# Cascade Ciphers Revisited: Indifferentiability Analysis

Chun Guo[1,2], Dongdai Lin[1⋆], and Meicheng Liu[1,3]

[1] State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences, Beijing 100093, China

[2] University of Chinese Academy of Sciences, China

[3] Nanyang Technological University, Central Area, Singapore

{guochun,ddlin,liumeicheng}@iie.ac.cn

**Abstract.** Shannon defined an ideal $(\kappa, n)$-blockcipher as a secrecy system consisting of $2^\kappa$ independent $n$-bit random permutations.

This work revisits the following question: in the ideal cipher model, can a cascade of several ideal $(\kappa, n)$-blockciphers realize $2^{2\kappa}$ independent $n$-bit random permutations, i.e. an ideal $(2\kappa, n)$-blockcipher? The motivation goes back to Shannon's theory on product secrecy systems, and similar question was considered by Even and Goldreich (CRYPTO '83) in different settings. Towards giving an answer, this work analyzes cascading independent ideal $(\kappa, n)$-blockciphers with two alternated independent keys in the indifferentiability framework of Maurer et al. (TCC 2004), and proves that for such alternating-key cascade, four stages is necessary and sufficient to achieve indifferentiability from an ideal $(2\kappa, n)$-blockcipher. This shows cascade capable of achieving key-length extension in the settings where keys are *not necessarily secret*.

**Keywords:** blockcipher, cascade, ideal cipher, indifferentiability.

⋆ Corresponding author.

# Table of Contents



## 1 Introduction

A blockcipher with key length $\kappa$ and block size $n$ is a set of efficiently computable (and invertible) permutations $E_k$ on the set $\{0,1\}^n$ indexed by a $\kappa$-bit key $k$, and is often referred to as a $(\kappa, n)$-blockcipher. When the permutations are $2^\kappa$ *independent and random* ones, it constitutes an ideal $(\kappa, n)$-blockcipher $\mathbf{IC}[\kappa, n]$.

A cascade cipher is defined as a concatenation of blockcipher systems, henceforth referred to as its *stages*. The cascade of $l$ ciphers ($l$-cascade) is

$$E_l(k_l, E_{l-1}(k_{l-1}, \dots (E_1(k_1, x)) \dots)).$$

In the secret-key setting, cascade is a natural and important way to amplify security, and has been deeply understood for this purpose, cf. two recent works [DLMS14,GLS$^+$15]. (We remark that this line of works, which will be recalled later, usually considers cascading the *same* blockcipher, which is different to the setting of *independent* blockciphers in this work.) However, it is far less understood when used for other purposes and in settings different from secret-key setting. In this sense, the power of cascade has not been fully developed.

This work considers cascade ciphers from such a different perspective. Among the motivations is to investigate the structure of the set of transformations performed by cascade ciphers, which dates back to Shannon: the cascade of $l$ independent ideal $(\kappa, n)$-ciphers is a special case of product secrecy system, and is a set of $2^{l\kappa}$ $n$-bit permutations [Sha49]; but Shannon did not provide additional knowledge of these permutations. More clearly, we want to determine what set of permutations the cascade can perform? Over the past decades, this question received very few attention – we are only aware of the research of Even and Goldreich in 1980s, which proved that $l$-cascade does *not* realize $2^{l\kappa}$ *independent* permutations (although $l$-cascade does realize $2^{l\kappa}$ *different* permutations w.h.p.): the behavior of them could be determined using only $l \cdot 2^\kappa$ exhaustive experiments [EG83]. Since their (negative) result, a problem is whether the cascade of ideal $(\kappa, n)$-ciphers could "behave like" $2^{l\kappa}$ independent random permutations for $l \geq 2$ (say, an ideal $(l\kappa, n)$-cipher) and under which conditions could it be?

The experiments considered in [EG83] do not allow the adversary access the underlying ciphers. However, we would like to consider the above problem in the *ideal cipher model* (ICM), as recent studies on cascade ciphers are typically conducted in ICM – say, the adversary is free to query the underlying ideal ciphers (such proofs are accepted as evidence of security against generic attacks despite the uninstantiability of ICM [CGH04,Bla06]). Then our problem can also be seen as the natural extension of the key-length extension problem from the traditional secret-key setting to the setting of *public* key, and the goal is formalized by the notion of *indifferentiability* of Maurer et al. [MRH04].

Note that cascade ciphers are idealized blockciphers; here we "pause" and give a brief remark on indifferentiably secure idealized ciphers. Traditionally blockciphers are used to provide privacy in the secret-key setting, and being pseudorandom already suffices. However blockciphers also find extensive use in constructing other cryptographic primitives such as hash functions, in which cases mere pseudorandomness would be insufficient. For example, the Davies-Meyer mode of the pseudorandom 1-round Even-Mansour cipher [EM97] is not collision resistant. These motivated the line of works considering whether idealized blockciphers can be *indifferentiable from ideal ciphers* [ABD+13,LS13]. The indifferentiability analysis of cascade follows this line in some sense.

Motivated by the above discussion, we have the following more concrete reformulated question:

– *In the ICM, under which conditions and for how large value $l$ is the cascade of $l$ ideal ciphers* $\mathbf{IC}[\kappa, n]$ *indifferentiable from an ideal $(\kappa', n)$-blockcipher such that $\kappa' > \kappa$, e.g.* $\mathbf{IC}[2\kappa, n]$*?*

Sub-Key Reuse, and Independent Underlying Ciphers. It has been observed by Lampe and Seurin that the cascade of two $\mathbf{IC}[\kappa, n]$ (with two independent keys) is *not* indifferentiable from $\mathbf{IC}[2\kappa, n]$ [LS13]. Indeed, we show that 2-cascade does not even resist related-key distinguishing attack [Bih94]. Obviously, using $l$ independent keys in $l$-cascade is like using a 2-cascade; the case is a bit similar to the context of Even-Mansour ciphers [ABD+13,LS13]. Therefore, the keys of the underlying ideal ciphers must be somehow *reused*.

To ensure key-reusing while achieving key-length extension, a very natural and promising approach is interleaving two independent (sub-)keys in the underlying ciphers. The next concern is whether the underlying ciphers should be independent or not. Our motivation inherently requires cascading independent ciphers, as we aim to study the product of different secrecy systems. However, as what is usually used in practice is cascading the same blockcipher (a.k.a. multiple encryption), it would be nice if we could achieve a positive result on multiple encryptions. But we are not able to do so, because based on slide attack [BW99] we find a chosen-key distinguisher on $l$-encryption with two alternated keys (as discussed above) regardless of how large $l$ is. Indeed, it is never surprising that multiple encryptions can not achieve indifferentiability because *they are never meant to do so.* Albeit less relevant to practical settings, this distinguisher suffices to force us to revert to cascading independent blockciphers.

With the discussion above, we consider *interleaving two independent keys* in *$l$ independent underlying ciphers.* Denote this construction by $\mathsf{CC}_l$ and let $K = (k_1, k_2)$, then

$$\mathsf{CC}_l(K, x) = E_l(k_t, \ldots (E_3(k_1, E_2(k_2, E_1(k_1, x)))) \ldots)$$

where $t = 1$ when $l$ is odd, and $t = 2$ when $l$ is even. An illustration could be found in Fig. 4. The rest part of this work thereby mainly focus on working out the value of $l$ such that $\mathsf{CC}_l$ is indifferentiable from $\mathbf{IC}[2\kappa, n]$.

Results. As the main result, we prove that the 4-cascade $\mathsf{CC}_4$ is indifferentiable from $\mathbf{IC}[2\kappa, n]$. The security bound is $O(q^6/2^n)$, and the query and time complexity of the simulator are $O(q^4)$ and $O(q^7)$ respectively. Besides, 2-cascade $\mathsf{CC}_2$ and 3-cascade $\mathsf{CC}_3$ are also considered – and are attacked (thus *not* indifferentiable). Therefore, for cascades with alternating key schedule the number of stages *4* is tight.[4]

As discussed, this research provides a deeper understanding into the algebraic structure of the sets of permutations realized by cascade ciphers. It also sheds light on the structural and probabilistic aspects of cascade in the ICM. Perhaps more importantly, the main result shows that cascade is not only a solution to the key-length extension problem in the traditional secret-key setting, but also a solution in the recently popularized setting of public keys and indifferentiability, thus making a step into further developing the power of cascade – which is a "basic information theoretic question" [ABDCV98]. Note that Coron et al. already provided a solution to key-length extension problem in indifferentiability setting: they proved that the construction $E(H(K), x)$ (*hash-then-encrypt* mode) from a random oracle $H$ and an ideal cipher $E$ is indifferentiable [CDMS10] (and the security bound $O(q^2/2^\kappa)$ is probably better than ours). However, the success of the construction $E(H(K), x)$ says nothing about the power of cascade. Furthermore, cascade has the advantage of being capable in both secret-key and indifferentiability setting, while the key space of the construction $E(H(K), x)$ is essentially the same as $E$ in secret-key setting.

Practically speaking, although cascading independent blockciphers seems less popular than cascading the same one, the idea is relevant to practice yet: first, it is natural and it also dates back to Shannon; second, it is possible to be used in some cases, e.g. those discussed in [Sha49] and [AB81], or in the form of multiple

[4] The tightness does not necessarily holds for those with other key schedules. However, we have not found a schedule to cinch the positive result on 3-cascade, cf. Appendix A.

ECB mode using different ciphers [Bih98]. In this sense, we provide a possible approach to building a secure blockcipher with larger key space, which may be useful as there lack a candidate for such a blockcipher.[5]

Technical Issues. Our proof is built on Andreeva et al.'s analysis of $t$-round Even-Mansour with Random oracle key schedule (EMR$_t$) [ABD$^+$13]. We adapt their *tripwire paradigm* for building our indifferentiability simulator. We also use their technique *explicitly bookkeeping* to simplify the language.[6]

To correctly maintain chain completions in 4 stages, our simulator incorporates two modifications compared to Andreeva et al.'s simulator for 5-round EMR. First, upon the distinguisher issuing a new query, our simulator would "patiently" find the most suitable "starting point" for the chain completion, while Andreeva et al.'s simply starts the chain completion from the "vertex" specified by the new query. Second, our simulator explicitly exploits the properties of the structures formed by the *entire* history of queries and answers of the "targeted" $\mathbf{IC}[2\kappa, n]$ (denoted $\mathcal{H}_{\mathbf{IC}[2\kappa,n]}$). Note that in indifferentiability setting the simulator should *not* be able to access $\mathcal{H}_{\mathbf{IC}[2\kappa,n]}$. To settle this, our simulator sometimes obtains information about $\mathcal{H}_{\mathbf{IC}[2\kappa,n]}$ via the "check" procedure of Coron et al. [CHK$^+$14]. For more details cf. page 11.

Perhaps the most troubling obstacle is to find a simulator-termination argument, i.e. to prove that the simulator has a polynomial complexity. Indeed, we take the "tripwire configuration" designed by Andreeva et al. for EMR$_4$, but they did not succeed in this task for EMR$_4$ (and turned to EMR$_5$). To solve this problem, we combine the core idea of Coron et al.'s termination argument [CHK$^+$14] and our fine-grained observations on our simulator and 4-cascade. For more details cf. page 16.

Other Related Work. As mentioned, most of the previous works on cascade focus on security amplification in the traditional secret-key setting. We briefly recall this line of works. As to security lower bound, cascade was proved at least as secure as the strongest underlying cipher when the enemy cannot exploit information about the plaintext statistics, and at least as secure as the *first* underlying cipher in general cases [EG83,MM93]. As to security amplification, double-encryption was proved (only) slightly better than single-encryption [ABDCV98], while triple-encryption was indeed better than single and double [BR06]. This line of researches was followed by a series of works [GM09,Lee13,Gaz13], which culminate with two recent ones of Dai et al. [DLMS14] and Gaži et al. [GLS$^+$15]: the former proved tight bounds for $l$-cascade for all $l \geq 3$, while the latter took the number of queries to the cascade as an explicit parameter in the expression of advantages, resulting in a refined security analysis.

The aforementioned works as well as this paper focus on "plain" cascade. There is another key-length extension approach named *xor-cascade*, which works by xoring whitening keys. The idea dates back to the well-known FX-construction [KR01], and was further generalized and analyzed in [GT12,Lee13,Gaz13,GLS$^+$15,HT16]. Additionally, the line of work on security amplification in the standard model was initiated by Luby and Rackoff [LR86] and followed by e.g. [Mye03,MPR07,MT09,MT10,Tes11].

Finally, indifferentiability of idealized blockciphers was initiated by Coron et al., with a proof for 14-round Feistel networks [CHK$^+$14], which was later improved to 10 [DS15a,DSKT16] and 8 rounds [DS15b]. It was also extended to Even-Mansour [ABD$^+$13,LS13,GL15a] and confusion-diffusion networks [DSSL16].

Related Problems. As mentioned, for the same "tripwire configuration", we succeed in finding a simulator-termination argument in our context of $\mathsf{CC}_4$, while Andreeva et al. did not do for $\mathsf{EMR}_4$ and turned to $\mathsf{EMR}_5$. It's thus natural to ask if our termination argument could be adapted for $\mathsf{EMR}_4$. While there seems no obstacle, we have found a proof for the indifferentiability of $\mathsf{EMR}_3$ (and thus completely closing the gap between positive and negative sides in [ABD$^+$13]). We therefore eschew the analysis of $\mathsf{EMR}_4$. Our proof for $\mathsf{EMR}_3$ will be made public in several weeks.

Organization. Sect. 2 serves necessary preliminaries. Sect. 3 gives the attacks. Sect. 4 presents the main theorem as well as the key points of the proof. As the full proof for $\mathsf{CC}_4$ is too long, the pseudocode is given in Appendix B and the full proof is in Appendix C and D (more clearly, C gives the proof for simulator termination and non-abortion, while D proves the indistinguishability of the systems).

Finally, in Appendix A, we describe attacks against 3-cascade with a quite large class of key schedules.

[5] Cf. [CDMS10, page 275]: *as of 2009 it is unclear if we have a candidate block-cipher with key-size larger than block-size that behaves like an ideal cipher.*

[6] One may get an illusion as follows: if we prove the indifferentiability of the product of an ideal cipher and a random permutation, then the proof of cascade ciphers could be obtained through a trivial domain separation argument. However, it's not hard to notice $\mathbf{P} \circ \mathbf{IC}_1[\kappa, n]$ is indifferentiable from $\mathbf{IC}[\kappa, n]$, but (as mentioned) $\mathbf{IC}_2[\kappa, n] \circ \mathbf{IC}_1[\kappa, n]$ is *not* indifferentiable from $\mathbf{IC}[2\kappa, n]$.

## 2 Preliminaries

Throughout the remaining, the $\kappa$-bit sub-keys are written in lower-case letters, i.e. $k$, $k_1$, and $k_2$, while the $2\kappa$-bit master key is interchangeably written in the capital letter $K$ or the concatenation $(k_1, k_2)$. For simplicity, the notation $\mathbf{C}$ refers to an ideal cipher $\mathbf{IC}[2\kappa, n]$, and the notation $\mathbf{E}$ refers to a tuple of ideal ciphers $\mathbf{IC}[\kappa, n]$, say, $\mathbf{E} = (\mathbf{E}_1, \ldots, \mathbf{E}_l)$, where the value of $l$ depends on the concrete context. We assume that the interfaces of $\mathbf{C}$ are $\mathbf{C}.\mathrm{C}(K, z) := \{0,1\}^{2\kappa} \times \{0,1\}^n \to \{0,1\}^n$, and $\mathbf{C}.\mathrm{C}^{-1}(K, z) := \{0,1\}^{2\kappa} \times \{0,1\}^n \to \{0,1\}^n$, and the interfaces of $\mathbf{E}$ are $\mathbf{E}.\mathrm{E1}(k, z) := \{0,1\}^{\kappa} \times \{0,1\}^n \to \{0,1\}^n$, $\mathbf{E}.\mathrm{E1}^{-1}(k, z)$, $\ldots$, $\mathbf{E}.\mathrm{E}l(k, z)$, and $\mathbf{E}.\mathrm{E}l^{-1}(k, z)$.

**The Cascade Cipher in Question.** Given $l$ independent ideal ciphers $\mathbf{E} = (\mathbf{E}_1, \ldots, \mathbf{E}_l)$ and following the above convention, the cascade cipher $\mathsf{CC}_l^{\mathbf{E}}$ considered in this work is formally written as follows (as depicted in Fig. 4):

$$\mathsf{CC}_l^{\mathbf{E}}((k_1, k_2), x) = \mathrm{E}l(k_t, \ldots (\mathrm{E3}(k_1, \mathrm{E2}(k_2, \mathrm{E1}(k_1, x)))) \ldots),$$
$$(\mathsf{CC}_l^{\mathbf{E}})^{-1}((k_1, k_2), y) = \mathrm{E1}^{-1}(k_1, \mathrm{E2}^{-1}(k_2, \mathrm{E3}^{-1}(k_1, \ldots (\mathrm{E}l^{-1}(k_t, y)) \ldots))),$$

where $t = 1$ when $l$ is odd, and $t = 2$ when $l$ is even.

**Indifferentiability.** Indifferentiability framework [MRH04] addresses idealized constructions in settings where no underlying element (including building blocks and parameters) is secret. For concreteness, consider $\mathsf{CC}_4^{\mathbf{E}}$: a distinguisher $D^{\mathsf{CC}_4^{\mathbf{E}}, \mathbf{E}}$ with oracle access to the cascade and the underlying ideal ciphers is trying to distinguish $\mathsf{CC}_4^{\mathbf{E}}$ from $\mathbf{C}$. Then, a formal definition due to [LS13] is as follows.

**Definition 1 (Indifferentiability).** *The idealized blockcipher $\mathsf{CC}_4^{\mathbf{E}}$ with oracle access to ideal primitives $\mathbf{E}$ is said to be statistically and strongly $(q, \sigma, t, \varepsilon)$-indifferentiable from an ideal cipher $\mathbf{C}$ if there exists a simulator $\mathbf{S}^{\mathbf{C}}$ s.t. $\mathbf{S}$ makes at most $\sigma$ queries to $\mathbf{C}$, runs in time at most $t$, and for any distinguisher $D$ which issues at most $q$ queries, it holds*

$$\left| Pr[D^{\mathsf{CC}_4^{\mathbf{E}}, \mathbf{E}} = 1] - Pr[D^{\mathbf{C}, \mathbf{S}^{\mathbf{C}}} = 1] \right| \leq \varepsilon$$

Such a result means that $\mathsf{CC}_4^{\mathbf{E}}$ “behaves as” $\mathbf{C}$, in the sense that $\mathsf{CC}_4^{\mathbf{E}}$ can safely replace $\mathbf{C}$ whenever a moderate blow-up of the adversary's time and memory requirements is acceptable (cf. [RSS11,DGHM13] for the limitations of indifferentiability). Indeed, indifferentiability has been a de-facto standard security notion beyond traditional ones such as collision resistance and pseudorandomness, and has found application in various idealized constructions including hash function [CDMP05] and random permutation [CHK+14].

## 3 Attacks on $\mathsf{CC}_2$, $\mathsf{CC}_3$, and Multiple Encryption with Two Alternated Keys

Following the convention stipulated in Preliminaries (note an exception: for multiple encryption, the interfaces of $\mathbf{E}$ are E and $\mathrm{E}^{-1}$), we present the attacks as follows. Note that the construction $\mathsf{CC}_1$ is non-sense, as itself is an ideal cipher without any structure that can be studied.

### 3.1 (Related-key) Distinguisher for 2-Cascade

For $\mathsf{CC}_2^{\mathbf{E}}$, we give a distinguisher $D$, which works as follows:

(1) $D$ randomly chooses key differences $\Delta_{k_1}, \Delta_{k_2} \in \{0,1\}^{\kappa} \setminus \{0\}$ and $u \in \{0,1\}^n$;
(2) for an fixed arbitrary key $(k_1, k_2)$, $D$ queries $w := \mathrm{C}((k_1, k_2), u)$ and $w' := \mathrm{C}((k_1, k_2 \oplus \Delta_{k_2}), u)$;
(3) if $\mathrm{C}^{-1}((k_1 \oplus \Delta_{k_1}, k_2), w) = \mathrm{C}^{-1}((k_1 \oplus \Delta_{k_1}, k_2 \oplus \Delta_{k_2}), w')$, $D$ outputs 1, otherwise outputs 0.

Denote by $v$ the (secret) intermediate value $\mathrm{E1}(k_1, u)$ (the value in gray in Fig. 1). Then clearly $\mathrm{C}^{-1}((k_1 \oplus \Delta_{k_1}, k_2), w) = \mathrm{E1}^{-1}(k_1 \oplus \Delta_{k_1}, v) = \mathrm{C}^{-1}((k_1 \oplus \Delta_{k_1}, k_2 \oplus \Delta_{k_2}), w')$, as depicted in Fig. 1, and $D$ always outputs 1 when interacting with $\mathsf{CC}_2^{\mathbf{E}}$. Whereas when interacting with $\mathbf{IC}[2\kappa, n]$ – or two independent random permutations, – the probability is $1/2^n$.

Clearly, this distinguisher does not need to query the underlying ciphers. Thus it shows that 2-cascade does not perform $2^{2\kappa}$ independent permutations even if the underlying ideal ciphers are secret and *the adversary is only allowed to ask a few queries*, thus enhancing the conclusion of [EG83] that *cascades of secret ideal ciphers with independent keys can be distinguished from longer-key ideal ciphers using exponential number of queries.*

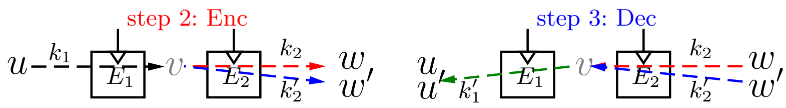


**Fig. 1.** Related-key attack on $\mathsf{CC}_2$. $k_1' = k_1 \oplus \Delta_{k_1}, k_2' = k_2 \oplus \Delta_{k_2}$. (Left) step 2: deviate at the 2nd stage; (Right) step 3: re-gather after (inverse of) the 2nd stage.

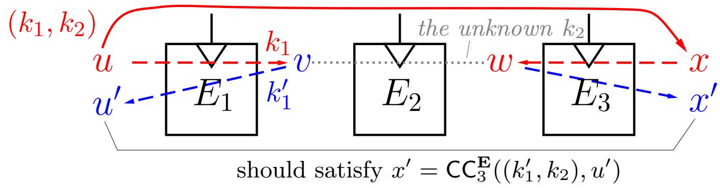


**Fig. 2.** Attack $\mathsf{CC}_3$.

### 3.2 Distinguisher for 3-Cascade(s)

The following equation is the basis of the distinguisher for $\mathsf{CC}_3$ (cf. Fig. 2)

$$\mathsf{CC}_3^{\mathbf{E}}((k_1', k_2), \mathrm{E1}^{-1}(k_1', \mathrm{E1}(k_1, u))) = \mathrm{E3}(k_1', \mathrm{E3}^{-1}(k_1, \mathsf{CC}_3^{\mathbf{E}}((k_1, k_2), u))).$$

By this, we consider a distinguisher $D$ which works as follows:

(1) $D$ randomly chooses $k_1 \in \{0,1\}^\kappa$ and $u \in \{0,1\}^n$, and queries $v := \mathrm{E1}(k_1, u)$;
(2) $D$ randomly chooses $k_2 \in \{0,1\}^\kappa$, and queries $x := \mathrm{C}((k_1, k_2), u)$;
(3) $D$ queries $w := \mathrm{E3}^{-1}(k_1, x)$;
(4) $D$ randomly chooses $k_1' \in \{0,1\}^\kappa \setminus \{k_1\}$, and queries $u' := \mathrm{E1}^{-1}(k_1', v)$ and $x' := \mathrm{E3}(k_1', w)$;
(5) $D$ outputs 1 if $x' = \mathrm{C}((k_1', k_2), u')$, and outputs 0 otherwise.

