Post-Compromise Security with Application-Level Key-Controls – with a comprehensive study of the 5G AKMA protocol –

Ioana Boureanu
University of Surrey
UK
i.boureanu@surrey.ac.uk

Cristina Onete
Universite de Limoges
France
cristina.onete@gmail.com

Steve Wesemeyer
University of Surrey
UK
s.wesemeyer@surrey.ac.uk

Leo Robert
Universite de Picardie Jules Verne
France
leo.robert@u-picardie.fr

Rhys Miller
University of Surrey
UK
rhysjohnmiller@googlemail.com

Pascal Lafourcade
Universite de Clermont-Auvergne
France
pascal.lafourcade@uca.fr

Fortunat Rajaona
University of Surrey
UK
s.rajaona@surrey.ac.uk

ABSTRACT

We propose PCS_AC – a new cryptographic framework for post-compromise security (PCS) in secure-communication protocols, which caters not only for the traditional PCS-corruption of key material, but also for adversarial controls of a PCS-nature *at the application level*. That is, we study and formalise when and how *an adversary can maintain or lose post-compromise advantages due to application-driven controls as well, rather than do so based on cryptographic manipulations and settings alone*. In fact, the motivation and ideas in our PCS_AC are in line with a recent document by NIST [15], where adversarial application-level influence on key compromise are discussed, and it is the first of this kind to be produced.

Since measures of PCS "healing" are discrete, we also mechanised an approximation of our cryptographic PCS_AC framework, in symbolic/Dolev-Yao models, as well; we do this in the two state-of-the-art protocol verifiers, ProVerif and Tamarin, and –in the process– we also compare and discuss different tool-assistance aspects in each. We include a summary of this in this version of the paper, too.

We apply PCS_AC to the 5G procedure called *AKMA (Authentication and Key Management for Application)*, showing new PCS-attacks therein and proposing patches, including a backwards-compatible one. We implement and test AKMA as well as of one of our PCS-driven patches, on top of Fraunhofer's Open5GCore.

We also cast 'TLS1.3 with session resumption and key updates' in PCS_AC in various ways, looking at different keys being potentially targeted by one type of attacker or another, amongst the multiple possible choices formalised in PCS_AC. This captures new and varied PCS dimensions of TLS, underlining the versatility and usefulness of our framework.

ACM Reference Format:
Ioana Boureanu, Cristina Onete, Steve Wesemeyer, Leo Robert, Rhys Miller, Pascal Lafourcade, and Fortunat Rajaona. 2025. Post-Compromise Security with Application-Level Key-Controls – with a comprehensive study of the 5G AKMA protocol –. In *ACM Asia Conference on Computer and Communications Security (ASIA CCS '25), August 25–29, 2025, Hanoi, Vietnam.* ACM, New York, NY, USA, 19 pages. https://doi.org/10.1145/3708821.3733910

1 INTRODUCTION

Post-compromise security (PCS) [22] is a timely notion, generally cast around communications on secure channels: intuitively, PCS holds if the security of a channel, primarily its confidentiality, can repair, or "*heal*", even after a full compromise by an adversary. In 2017, Cohn-Gordon *et al.* proposed a PCS framework [16], and recently, Blazy *et al.* [14] refined those in a computational metric for quantifying how fast key-establishment protocols can heal post-compromise.

But, all these past works only look at the key-derivation, i.e., the attacker's corruption and access are only w.r.t. keys inside key-derivation trees and ratcheting chains. Notably, the attacker does not control the application-related aspects of the key-derivations; e.g., Blazy *et al.*'s attacker do not control explicitly the triggers of the regeneration of the keys inside various levels of the key-derivation tree: i.e., they do not control the algorithmic software/hardware resets or parameters linked to the refreshing of keys, such as session-tickets expiry times or "times to live (TTLs)". Yet, different key-establishment protocols for different applications have an explicit logic and time-frame which determine how the keys can evolve, e.g., TLS, 3G/4G/5G procedures, and this can clearly influence the PCS of the protocol. For instance, if an attacker can keep a mobile phone charged and "alive" forever at one location within a stable network then this phone will not re-register on the network, nor will it refresh its various sessions keys, thus allowing an attacker to potentially prolong a given compromise. Conversely, if a session-key has a given TTL which the attacker cannot control, this TTL

ASIA CCS '25, August 25–29, 2025, Hanoi, Vietnam

ACM ISBN 979-8-4007-1410-8/2025/08…$15.00
https://doi.org/10.1145/3708821.3733910

can be a prime factor in increasing the healing speed of the protocol once a compromise has occurred.

Moreover, this type of application-level control on key-compromises came to the attention of NIST [15] and they are also calling for action. We answer that call herein.

We propose a *PCS Metric with Application Control (*PCS_AC*)*, a new PCS formalism and metric such that PCS is no longer "just" about key-manipulation compromises. Concretely, PCS_AC is the first (PCS) framework to bring and formalise application-level controls (e.g., someone manipulating the time-to-live of a session) into the cryptographic attackers' playground. The application-level adversarial additions also lead to our attacker model being more fine-grained, adding numerous and more complex dimensions to intruders, discerning and mixing various levels of threats at the application level (e.g., algorithmic, physical, etc), with multi-facetted key controls (e.g., hijacking application-level handles that affect different levels in a key hierarchy); all of this is formalised in a cryptographic model.

The depth of PCS attackers and security notions that PCS_AC brings is showcased in a comprehensive analysis we undertake, using PCS_AC, of the recent 5G protocol called *Authentication and Key Management for Applications based on 3GPP credentials in the 5G Systems*, *AKMA* for short [5]. AKMA lends itself extremely well to analysis via PCS_AC, since it is a procedure which mixes key-derivation, key-establishment and application-controls linked to these in intricate and different ways over the multiple parties involved. In short, in AKMA, the mobile operator (e.g., Orange) uses the existing mobile key-hierarchy to derive session keys for a third-party application server (e.g., BMW) to be used with Orange subscribers (e.g., BMW cars with Orange SIM cards inside); and, the latter two employ and govern over the time-to-live of these session keys via a protocol over which the core has no visibility or control.

Finally, PCS_AC's versatility and added-value, in terms of security analyses which combine application-level corruptions and key compromises, are clearly demonstrated also when we apply PCS_AC to TLS 1.3 [24]. Indeed, because of TLS' complex key-hierarchy, its intricate key-derivation options (e.g., TLS modes w/o resumption), and its multiple application-level controls (e.g., w.r.t. session tickets), there are numerous ways in which one can apply PCS_AC to TLS 1.3. Indeed, this array of choices available and the wealth of the analyses that PCS_AC brings to TLS 1.3 are dissected in Appendix E.

Our more detailed contributions are as follows:

CONTRIBUTIONS

- We introduce a new PCS framework, *PCS Metric with Application Control (*PCS_AC*)*, which gives PCS adversaries control over application-level parameters. In this way, attackers can manipulate not only cryptographic keys, but –importantly– also the application-logic which re-initiates or stalls the refreshing of keys.

  In Section 5 and Appendix B we show the value of PCS_AC: new attacks issued out of manipulating the application layers, in ways that impact on key security/compromises.

- We give a mechanisation of our PCS_AC framework (and, by proxy, [14] as well), in the state-of-the-art symbolic/Dolev-Yao [17] protocol-verifiers Proverif and Tamarin, and we discuss the advantages of each of the two mechanisations. We apply the mechanisation to AKMA and AKMA$^+$, and our findings in Proverif and Tamarin are in line with our computational analyses via PCS_AC. We give a summary of all of this herein.
- We first apply PCS_AC to *AKMA* [5], a key-establishment protocol intrinsically application-driven, and show PCS attacks therein that would not have been exhibited via prior PCS frameworks.

  We provide full cryptographic proofs for AKMA in our new PCS model.

  We also propose an efficient patch to AKMA, called AKMA$^+$, with formally proven improved security w.r.t. our strong PCS attackers, and which 3GPP is looking to adopt in the future version of the spec.

- We show an implementation of AKMA on top of Fraunhofer's Open5GCore, and use it to attest that our PCS-improved AKMA maintains good efficiency compared to the original AKMA.
  We believe this is also the first implementation of AKMA which is *publicly-available*.

- Also, PCS_AC is applied to TLS 1.3 with session resumption, session tickets and key updates, where the key-hierarchy is vast, the application controls numerous (including on time-to-live of tickets and session keys) and the various sensible types of attackers abound (e.g., passive, active, global, focused on "local"/temporary keys). This complex use case is suited to underline the versatility and usefulness of PCS_AC. Due to space constraints, this is shown in Appendix E.

2 BACKGROUND

2.1 PCS Notions

The most recent PCS framework by Blazy *et al.* [14] focuses on two-party secure-channel establishment only ignoring aspects linked to the application-level. So, it expresses PCS only in terms of number of key-evolutions, at various levels, measuring PCS-healing speed in numbers of key-evolutions.

That is insufficient for us, as we wish to add PCS measure modulo the application logic; however – to keep uniformity in this research field – we keep and extend some of their notions: the notions of stage, stage-based keys and adversaries, as well as a stage-based security notion. We summarise these below.

Stages & Horizontal/Vertical Key Evolutions. In Blazy *et al.* [14], an important concept is the notion of *stage*, denoted (x, y). Each stage corresponds to a bundle of ephemeral key-materials, such as *message-keys* – later used to authenticate and encrypt messages. In more details, a *stage* (x, y) refers to the bundle of keys that have been evolved/refreshed the y-th time in some dimension, and the x-th time in another dimension. More intuitively,

the x vertex refers to some "horizontal" key-evolutions denoting rather ephemeral keys being evolved/refreshed, whereas the y vertex refers to "vertical" key-evolutions whereby other, less ephemeral keys, or keys higher up in a key-hierarchy are evolved/refreshed.

More concretely, (communication-securing) keys can evolve from stage to stage, in two different ways: *horizontal* – from stage (x, y) to $(x + 1, y)$, generally providing weaker post-compromise healing, and *vertical* — from stage (x, y) to $(1, y + 1)$, generally providing stronger PCS.

Stage-based Keys. Blazy *et al.* [14] have: single-stage keys, which are only relevant during that stage and that session; cross-stage keys, which are used during at least two stages of the same session, but not in two different sessions; and cross-session keys, which are long-term keys used across multiple sessions.

Threat Model. Blazy *et al.* classify their adversaries in terms of three different characteristics:

- *Reach.* Attackers can learn (by corruption) keys at different levels of the key-schedule: *global* adversaries can learn all keys, *medium* attackers can only learn cross-stage and single-stage keys, while *local* attackers can only find single-stage keys.
- *Power.* Attackers may have different abilities to interfere with the communication channel. *Passive* attackers cannot inject data, while *active* adversaries can.
- *Access.* Protocols can be attacked by adversaries that lie outside the system (which are called *outsiders*) or even within it (*insider* adversaries).

Blazy's "Healing" Measures. *Post-compromise security* is a notion that captures, intuitively, how fast the confidentiality of a secure channel recovers after an attacker compromises at least one of the endpoints. In particular, if a protocol is proved to be (χ, Υ)-*PCS-secure* for a given-strength attacker, then proofs in Blazy *et al.* show that: (a) χ stages in the currently evolving key-chain and Υ subsequent chains' worth of stages have had their security compromised; (b) all other stages are secure.

3 PCS_AC: APPLICATION-DRIVEN PCS METRIC

• In Appendix A, we detail how and why our threat model, inspired by this NIST document [15], is realistic. This threat model (and the execution environment alongside it) is formalised next.

3.1 Key-evolution Protocols with Key-Control

Our model focuses on two-party key-exchange protocols that feature a dynamic key-evolution. The two endpoints retain some control over how and when the key-evolution occurs, at application level. We call such schemes Key-Exchange with Dynamic Evolution and Key-Control, in short KE-DECK.

A *key-control* will be a tuple, indexed by the name of the key and the stage of the key-evolution, which characterises (by means of a predicate) whether a specific key may evolve at a given stage, and how that evolution may take place (which party initiates and controls the evolution). The attacker will access and manipulate the key-controls, i.e., :

- The attacker only switches the key-controls for evolution off at a particular target stage;
- The attacker switches the key-controls for the evolution of a given key permanently off. Depending on the protocol, this might result in a denial of service;
- The attacker switches the key-controls for the evolution of several keys for a number of stages.

Now, we formalise PCS (in)security driven by key-controls, by modularly building on the framework by Blazy *et al.*.

The Execution Environment. We consider two types of endpoints with asymmetric roles: i.e., *clients* and *servers*. The sets of clients $\mathbb{C}$ and $\mathbb{S}$ are disjoint and, together with a *single super-user* $\hat{S} \notin \mathbb{C} \cup \mathbb{S}$, they give partition of the party set $\mathcal{P}$. All parties are probabilistic polynomial-time interactive Turing machines (PPT ITM).

KE-DECK protocols run in *sessions* between exactly one client and one server. We assume that any two parties P, Q will run at most one session of this protocol during their lifetime. The super-user may help the parties set up their session, but will not interact throughout it.

In contrast to [14], we abstract away certain application-message exchanges between parties[1], and instead focus only on the evolution of the keys and the control the parties may exert over which type of evolution occurs when. So, *stages* typical in asynchronous-messaging frameworks (including [14]) now simply mark intermediate key-evolutions.

Consider a key hierarchy (KH). Over it, a *key-schedule* is a finite-depth tree (specifically, each key has a single ancestor from KH), whereby edges between nodes denote one key-derivation from the parent to the child node; keys in a schedule may evolve across stages, differently. We associate parties $P \in \mathbb{C}$ and $Q \in \mathbb{S}$ with such a key-schedule.

