]. In other words, the EME standard suggests that there is no, or very little, common security guarantees for EME to refer to without considering specific CDM implementations. This point of view is unfortunate, for two reasons: (1) the essence of EME is to ensure content security, and a common model might help future DRM systems; (2) a major part of EME concerns the rights enforcement layer, whose security *has* already been discussed, in [12, 37].

In what follows, we propose some core security properties for EME, when considered as an enforcement layer. Then, we present a high-level description of seven security goals we define, covering all aspects of the EME workflow. We will use these goals later on to perform a complete computer-aided formal verification of the Widevine EME protocol.

As already mentioned in the introduction, the main goal of EME is the interoperability between DRMs. Thus, EME contains unspecified message formats whose instantiations are DRM specific. Even if the exact content of these messages is necessary to perform a complete security analysis, the security goals presented in this section are generic and reusable to continue analyzing other DRMs, e.g. PlayReady or FairPlay.

4.1 EME as an Enforcement Layer

EME defines an enforcement layer that works as follows. The license server encapsulates the secret decryption key (obtained from the CDN), also called the *Content Key* in the Widevine jargon, with the associated usage rights, or *policies* (from the content provider), into a license. This license is forwarded to the CDM that first checks the usage rights before loading the key into its memory for content decryption.

According to [37], the rights enforcement layer has two main roles. First, it protects the usage rights associated with content against tampering: an attacker should not be able to modify the usage rights. Second, it guarantees that usage rights cannot be bypassed: an attacker should not be able to use the content in a way not authorized by the usage rights. The actual enforcement of licenses is proprietary, and rarely studied in literature. Instead, we suggest four main properties:

1. Confidentiality of the decryption key, so that an attacker cannot obtain a digital copy of the content outside the CDM.

2. Integrity of the usage rights, so that an attacker cannot alter the usage rights delivered by the content provider. For instance, an attacker could not extend their rights by altering the license expiration time.

3. Authenticity of the usage rights, so that the CDM enforces the rights strictly as they only were defined by the party related to the owner of the decryption key.

4. Freshness of the license, so that the CDM can only use it once, regardless of its type: license-request or license-renewal.

In the next subsection, we express and expand the above properties as 7 precise security goals. Loosely, property 1 maps to goal 1, properties 2 and 3 map to the two goals 2 and 5, and property 4 maps to goals 3 and 6. As for goals 4 and 7, they are defined to ensure that content will be used only under the conditions defined by the usage rights, notably the expiration time. Strictly speaking, this aspect is part of the rights management layer, but we still decided to consider it, since the EME standard defines an abstraction of the session expiration time.

4.2 EME Security Goals

Recall that the essence of DRM systems is to allow content providers to protect their media from *piracy*. A naive definition for protection from piracy would be: preventing any media from being illegally distributed outside the DRM ecosystem. Here, we go further this vague definition, and propose several security goals depicting EME as an enforcement layer. These goals, although stated in natural language, have a clear meaning and will be formalized in the next sections.

The EME protocol does not deal with trust issues between the content provider, the license server and the CDN. Therefore, our security goals assume all three are trustworthy entities communicating securely. Henceforth, we use the generic term "Over-the-Top" (OTT) platform to refer to them. Decryption keys are not shared between different OTTs. We also assume a trusted authority issuing certificates to *trusted* OTTs and *trusted* CDM modules. Thus, our security goals are stated against an attacker who does not carry any valid certificate, whether acting as an OTT or a CDM. We also assume that the attacker cannot tamper with the CPU clock used by the CDM.

Goal 1. Confidentiality of the decryption key.

The decryption key remains secret.

Only trusted CDMs can acquire the decryption key of media content. Thus, assuming the encryption algorithm used is secure, media contents cannot be obtained by an attacker.

Goal 2. Integrity and authenticity of initial licenses.

The CDM must load licenses of type license-request *as they were generated* by the issuing entity.

This goal implies integrity of both the decryption key and the associated usage rights. It also implies that the CDM accepts licenses only if the included usage rights come from the same entity that owns the decryption key.

Goal 3. Freshness of initial licenses.

A given license of type license-request can be loaded at most once, and only by the CDM generating the corresponding request.

In the EME API, an OTT issues a license of type license-request to reply to a query of type `generateRequest`. This goal mandates that the generated license can only be loaded by the trusted CDM creating the `generateRequest`. It also mandates that a load can only happen once for a given license.

Otherwise, an obvious attack would be to replay licenses to the CDM in an offline mode without validation from the OTT.

Goal 4. Enforcing expiration time of initial licenses.

Before any renewal, the CDM is able to use the decryption key at a given time `T` only if the expiration time specified by the OTT in the initial license is greater than `T`.

The EME standard expects that all licenses are associated with an expiration time. In section 2 of [14], the standard states that "*a key is not usable for decryption if its license has expired*".

Goal 5. Integrity and authenticity of renewal licenses.

The CDM must load licenses of type license-renewal *as they were generated* by the issuing entity.

This goal implies integrity of the included usage rights. It also implies that the CDM accepts license-renewals only if they come from the same entity that issued the associated license of type license-request. In other words, only the OTT owning the decryption key loaded by a CDM can update its usage rights.

Goal 6. Freshness of renewal licenses.

A given license of type license-renewal can only be loaded once after a renewal event, and only by the CDM generating the corresponding request. This must hold even if multiple renewal licenses corresponding to the same renewal event have been emitted.

Similar to Goal 3, this goal mandates that the generated license can only be loaded by the trusted CDM triggering the event for license-renewal. Recall that a renewal event may generate multiple queries all containing the same unique identifier; a counter in the case of Widevine.

Goal 7. Enforcing expiration time of renewal licenses.

After a renewal, the CDM is able to use the decryption key at a given time `T` only if the expiration time specified by the OTT in a renewal license is greater than `T`.

This goal is the counterpart of Goal 4. Note that loading a renewal license on a CDM does not always extend expiration times; it may instead shorten them. The OTT, however, cannot force the CDM to actually load a (potentially shortening) renewal licence, and only knows it has been sent.

4.3 Threat Model

It should be noted that the threat model for DRM is quite different from the traditional threat model for Alice-Bob protocols. The traditional model often assumes that Alice and Bob trust each other. For DRM systems, the threat model cannot assume this symmetry of trust. In most cases, Alice, the content provider, cannot trust Bob, the content user. In fact, Alice cannot distinguish an honest user from a dishonest one. Even if Bob is honest, it does not mean that his computer has not been tampered with. This asymmetry explains why content providers, or license servers, only trust Bob's CDM, and not his browser, *i.e.* his EME user-agent.

Thus, our threat model involves an attacker that can inject calls to the EME API into any page. In addition, calls to the EME API from any page can be intercepted, thereby modifying or injecting arbitrary data in the parameters of method calls. This corresponds to the network attacks as defined in section 10.3 in [14]. We exclude all software-based attacks, based on input sanitization and CDM vulnerabilities. This is due to the fact that all DRM systems can be trivially broken, provided a CDM with software vulnerabilities. Our paper focuses on finding weaknesses on the EME protocol itself, assuming a secure CDM implementation. For instance, the attacker can be a user with valid subscription that installs a malicious browser plugin controlling all calls to EME to extend their rights, even after the end of their subscription.