Clearly $Pr[D^{\mathsf{CC}_3^{\mathbf{E}}, \mathbf{E}} = 1] = 1$. On the other hand, note that the value $k_2$ randomly chosen by $D$ is unknown to $\overline{S}$. Denote by $\mathsf{HitK}$ the event that $\overline{S}^{\mathbf{C}}$ queried $\mathbf{C}$ on $((k_1^*, k_2), u^*)$ for some $k_1^*$ and $u^*$ during the execution $D^{\mathbf{C}, \overline{S}^{\mathbf{C}}}$. Then, $Pr[x' = \mathrm{C}((k_1', k_2), u')] \leq Pr[x' = \mathrm{C}((k_1', k_2), u') \mid \neg\mathsf{HitK}] + Pr[\mathsf{HitK}] \leq 1/2^n + q_S/2^\kappa$, so that $D$'s advantage is at least $1 - q_S/2^\kappa - 1/2^n$. Typically, both $\kappa$ and $n$ are chosen to be polynomial functions of the security parameter. $\mathsf{CC}_3^{\mathbf{E}}$ is thereby not indifferentiable.

One may think appropriate key schedules could reduce the independence between the two halves $k_1$ and $k_2$ and thus "salvaging" 3-cascade. However, we find distinguishers for a large class of such key schedules, cf. Appendix A. Thus achieving positive (yet non-trivial) indifferentiability result on 3-cascade seems very hard.

### 3.3 Slide Attack on Multiple Encryption with Alternated Keys

To make a distinction, we denote $l$-encryption (using a single ideal cipher and two alternated keys) by $\mathsf{ME}_l^{\mathbf{E}}$. Formally,

$$\begin{aligned}\mathsf{ME}_l^{\mathbf{E}}((k_1, k_2), x) =& \mathrm{E}(k_t, \ldots (\mathrm{E}(k_1, \mathrm{E}(k_2, \mathrm{E}(k_1, x)))) \ldots), \\ (\mathsf{ME}_l^{\mathbf{E}})^{-1}((k_1, k_2), y) =& \mathrm{E}^{-1}(k_1, \mathrm{E}^{-1}(k_2, \mathrm{E}^{-1}(k_1, \ldots (\mathrm{E}^{-1}(k_t, y)) \ldots))),\end{aligned}$$

where $t = 1$ when $l$ is odd, and $t = 2$ when $l$ is even.

Consider the case when $l$ is even first. Then we have (cf. Fig. 3)

$$\mathsf{ME}_l^{\mathbf{E}}((k_2, k_1), \mathrm{E}(k_1, x)) = \mathrm{E}(k_1, \mathsf{ME}_l^{\mathbf{E}}((k_1, k_2), x)).$$

Thus we have a (very simple) distinguisher $D$ as follows:

(1) $D$ randomly chooses $k_1 \in \{0,1\}^\kappa$, $k_2 \in \{0,1\}^\kappa$, and $x \in \{0,1\}^n$, and queries $y := \mathrm{C}((k_1, k_2), x)$;
(2) $D$ queries $x' := \mathrm{E}(k_1, x)$ and $y' := \mathrm{E}(k_1, y)$;
(3) $D$ outputs 1 if $y' = \mathrm{C}((k_2, k_1), x')$ and outputs 0 otherwise.

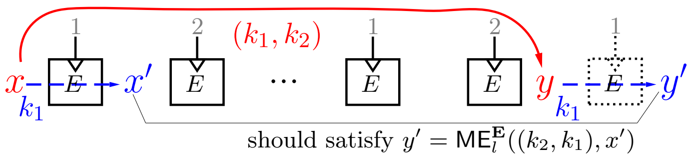


**Fig. 3.** Attack $l$-encryption with alternated keys, with $l$ being even.

Note that $k_2$ is unknown to the simulator $\overline{S}^{\mathbf{C}}$. The analysis is thus similar to the previous subsection and leads to the same advantage lower bound $1 - q_S/2^\kappa - 1/2^n$ (assuming $\overline{S}^{\mathbf{C}}$ makes $q_S$ queries).

Now consider odd $l$. In this case $2l$ is even; so if $D$ switches to query $y := \mathrm{C}((k_2, k_1), \mathrm{C}((k_1, k_2), x))$ in step (1) and check if $y' = \mathrm{C}((k_1, k_2), \mathrm{C}((k_2, k_1), x'))$ in step (3), then it will succeed. This might be a surprising example of composition leads to insecurity.

## 4 Indifferentiability of 4-Cascade

The main theorem of this work is presented as follows.

**Theorem 1.** *Assuming two independent $\kappa$-bit keys $(k_1, k_2)$ alternatively used in each stage, the cascade $\mathsf{CC}_4^{\mathbf{E}}$ of 4 independent ideal ciphers $\mathbf{E} = (\mathbf{E}_1, \mathbf{E}_2, \mathbf{E}_3, \mathbf{E}_4)$ is $(q, \sigma, t, \varepsilon)$-indifferentiable from an ideal cipher $\mathbf{IC}[2\kappa, n]$, where $\sigma = 8q^4 = O(q^4)$, $t = O(q^7)$, and $\varepsilon \leq \frac{210 \cdot q^6}{2^n} = O(\frac{q^6}{2^n})$.*

The central argument of indifferentiability is to design and present a simulator. Unfortunately, our simulator $\mathbf{S}$ is a bit complicated (cf. the 4-page code in Appendix B). It thus seems better to divide the presentation into two steps:

(1) Give an informal description for a related (and hopefully easier to understand) simulator $\mathbf{T}$, which is used in an imagined intermediate system $G_2$ in the proof. In this system, the ideal $(2\kappa, n)$-cipher of $\mathbf{T}$ offers an additional interface $\textsc{Check}(K, u, y)$ to $\mathbf{T}$, which allows $\mathbf{T}$ to know whether the query $(K, u, y)$ has appeared in the cipher's history. This is motivated by Coron et al. [CHK+14]. To make a distinction from the normal ideal cipher, we denote this modified ideal cipher by $\widetilde{C}$. Note that the interface $\textsc{Check}$ is hidden from $D$, thus $\widetilde{C}$ has no difference with a normal ideal cipher in the view of $D$.
(2) Describe how to obtain $\mathbf{S}$ from $\mathbf{T}$. *Indeed, for readers familiar with Coron et al.'s argument [CHK+14] this step is pretty obvious.*

In fact, when writing this proof, we indeed first design $\mathbf{T}$ as an intermediate step, and then obtain $\mathbf{S}$ by tweaking $\mathbf{T}$.

We first spend three subsections on presenting $\mathbf{T}$: 4.1 introduces basic ideas for simulation, in particular, the tripwire paradigm; 4.2 serves several instructive cases of interaction between the distinguisher $D$ and $(\widetilde{C}, \mathbf{T})$, and introduces the "layer-2" $\textsc{ProcessTree}$ procedures which "deal with" these cases; based on these underlying procedures, 4.3 describes how $\mathbf{T}$ handles queries and "passes on the control" to the "right" $\textsc{ProcessTree}$ procedure. Altogether, 4.2 and 4.3 form a bottom-up style overview of $\mathbf{T}$. After these, 4.4 show how to turn $\mathbf{T}$ to $\mathbf{S}$. Finally, 4.5 sketches the rest of the proof.

### 4.1 Simulator T: Basic Ideas

$\mathbf{T}$ offers eight interfaces to $D$ to emulate the ciphers, say, $\mathrm{E}i$ and $\mathrm{E}i^{-1}$ for $i = 1, 2, 3, 4$. To describe the interaction between $D$, $\mathbf{T}$, and $\widetilde{C}$, we use the notation $\mathrm{E}i(k, z) \to z'$ to mean that $D$ queries $\mathbf{T}.\mathrm{E}i$ on $(k, z)$ and $\mathbf{T}$ answers with $z'$, and $\mathrm{E}i^{-1}(k, z) \to z'$ vice versa. We similarly use $\mathrm{C}(K, z) \to z'$ to mean that either $D$ or $\mathbf{T}$ queries $\widetilde{C}.\mathrm{C}$ on $(K, z)$ and $\widetilde{C}$ returns $z'$, and $\mathrm{C}^{-1}(K, z) \to z'$ vice versa.

Informally speaking, $\mathbf{T}$ internally keeps already answered queries: after $D$ querying $\mathrm{E}i(k, z) \to z'$, it keeps a record $(i, k, z, z', \to)$, where $i$ indicates the index, $k, z, z'$ indicate the query and answer, and $\to$ indicates the query is a forward one; after $D$ querying $\mathrm{E}i^{-1}(k, z) \to z'$, it similarly keeps $(i, k, z', z, \leftarrow)$. Such tuples are called *$i$-queries*, and the terminology *E-query* refers to any $i$-query indifferently to the value of $i$. $\mathbf{T}$ may call $\mathrm{E}i/\mathrm{E}i^{-1}$ itself and internally create such records; it makes no distinction between such internally-created queries and those due to $D$'s actions. The queries that have been encountered and recorded are called *old*.

Upon a query from $D$, if it's old, then $\mathbf{T}$ simply replies with the recorded answers; otherwise, $\mathbf{T}$ may randomly sample an answer. For example, upon $D$ querying $\mathrm{E}i(k,z)$ such that no record of the form $(i,k,z,\cdot)$ pre-exists, $\mathbf{T}$ randomly samples a value $z'$ such that no record of the form $(i,k,\cdot,z')$ pre-exists,[7] creates a new record $(i,k,z,z',\rightarrow)$, and then replies with $z'$. To handily describe how the answer $z'$ is drawn, we follow [CS15] and make the randomness used by the simulator explicit through a tuple of 4 ideal ciphers $\mathbf{E} = (\mathbf{E}_1, \mathbf{E}_2, \mathbf{E}_3, \mathbf{E}_4)$. This means if $\mathbf{T}$ needs to assign a random answer to $\mathrm{E}i(k,z)$, $\mathbf{T}$ queries $\mathbf{E}$ to get $z' := \mathbf{E}.\mathrm{E}i(k,z)$.[8] We denote by $\mathbf{T}^{\widetilde{C},\mathbf{E}}$ the simulator accessing $\mathbf{E}$. However, for convenience, we keep saying "randomly sample" to refer to $\mathbf{T}$'s such actions.

$\mathbf{T}$ cannot answer all queries via randomly sampling, otherwise $\mathbf{T}$ will behave as four ideal ciphers *independent of* $\widetilde{C}$, whereas we need $(\widetilde{C}, \mathbf{T})$ to behave as $(\mathsf{CC}_4, \mathbf{E})$, i.e. answers from $\mathbf{T}$ should *depend on* $\widetilde{C}$. To this end, the basic idea is Coron et al.'s *simulation via chain completion* technique [CHK+14], which has been a routine for indifferentiability proof of idealized blockciphers.

**Chain Completion – Tripwire Paradigm.** We stipulate some terminology first. A triple $((k_1,k_2),u,y)$ such that $\mathrm{C}((k_1,k_2),u) \rightarrow y$ or $\mathrm{C}^{-1}((k_1,k_2),y) \rightarrow u$ has appeared is called a *C-query* (such query may be made by $D$ or $\mathbf{T}$, but this does not matter). A pair of E-queries $(i,k,z,z')$, $(i+1,k',z',z'')$ sharing the same intermediate value $z'$ is called *adjacent*. Two queries $((k_1,k_2),u,y)$ and $(1,k_1,u,v)$ (or $(4,k_2,x,y)$) are also *adjacent*. Then, a sequence of five adjacent queries

$$((k_1,k_2),u,y), (1,k_1,u,v), (2,k_2,v,w), (3,k_1,w,x), (4,k_2,x,y)$$

in the history of the interaction is called a $(k_1,k_2)$-*completed chain*. Also, such a chain identifies a cycle of values $u - v - w - x - y - (u)$.

Note that when interacting with $(\mathsf{CC}_4, \mathbf{E})$, the answers given by $\mathsf{CC}_4$ and $\mathbf{E}$ always form such completed chains. E.g. if $D$ asks $u \xrightarrow{\mathbf{E}.\mathrm{E1}_{k_1}} v \xrightarrow{\mathbf{E}.\mathrm{E2}_{k_2}} w \xrightarrow{\mathbf{E}.\mathrm{E3}_{k_1}} x \xrightarrow{\mathbf{E}.\mathrm{E4}_{k_2}} y$, then it must hold $\mathsf{CC}_4.\mathrm{C}((k_1,k_2),u) = y$. This forms the intuition of chain-completion technique: to generate similar interactions, $\mathbf{T}$ clearly needs to make its simulated answers form similar completed chains with $\widetilde{C}$'s answers. For this, note that for two adjacent E-queries it holds: (i) they (should) belong to the same completed chain; (ii) they can uniquely specify the chain. Therefore, $\mathbf{T}$ takes such adjacent pairs as "partial" chains, *detects* them, and *pre-emptively completes* them to completed chains. For example, after $D$ querying $\mathrm{E1}(k_1,u) \rightarrow v$ and $\mathrm{E2}(k_2,v) \rightarrow w$, $\mathbf{T}$ detects two adjacent $(1,k_1,u,v)$ and $(2,k_2,v,w)$. A possible strategy for $\mathbf{T}$ is to first internally call $\mathrm{E3}(k_1,w) \rightarrow x$ (with $x$ being a newly sampled random value) and create $(3,k_1,w,x,\rightarrow)$, then query $\widetilde{C}.\mathrm{C}^{-1}((k_1,k_2),u) \rightarrow y$. The second step ensures the existence of the C-query $((k_1,k_2),u,y)$. Now the only missing query of the chain $y - u - v - w - x$ is a 4-query $(4,k_2,x,y)$; $\mathbf{T}$ therefore create such a record $(4,k_2,x,y,\perp)$ to *adapt* and "close the cycle". The fifth coordinate $\perp$ of this record indicates that it's a query created for adaptation, henceforth referred to as *adapted queries*.

The queries newly created during chain-completion (including the adapted ones) may form new adjacent pairs, which may need to be completed as well and may lead to new queries and more adjacent pairs. $\mathbf{T}$ thus should devote to a recursive chain completion process; following [ABD+13], we call it a *chain reaction*.

If $\mathbf{T}$ completes chains for every encountered adjacent pair, then it may complete infinitely many chains, which is not acceptable. However, if $\mathbf{T}$ ignores too many adjacent pairs, then $D$ may be able to bypass $\mathbf{T}$'s chain-detection conditions and "trap" $\mathbf{T}$ in an over-constrained situation, e.g. try to create two contradictory records $(4,k_2,x,y,\perp)$ and $(4,k_2,x,y',\perp)$ to adapt two chains. To ensure $\mathbf{T}$ away from these two disasters, we have to choose a delicate chain detection strategy. To this end, note that detecting adjacent query-pairs resembles the strategy used for EMR [ABD+13]. Thus we take the *tripwire paradigm* introduced for EMR [ABD+13].

Informally, a *tripwire* is an ordered pair of the form $(i,i+1)$ or $(i+1,i)$ or $(1,4)$ or $(4,1)$. Using a tripwire configuration $(i,j)$ for $j = i+1$ or $j = i-1$ means that $\mathbf{T}$ will complete paths for adjacent pairs of $i$-query and $j$-query for which the $j$-query appears later than the $i$-query. On the other hand, using $(1,4)$ means that $\mathbf{T}$ will complete paths for pairs of queries $(1,k_1,u,v)$ and $(4,k_2,x,y)$ such that: (i) the 4-query appears later; (ii) the C-query $((k_1,k_2),u,y)$ exists in the history, i.e. $\widetilde{C}.\textsc{Check}((k_1,k_2),u,y)$ returns **true**. Vice versa for $(4,1)$. Whenever a tripwire is triggered (possibly by new adapted queries), $\mathbf{T}$ recurses to complete all the relevant chains.

[7] So that the simulated $i$-queries are consistent with a $(\kappa,n)$-blockcipher.

[8] Using such explicit randomness is equivalent to lazily sampling at the beginning of the experiment, as argued by Andreeva et al. [ABD+13,CS15].

For $\mathsf{CC}_4$, we adopt the following asymmetric configuration, cf. Fig. 4:

$$(1,2),(3,2),(3,4),(1,4).$$

This means only the simulated E2 and E4 serve as "beacons" for chain detection. As mentioned in Introduction, this configuration was designed for $\mathrm{EMR}_4$, yet abandoned due to the lack of a termination argument [ABD+13]. Whereas in our context we succeed in this task and finally achieve a proof for $\mathsf{CC}_4$.

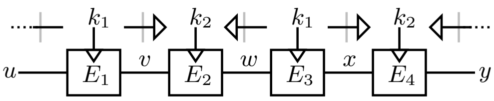


**Fig. 4.** Tripwire configuration for the simulation strategy for 4-cascade. A directed arrow from column $E_i$ to column $E_j$ indicates a tripwire $(i,j)$. The tripwires are $(1,2)$, $(3,2)$, $(3,4)$, $(1,4)$; note that the last one crosses $\mathbf{IC}[2\kappa,n]$.

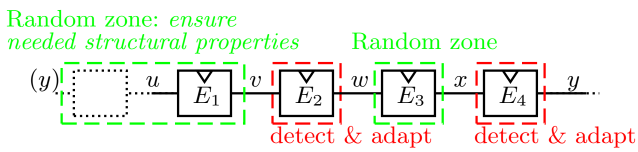


**Fig. 5.** Another illustration of the simulation strategy. The black dotted rectangle stands for the history of the ideal cipher $\mathbf{IC}[2\kappa,n]$.

The simulated E2 and E4 also facilitate adaptation, i.e. when completing chains, $\mathbf{T}$ always "closes cycles" by creating adapted 2- and 4-queries. In this way, the simulated E1 and E3 are reserved as two never adapted stages, and each 1- and 3-query would have one of their endpoints defined as a randomly sampled value. E.g. for the 3-query $(3,k_1,w,x,\rightarrow)$, the value $x$ was necessarily defined by an earlier randomly sampling action. Meanwhile, note that answers from $\widetilde{C}$ are random in the view of $D$ and $\mathbf{T}$, thus each C-query that appears during the interaction also has at least one "random endpoint". By the above, E1 along with $\widetilde{C}$ form the first "random zone", while E3 acts as the second one, cf. Fig. 5. Reserving two never adapted stages is indeed inspired by [ABD+13], which reserved the second and the forth simulated permutations (among five simulated ones) as never adapted ones.

Till now we have established the tripwire configuration. But this only makes the first step. Next we'll show how to handle such tripwires.

### 4.2 T Handling Tripwires: Trees, and Instructive Examples

Recall that tripwires are formed by a new 2- or 4-query and (probably more than one) pre-existing 1- or 3-queries in the random zones. The involved 1-/3-queries form structures, which should be considered by $\mathbf{T}$ upon detecting new tripwire(s). As each query in the random zones has a random endpoint (cf. the previous subsection), these structures possess special properties, which enables $\mathbf{T}$ to find the "most appropriate starting point" of the chain reaction and enforce the best order in which paths are completed.[9] Here how to react depends on the situation. To help understand our design, in this subsection, we serve several possible cases of interaction between the distinguisher $D$ and $(\widetilde{C},\mathbf{T})$ and introduce the underlying "layer-2" ProcessTree procedures which "deal with" these cases. We commence by briefing the properties of the structures formed by 1-, 3-, and C-queries (i.e. queries in random zones), as such properties are explicitly used in the remaining.

**Queries in Random Zones Give Rise to Tree-Structures.** Consider a bipartite graph $B_3$ built from all 3-queries. More clearly, $B_3$ takes $\{0,1\}^n$ and $\{0,1\}^n$ as the two shores, and includes an edge directed from the left-shore node $w$ to the right-shore node $x$ for each forward 3-query $(3,k_1,w,x,\rightarrow)$ and vice versa for each backward 3-query. A sequence of 3-queries thus may form directed paths in $B_3$, e.g. $w_1 \xrightarrow{\mathrm{E3}_{k_1^1}} x_1 \xrightarrow{\mathrm{E3}^{-1}_{k_1^2}} w_2 \xrightarrow{\mathrm{E3}_{k_1^3}} x_2 \ldots$

[9] Cf. [DS15b] for the importance of such chain-completion order (in another context).

If the answer of a later query collides with a pre-existing value, e.g. $\text{E3}^{-1}(k_1^4, x_2) \to w_3 = w_1$, then a cycle emerges, i.e. $w_1 - x_1 - w_2 - x_2 - (w_1)$. However, as long as the number of 3-queries is polynomial, such collisions are unlikely since the answers for later 3-queries are always randomly sampled. Thus $B_3$ is likely to be acyclic.

Similarly, for each $k_2$, consider a bipartite graph $CB(k_2)$ from *all* 1- and C-queries, which also takes $\{0,1\}^n$ and $\{0,1\}^n$ as the two shores, and includes an edge between the left-shore node $y$ and the right-shore node $v$ (denoted $(k_1, y, v)$) for each adjacent pair $((k_1, k_2), u, y)$ and $(1, k_1, u, v)$. The direction of the edge $(k_1, y, v)$ equals the direction of the later query among the adjacent pair. Due to the reserved randomness, $CB(k_2)$ is also acyclic. More clearly, w.h.p. connected components in $B_3$ and $CB(k_2)$ are *directed trees.*

These mean the mentioned structure (in the corresponding random zone, $B_3$ or $CB(k_2)$) involved by newly set off tripwires is also a directed tree. The subsequent chain reaction is "carried around" this tree (cf. Case 1 below). $D$ may use different query sequences to create various trees and force $\mathbf{T}$ to tackle; we serve 4 examples as follows.

**Case 1. The involved tree is not adjacent to any pre-existing 2-/4-query.** Since the graph $CB(k_2)$ is more complicated and somewhat novel, we consider examples around it. Assuming four edges $(k_1^1, y_1, v_1)$, $(k_1^2, y_2, v_1)$, $(k_1^3, y_2, v_2)$, and $(k_1^4, y_2, v_3)$,[10] and $D$ querying $\text{E2}(k_2, v_1)$, cf. Fig. 6 (left). Suppose $\mathbf{T}$ randomly samples $w_1$ as the answer and creates $(2, k_2, v_1, w_1, \to)$, a new 2-query. This leads $\mathbf{T}$ to detecting two $(1,2)$-tripwires due to $(2, k_2, v_1, w_1)$ and $(1, k_1^1, u_1, v_1)$ and $(1, k_1^2, u_2, v_1)$, completes the two associated chains one-by-one and creates two adapted 4-queries $(4, k_2, x_1, y_1, \perp)$ and $(4, k_2, x_2, y_2, \perp)$. $\mathbf{T}$ then notices that $(4, k_2, x_2, y_2)$ and $(1, k_1^3, u_3, v_2)$ and $(1, k_1^4, u_4, v_3)$ form two new $(1,4)$-tripwires, and completes them one-by-one and creates two adapted 2-queries. Such a recursive process would continue and "attach" adapted queries to the nodes in the tree one-by-one till reaching the "endpoints" $y_1$, $v_2$, and $v_3$ (they'll be defined as *leaves* of the tree). After this process, each node in the tree is adjacent to a 2- or 4-query with key $k_2$, cf Fig. 6 (right).