The focus of the security definition will be a particular kind of single-stage target key, which we denote by *target key* $\mathrm{t\_k}_{P,Q}$. But, we can examine the PCS characteristics of any key in the key-schedule (regardless of how high it might be in the hierarchy) – by casting it into the role of the target key, and tracing the impact of the evolution of that key and its ancestors in the key-schedule.

Keys can be prompted to evolve unilaterally or mutually, and might require contributions from one or both parties. We model the key-update logic in terms of *key-evolution controls*, which are essentially flags determining whether a particular key can be updated, as well as indicating to each party whether it can, unilaterally or interactively cause this.

Attributes. Each party $P \in \mathbb{C} \cup \mathbb{S}$ keeps track of:

- $(P.pk, P.sk)$: a pair of long-term public/private credentials that may be set to $\perp$ for specific protocols;
- $P.role \in \{C, S\}$: denoting whether the party is a client or a server (note that the role of each party is static and will not change ever);
- $P.KH$: the key-hierarchy associated with the protocol. Apart from *names*, e.g., k, keys are identified by their *level* in the

[1]Note that we do take into account generic, message-exchange-based attacks, as we model MiM interference in the protocol. Yet, in security notions we focus primarily on the way keys are used and evolved, and therein we abstract away some details of the communication – thus focusing on attacks which will immediately impact keys' security. This may elude PCS-related, adaptive attacks – which do not impact key-evolution (e.g., privacy threats).

hierarchy, i.e., $k.level$ 1, 2, 3, ... (from top to bottom). The bottom key is directly targeted in our PCS-security games and it is also denoted *target key* (t_k), whereas the others are also referred to, indiscriminately, as *non-target keys* (nt_k). The keys at the top of the key-hierarchy are called *top keys*, generically denoted K, and they do not evolve.

Instances π of a party P presumably partnered with Q are denoted: π_P^Q. Party instances inherit the long-term attributes of the party, and additionally keep track of:

- π_P^Q.ltsk: a long-term key associated with the unique session between P and a party Q whose role is different than that of P.
- π_P^Q.sid: the session identifier of the session that π_P^Q is running, which is a protocol-specific set of values that uniquely determine the key material computed by the two endpoints.
- π_P^Q.pid: the partner identifier of P, namely the party that P thinks it is talking to. This partner cannot have the same role as P.
- π_P^Q.stages: a list of tuples (st, v), of stages $st = (x, y)$, with values $v \in \{0, 1\}$ indicating whether the stage st was reached ($v = 1$) or not ($v = 0$). By abuse of notation we write $st \in \pi_P^Q$ if $(st, v) \in \pi_P^Q$.stages.
- $\pi_P^Q.Ke\_Tr$: key-evolution transcript $\pi_P^Q.T$, indexed by stages, describing all key-evolution data associated with a stage. We denote by $\pi_P^Q.T[st]$ the key-evolution data of instance $\pi_P^Q.T$ at stage st.
- π_P^Q.var: a set π_P^Q.var of ephemeral values used to compute stage keys, indexed by stages. If a value is used for more than one stage, it will appear under every single stage that it is required for.
  – One of the values included in P.var is the target key shared with a Q party: $\mathsf{t\_k}_{P,Q}$. As it evolves over a multitude of stages, we denote by $\mathsf{t\_k}_{P,Q}^{(x,y)}$ the value of this key at stage (x, y).
- π_P^Q.KControls: a list of tuples indexed by keys and stages, of the form $(key, st, \tau, \mathtt{aEvolve})$ where key is the name of a secret or a key included in $P.KH$, st is a stage, aEvolve is a boolean predicate indicating whether an evolution of that key is possible at this stage, and τ is the *key-control prompt*, which can be set to $\perp$ if party P cannot cause evolution of that key at all, NI if party P can non-interactively cause its update, to $P\cdot$ if party P can make the key evolve by interacting with its partner (but P initiates this update, and the partner follows), and to $\cdot P$ if the key-evolution process is interactive, but the partner goes first.

We define our primitive KE-DECK, noting that the common algorithms for sending and receiving messages are abstracted away into a potentially interactive protocol, $\Pi_{\mathtt{Evolve}}$, which instead focuses just on *key-evolution*.

Definition 1 (KE-DECK Protocol). *A Key-Exchange with Dynamic Evolution and Key-Control (*KE-DECK*) is a tuple of algorithms:* $\mathtt{KE\text{-}DECK} = (\mathtt{aSetup}, \mathtt{aPGen}, \Pi_{\mathtt{UReg}}, \Pi_{\mathtt{Evolve}})$*:*

$\underline{\mathtt{aSetup}(1^\lambda) \to (\hat{S}.\mathsf{sk}, \hat{S}.\mathsf{pk}, \mathsf{pparam})}$ *: outputs the public/private long-term keys of super-user* $\hat{S}$ *and the public system parameters* pparam *implicitly taken in input by all other algorithms.*

$\underline{\mathtt{aPGen}(1^\lambda, P, \mathbf{role}) \to P}$ *: run by a party* P *to output the handle of a party* P*, whose role is then set to either client (=C) or server (=S).*

$\underline{\Pi_{\mathtt{UReg}}(P, \hat{S}) \to (\{\mathsf{sk}, \mathsf{pk}\}, b)}$ *: an interactive protocol run by party* P *and super- user* $\hat{S}$*. The latter outputs a bit* b *(set to* 1 *for a successful registration), while the former outputs public/private credentials inside* $P.Attr$ *(which* $\hat{S}$ *may also learn). The super-user keeps track of a registration database indexed by user name.*

$\underline{\Pi_{\mathtt{Start}}(P, \mathsf{role}, \mathsf{pid}, \hat{S}) \to (\pi_P^{\mathsf{pid}}, b)}$ *: run interactively between* P *and super-user* $\hat{S}$*, so as to create an instance of* P *meant to be talking to an instance of* pid*, if the latter is possible (allowed by their roles). If successful,* $\hat{S}$ *outputs* $b = 1$*, while* P *outputs a handle* π_P^{pid}*. Some initial key material might be computed during this phase (like first stages of top keys in P's KH).*

$\underline{\Pi_{\mathtt{Evolve}}(\pi_P^Q, \pi_Q^P, st, key, AD) \to (\pi_P^Q, \pi_Q^P, AD^*) \cup \perp}$ *. This algorithm represents evolution at stage* $st = (x, y)$ *in the session between* P *and* Q *with additional data* AD*, where* P *is the initiator of the evolution. Essentially, the algorithm finds the value of* $P.\mathsf{KControls}_{P,Q}^{(x,y)}$ *and, if the flag is either* NI *or* $P\cdot$*, then the algorithmic steps for evolution (as described by* aDesc *will be assumed to occur. The evolution might require knowledge of some, or the entire – state of one or both parties* P *and* Q*. If the evolution is successful, the output are updated party instances* π_P^Q *and* π_Q^P*, as well as a potential list of ciphertexts resulting from the transmission, and corresponding* AD^**. If the evolution fails, a special symbol* $\perp$ *is output.*

Now, we give the definition of matching conversation: i.e., a communicating party is indeed as perceived.

Definition 2 (Matching Conversation). *Let* P *be a client,* Q *a server registered and running a* KE-DECK *protocol through instances* $\pi_P^\cdot$ *and* $\pi_Q^\cdot$ *respectively. We say these two instances have* matching conversation *if, and only if:* $\pi_P^\cdot.\mathsf{sid} = \pi_Q^\cdot.\mathsf{sid}$ *and* $\pi_P^\cdot.\mathsf{pid} = Q$*, and* $\pi_Q^\cdot.\mathsf{pid} = P$*.*

KE-DECK's Correctness. If π_P^Q and π_Q^P have matching conversation, then a KE-DECK protocol is *correct* if both conditions hold (in the absence of an adversary):

- For each $st = (x, y)$, both instances have identical secret & key values, down to the target key $\mathsf{t\_k}_{P,Q}^{(x,y)}$ and $\mathsf{t\_k}_{Q,P}^{(x,y)}$;
- For each $st = (x, y)$, both instances have compatible key-controls for all keys $key \in KH$: in other words, for all entry $(key, st, \tau, \mathtt{aEvolve}) \in \pi_P^Q.\mathsf{KControls}$, it holds that there exists a "matching" entry $(key, st, \tau^*, \mathtt{aEvolve}) \in \pi_P^Q.\mathsf{KControls}$, with identical values for key, st, and aEvolve, and entry τ^* such that:
  – If $\tau = \perp$ then $\tau^* = NI$ and vice-versa;
  – If $\tau = P\cdot$, then $\tau^* = \cdot Q$;
  – If $\tau = \cdot P$, then $\tau^* = Q\cdot$.

- Whenever P and/or Q use $\Pi_{\texttt{Evolve}}$, to update, then π_P^Q and π_Q^P still have matching conversation.

Stages. A *stage* is the period of time between two successive evolutions (not necessarily of the same key) in KE-DECK. If the parties prompt the evolution of the target key, then stages evolve horizontally (from (x, y) to $(x + 1, y)$). Else, stages evolve vertically (from (x, y) to $(x, y + 1)$).

Keys. We also write $\mathsf{k}^x(P, Q)$ for the x-th value of the key k (shared by P and Q and part of some stage $(x, \cdot)$ or $(\cdot, x)$).

Key Controls. In KE-DECK, at each stage $st = (x, y)$, each party in each instance has potentially-different *key-controls* $(key, st, \tau, \texttt{aEvolve}) \in P.\text{KControls}$, for each key $key \in KH$. In particular, aEvolve indicates whether an evolution of that key is possible at stage st. Note that, if $key = \mathsf{t\_k}_{P,\cdot}$ or $\mathsf{t\_k}_{\cdot,P}$, then, if aEvolve indicates that an evolution is possible, then that evolution is horizontal; else, it is vertical.

Key-Controls Conditions. The *value* τ indicates the conditions for a specific evolution. For instance, if party P has no say in the evolution, then the value $\tau = \perp$. If P is unilaterally responsible for an evolution (*i.e.*, the evolution takes only input from party P and can be triggered by that party) then $\tau = NI$. Also, we consider evolutions that are interactive, requiring input from both P and its partner; either P goes first – in which case $\tau = P\cdot$, or its partner goes first: $\tau = \cdot P$.

In the following section, we show how the adversary can hijack the key-controls, for instance by modifying the values of aEvolve for one or multiple stages. Obviously, the attacker's success depends on the robustness of the scheme with respect to such changes (for instance, if aEvolve is turned off entirely for 10 stages, maybe the partner of the corrupted party will be alerted to the presence of the attacker and shut down that instance). The goal of a protocol designer will be to either render key-control attacks impossible or to ensure that if key-controls are corrupted on the one side, then the partner of that instance will detect the attack.

3.2 Adversarial Model

We have a taxonomy of keys as follows:

Stage-specific Keys: keys occurring in only one stage of one protocol instance π_P^Q (typically including, but not necessarily limited to, the target keys). Namely, we require that for all instances π_P^Q and any two stages st and st', the value stored for stage-specific keys $key \in KH$ is different. We denote the set of all stage-specific keys of instance π_P^Q by $\pi_P^Q.1\textit{Stage-Keys}$.

Cross-stage Keys: keys that repeat in at least two stages: there exists an instance π_P^Q and distinct stages $st \in \pi_P^Q$ and $st' \in \pi_P^Q$, such that $k \in \pi_P^Q.\text{var}[st]$ and $k \in \pi_P^Q.\text{var}[st']$, but $k \notin \pi_P^{\cdot}.\text{var}$ for another instance from π_P^Q. We denote the set of cross-stage keys belonging to instance π_P^i as $\pi_P^Q.X\textit{Stage-Keys}$.

Cross-session Keys: keys that (intentionally) repeat in at least two sessions[2]: a key k is cross-session if there exist distinct instances $\pi_P^{\cdot}$, $\pi_P^{\cdot *}$ of registered party P, and (potentially distinct) stages $st \in \pi_P^{\cdot}$ and $st' \in \pi_P^{\cdot *}$ such that $k \in \pi_P^{\cdot}.\text{var}[st]$ and $k \in \pi_P^{\cdot *}.\text{var}[st']$. By definition, P's $\text{ltsk}(P, \cdot)$ and sk are cross-session keys. We denote P's cross-session keys as $P.X\text{sid}$.

Clearly, these three sets of keys are disjoint and form a true partition of the set of keys.

Our Adversary. The adversary's goal will be to distinguish, from random, a target key $\mathsf{t\_k}(P, Q)$ freshly and honestly generated. So, each instance needs to also store the attribute $\pi_P.\text{b}[st]$: a challenge bit randomly chosen for each instance for stage st. If $b = 1$, the output is the real key $\mathsf{t\_k}^{\text{st}}(P, Q)$ at stage st=(x, y), else the output is a random key.