5 Formal Verification using TAMARIN

We analyzed the Widevine protocol using the protocol verification tool TAMARIN, to determine whether the security goals detailed in Section 4 are satisfied. In this section, after a brief introduction to TAMARIN, we explain our TAMARIN models as well as the difficulties we have encountered in providing a faithful representation of some parts of the API under study.

5.1 TAMARIN in a Nutshell

The TAMARIN prover [28] is a security protocol verification tool that supports both falsification and unbounded verification in the symbolic model. As usual in this setting, messages are represented by terms, and the cryptography is assumed to be perfect, *e.g.* nonces are unguessable, and recovering the content of a ciphertext is only possible when knowing the key. Security protocols are specified as multiset rewriting systems, which makes the tool particularly suitable to model stateful protocols such as APIs for which some contents (*e.g.* key material, counter values) are stored over a long period. This aspect is well-known to be difficult to handle, and despite some recent advances made in PROVERIF [4,5], another well-known symbolic protocol verifier, TAMARIN seemed to be better suited to our case study. In particular, a recent extension introduces subterm-based proof techniques [9], which helps dealing with counters, making it particularly attractive.

TAMARIN relies on multiset rewriting, a formalism that is commonly used to model concurrent systems, since it naturally supports independent transitions. The state of the system, *i.e.* the contents of each agent's memory, the network, and the attacker's knowledge, is globally modeled by a multiset of *facts*. Each fact is a name and a sequence of terms representing data. For instance, in our models, a fact `!MovieGen($title, ~movie)` stores the information that a `~movie` has been generated, with its `$title` – in TAMARIN, values prefixed with `$` represent publicly known values, and values prefixed with `~` are fresh values, that are *a priori* secret.

A rewrite rule in TAMARIN has a name, *e.g.* `GenMovie`, and three components, each of which is a multiset of facts: the rule's left- and right-hand sides, and its labels. Executing this rule consumes (from the global fact multiset) the facts on the left, produces the facts on the right, and adds the label to the execution trace. Facts prefixed with `!` are *persistent*, *i.e.* they are not consumed, and can thus never be deleted or changed. For instance, the following rule `GenMovie` generates new `!MovieGen` facts with different movies, obtained using a built-in `Fr` fact that produces fresh values.

```
rule GenMovie:  [ Fr(~movie) ]
              –[ LMovieGen($title, ~movie) ]→
               [ !MovieGen($title, ~movie) ]
```

TAMARIN analyzes protocols in the so-called Dolev-Yao model, where the attacker is able to receive, intercept, modify, and forge messages. A built-in fact `K(m)` records that the attacker knows message `m`. TAMARIN has built-in rules for the attacker, that let them perform computations on messages they know, such as encrypting when they know both the plaintext and the key. Two built-in facts, `In(m)` and `Out(m)`, let protocol rules input and output messages from and to the network, which is supposed controlled by the attacker.

Once the protocol is modeled, we express security properties using *lemmas*. These are first-order formulas, written using the facts that label the rewriting rules. They can express properties of the protocol's traces. Execution steps in a trace are identified by indices `#i`, prefixed with `#`. For instance, the lemma below states that for all traces where a `movie` has been generated at step `#i` (label `LMovieGen`, from rule `GenMovie`), at no point `#j` in the trace does the attacker know `movie`. In other words, this expresses that movies are never known by the attacker.

```
lemma movieSecrecy: "∀ #i title movie.
   LMovieGen(title, movie)@#i ⇒ ¬ (∃ #j. K(movie)@#j)"
```

Restrictions are another useful feature of TAMARIN. They are formulas similar to lemmas, except that instead of proof obligations, they are assumptions, *i.e.* they tell TAMARIN to restrict its analysis to traces where they hold. As an example, we may use a restriction to model the fact that the OTT, when renewing a license, may attach a policy to it, marking it as further renewable or not. Assuming that the corresponding rule modeling the OTT features a fact `License`, where the policy field is left unspecified, we can write a restriction to enforce that only traces where the policy is instantiated by `'renewable'` or `'stop'` will be considered in the analysis.

```
restriction LicenseRenewOrStop:
"∀ #i l. License(l)@#i ⇒(l = 'renewable' | l = 'stop')"
```

When provided with a lemma, TAMARIN attempts to prove it, or to find a trace that contradicts it. This trace contains the sequence of rules that are applied and the actions the attacker took to violate the property. As the proof search is undecidable, TAMARIN may also not terminate. In that case, it is possible to use an interactive mode and to prove the property manually by guiding TAMARIN regarding the states to explore. Once the proof is done, or an attack is found manually, it is possible to store it, or to write an oracle that guides TAMARIN during its exploration. The idea is that, even if the security analysis requires some manual guidance at first, it is then possible to reproduce the result automatically.

5.2 Our TAMARIN Models

As explained in the previous section, one difficulty was to write the models themselves. In the absence of a clear, detailed specification, the reverse engineering stage gradually made it possible to clarify various points and obtain a representation as faithful as possible of the protocol to be studied.

We model four variants of Widevine: we analyze both the Android version, where the KCB is part of the renewal request, and the desktop version, where it is not. For each version, we consider both the actual protocol and the patched version, which includes the mitigation we propose. These four TAMARIN studies are rather similar regarding the protocol models, as they share a common basis, with only minor changes to account for the presence of the KCB and patch.

Our model of the Widevine protocol is composed of 18 multiset rewriting rules in total, divided in three sets: 6 rules for the initialization of pre-existing data (movies, content keys, public key infrastructure, *etc.*), 8 rules for the role of the CDM, and 4 rules for the role of the OTT. These rules model an unbounded number of instances of CDMs and OTTs running the protocol in parallel.

As an example, the following rule `CDMGenerateRequest` (simplified for legibility) models the generation of a license request by a CDM.

```
rule CDMGenerateRequest:
  let request = <~rID, ~keyID, %t,
                 enc(clientID, ~kPrivacy),
                 aenc(~kPrivacy, pk(~kOTT)), ~n>
      signReq = sign(request, kDevice)
  in
  [ In(%t), In(~keyID), Fr(~rID), Fr(~n), Fr(~kPrivacy),
   !OTTKey(~kOTT), !CDMSession(~sID, kDevice, clientID)]
 –[ GTime(%t), LCDMGenR(~requestID, ~sID, %t) ]→
  [ Out(<request, signReq>), CDMNonce(~n, ~sID),
   !CDMState(~rID, ~sID, ~kOTT, request, %t),
    CDMKeys(~rID, ~sID, ∅),
    CDMContentKey(~rID, ~sID, ∅, %1) ]
```

In this rule, a CDM session, with a session state `!CDMSession(~sID, kDevice, clientID)` containing a session identifier, a device key, and a client ID, generates a random request ID `~rID`, a nonce `~n` and a privacy key `~kPrivacy`. It uses these values to compute the license request, and signs it with its device key. It outputs the resulting message `<request, signReq>`, and stores all relevant data in facts **`CDMNonce`**, **`!CDMState`**, **`CDMKeys`**, **`CDMContentKeys`**. The role of these facts is further discussed in Section 5.4. The rule is labeled with fact **`LCDMGenR`**, recording the request generation. Note that the request contains a value **`%t`**, representing the current date, also included in an input and in label **`GTime`**: these are used as part of our model of time, which we discuss in Section 5.5.