To capture these considerations, we take Andreeva et al.'s terminology *pebbling* and *live tree* [ABD+13]. Informally, under a key $k_2$, a left-shore node $y$ (a right-shore node $v$, resp.) is *pebbled* if it has been adjacent to a 4-query (2-query, resp.) with key $k_2$. The *live tree anchored at a non-pebbled node* $z$ (denoted $Li(k_2, z)$; indifferently of $z$ in left-shore or right-shore) is the tree obtained by "dangling" the connected component (in $CB(k_2)$) containing $z$ by $z$, such that $z$ is the root, and then pruning all portions of this "dangled" tree that lie beneath a node pebbled under $k_2$. E.g. in this example, the four edges give rise to a tree $Li(k_2, v_1)$ with $y_1$, $v_2$, and $v_3$ as leaves. Note that as all the nodes under the pebbled ones have been pruned, only leaves can be pebbled nodes.

Under these definitions, Case 1 is translated as *the involved live tree $Li(k_2, v_1)$ has no pebbled leaf (no leaf pebbled under $k_2$).* The contrary cases are as follows.

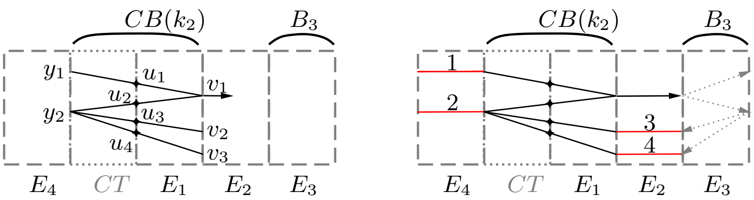


**Fig. 6.** Figures for Case 1. The directed edge indicates the query $\text{E2}(k_2, v_1)$. The keys $k_1^1$, etc. are omitted as they are less interested here. (Left) the state of the query history before the chain reaction; (Right) the state after the reaction (with the values $y_1$, etc. omitted for clearness). Each (red) edge with a number on it indicates an adapted query, and these associated numbers indicate the order of their creation. The newly created 3-queries are not interested here and are thus drawn in dotted (and gray).

**Case 2. The involved tree has one pebbled leaf.** The case would be different, if a leaf in the involved live tree in $CB(k_2)$ has been pebbled under $k_2$ before the reaction. For this, assuming four edges as before and a pre-existing 2-query $(2, k_2, v_3, w_3)$. In this case, upon $D$ querying $\text{E2}(k_2, v_1)$ which sets off new $(1,2)$-tripwires, if $\mathbf{T}$ insists on starting its chain reaction from $v_1$, then after it creates $(4, k_2, x_2, y_2, \perp)$ (as done in Case 1), it

[10] In detail, four pairs of queries $((k_1^1, k_2), u_1, y_1)$, $(1, k_1^1, u_1, v_1)$, $((k_1^2, k_2), u_2, y_2)$, $(1, k_1^2, u_2, v_1)$, $((k_1^3, k_2), u_3, y_2)$, $(1, k_1^3, u_3, v_2)$, $((k_1^4, k_2), u_4, y_2)$, and $(1, k_1^4, u_4, v_3)$.

would find $v_3$ already pebbled. This means **T** finds a chain $x_2 - y_2 - u_4 - v_3 - w_3$ that can only be adapted at E3. So **T** has to either admit a failure, or destroy the randomness of E3 (which would significantly complicate the proof), cf. Fig. 7 (left).

Here we give a brief note: starting the chain reaction from the vertex specified by $D$'s query is indeed the choice of Andreeva et al. [ABD+13] for EMR$_5$. But their simulator has 5 rounds to "play with" (more precisely, 3 rounds for adaptation, which is one more than our 2 stages) and thus could succeed, while ours **T** works in a more constrained setting of 4 stages and cannot follow this strategy.

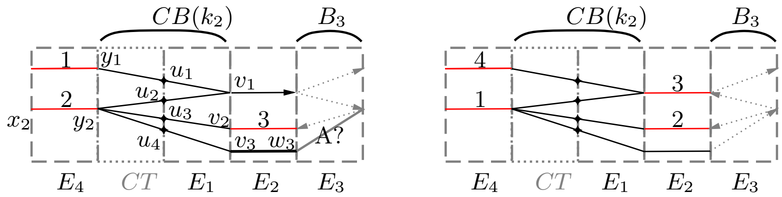


**Fig. 7.** Figures for Case 2. The pre-existing query $(2, k_2, v_3, w_3)$ is indicated by the black bold undirected edge. (Left) the unsuccessful trial: **S** has to adapt in E3; (Right) a more appropriate operation sequence leads to success.

However, if **T** could be a bit more patient, then it would easily overcome the above dilemma: after **T** notices that the last query E2$(k_2, v_1)$ from $D$ may set off new tripwires, instead of *immediately creating* $(2, k_2, v_1, w_1, \rightarrow)$, **T** first *traverses* in the live tree $Li(k_2, v_1)$ (in $CB(k_2)$) to figure out how many nodes have been pebbled under $k_2$. **T** therefore finds the pebbled node $v_3$, and then starts the chain reaction from $v_3$. More clearly, **T** adapts by "attaching" adapted queries to $y_2$, $v_2$, $v_1$, and $y_1$ successively, cf. Fig. 7 (right). After this, there is a newly created *adapted* query attached to $v_1$ (in contrast to the previous Case 1, in which the 2-query attached to $v_1$ is created by *randomly sampling*), so that **T** is able to answer $D$'s original query E2$(k_2, v_1)$.

The aforementioned traversal is performed by a procedure FindPebLeafCB. On the other hand, the subsequent chain reaction is performed by a procedure ProcessCBSubTree, which finishes the task by recursively calling itself. We give an overview of them, before we present the next instructive example.

Traversing in CB: FindPebLeafCB. The access to the whole history of $\widetilde{C}$ seems necessary for **T** to construct the graph $CB(k_2)$ and then traverse in it. However, **T** can only call $\widetilde{C}$.Check to verify the existence of certain C-queries. Thus we have to implement FindPebLeafCB based on Check.

Our solution is as follows. Note that the traversal algorithm has two scenarios. The first is when it reaches a left-shore node $y$, and would like to determine all the edges adjacent to $y$ (and then "jumps" to the children of $y$ via these edges). In this case, if an edge $(k_1, y, v)$ exists for some $k_1, v$, then the corresponding 1-query $(1, k_1, u, v)$ necessarily exists. Therefore, **T** could call $\widetilde{C}$.Check$((k_1, k_2), u, y)$ for each pre-existing 1-query $(1, k_1, u, v)$ to determine whether the C-query $((k_1, k_2), u, y)$ exists, which further indicates whether the edge $(k_1, y, v)$ exists.

The second case is when the algorithm reaches a right-shore node $v$, and would like to determine all the edges adjacent to $v$ and "jump" to the children of $v$ through them. Still, if $(k_1, y, v)$ exists for some $k_1, v$, then the 1-query $(1, k_1, u, v)$ must exist. But different to the first case, here **T** does not know for which $y$ should it call Check$((k_1, k_2), u, y)$. However, we observe that after "grasping" the 1-query $(1, k_1, u, v)$, **T** could simply queries $\widetilde{C}$.C$((k_1, k_2), u) \rightarrow y$ to obtain the left-shore node $y$, and take $(k_1, y, v)$ as if it indeed pre-exists. The underlying observation is as follows: after **T** obtains such $y$, **T** will complete some tripwires formed by relevant edges and will attach a new 2-query to $v$ at some time, which would form a $(1, 2)$-tripwire with $(1, k_1, u, v)$ and force **T** to issue the C-query C$((k_1, k_2), u)$; by this, the query C$((k_1, k_2), u) \rightarrow y$ will be made by **T** sooner or later, and it makes no difference if we let it appear earlier. These constitute the ideas of FindPebLeafCB.

ProcessCBSubTree Procedure. We first give some insights on ProcessCBSubTree. Note that in the above example, when **T** is to start chain reaction, it is expected to "pebble" the nodes in the live tree $Li(k_2, y_2)$ (which captures the same nodes as $Li(k_2, v_1)$ but takes a different root), and it has "grasped" $v_3$, the unique pebbled leaf of $Li(k_2, y_2)$, cf. Fig. 8 (left). Later after **T** attaches an adapted query to $y_2$, **T** is faced with a similar situation for each child of $y_2$, cf. Fig. 8 (right): **T** is to "deal with" $Li(k_2, v_1)$ and $Li(k_2, v_2)$ (warning: they refer to the live trees *at the current point*), and **T** has "grasped" $y_2$, the unique pebbled leaf of both $Li(k_2, v_1)$ and $Li(k_2, v_2)$. It can be seen that similar situations are successively presented to **T** till the end of the chain reaction.

These observations motivate the recursive implementation of ProcessCBSubTree. More clearly, to start a chain reaction as above we let **T** make a call to ProcessCBSubTree$(k_1^4, k_2, y_2, v_3, left)$. Among the arguments, the tuple $(k_1^4, y_2, v_3)$ identifies the edge $(k_1^4, y_2, v_3)$, and the fifth argument $left$ means that among the two endpoints of $(k_1^4, y_2, v_3)$, the non-pebbled one/the root of the to-be-processed tree is in the left shore of $CB(k_2)$. This ProcessCBSubTree-call would have two steps:

(1) complete the chain identified by its arguments. In the example the chain is $y_2 - u_4 - v_3 - w_3$, and ProcessCBSubTree first calls E3$(k_1^4, w_3)$ to create $(3, k_1^4, w_3, x_2, \rightarrow)$ and then creates an adapted query $(4, k_2, x_2, y_2, \perp)$.
Right before creating this adapted query, $x_2$ may have been "occupied", i.e. there has been a 4-query $(4, k_2, x_2, y_2')$ for some $y_2'$. In this case, creating $(4, k_2, x_2, y_2, \perp)$ would certainly cause **T** fail to emulate the forth ideal cipher; we thus let **T** *abort*. Consequently, $D$ would know it is interacting with the simulated world. We thus need to prove this situation occurs with negligible probability. Jumping ahead, the proof is in page 17.

(2) for each child $z$ of $y_2$, call itself with the arguments identified by the edge between $y_2$ and $z$. In the example, it makes two calls ProcessCBSubTree$(k_1^2, k_2, y_2, v_1, right)$ and ProcessCBSubTree$(k_1^3, k_2, y_2, v_2, right)$ – note that the involved non-pebbled nodes $v_1$ and $v_2$ are in the *right* shore, as identified by the fifth arguments of these sub-calls.

The ProcessCBSubTree-calls with *right* as the fifth argument runs symmetrically. By such a recursively-calling mechanism, the chain reaction "burns" from the starting point "through" the whole live tree. As the effects, **T** would alternatively create adapted 2- and 4-queries and finally pebble all the nodes in the live tree – as we expect.

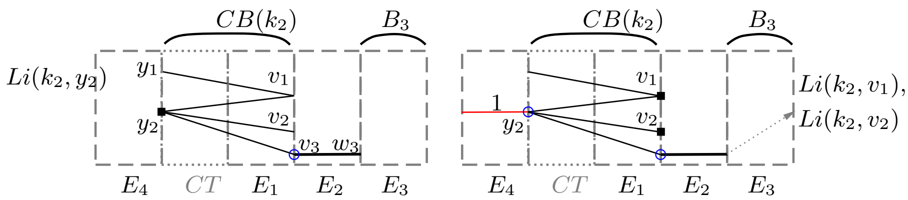


**Fig. 8.** Illustration for the recursive process. The blue circles indicate pebbled nodes, while the black squares indicate the roots of the live trees. (Left) **S** is to deal with $Li(k_2, y_2)$; (Right) After attaching an adapted query to $y_2$, **S** is to deal with $Li(k_2, v_1)$ and $Li(k_2, v_2)$.

Although the above two cases focus on $CB(k_2)$, the discussion can be easily transferred to $B_3$: similar terminology are used and similar interactions are considered. **T** traverses in $B_3$, and performs similar subsequent chain reactions. As $B_3$ is built from the 3-queries simulated by **T**, it's much easier to traverse in $B_3$, and this is implemented as a procedure FindPebLeafB3. On the other hand, the chain reactions around $B_3$ is performed by a procedure ProcessB3SubTree via recursion (similarly to ProcessCBSubTree).

**Case 3. The involved tree has more than one pebbled leaves.** Live trees involved in previous examples have at most one pebbled leaf. If the tree has more than one such leaves, then **T** cannot adapt. For this, assuming the tree $y_1 - v_1 - y_2 - (v_2, v_3)$ as before and its two leaves $y_1$ and $v_3$ pebbled due to $(4, k_2, x_1, y_1)$ and $(2, k_2, v_3, w_3)$ respectively, while $v_1$, $y_2$, and $v_2$ are not pebbled. Now upon the query E2$(k_2, v_1)$, if **T** starts chain reaction from $y_1$, then it will fail when reaching the pebbled leaf $v_3$, cf. Fig. 9 (left); if **T** starts from $v_3$ then it will fail when reaching $y_1$, cf. Fig. 9 (right). This case thus cannot be resolved. To summarize, if $D$ can create a live tree in $CB(k_2)$ with (at least) two leaves pebbled under $k_2$, then the strategy in Fig. 5 would fail and thus be useless.

Fortunately, we prove that *under the strategy in Fig. 5, such a "doubly-pebbled" live tree is unlikely to appear.* More clearly, we prove a claim similar to [ABD+13]: as long as $CB(k_2)$ remains acyclic (which is indeed likely), *if a node in $CB(k_2)$ is pebbled under $k_2$, then its parent is also pebbled under $k_2$*, so that *it is never possible that two leaves of a tree are pebbled while the root is not* (so, a live tree, whose root has to be non-pebbled, cannot have two pebbled leaves). Similar claims are proved for structures in $B_3$. These eliminate the apparent failure possibilities and enable us to step further.

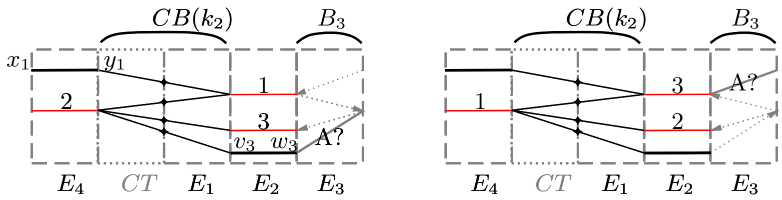


**Fig. 9.** Figures for Case 3. **S** has to adapt in E3 regardless of which sequence of operations it takes.

**Case 4. Dual-tree: two trees are involved, and the procedure** ProcessDualTree. In all the above cases only one live tree is involved in the subsequent reaction. Indeed, $D$ could force **T** to consider two trees linked by a 2- or 4-query, and this constitutes the most complicated case. For this consider the following example. Assume that $D$ first makes $\mathrm{E2}(k_2, v) \to w$, then makes $\mathrm{E1}^{-1}(k_1, v) \to u$, $\mathrm{C}((k_1, k_2), u) \to y$, $\mathrm{E3}(k_1, w) \to x$, and then makes a dozen of 1-, 3-, and C-queries to enlarge the two live trees $Li(k_2, y)$ and $Li(k_2, x)$ without setting off any tripwire (cf. Fig. 10 (left)), and finally uses a query e.g. $\mathrm{E4}(k_2, x)$ to "light the fuse".

We now give some insights on how to handle this case. We first bring out two features of the structure upon the "fuse-lighting" query:

(i) w.h.p. $v$ is the unique pebbled leaf of $Li(k_2, y)$, because as stressed in Case 3 $Li(k_2, y)$ is unlikely to contain more than one pebbled leaves before the chain reaction. Similarly, $w$ is the unique pebbled leaf of $Li(k_2, x)$;
(ii) the chain $y - u - v - w - x$ will be completed, once we add the 4-query $(4, k_2, x, y)$.

We will informally call such structures *dual-trees*, as it consists of two trees $Li(k_2, x)$ and $Li(k_2, y)$ linked by a 2-query $(2, k_2, v, w)$.

We then describe how **T** reacts. As the first step, **T** traverses in $Li(k_2, x)$ (as described in Case 2). As a result, **T** would find the pebbled leaf $w$ and further $v$, $u$, $y$, and be aware of the whole situation. To handle such a dual-tree, **T** makes a call to a procedure ProcessDualTree$(k_1, k_2, x, y, 4)$ (the fifth argument indicates that the "missing query" is a 4-query, as mentioned), which creates the adapted query $(4, k_2, x, y, \perp)$, cf. Fig. 10 (left).

After this **T** (more precisely, the ProcessDualTree-call) should consider all the $(1, 4)$- and $(3, 4)$-tripwires newly set off by $(4, k_2, x, y)$ – that is to say, consider all the pre-existing edges in $B_3$ and $CB(k_2)$ that is adjacent to $(4, k_2, x, y)$. Here the involved edges (and structures) fall into three possibilities. For clearness, we give three examples as follows:

(i) for an edge $(3, k_1^1, w^1, x)$ in $B_3$, if the edge $(k_1^1, y, v^1)$ already exists in $CB(k_2)$, then along with $(4, k_2, x, y)$ they indeed form another (smaller) dual-tree structure, cf. the black subtrees in Fig. 10 (right). In particular, $Li(k_2, v^1)$ and $Li(k_2, w^1)$ are linked by the 4-query $(4, k_2, x, y)$, and the chain $w^1 - x - y - v^1$ will be completed if we add the 2-query $(2, k_2, v^1, w^1)$. In this case **T** makes a call to ProcessDualTree$(k_1^1, k_2, v^1, w^1, 2)$ to handle this (smaller) dual-tree;
(ii) on the contrary, if for an edge $(3, k_1^2, w^2, x)$ in $B_3$ there is no edge of the form $(k_1^2, y, \cdot)$ in $CB(k_2)$, then only a $(3, 4)$-tripwire is set off, and **T** would be led to handle a live tree $Li(k_2, w^2)$ in $B_3$ (cf. the blue subtree in Fig. 10 (right)). In this case **T** makes a call to ProcessB3SubTree$(k_1^2, k_2, w^2, x, left)$;
(iii) on the other hand, for an edge $(k_1^3, y, v^3)$ in $CB(k_2)$, if there does not exist any edge of the form $(3, k_1^3, \cdot, x)$ in $B_3$ (cf. the green subtree in Fig. 10 (right)), then **T** makes a call to ProcessCBSubTree$(k_1^3, k_2, y, v^3, right)$ to handle the live tree $Li(k_2, v^3)$.

The ProcessDualTree-calls with 2 as the fifth argument runs similarly. For elaboration consider ProcessDualTree$(k_1^1, k_2, v^1, w^1, 2)$: it first creates $(2, k_2, v^1, w^1, \perp)$ and then makes several sub-calls similarly as ProcessDualTree$(k_1, k_2, x, y, 4)$. The dual-tree structure would thus be processed in a series of recursive calls as above. Once all the subsequent calls are finished without abortion, all the nodes in the dual-tree have been pebbled, and **T** could answer the original "fuse-lighting" query $\mathrm{E4}(k_2, x)$.

In the rest of the paper we would say "layer-2 ProcessTree-procedure" to indifferently refer to ProcessB3SubTree, ProcessCBSubTree, or ProcessDualTree. For the sake of page limits and cleanness we defer the technical details of these procedures to the code, Appendix B.

### 4.3 T Handling Queries: An Overview

Based on the underlying mechanism above, we now describe how **T** handles different queries. First, old queries are simply answered with the recorded values. Second, upon a new 1- or 3-query, **T** simply randomly samples a

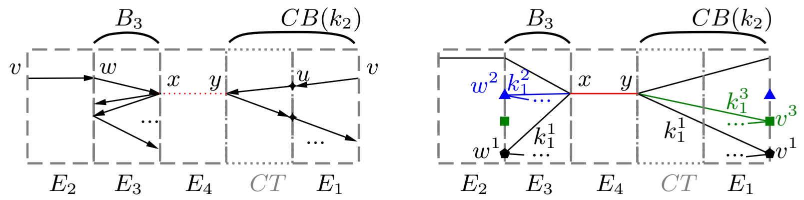


**Fig. 10.** Figures for Case 4. (Left) before the reaction – **S** is to create the adapted 4-query $(4, k_2, x, y, \perp)$; (Right) the state right after $(4, k_2, x, y, \perp)$ is created (with more concretely drawn edges). Note that different possibilities for sub-calls are differentiated by different colors and polygons.

value as the answer, and creates a (new) query record. The process around new 2- and 4-queries is complicated: **T** may have to traverse in the involved live tree, distinguish which (instructive) case the situation fits into, and call the right layer-2 ProcessTree procedure.

**New Query E2$^{-1}$($k_2, w$) and E4($k_2, x$).** Consider $\mathrm{E2}^{-1}(k_2, w)$ first. Upon such a new query, if there is no pre-existing 3-query of the form $(3, k_1, w, \cdot)$, then no $(3, 2)$-tripwire would be set off. In this case, **T** simply randomly samples $v$ and creates a record $(2, k_2, v, w, \leftarrow)$. If there does pre-exist 3-queries $(3, k_1, w, \cdot)$, then **T** calls FindPebLeafB3 to traverse in $Li(k_2, w)$ (in $B_3$) and check the number of leaves pebbled under $k_2$, and reacts depending on the case:

*(1) If there is no pebbled leaf in $Li(k_2, w)$,* then it fits into the instructive Case 1 (page 10), and **T** "starts the chain reaction from $w$". More clearly, **T** first creates $(2, k_2, v, w, \leftarrow)$ with randomly sampled $v$. Note that this makes $w$ pebbled, so that for each pre-existing 3-query $(3, k_1, w, x)$, $x$ becomes the root of a live tree with $w$ as one pebbled leaf. After this, for each such $(3, k_1, w, x)$ **T** makes a call to ProcessB3SubTree$(k_1, k_2, w, x, right)$. As sketched before (page 12), this procedure would "process" $Li(k_2, x)$ by recursion.

*(2) If there is exactly one pebbled leaf (denoted $z^\circ$),* then FindPebLeafB3 returns the edge in $Li(k_2, x)$ that contains $z^\circ$. Now **T** should further distinguish if it is in Case 2 (page 10) or Case 4 (page 13). Briefly speaking, if **T** finds $Li(k_2, x)$ "involved in" a dual-tree, then it calls ProcessDualTree; otherwise it calls ProcessB3SubTree to deal with $Li(k_2, x)$ alone. For clearness, according to which shore $z^\circ$ is in, we distinguish two cases:

- $z^\circ$ is in the left shore of $B_3$. Assume that the edge in $Li(k_2, x)$ that contains $z^\circ$ is $(3, k_1^\circ, w^\circ, x^\circ)$ (rewriting $z^\circ$ as $w^\circ$), while the 2-query attached to $w^\circ$ is $(2, k_2, v^\circ, w^\circ)$. Then:
  - if there's no pre-existing 1-query of the form $(1, k_1^\circ, \cdot, v^\circ)$, then it fits into Case 2 (page 10), and **T** simply calls ProcessB3SubTree$(k_1^\circ, k_2, w^\circ, x^\circ, right)$ to start the chain reaction (and handle $Li(k_2, x)$ alone).
  - if there's a 1-query $(1, k_1^\circ, u^\circ, v^\circ)$ for some $u^\circ$, then **T** queries $\widetilde{C}.\mathrm{C}((k_1^\circ, k_2), u^\circ) \to y^\circ$. At this point the (possibly new) node $y^\circ$ would likely be *non-pebbled*, and thus the live tree $Li(k_2, y^\circ)$ makes sense. As $Li(k_2, y^\circ)$ and $Li(k_2, x^\circ)$ are linked by $(2, k_2, v^\circ, w^\circ)$, it now fits into Case 4 (page 13). Therefore, **T** calls ProcessDualTree$(k_1^\circ, k_2, x^\circ, y^\circ, 4)$ to handle.
- $z^\circ$ is in the right shore of $B_3$. Assume that the edge in $Li(k_2, x)$ that contains $z^\circ$ is $(3, k_1^\circ, w^\circ, x^\circ)$ (rewriting $z^\circ$ as $x^\circ$) while the 4-query attached to $x^\circ$ is $(4, k_2, x^\circ, y^\circ)$. In this case, **T** calls Check$((k_1^\circ, k_2), u, y^\circ)$ for each pre-existing 1-query $(1, k_1^\circ, u^\circ, v^\circ)$ to determine whether an edge $(k_1^\circ, y^\circ, v^\circ)$ pre-exists in $CB(k_2)$. Depending on the result:
  - if none of the Check-calls return **true**, then there's no edge of the form $(k_1^\circ, y^\circ, \cdot)$ in $CB(k_2)$. Thus it fits into Case 2 (page 10) and **T** calls ProcessB3SubTree$(k_1^\circ, k_2, w^\circ, x^\circ, left)$ to tackle;
  - otherwise it fits into Case 4 (page 13), **T** finds the edge $(k_1^\circ, y^\circ, v^\circ)$ and then calls ProcessDualTree$(k_1^\circ, k_2, v^\circ, w^\circ, 2)$ to deal with the dual-tree formed by $Li(k_2, v^\circ)$ and $Li(k_2, w^\circ)$.