Oracles. Our framework re-uses six *oracles* from [14]:

oUReg(P): runs aKeyGen on party P *i.e.*, $\mathcal{A}$ can register malicious P to an honest $\hat{S}$.

oStart(P, role, pid, *hon*): for an honest P, the algorithm runs $\Pi_{\texttt{Start}}$ to create a new instance of an existing honest party with the role P.role, and its intended partner pid. If P is malicious, the algorithm runs $\Pi_{\texttt{Start}}$ with the indicated role and pid as intended partner. The value *hon* is a bit: if set, the challenger poses as $\hat{S}$, whereas if $hon = 0$, the adversary poses as $\hat{S}$.

oTest$_b(\pi_P, st)$: for honest parties, valid instances and stages, and a target key $\mathsf{t\_k}^{\text{st}}(P, Q)$ at stage st, it returns that key (if $\pi_P.\text{b}[st] = 1$), or a same-length, random key (if $\pi_P.\text{b}[st] = 0$). It can only be queried once.

oReveal.1Stage-Keys(π_P, st). It leaks the set $\{\mathsf{t\_k} \in \pi_P.1\text{Stage-Keys}\} \bigcap \pi_P.\text{var}[st]$ of stage-specific values, for stage st.

oReveal.XStage-Keys(π_P, st): for stage st, it leaks the set of non-target keys $\pi_P.X\text{Stage-Keys} \bigcap \pi_P.\text{var}[st]$ of cross-stage values.

oReveal.XSid(P): corrupts P, giving $\mathcal{A}$ access to $P.X\text{sid}$.

Since our framework is no longer centred on the sending/receiving of message (as Blazy *et al.*'s work was), two of their oracles (sending and receiving) are replaced by an oracle that prompts either honest or malicious key-evolution:

oKE(π_P, π_Q, st, AD). This oracle can be run in honest or malicious mode. In the first case, the value AD is input as $\perp$, and the oracle runs $\Pi_{\texttt{Evolve}}$ for these arguments. Else, AD takes an actual value (or tuple of values) that represent the messages sent by P during the purported evolution. Note that an adversarial input of AD can make the partner instance π_Q abort.

Unlike [14], our framework allows an adversary to control keys in multiple ways, over an arbitrary number of stages:

oHijack.1Stage($\pi_P^Q, k, [st_1, st_2, \ldots ALL], \tau^*, \texttt{aEvolve}^*$). This oracle will return $\perp$ unless: P is honest, Q exists, instance π_P^Q exists, $k \in KH$ such that $k \in \pi_P^Q.1\text{Stage-Keys}$, and all stages $st_1, \ldots$ are current or future stages of π_P^Q. Note that ALL is a special value, which indicates that the input value to this oracle

[2] We thus formally exclude collisions in randomness. This is just at definitional level, w.r.t. classifying/defining types of keys, and it is not about collision in probabilistic events inside proofs. I.e., in proofs, there is a chance ck_1 will later reappear, for instance, as ck_{101}.

concerns all current and future stages of π_P^Q. The oracle allows the adversary to control the entry $(k, st_i, \tau, \texttt{aEvolve}) \in \pi_P^Q.\text{KControls}$ for all indicated st_i. If $\texttt{aEvolve}^* \neq \bot$, then the input τ^* is ignored, and the adversary modifies each entry $(k, st_i, \tau, \texttt{aEvolve}) \in \pi_P^Q.\text{KControls}$ to $(k, st_i, \tau, \texttt{aEvolve}^*)$. Else, if $\texttt{aEvolve}^* = \bot$, then the entries $(k, st_i, \tau, \texttt{aEvolve}) \in \pi_P^Q.\text{KControls}$ are changed to $(k, st_i, \tau^*, \texttt{aEvolve})$

$\text{oHijack.XStage}(\pi_P^Q, k, [st_1, st_2, \ldots ALL], \tau^*, \texttt{aEvolve}^*)$. This acts similarly to the above, but $k \in \pi_P^Q.X\text{Stage-Keys}$ (*i.e.*, for cross-stage keys).

The last two oracles add numerous dimensions to the types of adversaries captured in [14], as shown in Section 3.3.

3.3 Taxonomy of Adversaries

From [14], we import three dimensions of adversaries: *reach* (which keys are revealed); *power* (active or passive); *access* (is the adversary a person-in-the-middle or a super-user?); and *power* (active session-hijacking vs. passive).

In our framework, we seamlessly adapt the definition of *active adversaries* to include attackers that either run oKE in a malicious mode (with $AD \neq \bot$) or query one of the two hijacking oracles (oHijack.1Stage or oHijack.XStage). Intuitively, in the former case, the adversary is compelled to insert valid messages on behalf of an honest party in order to force an update, whereas in the latter, we assume it obtained illicit access to the victim device and it was thus able to modify the parameters present at the application layer.

Three New Adversarial Traits. To Blazy *et al.*'s traits (reach, power, and access), we add three new, which describe how strength in relation key-evolution controls:

Breadth: We distinguish between adversaries with *stage-specific* key-control access (which only query oHijack.1Stage), and *cross-stage* key-control access (they may also query oHijack.XStage).

Impact: We distinguish between *minimal* key-control hijacking (the attackers only modify aEvolve or τ for one stage), *moderate* key-control hijacking (the attackers modify aEvolve or τ, for multiple, but not all current and future stages), and *extreme* key-control hijacking (all current and future stages are compromised).

Layer: We distinguish between *algorithmic* key-control access (the attackers only modify aEvolve for each hijacked key-control), and *physical* key-control access (also modifying τ for at least one hijacking call).

Thus, an attacker making no key-control hijacking queries (compatible with Blazy *et al.*'s framework) will be, for instance, a medium passive outsider. If this now exerted key-control capacities, it can become, e.g., a medium passive outsider *with moderate stage-specific physical key-control.*

3.4 PCS Games & Definition in PCS_AC

Our generalisation and augmentation of notions in [14] enable us to also elegantly build upon their security definition.

(χ, Υ)-PCS security [14]. Informally, this security notion captures one counting the number of stages from the last key-reveal until the moment when security is recovered, *i.e.*, when an adversary only has a negligible advantage to distinguish the real target-key from a random target-key. Formally, the PCS attacker in [14] will, depending on its type, have access to different oracles. After the setup of all the honest parties and the super-user, the adversary will be free to use its oracles – and at some point, it will have to make its single oTest query. The attacker will then be free to use its other oracles and will eventually have to guess whether the bit input to oTest was 0 or 1. We denote by st^* – the stage at which oTest is queried, and by st – the last stage before st^* at which the adversary has revealed a key. The notion of (χ, Υ)-PCS security measures the distance between st and st^*, assuming that the adversary only wins with negligible advantage.

$\text{Exp}^{PCS}_{\textbf{KE-DECK}}(\lambda, \mathcal{A})$
$(\hat{S}.\text{sk}, \hat{S}.\text{pk}, \text{pparam}) \leftarrow C^{\text{aSetup}(1^\lambda)}$
$(\mathcal{P} = \{P_1, \cdots P_{\text{n}_\mathcal{P}}\} = \mathbb{C} \cup \mathbb{S}) \leftarrow C(\lambda, \text{n}_\mathcal{P})$
$(P.pk_i, P.sk_i) \leftarrow C^{\text{aKeyGen}(1^\lambda)} \quad \forall i \in \{1, \cdots, \text{n}_\mathcal{P}\}$
$O_{\text{type}} \leftarrow \{$ oUReg(·), oStart(·,·,·,·), oReveal[$\mathcal{A}$.reach](·,·), oKE(·,·,·,·), oHijack[$\mathcal{A}$.breadth](·,·,·,·), $\};$
$(\pi_P^\star, s^\star) \leftarrow \mathcal{A}^{O_{\text{type}}}(1^\lambda)$
$K \leftarrow \text{oTest}_{b^\star}(\pi_P^\star, s^\star)$
$d \leftarrow \mathcal{A}^{O_{\text{type}}}(\lambda, \text{n}_\mathcal{P}, K)$
$\mathcal{A}$ wins iff. $d = b^\star$ and $(\neg\text{oUReg}(P) \vee \neg\text{oUReg}(\pi_P^i.\text{pid})) = \top$
$k \cup \bot \leftarrow C^{\text{oForceEvolve}(\cdot,\cdot)}$

Figure 1: The PCS game $\text{Exp}^{PCS}_{\textbf{KE-DECK}}(\lambda, \mathcal{A})$ between adversary $\mathcal{A}$ and challenger C, parametrized by the security parameter λ and $\text{n}_\mathcal{P}$ *honest* parties. Depending on type, $\mathcal{A}$ can query a set of oracles O_{type}, including a subset of reveal oracles collectively denoted as oReveal[$\mathcal{A}$.reach], and a subset of hijacking oracles collectively denoted as oHijack[$\mathcal{A}$.breadth].

Our PCS Winning & Aborts. In our case, the adversary now has access to key-controls: for each key k and at each stage st, the adversary can modify whether a party expects an update for that key (as per our algorithm aEvolve), or it can modify how the party expects that update to happen, i.e., interactively, unilaterally, etc. (as per our parameter τ).

In our PCS game, adversaries can now hijack the key-controls for some stage st and thus win a number of stages "for free". If both parties involved in a session are affected by hijacking for a number of stages, then those stages are counted as trivial wins for the adversary. But, if only one instance in the exchange is hijacked, then this might cause the partner to abort; if this happens, the hijacking is essentially detected, and we say that the adversary lost.

More formally, say our attacker queries oTest at stage st^*, and that st is the last stage before st^* for which the attacker has revealed keys, or changed the evolution parameters of one instance, but not the other. Our measure (χ, Υ) is then measured as the minimum between the distance between st and st^* and the distance between st and the stage $\hat{st}$, when one of the two partners aborts.

Key Staleness. Whether the protocol heals or not, our adversary may also impact the expected freshness of keys. We call a *key* k *stale*, written $k^{\circlearrowleft}$, if the attacker stops it from evolving, although it should be able to evolve. We do not quantify all the keys which may have been made stale, but just the one(s) at the lowest level of the key-hierarchy.

In order to test for staleness, the challenger will simulate, subsequently to the attack, some verifications for the evolution of each key $k \in KH$ from the leaves to the root of KH; such verifications are *simulated*, so they do not change the states of the two instances.

More formally, the challenger, which knows the initial key-schedule and evolution conditions (before the attacker modified them) for all parties, will simulate successive evolutions, which we denoted by the following abbreviated "oracle":

• oForceEvolve(π_P^Q, π_Q^P): For each key k, starting from the leaves of KH and ending at the root, the challenger simulates successive key-evolutions based on the original protocol's evolution rules and the current states of PPQ and its partner in order to learn the earliest stage at which π_P^Q is meant to be able to evolve key k and actually manages to do so. The challenger output the first k for which this earliest evolution stage should be non-∞ (an update is possible), but is in fact ∞ (the attack made evolution impossible).

Our Resulting Metric. Thus, protocols will be associated with a $(\chi, \Upsilon, k^{\circlearrowleft})$-PCS value. By default, we will maintain the same metric as Blazy *et al.* and say that one scheme provides *more PCS-security in the presence of key-control hijacking* than another scheme if the former's Υ value is strictly smaller than the latter's Υ *or* if for equal Υ values, the first protocol's χ value is smaller than the second's.

Definition 3 (Extended $(\chi, \Upsilon, k^{\circlearrowleft})$-PCS security). *Consider a* KE-DECK *protocol* Π. *Assume an adversary* $\mathcal{A}$ *queries its unique* oTest *query for an instance* π_P^Q *at some stage* $st^* = (x^*, y^*)$. *Let* $st = (x, y)$ *be the last stage before the* oTest *query at which:*

- *the adversary queried its* oReveal.XStage *or* oReveal.1Stage *for either* π_P^i *or its matching instance; or*
- *The two conditions hold for the values* $(k, st, \tau,$ aEvolve$)$ *either in* π_P^Q *or in its partnering instance* π_Q^P*:*
  - *The adversary has modified either* aEvolve *or* τ *for* π_P^Q *or in its partnering instance* π_Q^P*, but not for both; and*
  - *The adversary has modified either* aEvolve *or* τ *for both* π_P^Q *and* π_Q^P *at the stage immediately preceding* st.

Let $\hat{st} = (\hat{x}, \hat{y})$ *be the earliest stage at which either* π_P^Q *or its partnering instance aborts the execution of the protocol session such that both instances have successfully attained stage* $\hat{st}$ *in a matching state.*

We call protocol Π $(\chi, \Upsilon, k^{\circlearrowleft})$*-PCS secure with key-control hijacking if the following conditions hold:*

- *Either* $\hat{st}$, st^*, *and* st *are such that* $\hat{y} - y < y^* - y$ *or* $\hat{y} - y = y^* - y$ *and* $\hat{x} - x < x^* - x$, *and then* $\hat{y} - y^* = \Upsilon$ *and* $\hat{x} - x^* = \chi$ *(if the distance between the abort and unilateral hijacking is smaller than that between* oTest *and the unilateral, then we assume healing occurs at the abort), or*
- *If* $\hat{st}$, st^*, *and* st *are such that* $\hat{y} - y^* > y^* - y$ *or* $\hat{y} - y^* = y^* - y$ *and* $\hat{x} - x^* > x^* - x$, *then the two following conditions hold simultaneously:*
  - *An adversary has a non-negligible advantage to win the game by testing at stage* st^* *such that:*
    - *If* $\Upsilon = 0$, $x^* < x + \chi$ *and* $y^* = y$;
    - *If* $\Upsilon > 0$, x^* *is arbitrary and* $y^* < y + \Upsilon$.
  - *If the adversary is also allowed to query* oReveal.XSid, *then* $\mathcal{A}$ *has a non-negligible chance to win for all instances of party P which are not yet instantiated, or have not yet reached stage* $st' = (x', y')$ *such that:*
    - *If* $\Upsilon = 0$, *then* $x' \geq \chi$ *and* $y' \geq 1$;
    - *If* $\Upsilon > 0$, *then* $x' > 1$ *and* $y' \geq \Upsilon$.
  - *Finally, the adversary has a negligible advantage to win if* oTest *is queried for* s_{Test} *other than those specified in the first of these three simultaneous conditions.*

Finally, it holds that the output of the challenger's run of oForceEvolve *returns* k.