5.3 Main Assumptions and Limitations

It is worth discussing the faithfulness of our models and analysis. We have attempted to construct a formal model of EME/Widevine that is as close to the real protocol as possible. However, a gap can occur between our model and reality.

Indeed, this mainly comes from the gap between our description of the CDM and its actual implementation. The EME specification, combined with the reverse engineering we performed, gives us a description of the behavior of the CDM. This description is fairly precise – going back and forth between the reverse engineering and formal modeling allowed us to extensively test the CDM to refine it, as described in Section 3.1. Still, we cannot be certain that we perfectly described the CDM: ultimately, we only know about the behaviors we could observe, and there could in principle exist some behaviors of the CDM that are possible but that we were unable to trigger.

Below, we discuss two particular modeling assumptions we made in our Tamarin analysis.

First, when loading a license response, the counter kept by the CDM is in a normal execution still at its initial value (zero). Indeed, in a normal, honest execution of the protocol, only one load is performed in any given CDM session. It is however unclear how that counter would be handled if an attacker was somehow able to cause a second "load" to be successfully executed in a session. We have not been able to test which behavior is implemented in practice, which would have required us to simulate an OTT sending several license responses. We therefore felt that the reasonable way to proceed in our analysis was not to make an arbitrary choice, but rather to discard executions where two loads are successfully performed in the same CDM session. We express this assumption in TAMARIN as a restriction.

It is of note that this restriction is actually in line with the most recent draft of the EME standard [32], which imposes that only one license request is sent by the CDM per session. We were in fact able to prove that, for the desktop version of the CDM, the unicity of the license request implies the unicity of the nonce, and in turn of the license load, per CDM session, *i.e.* to lift that assumption. In the Android version, however, the same argument cannot be made. Since a nonce is also generated for any license renewal request by the CDM to populate the KCB, the unicity of the nonce per session, from which that of the load follows, does not hold. Thus, we kept the unique load assumption for that version of the protocol.

Second, in some implementations, when receiving a license response, a single CDM API function call performs both the derivation of Asset and MAC keys, and then the loading of the license. Providing this function with badly constructed parameters may result in the CDM deriving and storing the keys, and then failing to load the license. That behavior was observed while reverse engineering some CDM implementations. To account for this scenario, we chose in our model to split the handling of a license response into two rules, one for deriving the keys, and the second for loading the license. We however do not require the loading to be performed immediately after the derivation – such restrictions are not easily dealt with in TAMARIN. This gives the attacker slightly more power than is actually possible. That is not an issue: the attack we found does not use this additional power, and we still could prove secure the patched versions of the protocol.

Regarding the OTT, as already mentioned in Section 3, we did not reverse engineer its behavior, and therefore we modeled a "minimal" OTT. Note that the attack we discovered in our analysis does not depend on internal or unspecified behaviors of the OTT: it can be performed with any OTT that replies to normal protocol messages as expected.

5.4 A Stateful API Manipulating Counters

Although EME prescribes the flow of messages between the CDM and OTT in a normal execution of the protocol, the related API provides the browser with methods that can in fact be called in any order. To accurately represent this behavior, we do not impose a sequential order for the rules in our TAMARIN models: any CDM rule can be executed at any point, and updates the CDM's internal state accordingly. This state is modeled using facts that store the relevant information – persistent facts, for data that is set once and never updated, and non-persistent facts for data that can be updated later on. For instance, in rule **`CDMGenerateRequest`** described earlier, a persistent fact **`!CDMState`** is created to store (among other data) the `~rID` and the initial license request, which are stored until the end of a session and never updated. On the other hand, **`CDMNonce`** and **`CDMContentKey`** are linear facts. They are used to store resp. the nonce sent in a license request, which can later be deleted when loading the matching response, and the memory cell used to store the content key and license policy, which are initially `'null'` and can later be updated when loading (renewal) responses. This is rather standard practice when modeling stateful protocols and APIs. It does however complicate reasoning, compared to stateless protocols, as it required us to show invariants on the states, to help TAMARIN in its backwards search.

A notable feature of the CDM state is the presence of a counter identifying an event of license renewal. This counter is incremented each time a renewal response is successfully loaded. We rely on a recent extension of TAMARIN [9], that is specifically designed to reason about increasing counters.

5.5 The Crucial Role of Time

A major feature of our TAMARIN analysis is the modeling of time. Widevine messages for license and renewal requests include timestamps indicating the date at which requests are generated, and responses contain a time-to-live indicating the duration of the license's validity. TAMARIN has no built-in notion of time – only an ordering of events in an execution trace, which protocol messages do not have access to.

The indices available in TAMARIN are useful to reason about the order of actions, but they cannot express durations, they cannot be used to compute values (e.g. the sum of two indices is meaningless), and they cannot be used as timestamps in messages (an agent sending the index of its action makes no sense). It is sometimes possible to abstract time away, and to write properties at the level of indices. This is usually done for instance when modelling authentication properties stating that when an agent performs an action at $\#i$, then another one has performed a specific action at $\#j$ with $\#j < \#i$.

For that reason, timestamps are usually abstracted away (or simply not modelled at all) when analyzing protocols in TAMARIN. However, in our case some security goals rely heavily on time, and we did not see a way to abstract it. Indeed, timestamps are essential to the security of the system, as they are required to state Goals 4 and 7 (*cf.* Section 4). For that reason, we propose an explicit model of time, accessible to protocol agents, which is to our knowledge the first in TAMARIN.

We model time as a global clock, *i.e.* an integer counter `%t`, representing the current date. Each protocol rule receives, by an input `In(%t)`, the current time from the attacker. That value `%t` can then be used and referred to when writing security lemmas. We let the attacker pick the clock value each rule receives, modeling the fact that they can decide to wait as long as they want before triggering the execution of a rule. We force them however to choose these values in a consistent way: time only goes forward. To do so, we label each rule with an event fact `GTime(%t)`, recording the time at which it is executed, and we consider the following restriction:

```
restriction TimeIncreasing: "∀ #i #j %t1 %t2.
 GTime(%t1)@#i & GTime(%t2)@#j & #i<#j ⇒ %t1<<%t2"
```

We can then use this global time to specify for instance that the CDM only uses a content key (using a rule not depicted here, labeled with fact `LCDMUseKey`) at time `%t` if the expiration date `%t0 %+ %delta` for the license, stored in its state (fact `LCDMContentKey`), has not passed yet. This is expressed by the following restriction (simplified, for clarity):

```
restriction UseKeyLegal:
  "∀ #i requestID sID  k %t %t0 %delta.
    LCDMUseKey(rID, sID, k)@#i
      & LCDMContentKey(rID, sID, k, %t0, %delta)@#i
      & GTime(%t)@#i
   ⇒  %t << %t0 %+ %delta"
```

The reasoning on time is again facilitated by the recent TAMARIN extension for counters [9]. In order to obtain better performance from TAMARIN, we actually only add the timer input and label to the protocol rules that effectively use the current time. Doing so produces the same executions, but greatly reduces the number of `GTime(%t)` facts TAMARIN has to consider, only keeping those that are really useful.