*(3) If there are more than one pebbled leaves,* then **T** aborts. As discussed (Case 3, page 12), w.h.p. this situation would not occur.

In each case, once the ProcessTree-procedure returns and abort does not occur, the 2-query $(2, k_2, v, w)$ must have been created, which allows **T** to return $v$ to answer the original query $\mathrm{E2}^{-1}(k_2, w)$.

Forward 4-queries are handled similarly. Upon a new query $\mathrm{E4}(k_2, x)$, if there is no pre-existing 3-query $(3, k_1, \cdot, x)$, then **T** randomly samples $y$ to answer. If there exists 3-queries $(3, k_1, \cdot, x)$, then new $(3, 4)$-tripwires are set off, and **T** calls FindPebLeafB3 to traverse in $Li(k_2, x)$. The subsequent process is similar as that around new $\mathrm{E2}^{-1}(k_2, w)$ and is thus omitted.

**New Query $\mathbf{E4^{-1}}(k_2, y)$ and $\mathbf{E2}(k_2, v)$.** Consider a new query $\mathrm{E4}^{-1}(k_2, y)$ first, upon which **T** first determines whether it sets off new $(1, 4)$-tripwires, by calling $\textsc{Check}((k_1, k_2), u, y)$ for each pre-existing 1-query $(1, k_1, u, v)$. If **T** detects no new $(1, 4)$-tripwire, then it answers with randomly sampled $x$ and creates $(4, k_2, x, y, \leftarrow)$; otherwise, $y$ is adjacent to at least one edge in $CB(k_2)$, and **T** calls FindPebLeafCB to traverse in the live tree $Li(k_2, y)$ and reacts depending on the pebbling state:

*(1) If there is no pebbled leaf,* then (it fits into Case 1 and) **T** first creates $(4, k_2, x, y, \leftarrow)$ with randomly sampled $x$ and then considers each pre-existing edge $(k_1, y, v)$ in $CB(k_2)$ (by calling Check, as described) and makes a call to $\textsc{ProcessCBSubTree}(k_1, k_2, y, v, right)$ for each of them.

*(2) If there is one pebbled leaf (denoted $z^\circ$),* then FindPebLeafCB returns the edge in $Li(k_2, x)$ that contains $z^\circ$. Similarly as described, **T** calls ProcessDualTree if it finds $Li(k_2, y)$ "involved in" a dual-tree, and calls ProcessCBSubTree otherwise. For elaboration, consider the case of $z^\circ$ in the left shore of $CB(k_2)$ (for cleanness, the symmetrical case is omitted here). Assume that the edge in $Li(k_2, y)$ that contains $z^\circ$ is $(k_1^\circ, y^\circ, v^\circ)$ while the 4-query attached to $y^\circ$ is $(4, k_2, x^\circ, y^\circ)$. Then:

– if there's no pre-existing 3-query of the form $(3, k_1^\circ, \cdot, x^\circ)$, then it fits into Case 2 (page 10) and **T** calls $\textsc{ProcessCBSubTree}(k_1^\circ, k_2, y^\circ, v^\circ, right)$ to deal with $Li(k_2, v^\circ)$;
– if there's a 3-query $(3, k_1^\circ, w^\circ, x^\circ)$ for some $w^\circ$, then it fits into Case 3 (page 12) and **T** calls $\textsc{ProcessDualTree}(k_1^\circ, k_2, v^\circ, w^\circ, 2)$ to deal with the dual-tree formed by $Li(k_2, v^\circ)$ and $Li(k_2, w^\circ)$.

*(3) If there are more than one pebbled leaves,* then **T** aborts.

**T** answers the original query $\mathrm{E4}^{-1}(k_2, y)$ with the recorded $x$ once the tree processing procedure returns without abortion.

Finally, upon a new query $\mathrm{E2}(k_2, v)$, if there is no pre-existing 1-query $(1, k_1, \cdot, v)$, then **T** randomly samples $u$ to answer. If there exists 1-queries $(1, k_1, \cdot, v)$, then new $(1, 2)$-tripwires are set off, and **T** calls FindPebLeafCB to traverse in $Li(k_2, v)$. The subsequent process is similar to that around new $\mathrm{E4}^{-1}(k_2, y)$, thus omitted.

### 4.4 Obtaining S from T

Note that $\mathbf{T}^{\widetilde{C}}$ takes advantage of the "illegal" interface Check offered by $\widetilde{C}$. As a "normal" ideal cipher **C** certainly does not provide "Check", $\mathbf{S^C}$ implements this procedure itself. In detail, compared to $\mathbf{T}^{\widetilde{C}}$, $\mathbf{S^C}$ incorporates the following two modifications:

(i) $\mathbf{S^C}$ has an additional procedure $\mathbf{S^C}.\textsc{Check}(K, u, y)$, which queries $\mathbf{C}.\mathrm{C}(K, u)$, and returns **true** if $\mathbf{C}.\mathrm{C}(K, u) = y$;
(ii) Each time $\mathbf{T}^{\widetilde{C}}$ calls $\widetilde{C}.\textsc{Check}(K, u, y)$, $\mathbf{S^C}$ calls $\mathbf{S^C}.\textsc{Check}(K, u, y)$ instead.

The idea dates back to [CHK+14]: briefly speaking, if $(K, u, y)$ really appeared in the history of **C**, then clearly $\mathbf{C}.\mathrm{C}(K, u) = y$ and $\mathbf{S^C}.\textsc{Check}(K, u, y)$ returns **true**; otherwise, as long as Check is called polynomial times, it holds $Pr[\mathbf{C}.\mathrm{C}(K, u) = y] = O(\frac{1}{2^n})$, and thus w.h.p. $\mathbf{S^C}.\textsc{Check}(K, u, y)$ returns **false**. Thus this modification is unlikely to cause essential difference, while **S** is a "legal" simulator for $\mathsf{CC}_4$.

*Remark.* Access to the *entire* history of **C** seems necessary for $\mathbf{S^C}$ to traverse in $CB(k_2)$, but should not be possible in indifferentiability setting. In particular, $D$'s queries to **C** are *never* leaked to **S**, and **S** cannot construct the described graph $CB(k_2)$. This is why we introduce $\mathbf{T}^{\widetilde{C}}$ first and take it as an intermediate step.

### 4.5 Proof Overview

This subsection sketches the key points of the proof of Theorem 1. The formal presentation is deferred to Appendix C and D.

Denote by $G_1$ the simulated system consisting of $\mathbf{C}$ and $\mathbf{S}$, and by $G_3$ the real system formed by $\mathsf{CC}_4$ and $\mathbf{E}$. Then we need to show the following two claims for any computationally unbounded distinguisher $D$:

(i) $G_1(\mathbf{C}, \mathbf{S}^{\mathbf{C}})$ and $G_3(\mathsf{CC}^{\mathbf{E}}, \mathbf{E})$ are indistinguishable for $D$.

(ii) The query and time complexity of $\mathbf{S}^{\mathbf{C}}$ in $D^{G_1(\mathbf{C},\mathbf{S}^{\mathbf{C}})}$ are polynomial (at least with overwhelming probability).

The aforementioned intermediate system $G_2$ is formed by $\widetilde{C}$ and $\mathbf{T}$. As sketched in subsection 4.4, $G_1$ and $G_2$ are indistinguishable, if $\mathbf{T}$ makes polynomial number of calls to $\widetilde{C}$.Check. Thus the crux is to bound the complexity of $\mathbf{T}$ in $G_2$ (i.e. termination argument), as well as proving the indistinguishability of $G_2$ and $G_3$. The latter goal is done via a randomness mapping argument, which mainly requires proving $\mathbf{T}$ always succeeds in adapting chains/never aborts due to adaptations. In all, we should focus on analyzing $\mathbf{T}$ – or $G_2$.

The next subsection introduces *distinguisher which completes all chains*, a standard approach to indifferentiability proofs for idealized blockciphers. The remaining three subsections sketches respectively the termination argument for $\mathbf{T}$, calculating the abort probability for $\mathbf{T}$, and the indistinguishability of systems. Moreover, the formal proof for $\mathbf{T}$'s termination is in Appendix C, Lemmata 12-14, 15-20.

**Distinguisher that Completes All Chains.** For a fixed deterministic distinguisher $D$, the corresponding completing-all-chain distinguisher $\overline{D}$ first runs $D$, then queries E for each $D$'s query to C, and finally outputs whatever $D$ outputs. More clearly, for each $D$'s query $\mathrm{C}((k_1, k_2), u) \to y$ or $\mathrm{C}^{-1}((k_1, k_2), y) \to u$, $\overline{D}$ sequentially queries $\mathrm{E1}(k_1, u) \to v$, $\mathrm{E2}(k_2, v) \to w$, $\mathrm{E3}(k_1, w) \to x$, and $\mathrm{E4}(k_2, x) \to y$. Clearly $\overline{D}$ has exactly the same advantage as $D$ in distinguishing $G_2$ and $G_3$; all the rest arguments thereby concentrate on this fixed $\overline{D}$. Limiting $D$ to deterministic ones is *wlog* since the advantage of a probabilistic distinguisher cannot exceed the corresponding deterministic version with the best random tapes.

Note that each query of $D$ results in at most one $\overline{D}$'s query of a certain type. For instance, consider a query of $D$: if it is a 1-query, then $\overline{D}$ makes the same 1-query when it runs $D$ to this step; if it is a C-query, then $\overline{D}$ makes a corresponding 1-query when it completes the chain; otherwise, $\overline{D}$ does not make 1-query (relative to this query). By this, if $D$ issues at most $q$ queries, then in any execution $\overline{D}^{G_i}$ we have:

- for $i = 1, 2, 3, 4$, the number of $i$-queries made by $\overline{D}$ is at most $q$;
- $\overline{D}$ makes at most $q$ C-queries.

These observations will be used to derive the bounds on complexity.

**Termination Argument for T.** Let $KSet_1$ and $KSet_2$ be the sets of keys $k_1$ and $k_2$ appeared in the interaction respectively. E.g. if $\mathbf{T}$ created a query $(1, k_1, u, v)$, then $k_1 \in KSet_1$. Further denote by $E_i$ the set of $i$-queries for $i = 1, 2, 3, 4$. Then the key observation is that $|E_3|$ can only be affected by a certain type of chain completions (one may deem such chains as the "outer" chains in similar arguments, e.g. [LS13]). The number of such chain completions does not exceed $2q \cdot |KSet_1|$. This cinches the bound on $|E_3|$, and further enforces $|KSet_2| \cdot |E_3|$ as the bound of the number of chains completed by $\mathbf{T}$.

We then provide an overview as follows. First, based on our analysis of $\mathbf{T}$'s process, $|KSet_1|$ stays constant during chain reactions, thus can only be enlarged due to $\overline{D}$'s 1- or 3-queries bringing in new $k_1$ values. As the *original* distinguisher $D$ makes at most $q$ queries, $\overline{D}$'s 1- and 3-queries bring in at most $q$ distinct $k_1$ values in total, and hence $|KSet_1| \le q$. Similarly, $|KSet_2| \le q$.

We then show $|E_3| \le 3q^2$. During chain reactions around live trees in $B_3$, no 3-query is created. Hence besides $\overline{D}$ issuing 3-queries, $|E_3|$ is only enlarged during chain reactions around live trees in $CB(k_2)$ (whether these trees are involved in dual-trees or not does not matter). Denote by $T^i_{CB}$ the $i$-th involved live tree in $CB(k_2)$, by $|T^i_{CB}|$ the number of edges in $T^i_{CB}$, and by $Rv(T^i_{CB})$ the number of right-shore vertexes in $T^i_{CB}$. Then $T^i_{CB}$ brings in at most $|KSet_1| \cdot Rv(T^i_{CB})$ increment to $|E_3|$: essentially, this is because during the chain reaction, for each right-shore node $v$, $\mathbf{T}$ may encounter at most $|KSet_1|$ $(1,2)$-tripwires; the amount $|KSet_1| \cdot Rv(T^i_{CB})$ is indeed the bound on the number of chain completions.

Chain reactions around $CB(k_2)$ may be the consequence of $\overline{D}$ issuing 2- or 4-queries. For a live tree $T^i_{CB}$ involved by query $\mathrm{E2}(k_2, v)$, it holds $Rv(T^i_{CB}) \le |T^i_{CB}|+1$;[11] for the other possibilities it holds $Rv(T^i_{CB}) \le |T^i_{CB}|$. As $\overline{D}$ makes at most $q$ E2 queries, chain reactions bring in at most $(q + \sum_i |T^i_{CB}|) \cdot |KSet_1|$ increment to $|E_3|$.

[11] This is because $\mathrm{E2}(k_2, v)$ itself identifies a right-shore node $v$.

Note that the edges in $T^i_{CB}$ are necessarily due to $\overline{D}$ querying $\widetilde{C}$ and thus $\sum_i |T^i_{CB}| \le q$: because once **T**'s action leads to creating a new C-query, **T** would soon complete the corresponding chain, and hence the edges in $CB(k_2)$ with at least one non-pebbled endpoint cannot have been formed by "**T**'s" C-queries. Therefore chain reactions bring in at most $2q^2$ increment to $|E_3|$. This plus the possible $q$ 3-queries from $\overline{D}$ yields $|E_3| \le 3q^2$.

By the above, **T** completes at most $|KSet_2|\cdot|E_3| \le 3q^3$ chains. As a consequence, for $i = 1, 2, 4$, the number of $i$-queries created by **T** is at most $3q^3$, which plus the $q$ $i$-queries from $\overline{D}$ yields the bounds $|E_2|, |E_4|, |E_1| \le 4q^3$. These further yield that **T** makes at most $|E_1| \cdot |E_4| \le 16q^6$ *distinct* calls to $\widetilde{C}$.CHECK. Finally, we exhibit a bijection between the chains completed by **T** and the C-queries $((k_1, k_2), u, y)$ appeared in the interaction, thus the number of such C-queries is at most $3q^3$.

**Bounding the Abort Probability for T.** Besides aborting due to adaptations (cf. the layer-2 PROCESSTREE procedures, page 11), **T** may also abort due to finding more than one pebbled leaves when traversing in a live tree, cf. subsection 4.3. To bound the probabilities, we incorporate a number of checks in $G_2$, which catch "bad events" and may cause abort at the earliest possible stage. Roughly speaking, right after **E** (the explicit randomness source of **T**) or $\widetilde{C}$ gives a new random value $z$, if $z$ has appeared at some place in the history, then $G_2$ aborts. We call $G_2$'s abortion due to these conditions *early-abortions.*

Since we have bounded the size of the history, it's simply a matter of accounting that the probability of early-abortions is at most $178q^6/2^n$ in total. The checks for early-abortions are sufficient conditions for the execution to maintain the "desired" features, i.e. if early-abortions do not occur, then the properties of the defined graph $B_3$ and $CB(k_2)$ (for each $k_2$) are as wished. More clearly, connected components in $B_3$ and $CB(k_2)$ are directed trees, and each "dangled" live tree has at most one pebbled leaf. By this, **T** never aborts due to finding more than one pebbled leaves during tree-traversing.

For the impossibility of abortion due to adaptations, since the PROCESSTREE procedures perform the chain reactions in a recursive manner, we use a recursive-style argument. To this end, for a layer-2 PROCESSTREE-call, if the involved structure possesses certain nice properties, then we call it *safe.* For example, for a call to PROCESSDUALTREE$(k_1, k_2, x, y, 4)$, if the involved structure is indeed a "dual-tree" as depicted in Fig. 10 (left), then the call is safe.

Conditioned on the absence of early-abortion, we first show that when handling queries from $D$, **T**'s calls to layer-2 PROCESSTREE procedures are safe. Then, for any safe-call, we prove:

(i) the adaptation in this call would not cause abort;
(ii) safeness is preserved during recursion: all the sub-calls to layer-2 PROCESSTREE procedures made in this call are safe.

Thus by induction, all calls to layer-2 PROCESSTREE procedures are safe. This implies adaptations never cause $G_2$ abort, and we have $Pr[\overline{D}^{G_2}$ aborts$] = Pr[$early-abortion$] \le 178q^6/2^n$.

**Transitions Between Systems.** To prove $G_1$ and $G_2$ indistinguishable as well as transit the results on **T** to **S**, we take the idea of [CHK+14]: essentially, $\overline{D}^{G_1}$ and $\overline{D}^{G_2}$ only deviate due to CHECK-calls returning different answers. Since there are at most $16q^6$ *distinct* CHECK-calls in non-aborting $G_2$ executions, the probability that $\overline{D}^{G_1}$ and $\overline{D}^{G_2}$ deviate is at most $Pr[G_2 \text{ aborts}] + Pr[\text{“}BadCheck''] \le 178q^6/2^n + 2 \cdot 16q^6/2^n \le 210q^6/2^n$. This also shows that with probability at least $1 - 210q^6/2^n$, the query and time complexity of **S** in $G_1$ are $8q^4$ and $O(q^7)$ respectively.

To prove $G_2$ and $G_3$ indistinguishable we take the randomness mapping argument [CHK+14]. Roughly speaking, for any $\widetilde{C}$ and **E** such that $\overline{D}^{G_2(\widetilde{C},\mathbf{T}^{\widetilde{C},\mathbf{E}})}$ does not abort, we are able to find some tuples $\mathbf{E}'$ such that $\overline{D}^{G_3(\mathsf{CC}_4^{\mathbf{E}'},\mathbf{E}')} = \overline{D}^{G_2(\widetilde{C},\mathbf{T}^{\widetilde{C},\mathbf{E}})}$. Here $\widetilde{C}$'s answers to the (at most $3q^3$) C-queries in $\overline{D}^{G_2(\widetilde{C},\mathbf{T}^{\widetilde{C},\mathbf{E}})}$ brings in an additional statistical distance of $\epsilon \le (3q^3)^2/2^n \le 9q^6/2^n$ between $G_2$ and $G_3$.

By all the results above, we use an argument to link $G_1$, $G_2$, and $G_3$ together to bound $\overline{D}$'s advantage in distinguishing $G_1$ and $G_3$: $Pr[\overline{D}^{G_1} = 1] - Pr[\overline{D}^{G_3} = 1] \le Pr[\text{“}BadCheck''] + Pr[\overline{D}^{G_2} \text{ aborts}] + \epsilon \le \frac{219q^6}{2^n}$.

## Acknowledgements

We are grateful to the anonymous referees of Eurocrypt 2016 and CRYPTO 2016, for their useful comments and corrections. Specially:

– the bottom-up style presentation of the simulator was suggested by a referee at Eurocrypt 2016;
– the focus of Appendix A was suggested by another referee at Eurocrypt 2016, cf. that section for details.

We also thank Yu Yu for the help during the rebuttal phase of Eurocrypt 2016.

This work is partially supported by National Key Basic Research Project of China (2011CB302400), National Science Foundation of China (61379139), (11526215), and the "Strategic Priority Research Program" of the Chinese Academy of Sciences, Grant No. XDA06010701.

## References

AB81. Charles A Asmuth and GR Blakley. An efficient algorithm for constructing a cryptosystem which is harder to break than two other cryptosystems. *Computers & Mathematics with Applications*, 7(6):447–450, 1981.

ABD+13. Elena Andreeva, Andrey Bogdanov, Yevgeniy Dodis, Bart Mennink, and John P. Steinberger. On the Indifferentiability of Key-Alternating Ciphers. Cryptology ePrint Archive, Report 2013/061, 2013. Extended abstract appeared at CRYPTO 2013.

ABDCV98. W. Aiello, M. Bellare, G. Di Crescenzo, and R. Venkatesan. Security Amplification by Composition: The Case of Doubly-iterated, Ideal Ciphers. In Hugo Krawczyk, editor, *Advances in Cryptology – CRYPTO '98*, volume 1462 of *Lecture Notes in Computer Science*, pages 390–407. Springer Berlin Heidelberg, 1998.

Bih94. Eli Biham. New Types of Cryptanalytic Attacks Using Related Keys. *Journal of Cryptology*, 7(4):229–246, 1994.

Bih98. Eli Biham. Cryptanalysis of Multiple Modes of Operation. *Journal of Cryptology*, 11(1):45–58, 1998.

Bla06. John Black. The Ideal-Cipher Model, Revisited: An Uninstantiable Blockcipher-Based Hash Function. In Matthew Robshaw, editor, *Fast Software Encryption*, volume 4047 of *Lecture Notes in Computer Science*, pages 328–340. Springer Berlin Heidelberg, 2006.

BR06. Mihir Bellare and Phillip Rogaway. The Security of Triple Encryption and a Framework for Code-Based Game-Playing Proofs. In Serge Vaudenay, editor, *Advances in Cryptology – EUROCRYPT 2006*, volume 4004 of *Lecture Notes in Computer Science*, pages 409–426. Springer Berlin Heidelberg, 2006.

BW99. Alex Biryukov and David Wagner. Slide Attacks. In Lars Knudsen, editor, *Fast Software Encryption*, volume 1636 of *Lecture Notes in Computer Science*, pages 245–259. Springer Berlin Heidelberg, 1999.

CDMP05. Jean-Sébastien Coron, Yevgeniy Dodis, Cécile Malinaud, and Prashant Puniya. Merkle-Damgård Revisited: How to Construct a Hash Function. In Victor Shoup, editor, *Advances in Cryptology – CRYPTO 2005*, volume 3621 of *Lecture Notes in Computer Science*, pages 430–448. Springer Berlin Heidelberg, 2005.

CDMS10. Jean-Sébastien Coron, Yevgeniy Dodis, Avradip Mandal, and Yannick Seurin. A Domain Extender for the Ideal Cipher. In Daniele Micciancio, editor, *Theory of Cryptography*, volume 5978 of *Lecture Notes in Computer Science*, pages 273–289. Springer Berlin Heidelberg, 2010.

CGH04. Ran Canetti, Oded Goldreich, and Shai Halevi. The Random Oracle Methodology, Revisited. *J. ACM*, 51(4):557–594, July 2004.

CHK+14. Jean-Sébastien Coron, Thomas Holenstein, Robin Künzler, Jacques Patarin, Yannick Seurin, and Stefano Tessaro. How to Build an Ideal Cipher: The Indifferentiability of the Feistel Construction. *Journal of Cryptology*, pages 1–54, 2014.