Let us discuss some aspects of Definition 3.

Firstly, let us look at its intuitive meaning. It says: following a compromise, the protocol can either take a number of (horizontal $-\Upsilon$ and vertical $-\chi$) stages before healing, or it can take that many stages before an abort. Also, Definition 3 equates aborting the session with healing (since we chose to identify this abortion as a detection of compromise). This is why, if an abort occurs before a healing, at some stage $\hat{st}$, we count the number of vertical and horizontal stages between $\hat{st}$ and the stage of compromise st^* and define those to be the healing distance for an abort.

Secondly, given the above meaning, let us underline that Definition 3 does not give any numerical values for stage values st^* and st: these are the stages at which oTest is queried and respectively when the last major compromise is done before oTest. Their precise values are arbitrary, so our definition is equivalent to saying "whatever the values of st^* and st, ... are, the bound holds". Also, formulations such as "for any value of st^* and st, ..." are not the same and would not make sense, since that would imply that there are multiple queries of oTest.

4 MECHANISING PCS_AC IN SYMBOLIC TOOLS

Section 3 presented PCS_AC – a computational/cryptographic framework. So, one appeal would be to mechanise PCS_AC in computational tools such as Squirrel [12]. But, on the one hand, this would be arguably close to Section 3 and its associated proofs in Appendix B, whereas what we aim to emphasise next is that several aspects of PCS_AC make the mechanisation of (an abstraction of) PCS_AC attractive specifically in symbolic/Dolev-Yao [18] tools; next, we name such reasons:

(1) Our security notion inside PCS games has *discrete* measures, i.e., the healing speed is counted in several steps until security is restored.
(2) The local, global, medium type of PCS_AC attackers can be easily modelled via the leaking of keys in specific stages within a symbolic protocol-model.
(3) Our attackers' gain/loss in key-evolution controls, i.e., the PCS_AC parameters τ and aEvolve in KControls, can be modelled elegantly in symbolic verifiers.
(4) Finally, we use two state-of-the-art symbolic verification tools, Proverif and Tamarin. The reason for using both is that, recently, each tool introduced distinct ways to model counters, which is crucial for mechanising PCS_AC, and we also wish to compare and contrast these recent additions for counter-modelling by each tool.

Note that this mechanisation of PCS_AC in symbolic/Dolev-Yao tools, also implies the mechanisation of the framework of Blazy *et al.* [14] (since PCS_AC adds application-controls to all its dimensions, from model, to attacker to security notions).

4.1 Mechanisation PCS_AC Symbolically

We used two state-of-the-art symbolic verification tools, `Proverif` and `Tamarin`, for our symbolic models.

Both modelisations are based, w.l.o.g., on a key hierarchy in PCS_AC of 3 vertical levels: K_1, K_2, K_3, where K_3 is derived from K_2 and K_2 is derived from K_1.

As per PCS_AC, we do not bound the number of horizontal evolutions on any of these keys.

We model different KDFs between these keys: one allows for guaranteed, nonce-based freshness at a lower level (e.g., $K_3{:=}KDF_a(K_2, \textit{random-nonce})$), one does not (e.g., $K_3{:=}KDF_b(K_2)$, where the freshness of K_3 depends only on the freshness of K_2).

• Note: All our `ProVerif` and `Tamarin` files are at [11].

4.2 Mechanisation of PCS_AC in ProVerif

`ProVerif` [13] is a Dolev-Yao protocol-verifier, fully automated and supporting an unbounded number of protocol sessions. Its syntax is based on applied pi-calculus [8]. Within a `ProVerif` model, *events* are predicates on a process' variables, tagging points in its executions/*traces*. Generally, to prove a protocol's security properties, these *events* are then reasoned about using *queries*.

PCS_AC in ProVerif. We used the `ProVerif` version 2.05, to mechanise PCS_AC.

The file `PCS_LISTS.pv` [11] contains our mechanisation of PCS_AC in `ProVerif`. This `ProVerif` model encodes PCS_AC over a three-level key hierarchy, and unbounded number of horizontal evolutions of keys therein. Its main characteristics are:

(1) To model horizontal key evolutions, we defined a theory for lists indexed by integers (`ProVerif`-built-in type *nat*). Then, each level of the hierarchy is managed via such a user-defined list: K_1_list, K_2_list, K_3_list. That is, K_p_list$[i]$ keeps the i-th value key K_p. A PCS_AC stage (x, y) is then a tuple (K_p_list$[i]$, K_q_list$[i]$, ...) for $p, q \in \{1, 2, 3\}, p \neq q$.
(2) As per PCS_AC, the key-refresh "application trigger" at each key-evolution is controlled by the boolean result of the `aEvolve` oracle. To simplify notation, we call this trigger α – for K_3 and we call it β – for K_2. So, if α is $True$, then the K_3 value evolves and a new entry will be added to the K_3_list. For simplicity, the top-level key, K_1, is refreshed at each iteration.
(3) To model the key refresh of each level in the hierarchy, we firstly use user-defined function (i.e., the `ProVerif`, `letfun` construct). Such a function uses the relevant α and β flags and the KDFs, to either derive a new key or look up the previous, old value using its corresponding key list. The resultant key is then added back to its list with the current index i thus allowing us to keep track of the key-derivation history.
(4) As per PCS_AC, we need a target-key. The most possibilities w.r.t. controls/leaks are if we select the lowest key. So, we make K_3 our target-key and allow it to leak at a fixed, but user-defined point n, e.g., $K_3\_list[n]$ leaks.
(5) Finally, we formulate `ProVerif` queries that check the knowledge to post-leakage points on the K_3_list depending on the different values possible for α, β and the adversary's control of each, as well as the use of the fresh vs. non-fresh KDFs KDF_a and KDF_b.

These `ProVerif` queries show, *a la* PCS_AC's security in Def. 3, under which of these conditions the adversary knows $K_3\_list[m]$ when he knew $K_3\_list[n]$, and what is the compromise length $m-n$.

Screenshot-examples of queries are given in Appendix C.

Once again, our model allows for an unlimited number of end-key/horizontal evolutions. It proves in less than 8 minutes on a laptop with an Intel® Core™ i7-1065G7 CPU @ 1.30GHz (4 cores) with 32GB RAM.

4.3 Mechanisation of PCS_AC in Tamarin

`Tamarin` [23] is a Dolev-Yao [18] protocol-verifier, which also supports an unbounded number of protocol sessions. `Tamarin` models are transition systems (TS), whereby the transitions are modelled via *rules* containing logical predicates (called *facts*) over user-defined protocol variables. One can inspect (all or one) possible executions/*traces* of the TS via first-order formulae over facts.

PCS_AC in Tamarin. We also used `Tamarin` [23], version 1.8.0, to mechanise PCS_AC. The file `PCS_LISTS_nats.spthy` [11] is our mechanisation, in `Tamarin`, of PCS_AC, over a 3-level key hierarchy and unbounded number of horizontal evolutions of the keys within. The main characteristics of the model are:

(1) We used the recently added support for integers to emulate counting, up to some fixed m (as needed by the PCS winning conditions in Def 3).
(2) Possible key-leakages were implemented via Tamarin rules which allow the i-th value of the target key K_3 to be leaked. Concretely, the model includes a "LeakBranch" rule and a "NoLeakBranch" rule representing the presence/absence of leakage of one key at a stage. The transition through these different rules is controlled using restrictions on the facts in those rules.
(3) To model the key-control options w.r.t. PCS_AC's KControls and different KDFs, at each key level, we have two essential branches, implemented via alternative `Tamarin` rules: one rule for a key-update at that level ("do_updateKXBranch", where KX represents either K_1, K_2 or K_3, as per the above), another that re-uses the old key for that level ("doNot_updateKXBranch").
(4) The i-th key K_p is represented by an action fact, "UpdateKX(kx, index, refreshToken)", where KX again represents either K_1, K_2 or K_3, index is the i-th position and the refreshToken indicates whether or not the key was refreshed or not during that stage in the model.

We used the above model to demonstrate all that is expected given Def. 3, i.e., we prove a lemma, "compromiseK_3_alpha", that demonstrates that, once K_3 was leaked, if the attacker controls its key-evolution trigger α, then it can make K_3 stale. See Figure 10 for this lemma. Similarly, our lemma "compromiseK_2_beta" proves the attacker can make K_3 stale, by controlling the upper key-trigger β if the KDF is without freshness, KDF_b. See Figure 11, for this lemma.

The model requires a user-supplied oracle[3] to facilitate automated proofs. Again, our model allows for an unlimited number of end-key/horizontal evolutions. It proves within 40 seconds on the same aforementioned, commonplace laptop.

[3]This is a user-define heuristic to guide the state-exploration in `Tamarin`'s TS during its proofs.

4.4 PCS_AC in ProVerif vs. Tamarin

We now compare our modelling and subsequent PCS verification of PCS_AC in ProVerif vs. Tamarin.

Modelling Counting for PCS_AC's PCS Conditions. Our first point of call for symbolically-modelling PCS_AC was the 2.04 version of ProVerif, because it provided a new built-in type *nat*, i.e., an approximation for the theory of natural numbers; this was continued in the current 2.05 version of ProVerif. This allows us to:

- create user-defined integer-indexed lists, which, together with ProVerif's user-defined functions (letfun), permitted us to create a natural model of any key-hierarchy and key-evolution on top of it, i.e., our KE-DECK, its party-instances executing over stages;
- create *nat*-based user-defined queries over structures to show the key-evolution can be attacker-controlled for *exactly m* stages, before ceasing to be so;
- encode in a natural way, the winning-conditions (m, n) inside the PCS games.

Tamarin 1.8.0 now also offers built-in support for natural numbers. This allowed us to store counters for the different levels of key evolutions, similarly to Proverif. However, Tamarin's lack of support for lists and conditionals meant that we had to keep track of these key evolutions via rules and state predicates. This resulted in lengthier and less readable code.

Modelling PCS_AC's Stateful Stages. In ProVerif, we used our own user-defined list theory, with integer indices, for this. ProVerif also offers a built-in type of *table* which could have been used to implement the key look-ups. However, *table* is private (i.e., all its elements stay unknown to the attacker) by default, which did not suit us. Also, the performance of *table* proved to be an order of magnitude slower than our user-defined lists.

In Tamarin, on the other hand, no lists were modelled to keep track of the key hierarchy, as aforementioned. That said, as mentioned in Section 4.3, the action facts "UpdateKX" proved sufficient to query a key's position in the model.

Modelling PCS_AC's Key-Refresh Triggers and Options. Options such as non-deterministic key-refresh triggers were implemented in ProVerif via parameters in user-defined functions, and we can then simulate adversarial access to these. As if-then-else conditionals are also supported in ProVerif, switching between options is trivial.

Tamarin lacks functions and conditionals. So, to attain the above, we implemented separated rules for each alternative branch of a behaviour resulting in a harder-to-read model with duplicated code sections.

Performance. In Proverif, we first encountered some performance issues, which we overcame. Concretely, instead of symbolic i variables (see our executability lemma on i,$i+1$) we used concrete values (e.g., 3, 4), then Proverif built a partially instantiated model of a certain depth (e.g., for 4 stages in the horizontal key-evolutions) and therefore there were an exponential number of partially-instantiated choices, and it performed badly. So, in the end, we did not use concrete values, and Proverif transforms the model entirely into a set of symbolic Horn clauses where resolution has no tractability issues.

Overall, in terms of performance, both tools proved their respective queries/lemmas quickly, with Tamarin outperforming ProVerif due to it is effective, multi-threaded implementation. However, one proof in Tamarin requires manual intervention[4].

Thus, both symbolic verification tools were able to model PCS_AC. ProVerif's letfun constructs, its native support of integers via *nat* and if-then-else allowed for a more natural encoding. For this mechanisation, ProVerif was therefore our preferred choice despite its slower overall performance.

5 APPLYING PCS_AC

We first apply our PCS_AC to the 5G AKMA protocol. Also, our theorems for AKMA show that PCS_AC exhibits attacks that [14] could not.

To apply this to AKMA, we first have to introduce it and we do so in the next Section 5.1.

5.1 5G AKMA & Our AKMA Improvement – AKMA$^+$

In 5G (5th Generation) mobile networks, a procedure for delegated authentication was added: i.e., a kind of "Single-Sign-On"-cum-"OAuth" called the *Authentication and Key Management for Applications based on 3GPP credentials in the 5G Systems*, *AKMA* for short [5]. Using AKMA, for instance, a driver inside a connected car *securely* employs a proprietary navigation system to an e-charging space or *securely* pays for road services and tolls automatically, just using the car's SIM-card subscribed to a mobile network for authentication to any of these 3rd-party systems.

Primer on 5G. In Figure 2, we present a simplified overview of the relevant 5G network entities for AKMA:

(1) the *User Equipment (UE)* – a device subscribing to mobile service;
(2) the *Radio Access Network (RAN)* – the 5G radio "towers/base-stations" also called *gNBs* that "ferry" the service to the UEs;
(3) the *5G core* [3, 4] – servers implementing the operator's logic, split into services: e.g., the *Authentication Server Function (AUSF)* and the *Access and Mobility Management Function (AMF)* authenticate the UE via the protocol called the *5G Registration / AKA (Authentication and Key Agreement)*; the *User Plan Functions (UPFs)* are gateways for routing the user to the internet; *Network Exposure Function (NEF)* is an API-based proxy for 5G to allow third-party applications' queries; e.g., some of these calls go to the core via the *Applications (AKMA) Anchor Function* (*AANF*);
(4) *Application Functions (AFs)* – 3rd-party application-servers leveraging, e.g., the network's authentication.