6 Results of the Formal Analysis

We describe here how we proceed to analyze the models presented in Section 5.2. We first explain our methodology in Section 6.1 before summarizing our findings in Section 6.2. In Section 6.3, we give an overview of how TAMARIN reasons by providing some details about the proofs of Goal 2.

6.1 Our Methodology

Initially, we focused on obtaining a faithful model of the protocol in its desktop version. In a second stage, we considered the model for the Android version (*i.e.* with a KCB in the license renewal requests). This only causes some minor changes in our security analysis. We summarize some hands-on details about our TAMARIN model and how to execute it in Appendix B. For each model, we started with some sanity checks to ascertain that models were correct. This is the purpose of the "executability lemma" available in each model. These TAMARIN lemmas express the fact that the normal intended protocol flow can be executed in our models. These lemmas can be found in each of our Tamarin files and are systematically checked before performing the security analysis. While this reduces the risk that modelling mistakes were made, it does not guarantee that our models exactly correspond to the implementation. Though unfortunate, this situation is common when formally verifying protocols.

Then, the security analysis is carried out in 3 stages:

Some basic secrecy properties (including Goal 1). We first state some lemmas to establish secrecy of various keys used in the protocol, in particular the content keys used to protect movies, and the session keys generated by OTTs. These basic properties were easy to prove for TAMARIN, and their proofs were obtained automatically. All these lemmas are marked with flag `[reuse]`, to let TAMARIN know they can and should be used when proving the other security goals.

Analysis of the initial part (i.e. Goals 2, 3, and 4). This part requires the writing of some specific intermediate lemmas, especially to reason about the content of some memory cells, *e.g.* those used to store the Asset and the MAC keys.

Analysis of the renewal part (i.e. Goals 5, 6, and 7). This part is by far the most involved. When trying to prove Goal 5, TAMARIN led us to the derive attack we presented in Section 3.3. Then, once this issue was resolved by applying the proposed patch, we applied the same methodology as the one for analyzing the initial part.

License policies. When performing a security analysis of a DRM system, it makes sense to check that license policies are respected, *i.e.* that the CDM can load a renewal license only if this is authorized by the policy. This goal cannot be stated at the EME level, as policies are not part of the EME specification. However, we can express it when analyzing Widevine EME, and we therefore decided to consider this extra goal, which we call Goal 8, in our security analysis.

6.2 Summary of our Results

Our findings are summarized in Table 1. While most of the basic secrecy properties were obtained automatically using the autoprove mode of TAMARIN, the proofs of Goals 2 to 8 required some intermediate lemmas. These proofs have thus been obtained at first using the interactive mode of TAMARIN. For reproducibility purposes, we then automated them, relying on the oracle mechanism available in TAMARIN.

	without fix		with fix	
KCB	with	without	with	without
Goal 1	✓	✓	✓	✓
Goal 2	✓	✓	✓	✓
Goal 3	✓	✓	✓	✓
Goal 4	✓	✓	✓	✓
Goal 5	✗	✗	✓	✓
Goal 6	✓	✓	✓	✓
Goal 7	✗	✗	✓	✓
Goal 8	✗	✗	✓	✓

Table 1: Summary of our results. ✓ indicates that the goal is proved automatically by TAMARIN, whereas ✗ means that the goal is false (variants of the derive attack).

All our experiments were performed on a laptop with a 2.30GHz Intel Core i7-1068NG7 CPU and 16GB of RAM. We use version 1.8.0 of TAMARIN. Our models are available for reproducibility purposes [11]. The verification time varies depending on the model, and the goals we are trying to establish. The easiest lemmas are proved in a few seconds, whereas the most difficult ones, *i.e.* Goals 5-8 of the patched Android version, require around 4 minutes.

6.3 Proving the Security Goals

Even if the proofs are done *in fine* in an automatic way, some expertise and manual effort were necessary to obtain them. As an illustration, we present the proof of Goals 2 and 5.

```
lemma Goal2:
"∀ #i1 #i2 rID sID k kAsset kMacS kMacC resp
      title movie ottID keyID.
 Load(rID, sID, k, kAsset, kMacS, kMacC, resp)@#i1
 & Movie(title, movie, ottID, keyID, k)@#i2
⇒ (∃ #j.   #j < #i1
 & OTTLR(rID, ottID, k, kAsset, kMacS, kMacC, resp)@#j)"
```

This lemma states that when the CDM loads a license `resp` for a movie with content key `k`, this license has been generated by an OTT before, *i.e.* at some step `#j` occurring before `#i1`.

The proof of `Goal2` relies on two secrecy lemmas. One establishes the secrecy of the session keys generated by an honest OTT, and the other is actually `Goal1`: secrecy of content keys. We also need an additional lemma, stating that when a command uses the Asset or MAC keys, then these keys have been initialized earlier by a call to the derive command, unless there are still `'null'`.

```
lemma CDMKeysInit[reuse, use_induction]:
  "∀ #i rID sID kAsset kMacS kMacC.
       Keys(rID, sID, kAsset, kMacS, kMacC)@#i
 ⇒ ( (∃ #j. (#j < #i)
        & Derive(rID, sID, kAsset, kMacS, kMacC)@#j)
    | (kAsset ='null' & kMacS = 'null' & kMacC = 'null'))"
```

This lemma, although not surprising, is not obvious to TAMARIN, and requires a proof by induction, as indicated by the flag `[use_induction]`. The induction step holds, as the memory cells storing those keys are either left unchanged, in most rules, or indeed initialized, in the derive rule.

Using these three lemmas, we are then able to conduct the proof of `Goal2` with TAMARIN. When a load command is successfully executed, the MAC received by the CDM:

- *(i)* either comes from the protocol;
- *(ii)* or has been forged by the attacker.

In the former case, due to the format of the received MAC, it has necessarily been emitted by a license-response command, and thus the claim holds. Indeed, in this case, the agreement on the different keys is clear, as the content key appears in the exchanged message. In the second case *(ii)*, the `CDMKeysInit` lemma tells us that the Asset and the MAC keys are:

- *(ii.a)* either `'null'`;
- *(ii.b)* or have been initialized by a derive command.

In case *(ii.a)*, as the load command also needs a message of the form `senc(k, kAsset)` in input, the branch can be closed: if `kAsset` was equal to `'null'`, then the secrecy of the content key `k` would be violated.

In case *(ii.b)*, the derive rule tells us that the MAC key is `KDF(..., kSession)`. As no rule produces such a term at an extractable/deducible position, the attacker must have built it using their own `kSession`. In that case, the message `senc(k, kAsset)` with `kAsset=KDF(...,kSession)`, needed in input of the Load rule, can either be produced by the OTT, or constructed by the attacker. The former case contradicts the fact that the `kSession` keys generated by the OTT are secret (the aforementioned secrecy lemma). The latter case implies that the attacker knows the content key `k`, leading again to a contradiction.