CS15. Benoît Cogliati and Yannick Seurin. On the Provable Security of the Iterated Even-Mansour Cipher Against Related-Key and Chosen-Key Attacks. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology – EUROCRYPT 2015*, volume 9056 of *Lecture Notes in Computer Science*, pages 584–613. Springer Berlin Heidelberg, 2015.

DGHM13. Grégory Demay, Peter Gaži, Martin Hirt, and Ueli Maurer. Resource-Restricted Indifferentiability. In Thomas Johansson and PhongQ. Nguyen, editors, *Advances in Cryptology – EUROCRYPT 2013*, volume 7881 of *Lecture Notes in Computer Science*, pages 664–683. Springer Berlin Heidelberg, 2013.

DLMS14. Yuanxi Dai, Jooyoung Lee, Bart Mennink, and John Steinberger. The Security of Multiple Encryption in the Ideal Cipher Model. In JuanA. Garay and Rosario Gennaro, editors, *Advances in Cryptology – CRYPTO 2014*, volume 8616 of *Lecture Notes in Computer Science*, pages 20–38. Springer Berlin Heidelberg, 2014.

DS15a. Yuanxi Dai and John Steinberger. Indifferentiability of 10-Round Feistel Networks. Cryptology ePrint Archive, Report 2015/874, 2015. http://eprint.iacr.org/.

DS15b. Yuanxi Dai and John Steinberger. Indifferentiability of 8-Round Feistel Networks. Cryptology ePrint Archive, Report 2015/1069, 2015. Appeared at CRYPTO 2016.

DSKT16. Dana Dachman-Soled, Jonathan Katz, and Aishwarya Thiruvengadam. 10-Round Feistel is Indifferentiable from an Ideal Cipher. In Marc Fischlin and Jean-Sébastien Coron, editors, *Advances in Cryptology – EUROCRYPT 2016*, volume 9666 of *Lecture Notes in Computer Science*, pages 649–678. Springer Berlin Heidelberg, 2016.

DSSL16. Yevgeniy Dodis, Martijn Stam, John Steinberger, and Tianren Liu. Indifferentiability of Confusion-Diffusion Networks. In Marc Fischlin and Jean-Sébastien Coron, editors, *Advances in Cryptology – EUROCRYPT 2016*, volume 9666 of *Lecture Notes in Computer Science*, pages 679–704. Springer Berlin Heidelberg, 2016.

EG83. Shimon Even and Oded Goldreich. On the Power of Cascade Ciphers. In David Chaum, editor, *Advances in Cryptology: Proceedings of Crypto '83*, Lecture Notes in Computer Science, pages 43–50. Springer Berlin Heidelberg, 1983.

EM97. Shimon Even and Yishay Mansour. A Construction of a Cipher From a Single Pseudorandom Permutation. *Journal of Cryptology*, 10(3):151–161, 1997.

Gaz13. Peter Gazi. Plain versus Randomized Cascading-Based Key-Length Extension for Block Ciphers. In Ran Canetti and JuanA. Garay, editors, *Advances in Cryptology – CRYPTO 2013*, volume 8042 of *Lecture Notes in Computer Science*, pages 551–570. Springer Berlin Heidelberg, 2013.

GL15a. Chun Guo and Dongdai Lin. A Synthetic Indifferentiability Analysis of Interleaved Double-Key Even-Mansour Ciphers. In Tetsu Iwata and JungHee Cheon, editors, *Advances in Cryptology – ASIACRYPT 2015*, volume 9453 of *Lecture Notes in Computer Science*, pages 389–410. Springer Berlin Heidelberg, 2015.

GL15b. Chun Guo and Dongdai Lin. On the Indifferentiability of Key-Alternating Feistel Ciphers with No Key Derivation. In Yevgeniy Dodis and JesperBuus Nielsen, editors, *Theory of Cryptography*, volume 9014 of *Lecture Notes in Computer Science*, pages 110–133. Springer Berlin Heidelberg, 2015. full version: http://eprint.iacr.org/.

GLS+15. Peter Gaži, Jooyoung Lee, Yannick Seurin, John Steinberger, and Stefano Tessaro. Relaxing Full-Codebook Security: A Refined Analysis of Key-Length Extension Schemes. In Gregor Leander, editor, *Fast Software Encryption*, volume 9054 of *Lecture Notes in Computer Science*, pages 319–341. Springer Berlin Heidelberg, 2015.

GM09. Peter Gaži and Ueli Maurer. Cascade Encryption Revisited. In Mitsuru Matsui, editor, *Advances in Cryptology – ASIACRYPT 2009*, volume 5912 of *Lecture Notes in Computer Science*, pages 37–51. Springer Berlin Heidelberg, 2009.

GT12. Peter Gaži and Stefano Tessaro. Efficient and Optimally Secure Key-Length Extension for Block Ciphers via Randomized Cascading. In David Pointcheval and Thomas Johansson, editors, *Advances in Cryptology – EUROCRYPT 2012*, volume 7237 of *Lecture Notes in Computer Science*, pages 63–80. Springer Berlin Heidelberg, 2012.

HT16. Viet Tung Hoang and Stefano Tessaro. Key-alternating Ciphers and Key-length Extension: Exact Bounds and Multi-user Security. Cryptology ePrint Archive, Report 2016/578, 2016. Extended abstract to appear at CRYPTO 2016.

KR01. Joe Kilian and Phillip Rogaway. How to protect des against exhaustive key search (an analysis of desx). *Journal of Cryptology*, 14(1):17–35, 2001.

Lee13. Jooyoung Lee. Towards Key-Length Extension with Optimal Security: Cascade Encryption and Xor-cascade Encryption. In Thomas Johansson and PhongQ. Nguyen, editors, *Advances in Cryptology – EUROCRYPT 2013*, volume 7881 of *Lecture Notes in Computer Science*, pages 405–425. Springer Berlin Heidelberg, 2013.

LR86. M Luby and C Rackoff. Pseudo-random Permutation Generators and Cryptographic Composition. In *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing*, STOC '86, pages 356–363, New York, NY, USA, 1986. ACM.

LS13. Rodolphe Lampe and Yannick Seurin. How to Construct an Ideal Cipher from a Small Set of Public Permutations. In Kazue Sako and Palash Sarkar, editors, *Advances in Cryptology – ASIACRYPT 2013*, volume 8269 of *Lecture Notes in Computer Science*, pages 444–463. Springer Berlin Heidelberg, 2013.

MM93. UeliM. Maurer and JamesL. Massey. Cascade ciphers: The importance of being first. *Journal of Cryptology*, 6(1):55–61, 1993.

MPR07. Ueli Maurer, Krzysztof Pietrzak, and Renato Renner. Indistinguishability Amplification. In Alfred Menezes, editor, *Advances in Cryptology – CRYPTO 2007*, volume 4622 of *Lecture Notes in Computer Science*, pages 130–149. Springer Berlin Heidelberg, 2007.

MRH04. Ueli Maurer, Renato Renner, and Clemens Holenstein. Indifferentiability, Impossibility Results on Reductions, and Applications to the Random Oracle Methodology. In Moni Naor, editor, *Theory of Cryptography*, volume 2951 of *Lecture Notes in Computer Science*, pages 21–39. Springer Berlin Heidelberg, 2004.

MT09. Ueli Maurer and Stefano Tessaro. Computational Indistinguishability Amplification: Tight Product Theorems for System Composition. In Shai Halevi, editor, *Advances in Cryptology – CRYPTO 2009*, volume 9215 of *Lecture Notes in Computer Science*, pages 355–373. Springer Berlin Heidelberg, 2009.

MT10. Ueli Maurer and Stefano Tessaro. A Hardcore Lemma for Computational Indistinguishability: Security Amplification for Arbitrarily Weak PRGs with Optimal Stretch. In Daniele Micciancio, editor, *Theory of Cryptography*, volume 5978 of *Lecture Notes in Computer Science*, pages 237–254. Springer Berlin Heidelberg, 2010.

Mye03. Myers. Efficient amplification of the security of weak pseudo-random function generators. *Journal of Cryptology*, 16(1):1–24, 2003.

RSS11. Thomas Ristenpart, Hovav Shacham, and Thomas Shrimpton. Careful with Composition: Limitations of the Indifferentiability Framework. In KennethG. Paterson, editor, *Advances in Cryptology – EUROCRYPT 2011*, volume 6632 of *Lecture Notes in Computer Science*, pages 487–506. Springer Berlin Heidelberg, 2011.

Sha49. Claude E Shannon. Communication Theory of Secrecy Systems. *Bell system technical journal*, 28(4):656–715, 1949.

Tes11. Stefano Tessaro. Security Amplification for the Cascade of Arbitrarily Weak PRPs: Tight Bounds via the Interactive Hardcore Lemma. In Yuval Ishai, editor, *Theory of Cryptography*, volume 6597 of *Lecture Notes in Computer Science*, pages 37–54. Springer Berlin Heidelberg, 2011.

## A Attacks on 3-Cascade with Stronger Key Schedules

During the first round review at Eurocrypt 2016, one of the referees suggested considering 3-cascade with keys $(k_1, k_2, k_1 \oplus k_2)$ as a possible future work. This slightly more sophisticated key schedule nicely blocks the attack against $\mathsf{CC}_3$ (subsection 3.2). To make a distinction, we denote this scheme by $\mathsf{CC}_3[k_1, k_2, k_1 \oplus k_2]$. We did not find any distinguisher either, until 11 April, 2016. The distinguisher $D$ is as follows:

(1) Randomly chooses four keys $k_1$, $k_1'$, $k_2$, $k_2'$, and $n$-bit value $v$;
(2) Queries $\mathrm{E1}^{-1}(k_1, v) \to u_1$ and $\mathrm{E1}^{-1}(k_1', v) \to u_2$;
(3) Makes eight C-queries:

$$u_1 \xrightarrow{\mathrm{C}(k_1,k_2)} \xrightarrow{\mathrm{C}^{-1}(k_1 \oplus k_2 \oplus k_2',k_2')} \xrightarrow{\mathrm{C}(k_1 \oplus k_2 \oplus k_2',k_2)} \xrightarrow{\mathrm{C}^{-1}(k_1,k_2')} u_1',$$

and

$$u_1' \xrightarrow{\mathrm{C}(k_1',k_2)} \xrightarrow{\mathrm{C}^{-1}(k_1' \oplus k_2 \oplus k_2',k_2')} \xrightarrow{\mathrm{C}(k_1 \oplus k_2 \oplus k_2',k_2')} \xrightarrow{\mathrm{C}^{-1}(k_1',k_2')} u_2';$$

(4) If $\mathrm{E1}(k_1, u_1') = \mathrm{E1}(k_1', u_2')$ then outputs 1, else outputs 0.

When interacting with $\mathsf{CC}_3[k_1, k_2, k_1 \oplus k_2]$, $D$ always outputs 1. The involved structure is depicted in Fig. 11; the queries that are really made by $D$ are drawn in black, while the others are drawn in gray. On the other hand, no simulator would be able to extract the involved $k_2$ and $k_2'$ from the interaction, thus failed.

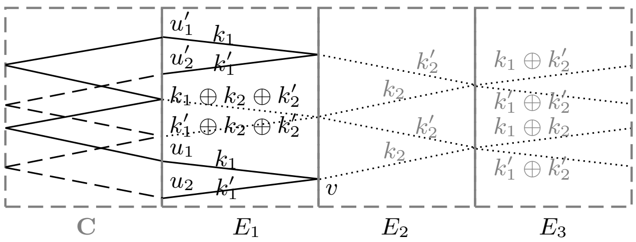


**Fig. 11.** Related-key Boomerang Structure in $\mathsf{CC}_3[k_1, k_2, k_1 \oplus k_2]$.

The key schedule could be generalized to a more general form: denote by $K$ the $2\kappa$-bit main key and $(k_1, k_2, k_3)$ the three round-key, then there exist three efficiently computable $2\kappa$-bit permutations $(\pi_1, \pi_2, \pi_3)$ such that:

– $(k_1, k_2) = \pi_1(K)$, $(k_2, k_3) = \pi_2(K)$, $(k_3, k_1) = \pi_3(K)$.

However, the above distinguisher could also be generalized to cover such schemes:

(1) Randomly chooses eight pair-wise distinct round-keys $k_2^1, k_2^2, \ldots, k_2^8$;
(2) Randomly chooses 2 distinct round-keys $k_1^1$ and $\overline{k_1^1}$ and $n$-bit value $v$;
(3) Queries $\mathrm{E1}^{-1}(k_1^1, v) \to u_1$ and $\mathrm{E1}^{-1}(\overline{k_1^1}, v) \to \overline{u_1}$;
(4) For $i = 1, 2, 3, 4$, repeats the following operations:
   (a) Computes $K_i \leftarrow \pi_1^{-1}(k_1^i, k_2^i)$, $\overline{K_i} \leftarrow \pi_1^{-1}(\overline{k_1^i}, k_2^i)$, $(k_3^i, k_1^i) \leftarrow \pi_3(K_i)$, and $(\overline{k_3^i}, \overline{k_1^i}) \leftarrow \pi_3(\overline{K_i})$;
   (b) Computes $K_i' \leftarrow \pi_3^{-1}(k_2^{i+1}, k_3^i)$, $\overline{K_i'} \leftarrow \pi_3^{-1}(k_2^{i+1}, \overline{k_3^i})$, $(k_1^{i+1}, k_2^{i+1}) \leftarrow \pi_3(K_i)$, and $(\overline{k_1^{i+1}}, k_2^{i+1}) \leftarrow \pi_3(\overline{K_i})$;
   (c) Queries $u_i \xrightarrow{\mathrm{C}(K_i)} \xrightarrow{\mathrm{C}^{-1}(K_i')} u_{i+1}$ and $\overline{u_i} \xrightarrow{\mathrm{C}(\overline{K_i})} \xrightarrow{\mathrm{C}^{-1}(\overline{K_i'})} \overline{u_{i+1}}$;
(5) Note that $u_5$ and $\overline{u_5}$ have been obtained. If $\mathrm{E1}^{-1}(\overline{k_1^5}, \mathrm{E1}(k_1^5, u_5)) = \overline{u_5}$ then outputs 1, and 0 otherwise.

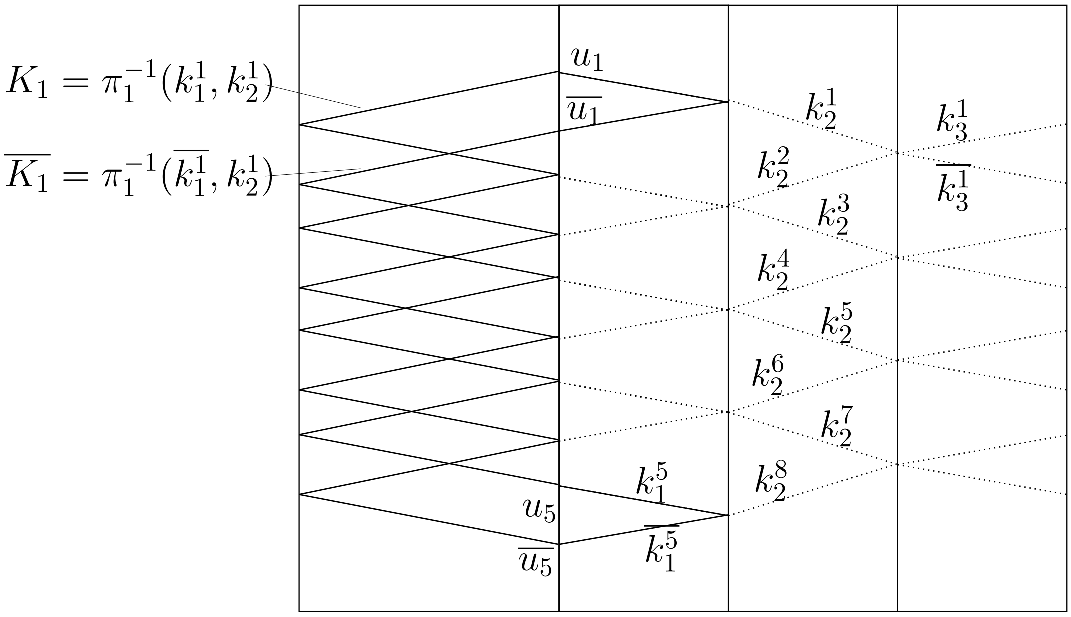


**Fig. 12.** Related-key Boomerang Structure in the general 3-cascade.

The involved structure is depicted in Fig. 12. As it is quite complicated, most of the key-labels are omitted; just take it as a coarse illustration. It can be seen from the figure that $D$ always outputs 1 when interacting with the variant of 3-cascade. On the other hand, for any $S^*$ that tries to simulate the three ciphers, the number of $\kappa$-bit keys $S^*$ can extracted from the above interaction is *four* (say, $k_1^1$, $\overline{k_1^1}$, $k_1^5$, and $\overline{k_1^5}$). But to figure out the value $\overline{u_5}$ $S^*$ has to recover the *eight* unknown $k_2$ values. This is clearly impossible.

As far as we know, the latter distinguisher seems to have no known analogue in the literature. On the other hand, as the general form covers a very large range of key schedules that could be imagined, it now seems very hard to dig out a positive yet non-trivial result on 3-cascade.

## B The Pseudocode

In the following two subsubsections, we deliver some additional implementation issues that is omitted in the the overview in subsection 4.1-4.4. The code is in the last subsubsection.

**Implementation Issues for $G_1$.**

Explicit Randomness of **S**. As mentioned in subsection 4.1, the randomness of the simulator **T** is made explicit via the 4 ideal ciphers **E**. Since **S** is modified from **T**, **S** also takes **E** as the randomness source, i.e. whenever **S** needs to assign a random answer to **S**.E$i(k,z)$, it queries **E** to get $z' := \mathbf{E}.\mathrm{E}i(k,z)$. As already mentioned, using such explicit randomness is indeed equivalent to lazily sampling at the beginning of the experiment.

**S** Maintaining History. To keep the query-records, **S** maintains a set *Queries*, storing tuples of the form $(i,k,z,z',dir,num)$. The first five coordinates are as described in subsection 4.1. While the additional sixth coordinate *num* is the value of a query counter *qnum* (initialized to 1 at the beginning of the interaction) when this record is created. The last coordinate – *num* – or last two coordinates – *dir* and *num* – are often omitted, when they are not of interest to the discussion at hand. We recall that whenever **S** newly simulates a query $(i,k,z,z')$ (as the result of either answering $D$'s query or **S**'s inner actions), we say it *creates a new $i$-query*, and a record as described is added to *Queries*.

We write $E_i$ for the set $\{(k,z,z') : \exists dir, num \text{ s.t. } (i,k,z,z',dir,num) \in Queries\}$, and denote by $ET$ the tuple of sets $(E_1,E_2,E_3,E_4)$. **S** would try to ensure the sets $E_1,\ldots,E_4$ to define four partial ciphers, i.e. for each tuple $(i,k,z)$, there is *at most one* $z'$ such that $(k,z,z') \in E_i$, and vice versa. Clearly, all the $i$-queries with $dir = \leftarrow$ or $\rightarrow$ are consistent with the randomness source $\mathbf{E}$, and indeed always define a partial cipher. However, since there exist *adapted* queries which are seldom consistent with $\mathbf{E}$, the situation may be broken in the following two cases:

(i) When $\mathbf{S}$ obtains a new random value $z$ from $\mathbf{E}$, $z$ may collide with a value of an adapted query. For example, when $\mathbf{S}$ obtains $w := \mathbf{E}.\mathrm{E2}(k_2, v)$ and tries to create $(2, k_2, v, w, \rightarrow)$, there may already exists a record $(2, k_2, v', w, \perp)$ for some $v' \neq v$;

(ii) When $\mathbf{S}$ tries to create an adapted query, it may contradict earlier-created queries. This possibility is also mentioned in the description of the layer-2 ProcessTree procedures, cf. page 11.

In these cases, we let $\mathbf{S}$ *abort*. This mechanism ensures $E_1, \ldots, E_4$ to define four partial ciphers, and we thus write $E_i[k]$ and $E_i[k]^{-1}$ for the sets $\{z : \exists z' \text{ s.t. } (k, z, z') \in E_i\}$ and $\{z' : \exists z \text{ s.t. } (k, z, z') \in E_i\}$ respectively, and write $E_i[k](z)$ ($E_i[k]^{-1}(z')$, resp.) for the (unique) corresponding $z'$ ($z$, resp.) when $z \in E_i[k]$ ($z' \in E_i[k]^{-1}$, resp.). Note that by the above mechanism, $\mathbf{S}$ *never overwrites anything*. Moreover, the sets $E_i$, $E_i[k]$, and $E_i[k]^{-1}$ changes as new records are added to *Queries*.

More Variables of $\mathbf{S}$. We let $\mathbf{S}$ maintain some additional sets. First is a set *Completed* which is used to keep a record of all the paths $\mathbf{S}$ has completed. The entries in *Completed* are 4-tuples $(k_1, k_2, v, 2)$ and $(k_1, k_2, x, 4)$, which keep the associated keys and the intermediate value $v$ and $x$ in the path respectively. Second, by the strategy, $\mathbf{S}$ would frequently check conditions of the form "$\exists k$ s.t. $z \in E_i[k]/E_i[k]^{-1}$" for some $z$. To simplify the code, we let $\mathbf{S}$ maintain 8 sets $\{LS\} = \{LS_1, LS_2, LS_3, LS_4\}$ and $\{RS\} = \{RS_1, RS_2, RS_3, RS_4\}$ to keep such values. More clearly, for $i \in \{1, 2, 3, 4\}$, $LS_i$ keeps all $z$ satisfying $\exists k \in \{0, 1\}^\kappa$ s.t. $z \in E_i[k]$, while $RS_i$ keeps all $z$ meeting $\exists k \in \{0, 1\}^\kappa$ s.t. $z \in E_i[k]^{-1}$. The abbreviation "LS" stands for *Left Shore* while "RS" stands for *Right Shore*. Finally, the two sets $KSet_1$ and $KSet_2$ mentioned in the termination argument (page 16) are explicitly maintained by $\mathbf{S}$.

**Implementation Issues for $G_2$.** All the aforementioned sets of $\mathbf{S}$ (i.e. *Queries*, $KSet_1$, $KSet_2$, *Completed*, $\{LS\}$, and $\{RS\}$) also appear in $\mathbf{T}$, and are similarly maintained. Besides, some additional issues are as follows.

Implementing $\widetilde{C}$. Since the imagined cipher $\widetilde{C}$ in $G_2$ is somewhat non-standard, we explicitly implement it. We also make the randomness of the implemented $\widetilde{C}$ explicit, via a normal ideal $(2\kappa, n)$-cipher $\mathbf{C}$. (We note that in the main body, page 17, we take $(\widetilde{C}, \mathbf{E})$ as the randomness source of $G_2$. Since we now replace the imagined $\widetilde{C}$ by our implemented $\widetilde{C}^{\mathbf{C}}$, the randomness source of $G_2$. changes from $(\widetilde{C}, \mathbf{E})$ to $(\mathbf{C}, \mathbf{E})$, and the formal proof for the transition from $G_2$ to $G_3$ would operate on $(\mathbf{C}, \mathbf{E})$, cf. subsection D.)