5G AKMA [5], or simply AKMA, is in fact a delegated authentication service in 5G: applications (AFs) outside the network "ask" the core to authenticate UEs. Thereafter, the core will compute a key called K_{AF} *(application function key)* and deliver it to the AF. Finally, a UE and an AF server will establish a channel secured with

[4]While this takes longer, having the option of manually guiding a proof is actually one of Tamarin's strengths.

said key. Figure 13 shows, at the high level, how the AKMA service have an "inverse" dependency on the main 5G authentication keys, all the way up to the so-called K_{AUSF} established at the end of the frequent *Registration/AKA* procedure [5] and sitting "high" in 5G's key-hierarchy.

5.1.1 AKMA-relevant Keys: K_{AKMA} & K_{AF}. The derivations of the AKMA-relevant keys are given next:

$$K_{AKMA} = KDF(const, K_{AUSF}, \text{"}AKMA\text{"}, \text{SUPI}) \quad (1)$$

$$K_{AF} = KDF(const, K_{AKMA}, AF\_ID) \quad (2)$$

As shown in Figure 13, K_{AKMA} is a key computed at the end of *Registration/AKA* procedure [2] and K_{AF} is computed asynchronously as part of the AKMA protocol, KDF is a hash, *const* symbolise hex constant values, *SUPI* is an ephemeral identifier of the UE, and AF_ID is constructed as $AF\_ID = AF||Ua*$ with $Ua*$ being the identifier[5] of the protocol eventually used at the application level between the UE and an application server associated with an AKMA Application Function (AF). An example of AF_ID is "hex(name_of_AF) || 0x01 0x00 0x00 0x00 0x02" where "0x01 0x00 0x00 0x00 0x02" stands for the Ua* security protocol for "HTTP digest authentication".

Note A: For a given AF and protocol Ua*, the AF_ID is a constant, so knowledge/freshness of a new K_{AF} is based on the freshness of its K_{AKMA} value.

Note B: Similarly, knowledge of a new K_{AKMA} is based on the knowledge of its K_{AUSF} and SUPI value.

5.1.2 ***Generating the AKMA-relevant Keys.*** AKMA is run between a UE, an AF (Application Function) and the core[6], i.e., the AAnF (Application Anchor Function).

At the end of 5G authentication (i.e., Registration/AKA [2]), an *AKMA-ready UE* and the AAnF will both hold a new K_{AKMA} key, one for all AFs it can communicate with; this instance of the K_{AKMA} key will be indexed under a *AKMA Key Identifier (A-KID).* See Figure 3. Based on this K_{AKMA} key, the UE and the AF associated with an A-KID can derive deriving a new K_{AF} key. This is shown in Figure 4 and described[7] as follows.

- The UE sends its A-KID on the AF's so-called Ua* channel/protocol.
- The AF sends the AAnF this A-KID and its own identifier, on the 5G-network's channels for this.

[5]This is specified in Annex H of 3GPP 33.220 TS [7].
[6]It is run between the AF and the AAnF – if the AF is internal to the operator, and between the AF, the AAnF and the NEF – otherwise.
[7]We describe this w.r.t. revision 17.07 of the 3GPP specifications. The current revision is 18, yet no difference is of essence here.

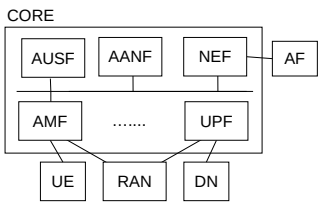

Figure 2: Overview of 5G – see [3, 4].

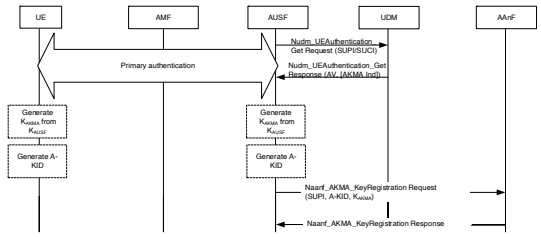

Figure 3: Deriving the K_{AKMA} key (Figure 6.1-1 [5]).

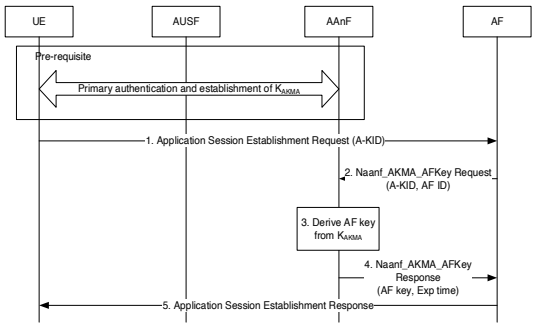

Figure 4: Deriving the K_{AF} key (Figure 6.2-1 [5]).

- When the AANF receives the request, it checks that the contacting AF can provide service to the UE linked to the A-KID, based on the UE's information.
  If successful, a new K_{AF} key and its time-to-live (*TTL*) is sent by the AAnF to the AF; otherwise, a descriptive error is sent back, instead.
- If successful, the AF sends a message to the UE confirming the start of a new AKMA/Ua* session. If unsuccessful, the AF sends an error message to the UE with the error cause.
- In case of success, the UE computes the K_{AF} key, just as the core did, using equation (2) above, and the new, K_{AF}-encrypted AKMA/Ua* session can now truly start between the UE and the AF.

5.1.3 ***Freshness of the AKMA-relevant Keys.*** The crux on new/fresh K_{AF}s is summarised below:

(1) For a given A-KID, two K_{AF}s in a series can be the same, as they can be issued out of the same K_{AKMA}, as the 3GPP specifications convey: (a) *"K_{AKMA} and A-KID can only be refreshed by a new successful primary authentication*[8]" (Section 6.1.1 [5]); (b) *"... when a new K_{AKMA} is derived, the K_{AF} will not be re-keyed automatically."* (Section 6.4.2 [5]); (c) *" When the K_{AF} lifetime expires and the K_{AKMA} has not changed in AAnF, according to the Annex A.4, the AKMA Application Key which is established based on the current AKMA Anchor Key KAKMA is not a new one"* (Section 5.2 [5]).

(2) The expiration-time of the K_{AF}s can be made longer/shorter via the third party Ua* protocol, as the 3GPP specifications convey: (a) *"The Ua* protocol shall be able to handle the expiration of KAF".* (Section 4.4.1 [5]); (b) *"If the Ua* protocol*

[8]This means Registration/AKA.

supports refresh of K_{AF}, the AF may refresh the K_{AF} at any time using the Ua protocol."* (Section 6.4.3 [5]).

5.1.4 Using AKMA & The Importance of Its K_{AF}s. To put certain discussions w.r.t. AKMA in a concrete context, consider Figure 5 below. Radio nodes connect the UE "out to the internet" via different gateways (UPFs), which are decided as part of Registration/AKA and/or handovers [2]. All this traffic is encrypted with keys shared with between the nodes and the UE. When the UE re-registers upon a connection loss, or when it changes physical location and enters the range for new nodes (see Fig. 5), these mobile-traffic encrypting keys also change (as may do the UPFs); this happens frequently, especially for a fast-moving UE, such as a car. AKMA brings more stability to this, by providing dedicated lines of traffic between the UE and certain servers in the "data network". For instance, in Fig. 5, a connected car can use the internet just as any SIM-enabled device, but also with have an AKMA-provided, dedicated channel to a BMW server. In this case, BMW holds an AF that runs AKMA with the UE/car and the core, and BMW also has, e.g., an HTTP server, that the UE/car has K_{AF} encrypted traffic to/from. The protocol run in the "K_{AF}-encrypted tunnel" between the UE/car and AF's/BMW's server is the aforesaid $Ua*$.

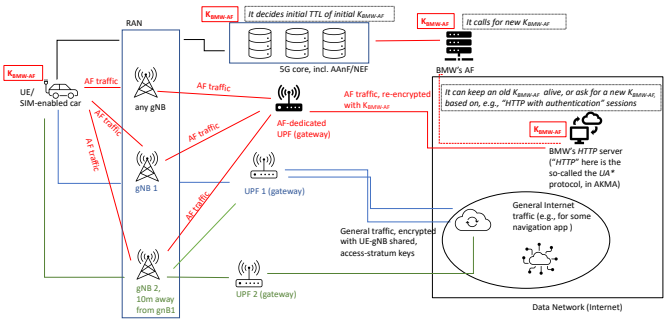

Figure 5: Using AKMA & K_{AF} Keys in Real Life

5.1.5 AKMA$^+$ *– An Idealised Version of AKMA.* As per Note A, Note B and point 1 above, for a new K_{AF} to be truly fresh, Registrations/AKAs ought to happen at least as often as the K_{AF}-regenerations, and this is not guaranteed. But, if we operate in a distribution of Registration/AKA-calls and of K_{AF} regenerations where the former is always more frequent than the latter, then we refer to the resulting, idealised AKMA procedure as AKMA$^+$.

5.2 Applying the PCS_AC to AKMA and AKMA$^+$

Let $\mathcal{P}$ be the set of honest users with unique identifiers, composed of clients $\mathbb{C}$, server $\mathbb{S}$, and a single super-user $\hat{S}$. In practice, the super-user $\hat{S}$ is the core (*e.g.*, the operator which provides services to the UE).

Setup. The server chooses the cryptographic suite (*e.g.*, signature algorithms, KDFs, etc.) to be used during the protocol. We assume that an existing procedure for the AF to get authenticated by $\hat{S}$ (be it with hardware solution or secure channel establishment) is used; we thus abstract the authenticating phase and assume that the correct parties are involved. Notice that a variant of AKMA keeps the core and AF as a unique component but this configuration is not considered, as it *hides* the vulnerability of AF. So we emphasise that the core and AF are distinct entities.

The server $\hat{S}$ generates a key pair $(\hat{S}.\mathsf{sk}, \hat{S}.\mathsf{pk})$ used in the secure-channel establishment with each party in $\mathcal{P}$. We also abstract this part and assume that communications between parties and $\hat{S}$ are secure. In particular, we assume that $\hat{S}$ and each party in $\mathcal{P}$ share a secret (which is, in practice, done within the SIM of the UE).

Key Generation. Each party $P \in \mathbb{C} \cup \mathbb{S}$ generates a key-pair $(P.pk, P.sk)$.

User Registration. Each party P has long-term symmetric key shared with $\hat{S}$, denoted $\mathsf{ltsk}(P, \hat{S})$.

Instance Initialisation. A client $C \in \mathbb{C}$ starts a session with a server $S \in \mathbb{S}$ by querying the core $\hat{S}$. The key-hierarchy $C.KH$ is constructed, then $\hat{S}$ initialises the session for S thus constructing $S.KH$.

Key Evolution. We abstract the actual sending/receiving phase and only consider the key evolution within the session.

Key Material. For AKMA, in PCS_AC this is:

- Cross-Session Keys: K;
- Cross-Stage Keys: $\mathsf{K}_{\mathsf{AKMA}}$;
- Single-Stage Keys: K_{AF}.

The PCS-security of AKMA. Having modelled AKMA as a KE-DECK, and split the key material in the corresponding categories, we can now analyse its PCS-security.

Theorem 1. *Consider the AKMA protocol modelled as a* KE-DECK *scheme. In the random oracle model (by replacing the KDFs with random oracles), and assuming AKE-security of the channels established between honest users and an honest $\hat{S}$, AKMA is:*

- $(\infty, 1)$*-PCS secure against local outsider without key-control;*
- $(\infty, 1, {K_{AF}}^{\circlearrowleft})$*-PCS secure against local outsider with single-stage key-control;*
- $(\infty, \infty, {K_{AKMA}}^{\circlearrowleft})$*-PCS secure against local outsider with cross-stage key-control;*
- $(\infty, 1)$*-PCS secure against medium outsider without key-control;*
- $(\infty, \infty, {K_{AF}}^{\circlearrowleft})$*-PCS secure against medium outsider with single-stage key-control;*
- $(\infty, \infty, {K_{AKMA}}^{\circlearrowleft})$*-PCS secure against medium outsider with cross-stage key-control;*
- (∞, ∞)*-PCS secure against insider or global.*

The proofs are given in Appendix B.1, each in two steps. First, we show a lower bound of the metric by explaining an attack (*i.e.*, there is no healing faster than this bound). Second, we show that the security holds beyond the given value. And, for $(\infty, \infty, \cdot)$ cases, we only need to give an actual attack (no healing/security is ever occurring/recovered).

5.3 PCS_AC onto AKMA$^+$, TLS1.3, and Others

In Theorem 2 (Appendix B.2), we move to not considering the *layer* adversarial-trait, as our metric considers the worst case: i.e., adversary targets the component with the maximal key control (for AKMA, this is AAnF, which chooses the time-to-live of K_{AF}). In tandem, for AKMA, the *impact* adversarial-trait makes little sense, since K_{AF} could be set with an endless time-to-live. However, this is

not possible in AKMA$^+$, since there any new K_{AKMA} will produce a new K_{AF}.

Indeed, PCS_AC can be applied to all protocols with repeated key-evolutions over a key-hierarchy. The more complex the key-hierarchy, the more subtle the application controls and the more attackers' type plausible, the better can PCS_AC show its value in systematisation of PCS-driven formal measuring on such an application.

One use-case that well suits this PCS-versatile setting is TLS1.3 with session resumption, session tickets and key updates. We use this complex use-case to underline the versatility and usefulness of PCS_AC. This is shown in Appendix E.