Implementing this proof strategy in an oracle, we obtain the following result, indicating that TAMARIN succeeds in proving all the lemmas. It also gives us the number of reasoning steps done by TAMARIN for each proof.

```
==================================================
SecrecykSession (all-traces): verified (25 steps)
Goal1 (all-traces): verified (4 steps)
CDMKeysInit (all-traces): verified (43 steps)
Goal2 (all-traces): verified (10 steps)
==================================================
```

Goal 5. The statement for `Goal5` is similar to `Goal2`, except that it concerns the renewal part.

```
lemma Goal5:
  "∀ #i1 #i2 rID sID k kAsset kMacS kMacC
       resp title movie ottID keyID.
  LoadRenew(rID,sID,k,kAsset,kMacS,kMacC,resp)@#i1
  & Movie(title, movie, ottID, keyID, k)@#i2
 ⇒  (∃ #j. #j < #i1
  & OTTRR(rID, ottID, k, kAsset, kMacS, kMacC, resp)@#j)"
```

Regarding its proof, things are a bit different. Actually, we have already seen that `Goal5` does not hold, because of the derive attack (*cf.* Section 3.3). We would like to pinpoint exactly where the previous reasoning fails. Basically, the issue comes from the fact that the content key `k` is not part of the response processed by the CDM, and thus to ensure agreement on it, more work is needed even in case *(i)*. The case *(ii.a)* and *(ii.b)* are also more complex for the same reason. We can *not* reason on the message `senc(k, kAsset)`, which is not present in a renewal response. Case *(ii.a)* can be handled by showing that `kAsset = 'null'` implies that the memory cell was not initialized, which is actually not possible (this requires some extra intermediate lemmas). Case *(ii.b)* is more problematic and leads to the derive attack. A way to prove this case would be to ensure that the `kSession` key as seen by the CDM is secret, and thus the attacker is not able to build `KDF(..., kSession)`. This secrecy property (of `kSession` from the CDM's point of view) does not hold in the unpatched protocol, but is fixed by the proposed patch.

7 Conclusion

In this paper, we performed the first in-depth and formal analysis of the W3C EME as implemented by Widevine. To the best of our knowledge, our work is the first to provide such analysis of any DRM system for video playback. To enable the analysis, we reverse-engineered various Widevine proprietary details, and we formulated precise security properties that reflect the notion of piracy. Our study indicates that our methodology is effective in automatically finding potential weaknesses. Building on a systematized workflow for the EME API allowed us to consider sophisticated combinations that can be achieved by the attacker in our threat model.

In the future, we aim at keeping our models up to date with upcoming work revealing novel insights about Widevine internals or about other DRM systems. We chose Widevine because of its dominance, but other DRM systems, such as PlayReady and FairPlay, are of course worth analyzing. This requires us to extend our work in two ways. First, we need to reverse-engineer these systems (as we do in Section 3 for Widevine), and also to provide a formal model of them (as we do in Sections 5 and 6 for Widevine). However, the security goals set out in Section 4 will remain unchanged, and we can even imagine that some parts of the formal model (e.g. the way we model time) could be reused.

In addition, there are multiple ways to improve our work, notably by considering a new threat model where valid OTT platforms can be dishonest. Namely, an attacker can successfully perform the `setServerCertificate` method to interact with the CDM through successfully initialized sessions. Lastly, we believe that it would be insightful to analyze EME through specialized models designed for the Web. Indeed, EME inherently uses many features of the Web, including scripts, origins (and hence, the notion of domains), the windows and document structure of browsers, and HTTPS. Therefore, basing future analysis of EME on a model that supports these features is crucial in order to be able to model EME as implemented by browsers (and not CDMs). Unlike our work, the related threat model is malicious web pages attempting to break privacy, or even to hijack opened sessions by a valid OTT.

Acknowledgments

This work received funding from the French National Research Agency (ANR) under the project DRAMA (grant agreement No. ANR-22-CE39-0005), and the France 2030 program managed by the ANR under the project SVP (grant agreement No. ANR-22-PECY-0006). This work benefited also from the support of the German Federal Ministry of Education and Research and the Hessian Ministry of Higher Education, Research, Science and the Arts within their joint support of the National Research Center for Applied Cybersecurity ATHENE.

References

[1] Apple. Apple FairPlay. https://developer.apple.com/streaming/fps/, 2023.

[2] Avalonswanderer. WideXtractor. https://github.com/Avalonswanderer/wideXtractor, 2022.

[3] Chetan Bansal, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, and Sergio Maffeis. Discovering concrete attacks on website authorization by formal analysis. *Journal of Computer Security*, 22(4):601–657, 2014.

[4] Bruno Blanchet. An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In *Proc. 14th IEEE Computer Security Foundations Workshop (CSFW'01)*, pages 82–96. IEEE Computer Society Press, 2001.

[5] Bruno Blanchet, Vincent Cheval, and Véronique Cortier. Proverif with Lemmas, Induction, Fast Subsumption, and Much More. In *Proc. 42nd IEEE Symposium on Security and Privacy (S&P'22)*. IEEE Computer Society Press, 2022.

[6] Kelsey Cairns, Harry Halpin, and Graham Steel. Security Analysis of the W3C Web Cryptography API. In *Proc. 3rd International Conference on Security Standardisation Research (SSR'16)*, volume 10074 of *LNCS*, pages 112–140. Springer, 2016.

[7] Can I Use. Encrypted Media Extensions. https://caniuse.com/eme, 2023.

[8] Chromium Blog. Chrome 26 Beta: Template Element & Unprefixed CSS Transitions . https://blog.chromium.org/2013/02/chrome-26-beta-template-element.html, 2013.

[9] Cas Cremers, Charlie Jacomme, and Philip Lukert. Subterm-based proof techniques for improving the automation and scope of security protocol analysis. In *Proc. 36th IEEE Computer Security Foundations Symposium (CSF'23)*. IEEE Computer Society Press, 2023.

[10] Jean Paul Degabriele, Anja Lehmann, Kenneth G. Paterson, Nigel P. Smart, and Mario Strefler. On the joint security of encryption and signature in EMV. In *CT-RSA*, volume 7178 of *Lecture Notes in Computer Science*, pages 116–135. Springer, 2012.

[11] Stéphanie Delaune, Joseph Lallemand, Gwendal Patat, Florian Roudot, and Mohamed Sabt. EME Widevine Tamarin Models. https://github.com/Avalonswanderer/eme_widevine_formal_verification/.

[12] Eric Diehl. A four-layer model for security of digital rights management. In *Proc. 8th ACM Workshop on Digital Rights Management*, pages 19–28. ACM, 2008.

[13] Quoc Huy Do, Pedram Hosseyni, Ralf Küsters, Guido Schmitz, Nils Wenzler, and Tim Würtele. A Formal Security Analysis of the W3C Web Payment APIs: Attacks and Verification. In *Proc. 43rd IEEE Symposium on Security and Privacy (S&P'22)*, pages 215–234. IEEE Computer Society, 2022.

[14] David Dorwin, Jerry Smith, Mark Watson, and Adrian Bateman. Encrypted Media Extensions. https://www.w3.org/TR/encrypted-media/, 2019.

[15] Daniel Fett, Ralf Küsters, and Guido Schmitz. An expressive model for the web infrastructure: Definition and application to the browser ID SSO system. In *IEEE Symposium on Security and Privacy*, pages 673–688. IEEE Computer Society, 2014.