In detail, we let $\widetilde{C}^{\mathbf{C}}$ maintain a set *CQueries* to keep its query history. Once receiving a query $\mathrm{C}(K, u)$, $\widetilde{C}^{\mathbf{C}}$ takes $y := \mathbf{C}.\mathrm{C}(K, u)$ as the answer and adds a record $((k_1, k_2), u, y, \rightarrow, qnum)$ via a procedure AddCQuery – here *CQueries* and $\mathbf{T}$.*Queries* share the same counter *qnum*, which is made global in $G_2$; and upon a query $\mathrm{C}^{-1}(K, y)$, $\widetilde{C}^{\mathbf{C}}$ answers with $u := \mathbf{C}.\mathrm{C}^{-1}(K, y)$ and adds a record $((k_1, k_2), u, y, \leftarrow, qnum)$ via AddCQuery. As all the tuples in *CQueries* are consistent with $\mathbf{C}$, *CQueries* always defines a partial blockcipher. The set *CQueries* and $\mathbf{T}$'s set *Queries* are used for the *explicitly bookkeeping* mechanism, which is lifted from [ABD+13] to simplify the arguments.

To simplify the language we use notations similar to *Queries*: we write $CTable[K]$ for the set

$$\{u : \exists y, dir, num \text{ s.t. } (K, u, y, dir, num) \in CQueries\},$$

and $CTable[K](u)$ for the corresponding $y$. Similarly for $CTable[K]^{-1}$ and $CTable[K]^{-1}(y)$.

Global Variables. In order to allow the two set-maintaining procedures AddQuery and AddCQuery to check "bad randomness" before creating new query-records (the mentioned *early-abortion* mechanism, cf. page 17), $\mathbf{T}$'s eight sets $LS_1, \ldots, LS_4, RS_1, \ldots, RS_4$ are made *global* (thus accessible to $\widetilde{C}$'s procedure AddCQuery). Furthermore, two additional (also global) sets $LS_0$ and $RS_0$ are used in $G_2$: $RS_0$ keeps all the values $u$ satisfying $\exists K \in \{0, 1\}^{2\kappa}$ and $y : (K, u, y) \in CQueries$, while $LS_0$ keeps all the values $y$ satisfying $\exists K \in \{0, 1\}^{2\kappa}$ and $u : (K, u, y) \in CQueries$. As $G_2$ is merely an intermediate system, the existence of such "global" variables is not problematic.

**The Code.** The first part of the code implements the simulated system $G_1$ along with the simulator $\mathbf{T}$ from $G_2$. When a line has a boxed variant next to it, $\mathbf{S}$ uses the original code, whereas $\mathbf{T}$ uses the boxed one. Additionally, the underlined red sentences only exist in $\mathbf{T}$.

**Simulated System** $G_1(\mathbf{C}, \mathbf{S}^{\mathbf{C}})$ // No need to implement the ideal cipher $\mathbf{C}$.

**Simulator** $\mathbf{S}^{\mathbf{C},\mathbf{E}}$: $\boxed{\textbf{Simulator } \mathbf{T}^{\widetilde{C}^{\mathbf{C}},\mathbf{E}}:}$

**Variables**

Sets $Queries$, $Completed$, $KSet_1$, $KSet_2$, $\{LS\} = \{LS_1, LS_2, LS_3, LS_4\}$, and $\{RS\} = \{RS_1, RS_2, RS_3, RS_4\}$; all initially empty

Integer $qnum$, initialized to 1

// Create a query using a random value from $\mathbf{E}$. The term "random assign" is from [LS13].

**private procedure** RandomAssign$(i, \delta, k, z)$
- **if** $\delta = +$ **then**
  - $z' := \mathbf{E}.\mathrm{E}i(k, z)$
  - **if** $z' \in E_i[k]^{-1}$ **then abort**
  - AddQuery$(i, k, z, z', \rightarrow)$
- **else**
  - $z' := \mathbf{E}.\mathrm{E}i^{-1}(k, z)$
  - **if** $z' \in E_i[k]$ **then abort**
  - AddQuery$(i, k, z', z, \leftarrow)$

// Create the record of a query. Also capture the early-abort conditions around new E-queries.

**private procedure** AddQuery$(i, k, z, z', dir)$
- **if** $dir = \rightarrow \wedge z' \in (RS_i \cup LS_{i+1 \bmod 5})$ **then abort** // Early-abortion in $G_2$.
- **else if** $dir = \leftarrow \wedge z \in (LS_i \cup RS_{i-1})$ **then abort** // Early-abortion in $G_2$.
- $Queries := Queries \cup \{(i, k, z, z', dir, qnum)\}$
- $qnum := qnum + 1$
- $LS_i := LS_i \cup \{z\}$
- $RS_i := RS_i \cup \{z'\}$
- **if** $i \in \{1, 3\}$ **then** $KSet_1 := KSet_1 \cup \{k\}$
- **else** $KSet_2 := KSet_2 \cup \{k\}$ // $i \in \{2, 4\}$

// The interfaces for 1- and 3-queries: simply randomly assign an answer.

**public procedure** E1$(k_1, u)$
- **if** $u \notin E_1[k_1]$ **then**
  - RandomAssign$(1, +, k_1, u)$
- **return** $E_1[k_1](u)$

**public procedure** E1$^{-1}(k_1, v)$
- **if** $v \notin E_1[k_1]^{-1}$ **then**
  - RandomAssign$(1, -, k_1, v)$
- **return** $E_1[k_1]^{-1}(v)$

**public procedure** E3$(k_1, w)$
- **if** $w \notin E_3[k_1]$ **then**
  - RandomAssign$(3, +, k_1, w)$
- **return** $E_3[k_1](w)$

**public procedure** E3$^{-1}(k_1, x)$
- **if** $x \notin E_3[k_1]^{-1}$ **then**
  - RandomAssign$(3, -, k_1, x)$
- **return** $E_3[k_1]^{-1}(x)$

// Interfaces for 2-queries: E2 considers live trees in $CB(k_2)$, while E2$^{-1}$ considers live trees in $B_3$.

**public procedure** E2$(k_2, v)$
- **if** $v \in E_2[k_2]$ **then return** $E_2[k_2](v)$
- **if** $v \notin RS_1$ **then** // Lazy sampling
  - RandomAssign$(2, +, k_2, v)$
  - **return** $E_2[k_2](v)$
- $OriginSet :=$ FindPebLeafCB$(k_2, v, right)$
- // Traverse in $Li(k_2, v)$
- **if** $OriginSet = \emptyset$ **then**
  - ProcessNonPebCBTree$(k_2, v, right)$
- **else if** $|OriginSet| > 1$ **then abort**
- **else**
  - $(k_1^\circ, y^\circ, v^\circ, pos) := OriginSet$
  - ProcessPebCBTree$(k_1^\circ, k_2, y^\circ, v^\circ, pos)$
- **return** $E_2[k_2](v)$

**public procedure** E2$^{-1}(k_2, w)$
- **if** $w \in E_2[k_2]^{-1}$ **then return** $E_2[k_2]^{-1}(w)$
- **if** $w \notin LS_3$ **then** // Lazy sampling
  - RandomAssign$(2, -, k_2, w)$
  - **return** $E_2[k_2]^{-1}(w)$
- $OriginSet :=$ FindPebLeafB3$(k_2, w, left)$
- // Traverse in $Li(k_2, w)$
- **if** $OriginSet = \emptyset$ **then**
  - ProcessNonPebB3Tree$(k_2, w, left)$
- **else if** $|OriginSet| > 1$ **then abort**
- **else**
  - $(k_1^\circ, w^\circ, x^\circ, pos) := OriginSet$
  - ProcessPebB3Tree$(k_1^\circ, k_2, w^\circ, x^\circ, pos)$
- **return** $E_2[k_2]^{-1}(w)$

// Interfaces for 4-queries.

**public procedure** E4$(k_2, x)$
- **if** $x \in E_4[k_2]$ **then return** $E_4[k_2](x)$
- **if** $x \notin RS_3$ **then**
  - // Lazy sampling
  - RandomAssign$(4, +, k_2, x)$
  - **return** $E_4[k_2](x)$
- $OriginSet :=$ FindPebLeafB3$(k_2, x, right)$
- // Traverse in $Li(k_2, x)$
- **if** $OriginSet = \emptyset$ **then**
  - ProcessNonPebB3Tree$(k_2, x, right)$
- **else if** $|OriginSet| > 1$ **then abort**
- **else**
  - $(k_1^\circ, w^\circ, x^\circ, pos) := OriginSet$
  - ProcessPebB3Tree$(k_1^\circ, k_2, w^\circ, x^\circ, pos)$
- **return** $E_4[k_2](x)$

**public procedure** E4$^{-1}(k_2, y)$

**if** $y \in E_4[k_2]^{-1}$ **then return** $E_4[k_2]^{-1}(y)$
**if** $\textsc{Exists14Tripwire}(k_2, y) = \textbf{false}$ **then**
// Lazy sampling
$\textsc{RandomAssign}(4, -, k_2, y)$
**return** $E_4[k_2]^{-1}(y)$
$OriginSet := \textsc{FindPebLeafCB}(k_2, y, left)$
// Traverse in $Li(k_2, y)$
**if** $OriginSet = \emptyset$ **then**
$\textsc{ProcessNonPebCBTree}(k_2, y, left)$
**else if** $|OriginSet| > 1$ **then abort**
**else**
$(k_1^\circ, y^\circ, v^\circ, pos) := OriginSet$
$\textsc{ProcessPebCBTree}(k_1^\circ, k_2, y^\circ, v^\circ, pos)$
**return** $E_4[k_2]^{-1}(y)$

**private procedure** $\textsc{Exists14Tripwire}(k_2, y)$
**forall** $k_1 \in KSet_1$ **do**
**if** $\textsc{FindEdgeInCB}(k_1, k_2, y) \neq \perp$ **then return true**
**return false**

**private procedure** $\textsc{FindEdgeInCB}(k_1, k_2, y)$
**forall** $u \in E_1[k_1]$ **do**
**if** $\textsc{Check}((k_1, k_2), u, y) = \textbf{true}$ $\boxed{\textbf{if } \widetilde{C}.\textsc{Check}((k_1, k_2), u, y) = \textbf{true}}$
**then return** $E_1[k_1](u)$
**return** $\perp$

**public procedure** $\textsc{Check}((k_1, k_2), u, y)$ // This procedure only exists in $G_1$
**return** $\mathbf{C}.\mathrm{C}((k_1, k_2), u) = y$
// The procedures on the trees: around trees in $B_3$
**private procedure** $\textsc{FindPebLeafB3}(k_2, z, pos)$ // Width-first traversal.
$OriginSet := \emptyset$
$SearchQueue.\textsc{Enqueue}(\perp, z, pos)$
**while** $SearchQueue \neq \emptyset$ **do**
$(past, z, pos) := SearchQueue.\textsc{Pop}()$
**forall** $k_1 \in KSet_1 \setminus \{past\}$ **do**
**if** $pos = right$ **then**
$x := z$
**if** $x \notin E_3[k_1]^{-1}$ **then continue**
$w := E_3[k_1]^{-1}(x)$
// If $w$ is pebbled, then stop going deeper from $x$.
**if** $w \in E_2[k_2]^{-1}$ **then**
$OriginSet := OriginSet \cup \{(k_1, w, x, right)\}$
**else**
$SearchQueue.\textsc{Enqueue}(k_1, w, left)$
**else** // $pos = left$
$w := z$
**if** $w \notin E_3[k_1]$ **then continue**
$x := E_3[k_1](w)$
**if** $x \in E_4[k_2]$ **then**
$OriginSet := OriginSet \cup \{(k_1, w, x, left)\}$
**else**
$SearchQueue.\textsc{Enqueue}(k_1, x, right)$
**return** $OriginSet$

**private procedure** $\textsc{ProcessNonPebB3Tree}(k_2, z, pos)$
**if** $pos = left$ **then**
$w := z$
$\textsc{RandomAssign}(2, -, k_2, w)$
**forall** $k_1 \in KSet_1$ **do**
**if** $w \notin E_3[k_1]$ **then continue**
$x := E_3[k_1](w)$
$\textsc{ProcessB3SubTree}(k_1, k_2, w, x, right)$
**else** // $pos = right$
$x := z$
$\textsc{RandomAssign}(4, +, k_2, x)$
**forall** $k_1 \in KSet_1$ **do**
**if** $x \notin E_3[k_1]^{-1}$ **then continue**
$w := E_3[k_1]^{-1}(x)$
$\textsc{ProcessB3SubTree}(k_1, k_2, w, x, left)$

The following procedure slightly deviates from the description presented in subsection 4.3: once the simulator detects a "dual-tree" (cf. page 13) to be processed, it first makes a call to FindPebLeafCB to perform the traversal in the live tree in $CB(k_2)$ before calling ProcessDualTree. The information returned by FindPebLeafCB is ignored. Indeed, this FindPebLeafCB-call can be eliminated. But note that a FindPebLeafCB-call may bring in new C-queries to the history, thus modifying the involved live tree. Therefore, we take this "traverse-in-advance" strategy, with the aim of bringing forward such possible modifications and making the subsequent process (in this case) the same as that in some other cases (cases where a new query $\mathrm{E4}^{-1}(k_2, y)$ or $\mathrm{E2}(k_2, v)$ appears and ProcessDualTree is subsequently called). This strategy helps simplify some arguments.

```
private procedure ProcessPebB3Tree(k_1, k_2, w, x, pos)
  if pos = left then
    y := E_4[k_2](x)
    Traversed := false
    v := FindEdgeInCB(k_1, k_2, y)
    if v = ⊥ then
      ProcessB3SubTree(k_1, k_2, w, x, left)
      return
    forall k'_1 ∈ KSet_1 \ {k_1} do
      if Traversed = true then continue
      if v ∉ E_1[k'_1]^{-1} then continue
      FindPebLeafCB(k_2, v, right)
      Traversed := true
    ProcessDualTree(k_1, k_2, v, w, 2)
  else // pos = right
    v := E_2[k_2]^{-1}(w)
    if v ∉ E_1[k_1]^{-1} then
      ProcessB3SubTree(k_1, k_2, w, x, right)
      return
    u := E_1[k_1]^{-1}(v)
    y := C.C((k_1, k_2), u)   [ y := \tilde{C}.C((k_1, k_2), u) ]
    Traversed := false
    forall k'_1 ∈ KSet_1 \ {k_1} do
      if Traversed = true then continue
      v' := FindEdgeInCB(k'_1, k_2, y)
      if v' = ⊥ then continue
      FindPebLeafCB(k_2, y, left)
      Traversed := true
    ProcessDualTree(k_1, k_2, x, y, 4)
```

```
// The procedures on the trees: around trees in CB(k_2)
private procedure FindPebLeafCB(k_2, z, pos)
  if (pos = left ∧ z ∈ E_4[k_2]^{-1}) ∨ (pos = right ∧ z ∈ E_2[k_2]) then return ∅
  OriginSet := ∅
  SearchQueue.Enqueue(⊥, z, pos)
  while SearchQueue ≠ ∅ do
    (past, z, pos) := SearchQueue.Pop()
    forall k_1 ∈ KSet_1 \ {past} do
      if pos = right then
        v := z
        if v ∉ E_1[k_1]^{-1} then continue
        u := E_1[k_1]^{-1}(v)
        y := C.C((k_1, k_2), u)   [ y := \tilde{C}.C((k_1, k_2), u) ]
        if y ∈ E_4[k_2]^{-1} then
          OriginSet := OriginSet ∪ {(k_1, y, v, right)}
        else
          SearchQueue.Enqueue(k_1, y, left)
      else // pos = left
        y := z
        v := FindEdgeInCB(k_1, k_2, y)
        if v = ⊥ then continue
        if v ∈ E_2[k_2] then
          OriginSet := OriginSet ∪ {(k_1, y, v, left)}
        else
          SearchQueue.Enqueue(k_1, v, right)
  return OriginSet
```

```
private procedure ProcessNonPebCBTree(k_2, z, pos)
  if pos = left then
    y := z
    RandomAssign(4, −, k_2, y)
    forall k_1 ∈ KSet_1 do
      v := FindEdgeInCB(k_1, k_2, y)
      if v = ⊥ then continue
      ProcessCBSubTree(k_1, k_2, y, v, right)
  else // pos = right
    v := z
    RandomAssign(2, +, k_2, v)
    forall k_1 ∈ KSet_1 do
      if v ∉ E_1[k_1]^{-1} then continue
      u := E_1[k_1]^{-1}(v)
      y := C.C((k_1, k_2), u)   [ y := \tilde{C}.C((k_1, k_2), u) ]
      ProcessCBSubTree(k_1, k_2, y, v, left)
```

```
private procedure ProcessPebCBTree(k_1, k_2, y, v, pos)
  if pos = left then
    w := E_2[k_2](v)
    if w ∈ E_3[k_1] then
      ProcessDualTree(k_1, k_2, E_3[k_1](w), y, 4)
    else
      ProcessCBSubTree(k_1, k_2, y, v, pos)
  else // pos = right
    x := E_4[k_2]^{-1}(y)
    if x ∈ E_3[k_1]^{-1} then
      ProcessDualTree(k_1, k_2, v, E_3[k_1]^{-1}(x), 2)
    else
      ProcessCBSubTree(k_1, k_2, y, v, pos)
```

// "Layer-2" ProcessTree procedures (cf. page 13)

According to the overview in page 11, ProcessB3SubTree only makes calls to ProcessB3SubTree. However, in the following code, it seems like that ProcessB3SubTree may calls ProcessCBSubTree and even ProcessDualTree (through the sub-call to RecurseNew2 and RecurseNew4). We remark that here the purpose of letting ProcessB3SubTree call layer-2 ProcessTree procedures indirectly (through RecurseNew2 and RecurseNew4) is to simply the implementation as well as increase modularity. Indeed, we will prove that only calls to ProcessB3SubTree can be made inside such sub-call to RecurseNew2 and RecurseNew4 (cf. the proof of lemma 7). Similarly remarked for the code of ProcessCBSubTree.

**private procedure** ProcessB3SubTree$(k_1, k_2, w, x, pos)$
  **if** $pos = left$ **then** // A $(3,4)$-tripwire.
    $y := E_4[k_2](x)$
    $u := \mathbf{C}.\mathrm{C}^{-1}((k_1,k_2),y)$ $\boxed{u := \widetilde{C}.\mathrm{C}^{-1}((k_1,k_2),y)}$
    $v := \mathrm{E1}(k_1,u)$
    Adapt$(2,k_2,v,w)$
    $Completed := Completed \cup \{(k_1,k_2,v,2)\}$
    $Completed := Completed \cup \{(k_1,k_2,x,4)\}$
    RecurseNew2$(k_1,k_2,v,w)$
  **else** // $pos = right$; a $(3,2)$-tripwire.
    $v := E_2^{-}[k_2](w)$
    $u := \mathrm{E1}^{-1}(k_1,v)$
    $y := \mathbf{C}.\mathrm{C}((k_1,k_2),u)$ $\boxed{y := \widetilde{C}.\mathrm{C}((k_1,k_2),u)}$
    Adapt$(4,k_2,x,y)$
    $Completed := Completed \cup \{(k_1,k_2,v,2)\}$
    $Completed := Completed \cup \{(k_1,k_2,x,4)\}$
    RecurseNew4$(k_1,k_2,x,y)$

**private procedure** ProcessCBSubTree$(k_1,k_2,y,v,pos)$
  **if** $pos = left$ **then** // A $(1,2)$-tripwire.
    $w := E_2[k_2](v)$
    $x := \mathrm{E3}(k_1,w)$
    Adapt$(4,k_2,x,y)$
    $Completed := Completed \cup \{(k_1,k_2,v,2)\}$
    $Completed := Completed \cup \{(k_1,k_2,x,4)\}$
    RecurseNew4$(k_1,k_2,x,y)$
  **else** // $pos = right$; a $(1,4)$-tripwire.
    $x := E_4[k_2]^{-1}(y)$
    $w := \mathrm{E3}^{-1}(k_1,x)$
    Adapt$(2,k_2,v,w)$
    $Completed := Completed \cup \{(k_1,k_2,v,2)\}$
    $Completed := Completed \cup \{(k_1,k_2,x,4)\}$
    RecurseNew2$(k_1,k_2,v,w)$

**private procedure** ProcessDualTree$(k_1,k_2,z,z',i)$
  Adapt$(i,k_2,z,z')$
  **if** $i = 2$ **then**
    $Completed := Completed \cup \{(k_1,k_2,z,2),(k_1,k_2,E_3[k_1](z'),4)\}$
    RecurseNew2$(k_1,k_2,z,z')$
  **else** // $i = 4$
    $Completed := Completed \cup \{(k_1,k_2,E_2[k_2]^{-1}(E_3[k_1]^{-1}(z)),2),(k_1,k_2,z,4)\}$
    RecurseNew4$(k_1,k_2,z,z')$

// Deals with the tripwires newly set off by a new 2-query $(2,k_2,v,w)$
**private procedure** RecurseNew2$(k_1,k_2,v,w)$
  **forall** $k_1' \in KSet_1 \setminus \{k_1\}$ **do**
    **if** $v \in E_1[k_1']^{-1} \wedge w \in E_3[k_1']$ **then**
      $u' := E_1[k_1']^{-1}(v)$
      $y' := \mathbf{C}.\mathrm{C}((k_1',k_2),u')$ $\boxed{y' := \widetilde{C}.\mathrm{C}((k_1',k_2),u')}$
      ProcessDualTree$(k_1',k_2,E_3[k_1'](w),y',4)$
    **else if** $v \in E_1[k_1']^{-1} \wedge w \notin E_3[k_1']$ **then**
      $u' := E_1[k_1']^{-1}(v)$
      $y' := \mathbf{C}.\mathrm{C}((k_1',k_2),u')$ $\boxed{y' := \widetilde{C}.\mathrm{C}((k_1',k_2),u')}$
      ProcessCBSubTree$(k_1',k_2,y',v,left)$
    **else if** $v \notin E_1[k_1']^{-1} \wedge w \in E_3[k_1']$ **then**
      ProcessB3SubTree$(k_1',k_2,w,E_3[k_1'](w),right)$

// Deals with the tripwires newly set off by a new 4-query $(4,k_2,x,y)$
**private procedure** RecurseNew4$(k_1,k_2,x,y)$
  **forall** $k_1' \in KSet_1 \setminus \{k_1\}$ **do**
    $v' :=$ FindEdgeInCB$(k_1',k_2,y)$
    **if** $x \in E_3[k_1']^{-1} \wedge v' \neq \perp$ **then**
      ProcessDualTree$(k_1',k_2,v',E_3[k_1']^{-1}(x),2)$
    **else if** $x \in E_3[k_1']^{-1} \wedge v' = \perp$ **then**
      ProcessB3SubTree$(k_1',k_2,E_3[k_1']^{-1}(x),x,left)$
    **else if** $x \notin E_3[k_1']^{-1} \wedge v' \neq \perp$ **then**
      ProcessCBSubTree$(k_1',k_2,y,v',right)$

**private procedure** $\text{ADAPT}(i, k_2, z, z')$
  **if** $z \in E_i[k_2] \vee z' \in E_i[k_2]^{-1}$ **then abort**
  $\text{ADDQUERY}(i, k_2, z, z', \perp)$

The second part of the code implements the intermediate system $G_2$. Note that the involved simulator $\mathbf{T}$ has been implemented in the code above.