5.4 Verifying AKMA and AKMA$^+$ in the Symbolic Mechanisation of PCS_AC

In Section 4, we showed for PCS_AC can be mechanised in Tamarin and/or Proverif. Now, we discuss how that can be applied to AKMA and AKMA$^+$. To do so, we just needed specialise the key-hierarchy in our mechanisation of PCS_AC to that in Section 4.1 (i.e., K_1=K_{AUSF}, K_2=K_{AKMA}, K_3= K_{AF}); two KDFs in the PCS_AC mechanisation already adhere to AKMA's KDFs, a la Notes A and B in Section 5.1.3, and another has added freshness as needed by AKMA$^+$'s KDFs. Both for AKMA and AKMA$^+$, we found the same results as in Section 5, but –this time– via our mechanisation of PCS_AC in Proverif and Tamarin.

6 IMPLEMENTING AKMA & AKMA$^+$

We show: (1) the first implementation of AKMA; (2) an implementation of AKMA$^+$ (Section 5.1.5), which we formally proved (Section 5) to be a PCS-driven improvement of AKMA; (3) an efficiency comparison between the two.

Our implementations are on Fraunhofer's 3GPP-compliant testbed called Open5GCore [20]. This is not open-source, so we cannot share all our entire code, but – at [11]– we share parts of it w.r.t. our experiments.

Overview of Open5GCore. Open5GCore is Fraunhofer's testbed for 5G networks written in C. The toolkit offers numerous features of the *5G core*. However, with respect to the Network Exposure Function (NEF), it only offers simplistic capabilities such as quality-of-service measures (see [21]) and does not support the AKMA procedure. We augmented Open5GCore's NEF capabilities by adding to it the AKMA procedure[9] in line with its latest 3GPP specification [6].

Adding AKMA to Open5GCore Firstly, we added all the AKMA protocol-messages to the Open5GCore. This had to be in line with a YAML file called Nnef_AKMA [1] that specifies how the NEF and AF interface inside the Open5GCore. This is via are simple API calls, following a specific format, emulating the YAML definitions. Some additional details on the YAML files are given in Appendix D.

Secondly, we implemented in C the logic behind these AKMA messages, i.e., we derived K_{AF} as per [5], using the objects/classes producing the "authentication vectors" which include the K_{AUSF} and K_{AKMA} keys.

[9]In Section 5.1, we presented the AKMA without an NEF, but this is simply a proxy between AFs and the AANF.

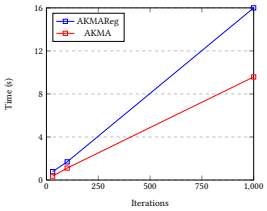

Figure 6: AKMA vs AKMAReg Execution Times

Adding AKMA$^+$ to Open5GCore To lift AKMA to AKMA$^+$, we need to add calls to Registration/AKA [2] every time a new K_{AF} is generated onto Open5GCore Thus, as part of the K_{AF} refresh, we implemented a call by the AF to the NEF/ AAnF, and in turn, a call by the AAnF to the AMF, to do Registration/AKA.

AKMA vs. AKMA$^+$ Experimentation

For this comparison, we fixed one UE, one AF, one AFID, one AKID, and considered that, when needed, full Registration/AKA [2] takes place. In this context, we compared the bandwidth depreciation (number of packets) in AKMA$^+$ vs. AKMA, as well as the execution times between the two in 1, 30, 1000 and 1000 runs of one vs. the other.

	Nb. of Packets for 30 Runs			
	at NEF	at UE	at AMF	Total
AKMA	780	120	116	1016
AKMA$^+$	780	240	191	1211

Table 1: 5G AKMA vs. AKMA$^+$ Nb. of Packets for 30 Runs

Packets' number are in Table 1: at the NEF, no change between the two procedures, since the NEF is not involved in Registration/AKA; at the UE, 4 more packets per call of AKMA$^+$ vs. AKMA; at the AMF, 25 packets more[10].

Figure 6 shows the execution times. There is no perceptible change between one single call of AKMA and one of AKMA$^+$(11,000 vs. 20,000 microseconds). Yet, AKMA and AKMA$^+$ take respectively: 359,000 and 782,000 microseconds (μs) for 30 executions, 1,110,000 vs. 1,690,000 μs for 100 executions, and 9 mil. vs. 16 mil. μs for 1000 executions.

Experiments were run on a multi-threaded installation of Open5GCore on a Ubuntu 18.04 virtual machine, with an Intel(R) Core(TM) i5 CPU 2.4GHz, and 8GB of *RAM*.

Disclosure and Possible Adoption. Given what we show formally and experimentally, we propose that AKMA *optionally* work as AKMA$^+$. We filed a disclosure form on this with 3GPP, and the conversations are ongoing.

7 RELATED WORK

In Sections 1, 3.2, 3.3, we discussed extensively, the differences between PCS_AC and Blazy *et al.* [14]. And, in turn, they formally compare with other known PCS models in [14]: e.g., there are separations between them and [16], which we inherit, One can also see PCS_AC as motivated by a recent PCS work in [19]. It shows that if active, a messaging-protocol adversary can prevent traditional PCS "healing", and even act as a man-in-the-middle indefinitely and without detection. We take that further and formalise such

[10]Yet, packets at AMF are larger than at the UE.

adversarial abilities but specifically at the application level, in general and not just for messaging, we taxonomise these controls, and even look at conditions for preventing the attacker to win at such enhanced PCS games (see Definition 3).

Finally, we note that, w.r.t. to symbolic verification, the AKMA as an authentication protocol, or from privacy perspectives, has been looked at very recently: in `ProVerif` – in [9, 10], and in `Tamarin` – in [25]. However, this is was w.r.t. classical Dolev-Yao agreement, not w.r.t. to *key-evolution* or PCS, *key-evolution* PCS as per `PCS_AC`.

8 CONCLUSIONS

We proposed `PCS_AC` – a new computational framework for post-compromise security (PCS), which adds adversarial controls at the application level. We gave a Dolev-Yao mechanisation of `PCS_AC`, in `ProVerif` and `Tamarin`. `PCS_AC` applies to all protocols with repeated key-evolutions, in line with what NIST recently calls for [15]. We show-cased `PCS_AC` on a study of 5G AKMA; we also suggested a backwards-compatible, PCS-driven improvement called AKMA$^+$, now under consideration by 3GPP. We implemented and tested AKMA and AKMA$^+$, on top of Fraunhofer's `Open5GCore`. We captured new and varied PCS dimensions of TLS1.3 with session resumption and key updates, via its casting under `PCS_AC` in various ways.

REFERENCES

[1] 3GPP. YAML OpenAPI Gitlab for 3GPP 5G Core Network (Release 17).
[2] 3GPP. Procedures for the 5G System. Technical Specification (TS) 23.502, 3rd Generation Partnership Project (3GPP), 10 2021. Version 16.7.0.
[3] 3GPP. 5G Security Assurance Specification (SCAS); Access and Mobility management Function (AMF). Technical Specification (TS) 33.512, 3GPP, 07 2022. Version 16.3.0.
[4] 3GPP. 5G Security Assurance Specification (SCAS) for the Session Management Function (SMF) network product class. Technical Specification (TS) 33.515, 3GPP, 07 2022. Version 16.2.0.
[5] 3GPP. Authentication and Key Management for Applications (AKMA) based on 3GPP credentials in the 5G System. Technical Specification (TS) 33.535, 3GPP, 09 2022. Version 17.
[6] 3GPP. Network Exposure Function Northbound APIs. Technical Specification (TS) 33.535, 3GPP, 07 2022. Version 17.6.0.
[7] 3GPP. Digital cellular telecommunications system (Phase 2+), ... Technical Specification (TS) 33.220, 3GPP, 01 2023. Version 17.4.0.
[8] Martín Abadi and Cédric Fournet. Mobile values, new names, and secure communication. *ACM Sigplan Notices*, 36(3):104–115, 2001.
[9] Gizem Akman, Philip Ginzboorg, Mohamed Taoufiq Damir, and Valtteri Niemi. Privacy-enhanced AKMA for multi-access edge computing mobility. *Comput.*, 12(1):2, 2023.
[10] Gizem Akman, Philip Ginzboorg, and Valtteri Niemi. AKMA for Secure Multi-access Edge Computing Mobility in 5G. In Osvaldo Gervasi, Beniamino Murgante, Sanjay Misra, Ana Maria A. C. Rocha, and Chiara Garau, editors, *Computational Science and Its Applications - ICCSA 2022 Workshops - Malaga, Spain, July 4-7, 2022, Proceedings, Part IV*, volume 13380 of *Lecture Notes in Computer Science*, pages 432–449. Springer, 2022.
[11] Anonymous. Anonymous Files, 2024. https://gitlab.com/Anonymous123AB/pcs-appl.
[12] David Baelde, Stéphanie Delaune, Charlie Jacomme, Adrien Koutsos, and Solène Moreau. An interactive prover for protocol verification in the computational model. In *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*, pages 537–554. IEEE, 2021.
[13] B. Blanchet. An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In *IEEE CSFW*, 2001.
[14] Olivier Blazy, Ioana Boureanu, Pascal Lafourcade, Cristina Onete, and Léo Robert. How fast do you heal? a taxonomy for post-compromise security in secure-channel establishment. In *Usenix Security Symposium*, 2023.
[15] Lily Chen. Recommendation for key derivation using pseudorandom functions. online, 2022. https://csrc.nist.gov/pubs/sp/800/108/r1/upd1/final.
[16] Katriel Cohn-Gordon, Cas Cremers, Benjamin Dowling, Luke Garratt, and Douglas Stebila. A Formal Security Analysis of the Signal Messaging Protocol. *Proceedings - 2nd IEEE European Symposium on Security and Privacy, EuroS and P 2017*, (July):451–466, 2017.
[17] Véronique Cortier, Steve Kremer, and Bogdan Warinschi. A survey of symbolic methods in computational analysis of cryptographic systems. *J. Autom. Reason.*, 46(3-4):225–259, 2011.
[18] D. Dolev and A. Yao. On the Security of Public-Key Protocols. *IEEE Trans. Inf. Theory 29*, 29(2), 1983.
[19] Benjamin Dowling and Britta Hale. Secure messaging authentication against active man-in-the-middle attacks. In *2021 IEEE European Symposium on Security and Privacy*, pages 54–70, 2021.
[20] Fraunhofer. Open5GCore, 2023.
[21] Fraunhofer. Open5GCore: Advanced QoS and Session Management, 2023.
[22] M. Marlinspike and T. Perrin. The double ratchet algorithm, 2016.
[23] S. Meier, B. Schmidt, C. Cremers, and D. Basin. The TAMARIN Prover for the Symbolic Analysis of Security Protocols. In *CAV*, pages 696–701, 2013.
[24] Eric Rescorla. The transport layer security (tls) protocol version 1.3. Technical report, 2018.
[25] Tengshun Yang, Shuling Wang, Bohua Zhan, Naijun Zhan, Jinghui Li, Shuangqing Xiang, Zhan Xiang, and Bifei Mao. Formal Analysis of 5G Authentication and Key Management for Applications (AKMA). *Journal of Systems Architecture*, 126:102478, 2022.

A REALISTIC NATURE OF OUR THREAT MODEL

We now explain our attacker's capabilities, showing that they are realistic, pragmatic and necessary today.

• What do we mean by application-layer or application controls, and how important are they?

Take a key-establishment protocol such as TLS that secures an application-layer protocol such as HTTP (to make it HTTPS). If the attacker controls some aspect at the HTTP(S) level, that may affect their ability to mount or prolong an attack on the session-keys established via TLS. The same is the case with the Ua* protocol(s) and the keys established via the AKMA procedure.

Our attacks and PCS proofs on AKMA (see Section B), and our attacks on 'TLS1.3 with session tickets' (see Section E) show just this.

• To have application controls, the attacker does NOT compromise a device/application entirely or forever.

Our attacker *only* needs to control some aspect of the application that deals with refreshing keys (e.g., generation of session tickets). Concretely, say he/she could have polluted a Javascript that deals with the TTL of session tickets in TLS. Or, he/she can manipulate a cookie for some time, until it gets cleared. Also, this control can be very ephemeral (e.g., an attacker has limited access to a server for 30 mins).

• There two different/independent PCS attacking powers: (1) manipulating application controls; (2) controlling/compromising keys in a hierarchy that is used by the application.

Concretely, imagine a key where K_2 depends on K_1, in that of K_1 is refreshed, then K_2 is refreshed. Then, let the attacker have power over the "application controls" of K_1 but not that of K_2. Then, when K_2 is refreshed, the attacker loses control over K_1 as well, no matter if the attacker manipulates the control (e.g., TTL) of K_1. That is, K_2 has now been refreshed due to an "upper" key regeneration and this supersedes manipulations the "application controls" of K_1.

This is what happens in AKMA and in TLS, as per our results (Section B and Section E), where the keys in a key hierarchy/tree can evolve independently (via application controls), as well as evolve due to "upper keys" refreshes.

• In mobile networks, to partly control the application layer of the UE/phone by keeping it temporarily in an 'alive/connected' state, the attacker does not need to run malicious code on the device.

Indeed, if an attacker only wants to keep the phone/UE from re-registering for a given period (e.g., say a 1h window, whilst they know some keys are due to be regenerated), all the attacker need to do is to keep the device in the same rough location for that time and turn the screen on/off a few times (e.g., spend a coffee break with the device's owner, looking at something on their phone every so often).