[16] Google Widevine. Widevine. https://widevine.com/, 2023.

[17] Iness Ben Guirat and Harry Halpin. Formal verification of the W3C web authentication protocol. In *Proc. 5th Annual Symposium and Bootcamp on Hot Topics in the Science of Security, (HoTSoS'18)*, pages 6:1–6:10. ACM, 2018.

[18] Stuart Haber and Benny Pinkas. Securely combining public-key cryptosystems. In *CCS*, pages 215–224. ACM, 2001.

[19] Tomer Hadad. Reversing the old Widevine Content Decryption Module. https://github.com/tomer8007/widevine-l3-decryptor/wiki/Reversing-the-old-Widevine-Content-Decryption-Module, 2021.

[20] Harry Halpin. The W3C web cryptography API: motivation and overview. In *Proc. 23rd International World Wide Web Conference, (WWW'14)*, pages 959–964. ACM, 2014.

[21] Harry Halpin. The Crisis of Standardizing DRM: The Case of W3C Encrypted Media Extensions. In *Proc. 7th International Conference on Security, Privacy, and Applied Cryptography Engineering, (SPACE'17)*, volume 10662 of *LNCS*, pages 10–29. Springer, 2017.

[22] Gregory L. Heileman, Pramod A. Jamkhedkar, Joud S. Khoury, and Curtis J. Hrncir. The drm game. In *Proc. tth ACM Workshop on Digital Rights Management, (DRM'07)*, pages 54–62. ACM, 2007.

[23] Jeff Hodges, J.C. Jones, Michael B. Jones, Akshay Kumar, and Emil Lundberg. Web Authentication: An API for accessing Public Key Credentials. https://www.w3.org/TR/webauthn-2/, 2021.

[24] Adrian Hope-Bailie, Ian Jacobs, Rouslan Solomakhin, and Jinho Bang. Payment Handler API. https://www.w3.org/TR/payment-handler/, 2023.

[25] Fortune Business Insights. Market Research Report. https://www.fortunebusinessinsights.com/video-streaming-market-103057, 2023.

[26] ISO Central Secretary. Information technology — Multimedia application format (MPEG-A) — Part 19: Common media application format (CMAF) for segmented media. Standard ISO/IEC 23000-19:2020, International Organization for Standardization, 2020.

[27] ISO Central Secretary. Information technology — Dynamic adaptive streaming over HTTP (DASH) — Part 1: Media presentation description and segment formats. Standard ISO/IEC 23009-1:2022, International Organization for Standardization, 2022.

[28] Simon Meier, Benedikt Schmidt, Cas Cremers, and David A. Basin. The TAMARIN Prover for the Symbolic Analysis of Security Protocols. In Natasha Sharygina and Helmut Veith, editors, *Proc. 25th International Conference on Computer Aided Verification, (CAV'13)*, volume 8044 of *LNCS*, pages 696–701. Springer, 2013.

[29] Microsoft. Microsoft PlayReady. https://www.microsoft.com/playready/, 2023.

[30] NSA. Ghidra Reverse Engineering Framework. https://github.com/NationalSecurityAgency/ghidra, 2019.

[31] R. Pantos and W. May. HTTP Live Streaming. RFC 8216, RFC Editor, August 2017.

[32] Joey Parrish and Greg Freedman. Encrypted Media Extensions. https://w3c.github.io/encrypted-media, 2023.

[33] Gwendal Patat, Mohamed Sabt, and Pierre-Alain Fouque. Exploring Widevine for Fun and Profit. In *Proc. 43rd IEEE Security and Privacy Workshops (WOOT'22)*, pages 277–288. IEEE Computer Society, 2022.

[34] Gwendal Patat, Mohamed Sabt, and Pierre-Alain Fouque. Your DRM Can Watch You Too: Exploring the Privacy Implications of Browsers (mis)Implementations of Widevine EME. *Proc. 23rd International Conference on Privacy Enhancing Technologies (PETS'23)*, 2023(4):306–321, 2023.

[35] Ole André Vadla Ravnås. Frida. https://github.com/frida/frida, 2013.

[36] B. Rosenblatt, W. Trippe, and S. Mooney. *Digital Rights Management: Business and Technology*. M&T Books. Wiley, 2002.

[37] Niels Rump. Definition, Aspects, and Overview. In *Digital Rights Management - Technological, Economic, Legal and Political Aspects*, volume 2770 of *LNCS*, pages 3–15, 2003.

[38] Beale Screamer. Microsoft's Digital Rights Management Scheme - Technical Details. https://cryptome.org/beale-sci-crypt.htm, 2001.

[39] Joey Sneddon. How To Watch Netflix on Ubuntu The Easy Way. https://www.omgubuntu.co.uk/2014/08/netflix-linux-html5-support-plugins, 2014.

[40] U.S. Chamber Staff. Quick Take: Your Primer on Digital Piracy and Its Impact on the U.S. Economy, 2019. https://www.uschamber.com/intellectual-property/quick-take-your-primer-digital-piracy-and-its-impact-the-us-economy.

[41] Stat Counter. Mobile Tablet Android Version Market Share Worldwide. https://gs.statcounter.com/android-version-market-share/mobile-tablet/worldwide, 2023.

[42] Stat Counter. Web Browser Market Share Worldwide. https://gs.statcounter.com/browser-market-share/all/worldwide/2023, 2024.

[43] W3C. W3C Publishes Encrypted Media Extensions (EME) as a W3C Recommendation. https://www.w3.org/2017/09/pressrelease-eme-recommendation.html.en, 2017.

[44] W3C. Media WG meeting. https://www.w3.org/2024/02/13-mediawg-minutes.html#t04, 2024.

[45] Mark Watson. Web Cryptography API. https://www.w3.org/TR/WebCryptoAPI/, 2017.

[46] Qi Zhao. Wideshears: Investigating and breaking Widevine on QTEE. BlackHat Asia, 2021.

A Widevine EME Internals

A summary of Widevine EME message content and Widevine internals are given in Table 2 and Table 3.

B Formal Security Analysis

Size of our TAMARIN models. We summarize some information regarding the size of our models in Table 4. We do not take into account the part of the files concerning the sanity check (executability lemma).

	without fix	with fix
#LoC protocols	510	510
#LoC properties	200	450
#lemmas (secrecy properties)	9	11
#lemmas (initial part)	5	5
#lemmas (renewal part)	_	15

Table 4: Summary regarding the size of our models. #LoC includes the comments, and #lemmas includes the intermediate lemmas stated and proved to establish our 7 main goals.

Command line to run experiment. Each model found in 'widevine.spthy' can be launched via the following command:

```
tamarin-prover --prove  --derivcheck-timeout=0
         --heuristic=O --oraclename='widevine.oracle'
         widevine.spthy -DSecrecy -DGoalInitialPart
```

We explain below the different elements:

- `--prove` means that the tool is launched using its automatic mode (no user interaction);
- `--derivcheck-timeout=0` is there to turn off some sanity checks that are useless once the model is written;
- `--heuristic=O --oraclename='widevine.oracle'` means that the oracle written in the file 'widevine.oracle' will be used to guide the proof search;
- `-DSecrecy -DGoalInitialPart` indicates that only the main part of the file as well as those defined in the macros `DSecrecy` and `DGoalInitialPart` will be taken into account.