**Intermediate System** $G_2(\widetilde{C}^{\mathbf{C}}, \mathbf{T}^{\widetilde{C}^{\mathbf{C}}})$
**Global Variables**
  Sets $LS_0$, $RS_0$, and $CQueries$; all initially empty
**Enhanced Ideal Cipher** $\widetilde{C}^{\mathbf{C}}$:
// Capture the early-abort conditions around new C-queries.
**private procedure** $\text{ADDCQUERY}((k_1, k_2), u, y, dir)$
  **if** $dir = \rightarrow \wedge y \in (LS_0 \cup RS_4)$ **then abort** // Early-abortion
  **else if** $u \in (RS_0 \cup LS_1)$ **then abort** // Early-abortion; $dir = \leftarrow$
  $CQueries := CQueries \cup \{((k_1, k_2), u, y, dir, qnum)\}$
  $qnum := qnum + 1$
  $RS_0 := RS_0 \cup \{u\}$
  $LS_0 := LS_0 \cup \{y\}$

**public procedure** $\text{C}((k_1, k_2), u)$
  **if** $u \notin CTable[(k_1, k_2)]$
    $y := \mathbf{C}.\text{C}((k_1, k_2), u)$
    $\text{ADDCQUERY}((k_1, k_2), u, y, \rightarrow)$
  **return** $CTable[(k_1, k_2)](u)$

**public procedure** $\text{C}^{-1}((k_1, k_2), y)$
  **if** $y \notin CTable[(k_1, k_2)]^{-1}$
    $u := \mathbf{C}.\text{C}^{-1}((k_1, k_2), y)$
    $\text{ADDCQUERY}((k_1, k_2), u, y, \leftarrow)$
  **return** $CTable[(k_1, k_2)]^{-1}(y)$

**public procedure** $\text{CHECK}((k_1, k_2), u, y)$
  **return** $CTable[(k_1, k_2)](u) = y$

## C Focus on $G_2$: Non-abortion and Termination Arguments

As mentioned in subsection 4.5, the indifferentiability can be reduced to results on $G_2$. This subsection gives the desired results, including non-abortion and termination arguments. For ease of discussion, we borrow the terminology *simulator cycle* from [ABD+13], which refers to the execution period from the point $\overline{D}$ makes a query till the point $\overline{D}$ receives the answer – or $G_2$ aborts.

INVARIANTS. Due to the early abort conditions incorporated by us (cf. page 17), the desired properties in *Queries* and *CQueries* are ensured for *any* point in *any* $G_2$ execution. To give a formal presentation, we first give a useful lemma, stating that each tuple in *Completed* corresponds to a completed path.

**Lemma 1.** *At any point in a $G_2$ execution, for any $(k_1, k_2, x, 4) \in Completed$, there exist $u, v, w, y \in \{0,1\}^n$ such that the following five queries have been in Queries and CQueries respectively:*

$$((k_1, k_2), u, y), (1, k_1, u, v), (2, k_2, v, w), (3, k_1, w, x), (4, k_2, x, y).$$

*Similar claim holds for any $(k_1, k_2, v, 2) \in Completed$.*

*Proof.* By inspection of the code, it can be seen that right before any tuple $(k_1, k_2, x, 4)$ is to be added to *Completed*, there is a corresponding call to ADAPT. The claim thereby holds for any $(k_1, k_2, x, 4)$ right after $(k_1, k_2, x, 4) \in Completed$ holds. As nothing can be overwritten, the claim keeps holding since then. Similarly for $(k_1, k_2, v, 2)$. □

We then present several invariants, which are somewhat similar to [ABD+13].

*Inv0.* 1-queries and 3-queries have $dir \in \{\leftarrow, \rightarrow\}$.

*Inv1. (About two E-queries to two consecutive cascade stages)* For $num > num'$, there does not exist two queries $(i, k, z, z', \rightarrow, num)$ and $(i{+}1, k', z', z'', dir, num')$ (in *Queries*); there does not exist two queries $(i{+}1, k', z', z'', \leftarrow, num)$ and $(i, k, z, z', dir, num')$ either.

*Inv2. (About two E-queries to the same cascade stage)* For $num > num'$, there does not exist two queries $(i, k, z, z', \rightarrow, num)$ and $(i, k', z'', z', dir, num')$; there does not exist two queries $(i, k, z', z, \leftarrow, num)$ and $(i, k', z', z'', dir, num')$ either.

*Inv3. (About two C-queries)* For $num > num'$, there does not exist two C-queries $(K, u, y, \rightarrow, num)$ and $(K', u', y, dir, num')$; there does not exist two C-queries $(K, u, y, \leftarrow, num)$ and $(K', u, y', dir, num')$.

*Inv4. (About a C-query and a 1/4-query)* New 1/4-queries and C-queries cannot hit each other:

– There does not exist a C-query $((k_1, k_2), u, y, dir_C, num_C)$ and a 1-query $(1, k_1, u, v, dir_1, num_1)$ such that either: (i) $dir_1 =\leftarrow$ and $num_1 > num_C$, or (ii) $dir_C =\leftarrow$ and $num_C > num_1$;
– There does not exist a C-query $((k_1, k_2), u, y, dir_C, num_C)$ and a 4-query $(4, k_2, x, y, dir_4, num_4)$ such that either: (i) $dir_4 =\rightarrow$ and $num_4 > num_C$, or (ii) $dir_C =\rightarrow$ and $num_C > num_4$.

*Inv5. (The tripwires function well)* In each of the following cases, the two involved queries are part of the same $(k, k')$-completed path, and the 4-tuples corresponding to the $(k, k')$-completed path are in *Completed*:

(i) There are two queries $(j, k, z', z'', dir, num)$ and $(i, k', z, z', dir', num')$ such that $(i, j) \in \{(1, 2), (3, 4)\}$ and $num > num'$;
(ii) There are two queries $(2, k_2, v, w, dir, num)$ and $(3, k_1, w, x, dir', num')$ such that $num > num'$;
(iii) There are two queries $(4, k_2, x, y, dir, num)$ and $(1, k_1, u, v, dir', num')$ such that both $num > num'$ and $\widetilde{C}.\textsc{Check}((k_1, k_2), u, y) = \textbf{true}$.

**Lemma 2.** *Invariants Inv0-Inv4 hold throughout any $G_2$ execution. Invariant Inv5 holds at the end of each simulator cycle as long as $G_2$ does not abort.*

*Proof.* Inv0 is obvious (which is indeed among the core ideas of the simulation strategy). Inv1 to Inv4 are ensured by the early abort conditions inside AddQuery and AddCQuery. More clearly, for Inv1, in each case, the value $z'$ must have been in $LS_{i+1}$ or $RS_i$ before AddQuery is called on the $num$-th query. By this, AddQuery would abort and not add the $num$-th query to *Queries*. For Inv2 we *wlog* consider two queries $(i, k, z, z', \rightarrow, num)$ and $(i, k', z'', z', dir, num')$. After $(i, k', z'', z', dir, num')$ is created, $z'$ must be added to $RS_i$, so that the later query $(i, k, z, z', \rightarrow, num)$ would cause AddQuery abort and would not be added to *Queries*. The proof of Inv3 and Inv4 follows the same line as Inv1 and Inv2 (differently: in some cases, the queries contradicting them would cause AddCQuery abort).

To show Inv5, we consider all possibilities of the creation of two queries meeting the constraints, and prove the claim for each of them.

Consider two queries $(1, k_1, u, v, dir', num')$ and $(2, k_2, v, w, dir, num)$ (i.e. $(i, j) = (1, 2)$) first. Since $num > num'$, it cannot be $dir =\leftarrow$ as otherwise contradicting Inv1. Hence $dir \in \{\rightarrow, \perp\}$. By construction, such a 2-query may be created due to the following possibilities:

(i) a call to RandomAssign$(2, +, k_2, v)$ in E2$(k_2, v)$;
(ii) a call to RandomAssign$(2, +, k_2, v)$ in ProcessNonPebB3Tree;
(iii) a call to Adapt$(2, k_2, v, w)$ in a call to layer-2 ProcessTree procedure.

However, possibility (i) is not possible: for a call E2$(k_2, v)$ to call RandomAssign, it has to be $v \notin RS_1$ before the E2-call, which contradicts the pre-existence of $(1, k_1, u, v)$. So we consider the remaining two possibilities:

– for possibility (ii): by construction, after ProcessNonPebCBTree creates the 2-query $(2, k_2, v, w, \rightarrow)$, it will check the set $E_1$ and find $v \in E_1[k_1]$ and call ProcessCBSubTree$(k_1, k_2, y, v, left)$ for $y = \mathrm{C}((k_1, k_2), u)$. By inspection of the code of ProcessCBSubTree, it can be seen once this call returns without abortion, $(k_1, k_2, v, 2) \in \mathit{Completed}$ would hold. By Lemma 1, this implies $(2, k_2, v, w)$ and $(1, k_1, u, v)$ in the same completed path;
– for possibility (iii): note that each of the three layer-2 ProcessTree procedures takes the first sub-key as an argument – to make a distinction, we denote by $k_1^\circ$ the "first-key argument" to the layer-2 ProcessTree-call which creates $(2, k_2, v, w)$. Then:
  - If $k_1^\circ = k_1$ and the ProcessTree-call returns without abortion, then by a quick inspection of the code, it can be seen the claims hold regardless of the type of the call. For example, if the call is ProcessB3SubTree$(k_1, k_2, w, x, \cdot)$, then $(k_1, k_2, x, 4) \in \mathit{Completed}$ after the call, which implies $(2, k_2, v, w)$ and $(1, k_1, u, v)$ in the same completed path (by Lemma 1).

- If $k_1^\circ \neq k_1$, then after the 2-query is created, the ProcessTree-call would call RecurseNew2 and iterate for $k_1' \in KSet_1 \setminus \{k_1^\circ\}$ and make a call to a layer-2 ProcessTree procedure when it iterates with $k_1' = k_1$. The latter ProcessTree-call indeed will be made due to $v \in E_1[k_1]^{-1}$ which is implied by the existence of $(1, k_1, u, v)$; and, by the analysis above, if it returns without abortion, then the claims hold.

The arguments for $(i,j) = (3,2), (3,4)$ follow the same line as the above. As to $(i,j) = (1,4)$, consider two queries $(4, k_2, x, y, dir, num)$ and $(1, k_1, u, v, dir', num')$ such that $\widetilde{C}.\textsc{Check}((k_1,k_2), u, y) = \textbf{true}$. The return value of Check implies the existence of the C-query $((k_1,k_2), u, y, dir_C, num_C)$. It cannot be $num_C > num$, as otherwise $num_C > num > num'$ and the creation of the C-query would contradict Inv4. Hence $num > num_C$ and the 4-query is the latest among the three, and $dir' \in \{\leftarrow, \perp\}$ due to Inv4. By construction, such a 4-query can only be created due to ProcessNonPebCBTree or layer-2 ProcessTree-calls (again, the query cannot be created due to $\mathrm{E4}^{-1}$, as the pre-existence of the 1- and C-query implies Exists14Tripwire$(k_2, y)$ returning **true**). Similarly to the case of $(i,j) = (1,2)$, in each of the cases, the tuples are in *Completed* and the path exists after the simulator cycle is finished. □

**The Bipartite Graphs $B_3$, $CB(k_2)$, and the Graph $B(k_2)$.** This subsubsection presents formal definitions for the graphs used in the proof, as well as discussions on their structural properties. More precisely, for each $k_2 \in \{0,1\}^\kappa$, we use an edge-labeled graph $B(k_2)$ to encode the information from *Queries* and *CQueries* relevant to 2-, 4-, and C-queries associated with $k_2$; and the two bipartite graphs $B_3$ and $CB(k_2)$ are two subgraphs in $B(k_2)$. Both $B_3$ and $CB(k_2)$ have shores $\{0,1\}^n$; note that $B_3$ is independent from $k_2$ as it is built from 3-queries. Further note that $B(k_2)$, $B_3$, and $CB(k_2)$ are all *time-dependent*.

We describe $B_3$ first. Edges of $B_3$ are directed and labeled, and constructed as follows: for every 3-query $(3, k_1, w, x, dir, num) \in Queries$, we construct an edge $(w, x)$ of label $k_1$, of direction $dir$ ($dir \in \{\rightarrow, \leftarrow\}$ by Inv0), and of an associated $num$ value equaling the $num$ value of the 3-query. This constitutes all edges of $B_3$. For convenience, we will use the 3-query $(3, k_1, w, x, dir, num)$ to refer to the corresponding edge. Due to Inv2, two distinct 3-queries cannot give rise to two edges of $B_3$ with the same endpoints, and hence $B_3$ contains no multiple edges.

We then describe $CB(k_2)$. For any $k_2 \in \{0,1\}^\kappa$, $CB(k_2)$ is constructed as follows: for every C-query $((k_1,k_2), u, y, dir_C, num_C) \in CQueries$ and every 1-query $(1, k_1, u, v, dir_1, num_1) \in Queries$ (the two queries must share the same $k_1$ and $u$), we construct an edge $(y, v)$ of label $k_1$, direction $dir_{(y,v)}$, and $num$ value $num_{(y,v)}$. The associated $num_{(y,v)}$ and $dir_{(y,v)}$ equal the corresponding parameters of the later query (say, if $num_C > num_1$, then $num_{(y,v)} = num_C$ and $dir_{(y,v)} = dir_C$, and vice versa). For convenience, we will use the 5-tuple $(k_1, y, v, dir_{(y,v)}, num_{(y,v)})$ to refer to such an edge (often abbreviated to $(k_1, y, v)$). This constitutes all edges of $CB(k_2)$. Note that for any $k_2$, each 1-query $(1, k_1, u, v)$ gives rise to at most one edge in $CB(k_2)$: because only C-queries of the form $((k_1,k_2), u, \cdot)$ are able to form edges with $(1, k_1, u, v)$, and the number of such C-queries is at most 1. Moreover, due to Inv2 and Inv3, two distinct pairs of queries cannot give rise to two edges of $CB(k_2)$ with the same endpoints, so that $CB(k_2)$ contains no multiple edges.

We note that if there is an edge $(k_1, y, v, d, num)$ with $d = \leftarrow$, then the involved C-query has to head towards $y$, i.e. of the form $((k_1,k_2), u, y, \rightarrow, n_C)$: because if not, then the 1-query $(1, k_1, u, v, d_1, n_1)$ must meet (i) $n_1 > n_C$ (due to Inv4); (ii) $d_1 = \rightarrow$ (due to $n_1 > n_C$ and Inv4), so that $d$ cannot be $\leftarrow$. Similarly, the 1-query $(1, k_1, u, v, d_1, n_1)$ involved in an edge $(k_1, y, v, d, num)$ with $d = \rightarrow$ has to meet $d_1 = \rightarrow$ as otherwise the involved C-query is later and heads towards $y$ and $d \neq \rightarrow$.

We now formally prove the acyclic property of $B_3$, which is almost the same as Lemma 12 of [ABD+13].

**Proposition 1.** *Connected components of $B_3$ are directed trees with edges directed away from the root, and the num values on the edges of any directed path in $B_3$ are strictly increasing.*

*Proof.* Due to Inv2, every vertex of $B_3$ has indegree at most 1. Moreover, since queries are totally ordered and a single query exactly raises a single edge, two adjacent edges in $B_3$ have different $num$ values. Due to Inv2, these $num$ values go from smaller to larger according to the edge directions, hence the connected component is also acyclic. These establish the claim. □

We then formally prove the acyclic property of $CB(k_2)$.

**Proposition 2.** *For any $k_2 \in \{0,1\}^\kappa$, connected components of $CB(k_2)$ are directed trees with edges directed away from the root, and the num values of the edges of any directed path in $CB(k_2)$ are strictly increasing.*

*Proof.* We first show that every vertex in $CB(k_2)$ has indegree at most 1. Consider a left-shore node $y$ first. As remarked, the C-query involved in an edge $(k_1, y, v, \leftarrow)$ has to head towards $y$. Due to Inv3, for each $y$, there exists at most one C-query that heads towards $y$. The above show that every left-shore node $y$ in $CB(k_2)$ has indegree at most 1. For right-shore node $v$ the argument follows the same line: as remarked, the 1-query involved in an edge $(k_1, y, v, \rightarrow)$ has to head towards $v$; a 1-query gives rise to at most one edge; due to Inv2, two different 1-queries cannot head towards the same $v$.

We then proceed to argue that the *num* values of two adjacent edges go from smaller to larger according to the edge directions:

(i) for two edges adjacent to the same right-shore node $v$, assume that the four involved queries are $((k_1, k_2), u, y, dir_1, num_1)$, $(1, k_1, u, v, \rightarrow, num_2)$, $(1, k_1', u', v, dir_3, num_3)$, and $((k_1', k_2), u', y', \rightarrow, num_4)$. By Inv2, it necessarily holds $num_3 > num_2$ and $dir_3 = \leftarrow$; then by Inv4, it holds $num_4 > num_3 > num_2$. Since the edge formed by $(1, k_1, u, v, \rightarrow, num_2)$ and $((k_1, k_2),$ $u, y, dir_1, num_1)$ is directed from $y$ to $v$, either $dir_1 = \leftarrow \wedge num_2 > num_1$, or $dir_1 = \rightarrow$ (which also implies $num_2 > num_1$). Therefore, the two *num* values associated to the two edges are $num_2$ and $num_4$ respectively, and $num_4 > num_2$ and the claim on the *num* values of the edges holds.

(ii) for two edges adjacent to the same left-shore node $y$, the proof is symmetrical.

By the analysis above, the component is acyclic. These establish the claim. ◻

With respect to the graph $CB(k_2)$, we have the following variant of Inv5.

**Lemma 3.** *For any $k_2 \in \{0,1\}^\kappa$, consider an edge $(k_1, y, v, dir, num)$ in $CB(k_2)$. At any moment of a $G_2$ execution such that Inv5 holds, if there exists a 2-query $(2, k_2, v, w, dir_2, num')$ (a 4-query $(4, k_4, x, y, dir_4, num')$, resp.) such that $num' > num$, then the queries of the edge $(k_1, y, v, dir, num)$ and the 2-query (4-query, resp.) are part of the same $(k_1, k_2)$-completed path.*

*Proof.* Assume that the two queries involved in the edge $(k_1, y, v, dir, num)$ are $((k_1, k_2), u, y, dir_C, num_C)$ and $(1, k_1, u, v, dir_1, num_1)$. If there exists a 2-query $(2, k_2, v, w, dir_2, num')$ satisfying all the assumptions, then it necessarily holds $num' > \mathrm{Max}\{num_C, num_1\} \geq num_1$; if there is a 4-query $(4, k_4, x, y, dir_4, num')$ satisfying all the assumptions, then: (i) $num' > \mathrm{Max}\{num_C, num_1\} \geq num_1$; (ii) due to the existence of $((k_1, k_2), u, y, dir_C, num_C)$, $\widetilde{C}.\textsc{Check}((k_1, k_2), u, y)$ returns **true**. In both cases, the claim holds by Inv5. ◻

We now present the graph $B(k_2)$ itself (for a certain $k_2$), which is basically obtained by linking the nodes in the left shore of $B_3$ and the right shore of $CB(k_2)$ with existing 2-queries $(2, k_2, v, w, dir, num)$, and linking the nodes in the left shore of $CB(k_2)$ and the right shore of $B_3$ with existing 4-queries $(4, k_2, x, y, dir, num)$. These additional edges correspond to queries labeled $k_2$; however since the entire graph $B(k_2)$ is already parameterized by $k_2$, there is no need to associate labels to these additional edges.

More precisely, for a fixed $k_2 \in \{0,1\}^\kappa$, $B(k_2)$ has four "shores" $\{0,1\}^n$ numbered 2, 3, 4, and 5:[12] a copy of $CB(k_2)$ is placed between shore 5 and shore 2, while a copy of $B_3$ is placed between shore 3 and shore 4. For $i = 2, 4$, a query $(i, k_2, z, z', dir, num)$ becomes a (possibly directed) unlabeled *$i$-edge* from node $z$ in shore $i$ to node $z'$ in shore $i + 1$, with direction consistent with *dir*.

Note that unlike the graph $B$ used in [ABD+13] (page 42), in this work, a completed path corresponds to a *cycle* in $B(k_2)$ which crosses all the shores.

The following paragraphs present the formal definitions of the other terminology borrowed from [ABD+13], say, *pebbling* and *live tree*.

Pebbling in $B(k_2)$. In $B(k_2)$, a node $z$ in shore 2, 3, 4, or 5 that is adjacent to a 2-edge or 4-edge is said to be *pebbled*.[13] For a pebbled node $z$ in shore 2 or 5 (3 or 4, resp.), we also say that $z$ is *a pebbled node of $CB(k_2)$* ($B_3$, resp.). Similarly to [ABD+13], the edge pebbling a node is necessarily unique; also, pebbling transfers upwards in a live tree, which is crucial for the proof.

**Lemma 4.** *For any $k_2 \in \{0,1\}^\kappa$, at any moment of a $G_2$ execution such that Inv5 holds, the connected components of $B_3/CB(k_2)$ are "pebbled upwards": if a node is pebbled, then its parent is also pebbled. Formally speaking,*

[12] They are numbered from 2 to 5 because they consist of inputs to $E_2$ ($v$), inputs to $E_3$ ($w$), inputs to $E_4$ ($x$) respectively, and $y \in CTable[K]^{-1}$.

[13] Note that in simulator overview (page 9), the notion *pebbling* is informally specified under certain $k_2$ for the sake of simplicity.

*(a) for a 3-query $(3, k_1, w, x, dir, num)$, if $dir = \rightarrow$ and $x$ is adjacent to a 4-query, then $w$ is adjacent to a 2-query; if $dir = \leftarrow$ and $w$ is adjacent to a 2-query, then $x$ is adjacent to a 4-query;*

*(b) for an edge $(k_1, y, v, dir, num)$ in $CB(k_2)$, if $dir = \rightarrow$ and $v$ is adjacent to a 2-query, then $y$ is adjacent to a 4-query; if $dir = \leftarrow$ and $y$ is adjacent to a 4-query, then $v$ is adjacent to a 2-query.*

*Proof.* For (a), we prove the first half of the claim, and the second half is symmetric. Assume that the involved 4-query is $(4, k_2, x, y, dir', num')$. Then it has to be $num' > num$, as otherwise contradicting Inv1; then as Inv5 holds, the two queries $(3, k_1, w, x, dir, num)$ and $(4, k_2, x, y, dir', num')$ are in the same $(k_1, k_2)$-completed path, so that $w$ is adjacent to a 2-query $(2, k_2, v, w)$.

For (b) we also prove the first half of the claim. Assume that the two queries involved in the edge are $((k_1, k_2), u, y, d_C, n_C)$ and $(1, k_1, u, v, \rightarrow, n_1)$ and the 2-query attached to $v$ is $(2, k_2, v, w, d_2, n_2)$. It necessarily be $num = n_1$. Due to Inv1, it has to be $n_2 > n_1$; hence $n_2 > num$ and (by Lemma 3) the three queries are in the same $(k_1, k_2)$-completed path, so that $y$ is adjacent to a 4-query. ◻

LIVE TREES. For $k_2 \in \{0,1\}^\kappa$, consider $B(k_2)$ and the graphs $B_3$ and $CB(k_2)$ in $B(k_2)$. For $z \in \{0,1\}^n$, denote by $B_3(z)$ ($CB(k_2, z)$, resp.) the connected component containing $z$ in $B_3$ ($CB(k_2)$, resp.). Then at any point in a $G_2$ execution and for any *non-pebbled* node $z$ of $B_3$ ($CB(k_2)$, resp.), define the *live tree anchored at $z$ in $B_3$ ($CB(k_2)$, resp.)* (denoted $Li(k_2, z)$) as the tree obtained by "dangling" $B_3(z)$ ($CB(k_2, z)$, resp.) by $z$, such that $z$ is the root, and then pruning all portions of this "dangled" tree that lie beneath a pebbled node (in $B(k_2)$). More clearly, $Li(k_2, z)$ in $B_3$ ($CB(k_2)$, resp.) is obtained from $B_3(z)$ ($CB(k_2, z)$, resp.) according to the following rules:

(1) Initially, $Li(k_2, z)$ is empty;
(2) Add $z$ into $Li(k_2, z)$ and take $z$ as the root;
(3) For any node $z' \in B_3(z)$ ($z' \in CB(k_2, z)$, resp.) ($z' \neq z$), if the path between $z'$ and $z$ does not pass through any pebbled node, then add $z'$ and the edges and nodes of the path into $Li(k_2, z)$ (if they have not been in $Li(k_2, z)$). Note that whether $z'$ is pebbled or not does not matter.