B PCS PROOFS FOR AKMA AND AKMA$^+$ IN PCS_AC

We consider all KDF modelled as random oracles. The security games are parametrised by a maximal number of stages $\mathsf{n}_{\mathsf{x-max}}$ in a given chain with a maximal number of chain $\mathsf{n}_{\mathsf{y-max}}$. The maximal number of clients and servers are denoted $|\mathbb{C}|$ and $|\mathbb{S}|$ respectively. The number of session created by any party is n_π. The set of a key k is given by $||$.

In all the proofs, we exclude collision from derivated keys. This comes from two fact: (1) the long-term symmetric keys associated to each session are computed by $\hat{S}$; and (2) the KDFs are assumed to have an output space large enough to avoid collisions. So the input are not colliding and the probability of outputing the same value with different inputs is negligible.

The proofs are done via game-hops where the initial game corresponds to the security game in Fig. 1, and the last game is designed such that the adversary has no advantage.

We can describe the first games common to all cases:

$\mathbb{G}_0$: This game corresponds to the original security game (Fig. 1 of Sec. 3.4). The advantage of $\mathcal{A}$ is Adv_0.

$\mathbb{G}_1$: The challenger guesses which P (from $\mathbb{C}$) and which Q (from $\mathbb{S}$) are chosen; it also chooses randomly the session from the maximal number of possible sessions n_π; it also chooses the targeted stage (for which $\mathcal{A}$ has queried oTest).

If the challenger does not guess correctly, then a random bit is drawn and the game aborts. Otherwise, the game continue and we have:

$$\mathsf{Adv}_0 \leq |\mathbb{C}| \cdot |\mathbb{S}| \cdot \mathsf{n}_\pi \cdot \mathsf{n}_{\mathsf{x-max}} \cdot \mathsf{n}_{\mathsf{y-max}} \cdot \mathsf{Adv}_1$$

B.1 Proof of Theorem 2 (PCS_AC security for AKMA)

Local outsider without key-control. This case is actually hard to model since the metric does depend on external factor than purely algorithmic/applicative. Indeed, our metric is given through *worst-case scenario* which considers the maximal number of compromised stages.

Let us consider the attack: the adversary reveal K_{AF} which can or cannot evolved. This imprecision is due to the specification of AKMA on the derivation of K_{AF}. The latter is derived with the upper key $\mathsf{K}_{\mathsf{AKMA}}$ but independently from the case where $\mathsf{K}^i_{\mathsf{AKMA}}$s old or new. This means that if $\mathsf{K}_{\mathsf{AKMA}}$ does not evolve then K_{AF} is always derived from the same value (which gives always the same K_{AF}). Notice that the evolution of $\mathsf{K}_{\mathsf{AKMA}}$ depends on various factor which are not considered in our model (other than adversarially ones). For instance, the user equipment could avoid idle state thus not executing a Reg/AKA procedure which in turns make the $\mathsf{K}_{\mathsf{AKMA}}$ static.

The worst-case scenario is thus from a not evolving $\mathsf{K}_{\mathsf{AKMA}}$ (which is possible without $\mathcal{A}$ intervening) which produces a static K_{AF}. This means that the key hierarchy is *stuck* in a single chain composed of one K_{AF} without being a key stale per se (the key still could evolved). Notice that the case with optimal healing could also occur. This means that the adversary has compromised one stage (which is, by assumption, possible) but cannot compromise future stage, and a new $\mathsf{K}_{\mathsf{AKMA}}$ is generated.

Recall that the adversary is local (the K_{AF} is revealed which defines the compromised stage), outsider (with no access to the super-user $\hat{S}$) and without key-control (so the key hierarchy is not changed by $\mathcal{A}$).

We can now resume to our security games.

$\mathbb{G}_2$: We ensure that the value $\mathsf{K}^{1,y^\star}_{\mathsf{AF}}$ is unique. If there are two

equal values in a session, or in two different (honest) sessions, then the challenger aborts, returning a random bit.

Recall that the random oracle model implies that a call to the KDF duplicates the output if the same inputs are used, or if true randomness repeats (with negligible probability), thus we have: $\mathsf{Adv}_1 \leq \mathsf{n}_{\mathsf{x-max}} \cdot \binom{\mathsf{n}_{\mathsf{x-max}} \cdot \mathsf{n}_{\mathsf{y-max}}}{2} \cdot 2^{-|\mathsf{K}_{\mathsf{AF}}|} + \mathsf{Adv}_2$.

At this point, we claim that $\mathsf{Adv}_2 \leq 2^{-|\mathsf{K}_{\mathsf{AKMA}}|} \cdot 2^{-|\mathsf{K}_{\mathsf{AF}}|}$. This means that the adversary has now two possiblities to guess the target key: guessing from random directly the target key or passing to the random oracle the value $\mathsf{K}_{\mathsf{AKMA}}$.

Notice that the case we cover is the same as a local outsider with single-stage key-control. In this case, the K_{AF} is specifically made *stale* by the adversary (where in the previous case, it was only assumed in the worst-case).

Local outsider with cross-stage key-control. We now explicitly give an attack for this adversary, with no heal: since $\mathcal{A}$ is local, it can reveal the K_{AF} and its *breadth* trait allows it to make the $\mathsf{K}_{\mathsf{AKMA}}$ a key stale (i.e., stopping its evolution). This revealed key equates to new K_{AF} being derived from the same $\mathsf{K}_{\mathsf{AKMA}}$.

Notice that this case also covers a medium outsider adversary with cross-stage key-control (the keys revealed include the previous one).

Medium outsider without key-control. We show that, in this case, the protocol is $(\infty, 1)$.

First, the adversary can reveal (since it is medium) the $\mathsf{K}_{\mathsf{AKMA}}$ thus all derived keys are accessible. This attack shows that a single full chain is compromised. We now show that no other (future) stage are compromised.

$\mathbb{G}_2$: We ensure that the value $\mathsf{K}_{\mathsf{AKMA}}$ for chain y^*+1 is unique. If there are two equal values in a session, or in two different (honest) sessions, then the challenger aborts and returns a random bit.

We consider the random oracle model, thus we have: $\mathsf{Adv}_1 \leq \mathsf{n}_{\mathsf{x-max}} \cdot \binom{\mathsf{n}_{\mathsf{x-max}} \cdot \mathsf{n}_{\mathsf{y-max}}}{2} \cdot 2^{-|\mathsf{K}_{\mathsf{AKMA}}|} + \mathsf{Adv}_2$.

At this point, we claim that $\mathsf{Adv}_2 \leq 2^{-|\mathsf{K}_{\mathsf{AKMA}}|} \cdot 2^{-|\mathsf{ltsk}(P,Q)|}$. This means that the adversary has now two possiblities to guess the $\mathsf{K}_{\mathsf{AKMA}}$: guessing from random directly or passing to the random oracle the value $\mathsf{ltsk}(P, Q)$ which is a global key.

Insider and Global adversaries. Those cases give the worst healing (*i.e.*, no healing at all) because the keys revealed (for global) are too high in the key hierarchy, or the information from the super-user (*i.e.*, insider) are leaked.

B.2 PCS_AC PCS for AKMA$^+$

Theorem 2. *Consider the* AKMA$^+$ *protocol modelled as a* KE-DECK *scheme. The following results hold in the random oracle model (by replacing the KDFs with random oracles), and assuming the AKE security of the channels established between honest users and an honest* $\hat{S}$, AKMA$^+$ *is:*

- $(1, 0)$*-PCS secure against local outsider without key-control;*
- $(1, 0)$*-PCS secure against local outsider with single-stage key-control;*
- $(\infty, \infty, {K_{AKMA}}^{\circlearrowleft})$*-PCS secure against local outsider with cross-stage key-control;*
- $(1, 0)$*-PCS secure against medium outsider without key-control;*
- $(1, 0)$*-PCS secure against medium outsider with single-stage key-control;*
- $(\infty, \infty, {K_{AKMA}}^{\circlearrowleft})$*-PCS secure against medium outsider with cross-stage key-control;*
- (∞, ∞)*-PCS secure against insider or global.*

The proofs follow similarly. These improvements for AKMA$^+$ come from the fact that the control for single-stage key K_{AF} becomes useless: when a new $\mathsf{K}_{\mathsf{AKMA}}$ is generated, a new K_{AF} is also generated and independently for the time-to-live of K_{AF}. However, there are two cases that are not improved: (1) for adversaries with cross-stage key-control, and (2) for insider/global adversaries. In both cases, the key-control is higher in the key hierarchy than the target key, so a key-stale attack has a consequence of no healing.

C PROVERIF & TAMARIN CODE SNIPPETS

C.0.1 `Proverif` *Code Snippets.* We recall that in ProVerif, for ease and with loss of generality, we take a vertical hierarchy of three keys: K_1 feeding into K_2, which feeds into K_3. The "key-refresh application trigger" at each key-evolution is controlled by the boolean result of the *aEvolve* oracle in our PCS cryptographic model. To simplify notation, we call this trigger α – for K_3 and we call it β – for K_2.

We give an example of such a query in Figure 7 which shows that if the K_3 key is known to the attacker and is still the same at step 3 then it was the same in the previous step 2.

```
(* This shows that it is possible for the
    attacker to know k3 at several stages.
If the attacker knows k3 and k3 has not
    changed at iteration 3 then it was also
    the same at step 2. *)

query k3:key, st1,st2:stamp, k3list:keylist,
     alpha1, alpha2:bool; attacker(k3) &&
event (K3_Refresh(3, k3, st2, alpha2)) ==>
    event (K3_Refresh(2, k3, st1, alpha1)).
```

Figure 7: `ProVerif` query w.r.t. knowing target-key K_3

The following `Proverif` code snippets illustrate the knowledge of the attacker and how the control of either or both of α and β allows the attacker to control the "freshness" of the target key K_3.

C.0.2 `Tamarin` *Code Snippets.* The following code snippets show `Tamarin` lemmas expressing similar attacker behaviour and knowledge with respect to α, β and the "freshness" of the target key K_3.

D IMPLEMENTATION ON OPEN5GCORE – DETAILS

The `Open5GCore` `YAML` files provide an image of the AF, the AAnF and/or any part of the core as web servers, that accept HTTP and JSON requests, which, in turn, emulate the 5G procedures (such as the 3GPP specifications). For instance, a partial JSON request for AKMA looks like

```
(* This shows that if the attacker controls
    alpha then they can ensure that from the
     point of leakage onwards they know the
    secret k3 by ensuring that alpha is
    false in the next step. *)

query k3:key, st1,st2:stamp, k3list:keylist;
     event (effective_leakage_stamped(af, k3
    , 1, st1, k3list)) && event (K3_Refresh
    (2, k3, st2, false)) .
```

Figure 8: `Proverif` query showing attacker controlling α

```
(* This shows that if the attacker controls
    beta then they can ensure that, provided
     they stop k2 from changing, they will
    know k3 (once leaked) even if k3 "
    refreshes" as k2 remains the same *)
query k2,k3:key, st0,st1,st2:stamp; event (
    K2_Refresh(0, k2, st1, false)) (* if k2
    is not updated in step 0 *) && event (
    K3_Refresh(0, k3, st1, true)) (* and we
    force a refresh of k3 *) && event (
    K2_Refresh(1, k2, st2, false))  (* and
    k2 is not updated in step 1 *) (* then
    k3 stays the same even when forced to
    refresh *) && event (K3_Refresh(1, k3,
    st2, true)) .
```

Figure 9: `Proverif` query showing attacker controlling β

```
/* This shows that if the attacker controls
    alpha then they can ensure that from the
     point of leakage onwards they know the
    secret k3 */

lemma compromiseK3_alpha:
"All #t01  #t02 #t03 k3 k3_new .
Leak('1', k3, 'True') @ t01 & UpdateKAF(
    k3_new, '1'+'1', 'alpha_0') @ t02
& Secret('1'+'1',k3_new) @ t03
==> k3_new=k3
"
```

Figure 10: `Tamarin` lemma – attacker controlling α

```
jsonRequest = {..., "afId" : "23", "aKId" :
    "3234"}
```

```
/* This shows that if the attacker controls
    beta then they can ensure that,
provided they stop K2 from changing before
    the point of leakage and afterwards,
They will know K3 no matter if K3 updates or
     not as K2 remains the same */
lemma compromiseKAF_beta:
"All #t01  #t02 #t03 #t04 k2 k2_new k3
    k3_new .
UpdateK2(k2, '1', 'beta_0') @ t01 & Leak('1'
    , k3, 'True') @ t02
& UpdateK2(k2_new, '1'+'1', 'beta_0') @ t03
    & Secret('1'+'1',k3_new) @ t04
==>
k3_new=k3
"
```

Figure 11: `Tamarin` lemma – attacker controlling β

and, to trigger the AKMA procedure for the above, we make a process call with an HTTP POST containing this `jsonRequest` inside, e.g.,:

```
subprocess.Popen(['curl', '-i', '--http2-
    prior-knowledge', '-X', 'POST', '-d',
    dumps(jsonRequest), "http://192.zzz.yy.
    xx:pppp/3gpp-akma/v1/retrieve", '-H',  "
    accept: application/json"], stdout=
    subprocess.DEVNULL, stderr=subprocess.
    STDOUT).
```

E TLS1.3 WITH SESSION TICKETS IN `PCS_AC`

E.1 TLS 1.3 with Session-Resumption & Key Updates

The network protocol TLS (Transport Layer Security) is widely deployed and aims at providing high level security to Internet traffic. Throughout the years, various versions of TLS were designed and standardised. The most recent of these is TLS 1.3, whose design revolutionised TLS in many ways, including its key-derivation, its session-resumption mechanisms, and the encryption of part of the handshake.