Some insights regarding Goal 6 and its proof. Regarding freshness of the license during a load (Goal 3), the situation is rather simple. This property is ensured by the nonce generated by the CDM and consume when the response (which contains also the nonce) is loaded by the CDM. The TAMARIN proof is very easy and done in very few steps. Establishing Goal 6 (freshness of the license during a renewal load) is more involved. The difficulty comes from the fact that the renewal part does not rely on the use of a nonce to ensure freshness of the response that is loaded. The freshness is ensured through the use of a counter that is increased by the CDM each time a renewal response is loaded. We thus have to state and prove some lemmas (by induction) that roughly state that the counter is always increasing, and even strictly increases each time a renewal load is performed. Then, Goal 6 easily follows.

```
==========================================================
CounterIncrease (all-traces): verified (259 steps)
CounterIncreaseStrictly (all-traces): verified (3941
    steps)
Goal6 (all-traces): verified (18 steps)
==========================================================
```

EME Message	EME Method	Widevine EME Message Content
License Request	`generateRequest`	body = {Request ID, nonce, $\{\text{ClientID}\}_{\text{privacyKey}}$, $\{\text{privacyKey}\}_{\text{serviceCertificate}_{\text{pub}}}$, KID, T1}, signature = sign(body, $\text{DeviceRSAKey}_{\text{priv}}$)
License Response	`update`	body = {$\{\text{SessionKey}\}_{\text{DeviceRSAKey}_{\text{pub}}}$, response = Request ID, T1, KID, $\{\text{ContentKey}\}_{\text{AssetKey}}$, $\{\text{KCB}\}_{\text{ContentKey}}$, Policies}, hmac = HMAC(response, MAC Server Key)
Renewal Request	`onMessage`	body = {Request ID, $\{\text{ClientID}\}_{\text{privacyKey}}$, $\{\text{privacyKey}\}_{\text{serviceCertificate}_{\text{pub}}}$, T1, T2, counter, *nonce*†}, hmac = HMAC(body, MAC Client Key)
Renewal Response	`update`	body = {Request ID, T1, T2, Updated counter, Updated Policies, *Updated KCB*†}, hmac = HMAC(body, MAC Server Key)

† only on Android.

Table 2: Widevine EME Message Content.

Steps	EME Protocol	EME Method	Widevine Method	Widevine Internal Memory
1	Initial State	`createSession`	OpenSession	InitialKeys = {$\text{DeviceRSAKey}_{\text{pub/priv}}$, $\text{serviceCertificate}_{\text{pub}}$}, Session ID
2	License Request	`generateRequest`	PrepareKeyRequest	InitialKeys, IDs = {Session ID, Request ID}, nonce, T1, request
3	License Processing	`update`	3.a DeriveKeysFromSessionKey	InitialKeys, IDs, nonce, T1, request, DerivedKeys = {MAC Client Key, MAC Server Key, Asset Key}
			3.b LoadKeys	InitialKeys, IDs, T1, request, DerivedKeys, Content Key, KCB, Policies
4	Renewal Request	`onMessage`	PrepareKeyUpdateRequest	InitialKeys, IDs, T1, request, DerivedKeys, Content Key, KCB, Policies, T2, counter, *nonce*†
5	Renewal Processing	`update`	RefreshKeys	InitialKeys, IDs, T1, request, DerivedKeys, Content Key, T2, Updated counter, *Updated KCB*†, Updated Policies

† only on Android using KCB in the renewal response.

Table 3: Widevine CDM Internals during EME workflow.

USENIX Security '24 Artifact Appendix: Formal Security Analysis of Widevine through the W3C EME Standard

Stéphanie Delaune
Univ Rennes, CNRS, IRISA, France

Joseph Lallemand
Univ Rennes, CNRS, IRISA, France

Gwendal Patat
Fraunhofer SIT | ATHENE, Germany

Florian Roudot
Univ Rennes, CNRS, IRISA, France

Mohamed Sabt
Univ Rennes, CNRS, IRISA, France

A Artifact Appendix

This artifact appendix describes the experiments and case studies conducted in the research paper *Formal Security Analysis of Widevine through the W3C EME Standard*.

A.1 Abstract

Streaming services such as Netflix, Amazon Prime Video, or Disney+ rely on the widespread EME standard to deliver their content to end users on all major web browsers. While providing an abstraction layer to the underlying DRM protocols of each device, the security of this API has never been formally studied. One of the core objectives of this research is to provide a formal security analysis of Widevine, the most deployed DRM instantiating EME.

Relying on TAMARIN, we study two variants of the Widevine DRM system. Our investigation highlights a vulnerability that could allow for unlimited media consumption. Additionally, we present a fix suitable for both mobile and desktop platforms, and formally prove it secure using TAMARIN.

We provide means to reproduce all the proofs and analyses we performed using the TAMARIN prover. We study both the Android version of Widevine (where the renewal request contains a so-called Key Control Block – KCB), and the desktop version (where it does not). For each version, we consider both the actual protocol and the fixed version which includes the mitigation we propose. Altogether, we thus present four protocol models as the artifact for our paper.

A.2 Description & Requirements

A.2.1 Security, Privacy, and Ethical Concerns

Executing the models provided as artifact does not present any security or ethical risks. The vulnerability we found has been responsibly disclosed to Widevine and the W3C, as discussed in the paper.

A.2.2 How to Access

Our artifact is available in a public GitHub repository, with detailed instructions to replicate the experiments discussed in the original paper. https://github.com/Avalonswanderer/eme_widevine_formal_verification/releases/tag/v1.0.

A.2.3 Hardware Dependencies

Running our artifact does not require any specific hardware. We used a fairly standard laptop (2.30GHz Intel Core i7-1068NG7 CPU, 16GB of RAM), and were able to verify all our models in a few minutes.

A.2.4 Software dependencies

Our artifact relies on the following dependencies:

- The TAMARIN Prover[1]. We used version 1.8.0, which is the latest release at the time of writing. TAMARIN itself depends on Haskell-stack[2], GraphViz[3], and Maude[4] (versions 2.7.1 to 3.3.1 are recommended by TAMARIN). Installation instructions for TAMARIN and its dependencies can be found at https://tamarin-prover.com/manual/master/book/002_installation.html. Note that TAMARIN runs natively on Linux or macOS, but not on Windows systems (WSL may be used there).

- Python3[5], which can be installed from most package managers or manually. We used version 3.12, but any relatively recent version should work as well.

[1] https://tamarin-prover.com/
[2] https://github.com/commercialhaskell/stack
[3] https://www.graphviz.org
[4] https://github.com/maude-lang/Maude
[5] https://www.python.org/downloads/

A.2.5 Benchmarks

None.

A.3 Setup

A.3.1 Installation

1. Install TAMARIN (and its dependencies), either with a package manager, by manually downloading the binaries, or by compiling it from sources. Instructions are provided at https://tamarin-prover.com/manual/master/book/002_installation.html.
2. Install Python3.
3. Retrieve the files from our repository.