In this section, our goal will be to analyse how our framework applies to TLS 1.3. Although a complete study of this protocol would, in and of itself, require a full paper, our goal is to provide an intuition that shows the versatility of our framework in capturing potential vulnerabilities at various levels of the key-derivation hierarchy.

To begin with, we briefly recall the key-schedule and some session-resumption/key-update mechanisms of TLS 1.3. This is by no means a full description of the protocol (indeed, we do not – for instance – focus at all on the messages exchanged throughout the protocol). For further details, we direct the interested reader to RFC 8446, which standardises TLS 1.3.

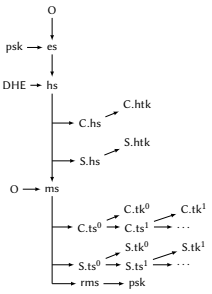

Figure 12: Key Schedule & Evolution in TLS 1.3 (simplified)

The key-schedule of TLS 1.3. TLS sessions are run between a client C and a server S. If the client and server do not share any key material (for instance material exchanged out of band or through a previous TLS session), then the client and server engage in a *full* TLS handshake. The latter begins with a parameter-, nonce-, and handshake-element exchange, during which the two parties:
– Agree on cipher-suites and extensions they want to use;
– Choose two nonces which are meant to provide the freshness of computed keys;
– Exchange key material that enables them both to compute an ephemeral Diffie-Hellman secret DHE.

These messages are exchanged in clear (unencrypted), and they are unauthenticated at this stage – and allow C and S to use DHE in order to compute a *handshake secret* hs, which is expanded to a client handshake secret C.hs and a server handshake secret S.hs. The client and server then derive handshake keys C.htk, S.htk, the former derived from C.hs and the latter, derived from S.hs by means of a KDF:

$$\mathsf{C.htk} = \mathsf{HKDF.aXP}(\mathsf{C.hs}, \texttt{label.chtk}, \tau)$$
$$\mathsf{S.htk} = \mathsf{HKDF.aXP}(\mathsf{S.hs}, \texttt{label.shtk}, \tau),$$

where τ is the current session transcript in each case.

The remainder of the handshake between the client and server is encrypted with these two keys. It serves to:
– Authenticate the server and the (unencrypted) beginning of the handshake, through a signature;
– Confirm the key material through the exchange of encrypted finished messages;
– Compute a master secret ms, which will then be expanded to several secrets, including: the client traffic secret C.ts, the server traffic secret S.ts, and the resumption master secret rms. An abstraction of this key-hierarchy is in the Figure 12. Note that in a full handshake the top psk is set to 0. The two traffic secrets C.ts and S.ts are expanded to the client traffic key C.tk and the server traffic key S.tk, respectively:

$$\mathsf{C.tk} = \mathsf{HKDF.aXP}(\mathsf{C.ts}, \texttt{label.ctk}, \tau)$$
$$\mathsf{S.tk} = \mathsf{HKDF.aXP}(\mathsf{S.ts}, \texttt{label.stk}, \tau).$$

Updating the traffic key. The traffic secrets in TLS 1.3 are updateable (in long sessions) by means of a KDF expansion:

$$\mathsf{C.ts}^{N+1} = \mathsf{HKDF.aXP}(\mathsf{C.ts}^{N}, \texttt{traffic update}).$$

Each party is responsible for its own traffic-secret update (and for later deleting the stale secret).

Session resumption. If the client and server share some key material (through out-of-band exchange or having run a prior, full TLS session), they can abbreviate their next handshake and use the shared key-material in the key-schedule. In that abbreviated handshake psk, which is generated from the resumption master secret rms of a prior, full TLS session, is used for two purposes:
– It provides implicit mutual authentication as (hopefully) only the two endpoints of the original handshake (which had authenticated to each other) should know psk;
– It enables the derivation of an early secret es, a new handshake secret based on psk and fresh session-specific nonces, and a new master secret based on hs and a potentially-new DHE value.

There are several ways of resuming a TLS session, which influence the security of the traffic keys.

The simplest resumption is 0-RTT, in which the psk and a client-side nonce provide the freshness for the client's early traffic secret.

In PSK-only session-resumption, the derivation continues, with the handshake secret (hs) derived from the early secret and the new session's transcript (including the two fresh nonces). Subsequently, key-derivation follows as in a full handshake.

In PSK+DHE session-resumption, the handshake secret is derived from es and the new session's DHE value, as well as the new session's transcript.

Whichever resumption will eventually be chosen, it depends on the client and server both having access to rms and the value psk derived from it. There are two ways of allowing the endpoints access to this shared information:
– The client and server both store these values for each completed session;
– The client stores resumption data, but the server does not, instead sending it in encrypted form to the server in the form of a session ticket. The client sends the session ticket to the server in the abbreviated handshake, allowing S to decrypt it and retrieve the pre-shared key material.

TLS 1.3 and our framework. TLS 1.3 is a complex and many-layered protocol, and our framework allows us to focus on multiple aspects of it. If we consider the key-schedule depicted in Figure 12, then we can immediately classify the following keys:
– Single-stage keys: the client- and server-traffic secrets C.ts, S.ts, as well as the traffic keys derived from it are clearly single-stage keys, which are removed from memory as soon as they are updated (either through key-updates or by ending the current session). We can also classify the Diffie-Hellman value DHE, as well as the early secret es, the handshake secret hs, and the master secret ms, which only need to survive until they are expanded into intermediate, independent secrets.
– Cross-session keys: the server's long-term key (used for authentication) and potentially the long-term secret used to encrypt session tickets are both cross-session secrets.

This leaves a number of secrets and keys in a grey area.

The resumption values: psk and rms Technically, the pre-shared key originates from the resumption master secret and ticket nonce computed in one session, and is used in a subsequent and distinct

session. One can therefore wonder whether psk is then a *cross-session* key – which would not allow us to apply our framework (and key-controls) on it.

However, note that in fact, the values encapsulated by each ticket are the resumption secret and a session nonce, and the psk is only derived and used during the next session. In this case, rms becomes a cross-session key, whereas psk is a *single-stage* key (derived and used only at the beginning of the TLS key-schedule).

This is interesting because, as we discuss below, there are ways of abusing session resumption in order to prolong the attack window of a passive local outsider.

Ticket lifetime Another interesting aspect of resuming sessions via session tickets is the notion of ticket lifetime (which can run up to a week). Technically, ticket lifetime is a property of the session ticket and not of a specific key; it is a value that accompanies the rms and ticket-nonce of each ticket, and thus characterizes the lifetime, not of the rms, but of the psk that will be computed from it with the aid of the ticket nonce.

This is why in the analysis below the ticket lifetime will be assumed to be part of the key-controls for the stage-specific psk key.

E.2 Applying PCS_AC to TLS 1.3

There are multiple ways in which we can apply our framework to TLS 1.3, and we present some of these below.

The target keys, in each case, are the traffic keys used to encrypt messages: potentially the early traffic keys (derived from es) or traffic keys C.tk and S.tk. The goal of the attacker is to maximize its ability to compromise the target keys by using traditional means (corruption, key-revelation, active message-sending attacks), as well as key-control hijacking.

Case 1: key updates. During long TLS sessions, both endpoints can choose to update their keys as often as they deem necessary. However, note that this update is unilateral: a client cannot, say, force a server to update its traffic key, nor vice-versa. Say that the attacker is able to ensure that the client never updates its traffic secret, no matter how long a session is. Then, once it compromises the client's current traffic key, it is able to break security for all client-side traffic for the remainder of that (endless) session, even if the server does update its own traffic secret.

We can model this in our framework as a set of key controls such that, on the client side, each C.ts has τ set to *NI* and, by default, aEvolve is set to true, because in principle the client is always allowed to evolve its own traffic secrets in an independent, non-interactive way (same on the server side, with S.ts). In this case, the hijacker triggers the aEvolve control, setting that flag to false.

This threat can be avoided by, for instance, timing out sessions that have not been updated with sufficient frequency, in the case of both partners. This would force the existence of a new session (and new keys).

Case 2: long psk **lifetimes.** In RFC 8446, ticket lifetimes are restricted to a week. Thus, an attacker that has learned psk can benefit from hijacking the following controls:
– Ticket lifetime: the attacker will maximize the lifetime of all tickets, making each psk valid for an entire week; – PSK resumption: By choosing PSK resumption, an attacker ensures that the key derivation remains symmetric, and thus predictable for the adversary;
Resumption: The adversary can force servers to always issue (multiple) resumption tickets at the end of each session, thus continuing to be able to derive keys from the same, single-stage psk. On the client side, the adversary can prompt clients to resume sessions whenever possible.

We capture this threat in our framework by working with psk as a single-stage key, and then treating ticket-lifetime, resumption-type, and the prompt for resumption as key-controls that can be hijacked. A particularly elegant trait of our framework is that we are able to separate the following key-control actions: (1) forcing servers to always issue tickets for resumption; (2) forcing clients to always use resumption; (3) choosing the type of resumption. The type of resumption chosen by the client, as well as whether or not it chooses to resume tickets it has received, can be modelled in our framework as a τ control (since it prompts whether or not the client and prompt a particular evolution), whereas the issuing of tickets can be chalked up as an aEvolve control (since it influences the existence or not of an evolution). An adversary that is able to hijack all the three controls is thus a *physical-layer* attacker, whereas one that only has partial control is an *algorithmic-layer* attacker.

Case 3: use of 0-RTT. An attacker could also choose to target the client's early traffic key. This is the weakest key in the entire key-schedule. In the TLS RFC, there are several ways in which server settings can affect how effectively the attacker can abuse this:
– The RFC specifies that servers should not allow 0-RTT encryption whenever they suspect psk to be unfresh (*i.e.*, if a ticket is resumed much later than its issue time). An attacker could benefit from allowing 0-RTT client traffic even for stale tickets.
– The client can continue encrypting messages with early traffic keys until the server continues the handshake (and move from 0- to 1-RTT). An attacker can delay this so, by default, servers choose not to continue the handshake, thus keeping the communication 1-sided.

Note that, in the former threat, the attacker would only need to hijack the aEvolve for a single stage, in order to compromise the early-traffic keys. The latter threat is a single-stage τ-type control on the server side, allowing the latter to prompt, or not, the evolution to 1-RTT keys.

E.3 TLS 1.3 (in-)security

Above we gave a description of part of key-hierarchy in TLS 1.3 and its keys' modelling in our framework, as well as discussed the application controls (*e.g.*, time to live) that come with them.

Onto these, we can impose different attackers as classified in our framework and therefore make and prove different statements about the PCS_AC-security of TLS 1.3. As we said before, an exhaustive and fully-explained analysis of this needs a paper in its own right. So, we give an example of one such PCS_AC-insecurity statement for TLS 1.3 for a relatively weak attacker in the PCS_AC framework.

Proposition 1. *The TLS 1.3 session resumption PSK mode with ticket is* (∞, ∞)*-PCS secure against an* PCS_AC *attacker that is: passive, extreme, physical, local outsider, with stage-specific key-control.*

This proposition essentially reflects the intuition behind case 2. The proposition claims that the described adversary (passive, extreme, physical, local outsider with stage-specific key-control) is able to destroy the protocol's healing entirely (∞, ∞). The adversary in question targets the traffic keys of a session between endpoints C and S which:

– Needs to recover one psk value shared by C and S at some stage s (since psk is a stage-specific key, this attacker is *local*);

– Needs to hijack both key-controls for psk (τ and `aEvolve`) in order to ensure that the server always issues exactly one ticket at the end of each session with C, that the client always resumes, and that it also only runs PSK-based resumption. This makes the adversary *physical*. All stages of evolution of psk need to be toggled, hence the attacker is *extreme*. Finally, the key whose controls are hijacked is stage-specific, thus the attacker has *stage-specific controls*;

– Does not need to inject any messages into the protocol (is *passive*), and is a regular PitM attacker, able to intercept all communications between C and S (is an *outsider*).

This attacker can learn all future client- and server-traffic keys for all future sessions run between C and S, since the endpoint are configured to keep resuming sessions starting from one psk known to the attacker. (In PSK-only resumption, knowledge of the psk and of the nonces suffices to compute both the handshake and the traffic keys.)

Even a weaker attacker (minimal rather than extreme), which performs the attack described above, but hijacks the controls only for one stage of psk is able to damage the security of TLS 1.3, providing $(\infty, 2)$-`PCS_AC` security.

F KEY HIERARCHY IN 5G & IN AKMA

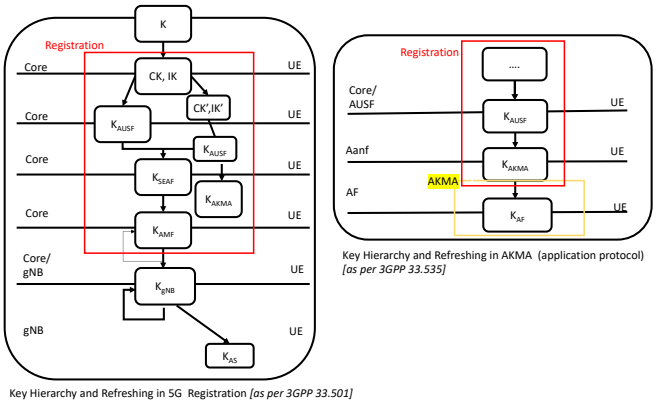

Figure 13: Key-Hierarchy in 5G (Left) & AKMA (Right)