A.3.2 Basic Test

To check that TAMARIN is properly installed, run

```
$ tamarin-prover test
```

You should see a diagnostic message, ensuring that

- Maude is correctly installed;
- GraphViz is correctly installed;
- unification works properly.

It should conclude with:

```
All tests successful.
The tamarin-prover should work as intended.

           :-) happy proving (-:
```

A.4 Evaluation Workflow

Our repository contains a `README` file and four subfolders: `WithKCB`, `WithoutKCB`, `FixWithKCB`, and `FixWithoutKCB`. These four folders correspond to the four models we developed, to analyze the Android version (`WithKCB`), the desktop version (`WithoutKCB`), and the mitigation we proposed for both versions (`FixWithKCB` and `FixWithoutKCB`).

For the security analysis, we consider 7 security goals that are explained in the paper. In a nutshell, they are:

Goal 1: Confidentiality of the decryption key.

Goal 2: Integrity and authenticity of initial licenses.

Goal 3: Freshness of initial licenses.

Goal 4: Enforcing expiration time of initial licenses.

Goal 5: Integrity and authenticity of renewal licenses.

Goal 6: Freshness of renewal licenses.

Goal 7: Enforcing expiration time of renewal licenses.

In addition, we prove an "executability lemma", as a form of sanity check. This lemma expresses the fact that the normal intended protocol flow can be executed. Its purpose is to ascertain that our models are correct.

We also consider an extra security goal, expressing that the license policies are respected, *i.e.* that the CDM can load a renewal license only if it is authorized by the policy. This goal cannot be stated at the EME level (the API studied in this paper), as policies are not part of EME. Nevertheless, we still include this extra goal, which we call **Goal 8**, for the specific case of Widevine.

A.4.1 Major Claims

Our findings are summarized in Table 1. Most of the proofs required some intermediate lemmas in order to conclude. These proofs have been obtained at first using the interactive mode of TAMARIN. However, for reproducibility purposes, we then automated them, relying on the oracle mechanism available in TAMARIN. The experiments below can therefore be run non-interactively from the command line.

	without fix		with fix	
KCB	with	without	with	without
Goal 1	✓	✓	✓	✓
Goal 2	✓	✓	✓	✓
Goal 3	✓	✓	✓	✓
Goal 4	✓	✓	✓	✓
Goal 5	✗	✗	✓	✓
Goal 6	✓	✓	✓	✓
Goal 7	✗	✗	✓	✓
Goal 8	✗	✗	✓	✓

Table 1: Summary of our results.

Goals marked with ✓ in the table are satisfied, and are proved automatically by TAMARIN in the respective files. Goals against which we found attacks are marked ✗ in the table (see the paper for a description of the attacks). In fact, we used TAMARIN to discover the attack on **Goal 5**, and stored the attack trace in each corresponding TAMARIN file for reproducibility. The goals in a shaded box in the table are broken as well as a consequence of the attack on **Goal 5**. We did not use TAMARIN to derive the same attack again for each of them, and thus the corresponding files do not contain any proof or attack for these goals.

A.4.2 Experiments

All our experiments were performed on a standard laptop, as mentioned earlier. The verification time varies depending on the model, and the goals we are trying to establish. The easiest lemmas are proved in a few seconds, whereas the most difficult ones require around 4 minutes.

Below, we explain the different types of experiments, as well as the results that are expected when running them.

(E1): Proving **Goal 1** in TAMARIN.
Preparation: Go to the folder containing the model for which you want to establish **Goal 1**.
Execution: Run the command given in the `README` file, and recalled below:

```
 tamarin-prover --prove
  --derivcheck-timeout=0
  --heuristic=O
  --oraclename='widevine.oracle'
  widevine.spthy -DSecrecy -DGoal1
```

Results: After execution, you should obtain a summary stating that `ContentKeySecrecy`, *i.e.* **Goal 1**, (and actually all required intermediate lemmas) are verified.

```
==================================================
summary of summaries:
analyzed: widevine.spthy
processing time: 9.16s

OTT2 (all-traces): verified (6 steps)
...
contentKeySecrecy (all-traces): verified (2 steps)
==================================================
```

(E2): Proving **Goal 2**, **Goal 3**, and **Goal 4** in TAMARIN.
Preparation: Go to the folder containing the model for which you want to establish **Goals 2**, **3**, and **4**.
Execution: Run the command given in the `README` file, and recalled below:

```
tamarin-prover --prove
  --derivcheck-timeout=0
  --heuristic=O
  --oraclename='widevine.oracle'
  widevine.spthy -DSecrecy -DGoalInitialPart
```

Results: After execution, you should obtain a summary stating that all these goals, as well as the required intermediate lemmas, are verified. The correspondence between goals and lemmas is as follows:

- **Goal 2**: `OTTLicenseResponseBeforeLoad`
- **Goal 3**: `LoadRespUnique`
- **Goal 4**: `UseAuthorised`

```
==================================================
summary of summaries:
analyzed: widevine.spthy
processing time: 9.96s

OTT2 (all-traces): verified (6 steps)
...
UseAuthorised (all-traces): verified (4 steps)
==================================================
```

(E3): Proving **Goal 6** (on all models), as well as **Goals 5**, **7**, and **8** on the fixed versions is similar:

```
tamarin-prover --prove
  --derivcheck-timeout=0
  --heuristic=O
  --oraclename='widevine.oracle'
  widevine.spthy -DSecrecy -DGoal6
tamarin-prover --prove
  --derivcheck-timeout=0 --heuristic=O
  --oraclename='widevine.oracle'
 widevine.spthy -DSecrecy -DGoal578
```

The correspondence between goals and lemmas is as follows:

- **Goal 5**: `OTTRefreshResponseBeforeLoadRefresh`
- **Goal 6**: `LoadRefreshRespUnique`
- **Goal 7**: `UseAuthorised`
- **Goal 8**: `LoadRefreshOnlyIfRenewable`

The runtime is slightly longer for these experiments. TAMARIN should still conclude within a few minutes.

(E4): For **Goal 5**, which does not hold in the models `WithoutKCB` and `WithKCB`, the attack trace has been stored in the corresponding file, and can be replayed[6]
Preparation: Go to the folder containing the model for which you want to establish that **Goal 5** fails.
Execution: Run the command given in the `README` file, and recalled below:

```
tamarin-prover
   --derivcheck-timeout=0
   --heuristic=O
    --oraclename='widevine.oracle'
    widevine.spthy -DSecrecy -DGoal578
```

Results: After execution, you should obtain a summary stating that **Goal 5**, which is to say lemma `OTTRefreshResponseBeforeLoadRefresh`, is falsified, and that an attack trace has been found. All the other lemmas are marked "analysis incomplete" as we did *not* ask TAMARIN to prove them (note that the `--prove` option is not part of the command).

```
==================================================
summary of summaries:
analyzed: widevine.spthy
processing time: 13.28s
...
OTTRefreshResponseBeforeLoadRefresh (all-traces):
    falsified - found trace (77 steps)
...
==================================================
```

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at `https://secartifacts.github.io/usenixsec2024/`.

[6]The proof scripts are not complete and contain some by sorry instructions. This is normal as there is an attack, and thus this goal can not be proved.