Note that this definition is the same as the *generalized live tree* in [ABD+13]. Also note that, by this definition,

– if $z$ is not adjacent to any edges, then $Li(k_2, z) = \{z\}$;
– the pebbled nodes in $Li(k_2, z)$ can only be leaves (as all the portions beneath the pebbled nodes have been pruned).

For convenience, for any *pebbled* node $z$, define $Li(k_2, z) := \{z\}$.

We then prove that at any point such that Inv5 holds, there is at most one pebbled node in a live tree $Li(k_2, z)$. As mentioned, this is the core idea behind the simulator design as well as the non-abortion argument.

**Lemma 5.** *At any point in a $G_2$ execution such that Inv5 holds (for example, the point before/after a simulator cycle) and for any $k_2 \in \{0,1\}^\kappa$, there is at most one pebbled node in a live tree $Li(k_2, z)$.*

*Proof.* If $z$ itself is pebbled then $Li(k_2, z) = z$ and the claim holds. In case of $z$ is non-pebbled, assume that there are two pebbled nodes $z'$ and $z''$ in $Li(k_2, z)$. As per the remark above, both $z'$ and $z''$ must be leaves, and the path between $z'$ and $z''$ passes through $z$ (an illustration is $z' - \ldots - z - \ldots - z''$). Then consider the "original" connected component $B_3(z)$ ($CB(k_2, z)$, resp.). As the connected component is a directed tree (Propositions 1 and 2), it can be seen that at least one of the following two paths is directed from $z$:

– the path between $z$ and $z'$;
– the path between $z$ and $z''$.

So that by Lemma 4, $z$ must be the parent of a pebbled node, and must be pebbled. This contradicts our assumption that $z$ is non-pebbled. ◻

The *size* of a live tree $Li(k_2, z)$ is defined as *the number of edges* in $Li(k_2, z)$, and is denoted by $|Li(k_2, z)|$.[14] For a tree $T$ and a node $z$ in $T$, we write $SubT(T, z)$ for the *subtree* of $T$ rooted at $z$; if $z$ is the root, then $SubT(T, z) = T$.

[14] Note that this deviates from the analogue terminology in [ABD+13, page 44]: the term "size" in [ABD+13] refers to the number of non-pebbled nodes.

Extending Live Trees. We now consider the effects of the procedure FindPebLeafCB. We start by defining *complete live trees*: at any point in a $G_2$ execution and for any $k_2 \in \{0,1\}^\kappa$, a live tree $Li$ in $CB(k_2)$ is *complete*, if for any *non-pebbled* right-shore node $v$ in $Li$ and any $k_1$, $\exists(1, k_1, u, v) \in Queries \Rightarrow \exists((k_1, k_2), u, y) \in CQueries$. In other words, for no pair $(k_1, v)$ does the following hold:

$$v \in Li \wedge v \in E_1[k_1]^{-1} \wedge E_1[k_1]^{-1}(v) \notin CTable[(k_1, k_2)].$$

The motivations behind this definition are two-fold: first, we observe that during $G_2$ processing a complete tree $Li$, the number of newly-created 3-queries is at most $|Li|$; second, a call to FindPebLeafCB$(k_2, z, pos)$ would turn the live tree $Li(k_2, z)$ to complete, if it returns without abortion. The former observation will be formally discussed in Lemmata 8 and 9, while the latter is captured by the following lemma. To simplify the presentation, we introduce a notation $T_{k_2,z}$, which denotes the snapshot of the live tree $Li(k_2, z)$ standing at the point right before the call in question. This notation will be used not only in the following lemma, but also in all the lemmata in the remaining of this subsection. Similarly to [ABD+13], this notation is used to avoid ambiguity, as the tree $Li(k_2, z)$ changes during $G_2$ processing.

**Lemma 6.** *Right after a call to* FindPebLeafCB$(k_2, z, pos)$ *returns without abortion, the following hold:*

*(a) Each C-query created in this call is a part of an edge in $Li(k_2, z)$; and all the nodes "newly added"*[15] *to $Li(k_2, z)$ are non-pebbled leaves;*
*(b) $Li(k_2, z)$ is complete;*
*(c) if $pos = left$ then $|Li(k_2, z)| \le |T_{k_2,z}| \cdot |KSet_1|$; otherwise ($pos = right$) $|Li(k_2, z)| \le Max\{|T_{k_2,z}|, 1\} \cdot |KSet_1|$;*

*Proof.* We start by reminding that in this lemma, the notation $T_{k_2,z}$ is the snapshot of $Li(k_2, z)$ before the FindPebLeafCB-call.

We then show the three statements by carefully analyzing the process of FindPebLeafCB. If $z$ is pebbled before the FindPebLeafCB-call, then it simply returns $\emptyset$, and the claims clearly hold.[16] When $z$ is non-pebbled, the discussions are divided into two cases depending on the arguments of the call:

*Case 1: the call is* FindPebLeafCB$(k_2, y, left)$. If $T_{k_2,y} = \{y\}$, then for no $(k_1, u)$ in the history does Check return **true**, and FindEdgeInCB returns $\perp$ for all $k_1$, so that FindPebLeafCB has no effect.[17] We thereby assume that the children of $y$ in $T_{k_2,y}$ are $v_1, \ldots, v_l$, and the associated edges are $(k_1^1, y, v_1), \ldots, (k_1^l, y, v_l)$. Then for $i = 1, \ldots, l$, FindPebLeafCB will find $v_i$ via FindEdgeInCB, and pushes it into either $OriginSet$ or $SearchQueue$ depending on its pebbling state:

- if $v_i$ has not been pebbled, then $(k_1^i, v_i, right)$ is pushed into $SearchQueue$;
- if $v_i$ has been pebbled, then $(k_1^i, k_2, y, v_i, left)$ is added into $OriginSet$, and FindPebLeafCB will not go deeper in $T_{k_2,y}$ from $v_i$.

After the above process around the root/level 1 node of $T_{k_2,y}$, FindPebLeafCB only adds some nodes to $SearchQueue$ or $OriginSet$, and does not modify $Li_{k_2,y}$.

Then FindPebLeafCB proceeds to pop the level 2 nodes from $SearchQueue$. Assume that the following hold for a level 2 node denoted $v_i$:

(i) the children of $v_i$ in $T_{k_2,y}$ are $y_i^1, \ldots, y_i^s$, with associated edges $(k_1^{i,1}, y_i^1, v_i), \ldots, (k_1^{i,s}, y_i^s, v_i)$;
(ii) there are $\overline{s}$ 1-queries $(1, \overline{k_1^{i,j}}, \overline{u_i^j}, v_i)$ $(j = 1, \ldots, \overline{s})$ such that $\overline{u_i^j} \notin CTable[(\overline{k_1^{i,j}}, k_2)]$. Clearly $k_1^i \ne k_1^{i,1} \ne \ldots \ne k_1^{i,s} \ne \overline{k_1^{i,1}} \ne \ldots \ne \overline{k_1^{i,\overline{s}}}$ and $s + \overline{s} + 1 \le |KSet_1|$.

Then after $v_i$ is popped, the operations inside the **forall** loop depend on the concrete conditions:

(i) When the **forall** loop iterates with $k_1 = k_1^i$, nothing happens as FindPebLeafCB finds $k_1 = past(= k_1^i)$.

[15] Say, the nodes that were not in $T_{k_2,z}$ but are in $Li(k_2, z)$. It should be reminded that by the assumption of the lemma, $Li(k_2, z)$ refers to the live tree anchored at $z$ *right after* the FindPebLeafCB-call.

[16] Indeed, FindPebLeafCB would never be called on pebbled node $z$. However, to show this requires analyzing the process of simulator cycles, which cannot be accomplished in this paragraph. We thereby add the "pebbling-check" operation at the beginning of FindPebLeafCB.

[17] Indeed, similarly to the previous footnote, FindPebLeafCB is never called in such a case by construction. But we do not have to care about this fact.

(ii) When the **forall** loop iterates with $k_1 = k_1^{i,j}$, the subsequent query to $\widetilde{C}$ does not create any new C-queries. FindPebLeafCB simply obtains the level 3 node $y_i^j$ and pushes it into either $SearchQueue$ or $OriginSet$ depending on its pebbling state (as previously pushing level 2 nodes).

(iii) When the **forall** loop iterates with $k_1 = \overline{k_1^{i,j}}$, FindPebLeafCB obtains $\overline{u_i^j}$ by accessing $E_1^{-1}$, and then queries $\widetilde{C}.\mathrm{C}((\overline{k_1^{i,j}}, k_2), \overline{u_i^j})$. As $\overline{u_i^j} \notin CTable[(\overline{k_1^{i,j}}, k_2)]$, this operation creates a new C-query $((\overline{k_1^{i,j}}, k_2), \overline{u_i^j}, \overline{y_i^j})$ (with $\overline{y_i^j} = \mathbf{C}.\mathrm{C}((\overline{k_1^{i,j}}, k_2), \overline{u_i^j})$), thus adding an edge $(\overline{k_1^{i,j}}, \overline{y_i^j}, v_i)$ into $Li_{k_2,y}$. Furthermore, as $G_2$ does not abort (by assumption), $\forall k_1' \in \{0,1\}^\kappa \backslash \{\overline{k_1^{i,j}}\}, \overline{y_i^j} \notin CTable[(k_1', k_2)]^{-1}$ immediately holds due to Inv3, so that $\overline{y_i^j}$ turns out to be a leaf of $Li_{k_2,y}$; and $\overline{y_i^j} \notin E_4[k_2]^{-1}$ immediately holds right after the C-query is created (due to Inv4), so that $\overline{y_i^j}$ is non-pebbled. As a consequence, $\overline{y_i^j}$ will later be innocuously popped, and keeps non-pebbled till the end of FindPebLeafCB.

It can be easily checked that (a) holds with respect to the above process. Furthermore, after the above process, for each level 1 node $v_i$ in $Li_{k_2,y}$:

- at most $|KSet_1|$ edges in $Li_{k_2,y}$ are adjacent to $v_i$;
- for any $k_1$, $\exists(1, k_1, u, v_i) \in Queries \Rightarrow \exists((k_1, k_2), u, y) \in CQueries$.

Later when the newly added node $y_i^j$ is popped from $SearchQueue$, the subsequent process is similar to that around the level 1 node $y$, which has been described. Subsequently, level 4 nodes in $Li_{k_2,y}$ are processed similar to the level 2 nodes as described, and have similar effects. The above are interleavingly repeated till all the nodes in $T_{k_2,y}$ are visited.

In summary, note that only in the cases captured by the above case (iii) does FindPebLeafCB add nodes into $Li_{k_2,y}$ and create new C-queries. By the analysis, such nodes are non-pebbled leaves, and such new C-queries are clearly parts of the edges in $Li(k_2, y)$; these establish (a). Moreover, $\forall k_1, \exists(1, k_1, u, v) \in Queries \Rightarrow \exists((k_1, k_2), u, y) \in CQueries$ holds for any non-pebbled right-shore node $v$ in $Li(k_2, y)$ after all of them are visited, so that (b) holds. Finally, at the end of FindPebLeafCB, each right-shore node $v$ in $Li_{k_2,y}$ is adjacent to at most $|KSet_1|$ edges. As the number of right-shore nodes is at most $|T_{k_2,y}|$ (when $pos = left$), we obtain $|Li(k_2, y)| \le |T_{k_2,y}| \cdot |KSet_1|$ and establish (c). These complete the analysis of Case 1.

*Case 2: the call is* FindPebLeafCB$(k_2, v, right)$. When $|T_{k_2,v}| \ge 1$, then it can be checked that the process has no essential difference with Case 1, and the statements hold. In particular, when $|T_{k_2,v}| \ge 1$, $T_{k_2,v}$ has at least one left-shore node, and hence the number of right-shore nodes in $T_{k_2,v}$ is at most $|T_{k_2,v}|$ and $|Li(k_2, v)| \le |T_{k_2,v}| \cdot |KSet_1| \le \mathrm{Max}\{|T_{k_2,v}|, 1\} \cdot |KSet_1|$. So we focus on the case where $T_{k_2,v} = \{v\}$. Assume that there are $l$ pre-existing 1-queries $(1, k_1^1, u_1, v), \ldots, (1, k_1^l, u_l, v)$ (note that $\forall i \in \{1, \ldots, l\}, u_i \notin CTable[(k_1^i, k_2)]$ by the assumption $T_{k_2,v} = \{v\}$). Then for $i = 1, \ldots, l$, FindPebLeafCB obtains $u_i$ by accessing $E_1^{-1}$, and then queries $\widetilde{C}$, which results in creating a new C-query $((k_1^i, k_2), u_i, y_i)$ and adding an edge $(k_1^i, y_i, v)$ to $Li_{k_2,v}$. Right after the C-query is created, $\forall k_1' \in \{0,1\}^\kappa \backslash \{k_1^i\}, y_i \notin CTable[(k_1', k_2)]^{-1}$ immediately holds due to Inv3, so that $y_i$ turns out to be a leaf of $Li_{k_2,v}$; and $y_i \notin E_4[k_2]^{-1}$ immediately holds due to Inv4, so that $y_i$ is non-pebbled. Consequently, $y_i$ will later be innocuously popped, and keeps non-pebbled till the end of FindPebLeafCB. In this case, FindPebLeafCB exactly makes $l$ C-queries and adds $l$ leaves into $Li(k_2, v)$, and (a), (b) clearly hold. On the other hand, in this case, $T_{k_2,v}$ has one right-shore nodes and $|T_{k_2,v}| = 0$, hence (c) ($|Li(k_2, v)| \le |KSet_1| \le \mathrm{Max}\{|T_{k_2,v}|, 1\} \cdot |KSet_1|$ when $pos = right$) holds. □

As a corollary of Lemma 6 (a), the number and positions of pebbled nodes in $Li_{k_2,z}$ are both invariant after $Li_{k_2,z}$ is "extended" by FindPebLeafCB.

**Inside a Simulator Cycle.** Based on the above observations, this subsubsection analyzes the process and effects of simulator cycles. Recall that a simulator cycle is triggered by a query from $\overline{D}$, and refers to the execution period between the query is made till the point the query is answered or $G_2$ aborts. The case of $\overline{D}$ making a query which has been in the history is clearly not interesting. Whereas according to the strategy (cf. page 13) and the code, in the following two cases (of $\overline{D}$ making a new query), the subsequent simulator cycle only consists querying $\mathbf{E}$ to obtain the random answer, and the corresponding set is simply enlarged by 1:

(a) $\overline{D}$ makes a new 1- or 3-query;
(b) $\overline{D}$ makes a new 2- or 4-query, but the query is not adjacent to any pre-existing queries (say, not able to set off any tripwire).

We thereby focus on the remaining cases. For clearness, we briefly summarize the hierarchy of the code/process of $G_2$ in the next paragraph before we present the (pretty long) main analysis.

Hierarchy of the Code and Procedures. Consider $\overline{D}$ issuing a new 2- or 4-query. In general, the subsequent simulator cycle proceeds as follows:

(1) checks the state around the new query by both accessing the sets and calling the traversal procedure FindPebLeafB3 or FindPebLeafCB;
(2) calls a procedure among ProcessNonPebB3Tree, ProcessPebB3Tree, ProcessNonPebCBTree, and ProcessPebCBTree – depending on the concrete state. They will thus be called *layer-1* ProcessTree procedures;
(3) makes a series of calls to the "layer-2" ProcessTree procedure(s) (cf. page 13) – ProcessB3SubTree, ProcessCBSubTree, and ProcessDualTree – recursively, depending on concrete conditions.

With the above in mind, the following paragraphs first analyze layer-2 ProcessTree procedures (as the results are the most elementary ones and are necessary for the other analyses), then analyze layer-1 ProcessTree procedures, and finally gather them to yield the results on 2- and 4-queries.

Around Layer-2 ProcessTree Procedures. The analysis starts with formally defining *safe* ProcessTree-calls. The term "safe" is due to [LS13].

**Definition 2.** *A call to* ProcessB3SubTree *is* safe *if depending on its arguments, the following hold right before the call is made:*

*(i) when the arguments to the call are $(k_1, k_2, w, x, left)$, let $y := E_4[k_2](x)$, then: (a) $x$ is the unique pebbled leaf of $Li(k_2, w)$,*[18] *and (b) there does not exist an edge $(k_1, y, v)$ in $CB(k_2)$ (cf. Fig. 13 (right));*[19]
*(ii) when the arguments to the call are $(k_1, k_2, w, x, right)$, let $v := E_2[k_2]^{-1}(w)$, then: (a) $w$ is the unique pebbled leaf of $Li(k_2, x)$, and (b) $v \notin E_1[k_1]^{-1}$ (cf. Fig. 13 (left)).*

**Definition 3.** *A call to* ProcessCBSubTree *is* safe *if depending on its arguments, the following hold right before the call is made:*

*(i) when the arguments to the call are $(k_1, k_2, y, v, left)$, let $w := E_2[k_2](v)$, then: (a) $v$ is the unique pebbled leaf of $Li(k_2, y)$, and (b) $w \notin E_3[k_1]$, and (c) $Li(k_2, y)$ is complete (cf. Fig. 14 (left));*
*(ii) when the arguments to the call are $(k_1, k_2, y, v, right)$, let $x := E_4[k_2]^{-1}(y)$, then: (a) $y$ is the unique pebbled leaf of $Li(k_2, v)$, and (b) $x \notin E_3[k_1]^{-1}$, and (c) $Li(k_2, v)$ is complete (cf. Fig. 14 (right)).*

**Definition 4.** *A call to* ProcessDualTree *is* safe *if depending on its arguments, the following hold right before the call is made:*

*(i) if the arguments to the call are $(k_1, k_2, v, w, 2)$, then: (a) there exist four queries $((k_1, k_2), u, y)$, $(1, k_1, u, v)$, $(3, k_1, w, x)$, and $(4, k_2, x, y)$, and (b) $u$ ($x$, resp.) is the unique pebbled leaf of $Li(k_2, v)$ ($Li(k_2, w)$, resp.), and (c) $Li(k_2, v)$ is complete (cf. Fig. 15 (left));*
*(ii) if the arguments to the call are $(k_1, k_2, x, y, 4)$, then: (a) there exist four queries $((k_1, k_2), u, y)$, $(1, k_1, u, v)$, $(2, k_2, v, w)$, and $(3, k_1, w, x)$, and (b) $w$ ($v$, resp.) is the unique pebbled leaf of $Li(k_2, x)$ ($Li(k_2, y)$, resp.), and (c) $Li(k_2, y)$ is complete (cf. Fig. 15 (right)).*

Recall from the previous subsubsection that the notation $T_{k_2,z}$ refers to the snapshot of the live tree $Li(k_2, z)$ standing at the point right before the call in question is made. Then, the following lemmata analyze each layer-2 ProcessTree procedure. In each case, the "interesting" influence on the sets is presented, and it is proved that $G_2$ never aborts due to adaptations. The first one deals with safe calls to ProcessB3SubTree.

**Lemma 7.** *The following hold for a safe call to* ProcessB3SubTree*:*

*(a) All the calls to layer-2* ProcessTree *procedures made in this call are safe, and $G_2$ never aborts due to calls to* Adapt *in this call;*
*(b) $|KSet_1|$, $|KSet_2|$, and $|E_3|$ stay constant after this call.*

*Proof.* We first analyze the flow of ProcessB3SubTree. For clearness, we divide the flow into two steps: first is a *chain-completion phase*, then is *a call to* RecurseNew. Depending on the arguments, there are two possibilities:

[18] Note that this implicitly requires: (i) $(3, k_1, w, x)$ pre-exists; (ii) $w$ is non-pebbled in $B(k_2)$. Similarly for those below.
[19] Formally speaking, either $y \notin CTable[(k_1, k_2)]^{-1}$ (Fig. 13 (right-upper)), or $CTable[(k_1, k_2)]^{-1}(y) \notin E_1[k_1]$ (Fig. 13 (right-lower)).

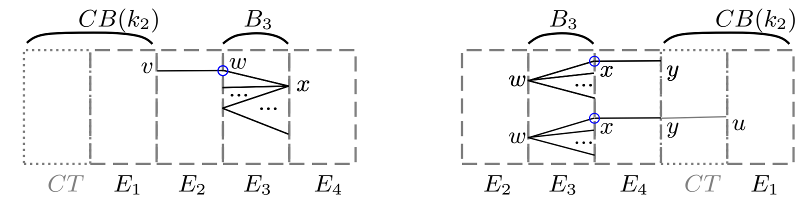


**Fig. 13.** (Left) a safe call to ProcessB3SubTree($k_1, k_2, w, x, right$); (Right) a safe call to ProcessB3SubTree($k_1, k_2, w, x, left$). The two structures in the right half identify the two possibilities. The blue circles identify the pebbled leaves; the same for the other two figures for safe layer-2 ProcessTree calls.

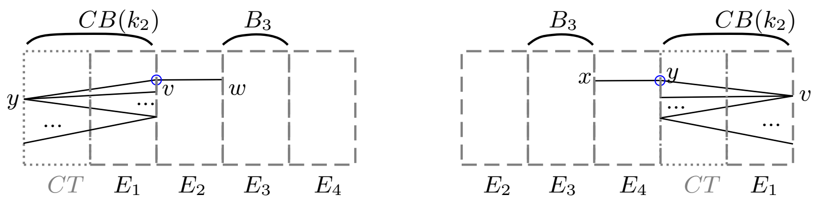


**Fig. 14.** (Left) a safe call to ProcessCBSubTree($k_1, k_2, y, v, left$); (Right) a safe call to ProcessCBSubTree($k_1, k_2, y, v, right$).

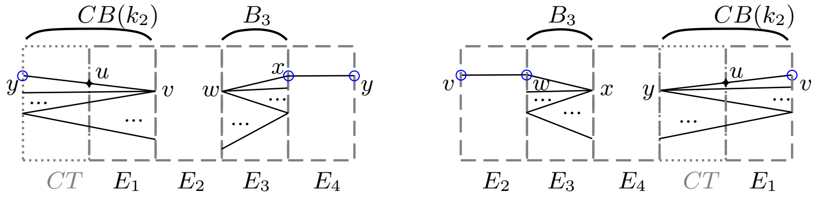


**Fig. 15.** (Left) a safe call to ProcessDualTree($k_1, k_2, v, w, 2$); (Right) a safe call to ProcessDualTree($k_1, k_2, x, y, 4$).

*Case 1: the call is* PROCESSB3SUBTREE$(k_1, k_2, w, x, left)$. Let $y := E_4[k_2](x)$. Then the chain-completion phase consists of three steps: (1) queries $\widetilde{C}$; (2) calls E1; (3) calls ADAPT – for ease of reference, this call will be called *level 1* ADAPT-call in this proof. We argue that a new 1-query $(1, k_1, u, v, \to)$ must be created during step (2). To this end, recall the definition of safe PROCESSB3SUBTREE-calls: either $y \notin CTable[(k_1, k_2)]^{-1}$ or $u \notin E_1[k_1]$ (for $u = CTable[(k_1, k_2)]^{-1}(y)$) before the call. In the latter case, $G_2$ clearly creates a new 1-query $(1, k_1, u, v, \to)$; in the former case, $G_2$ first creates a new C-query $((k_1, k_2), u, y, \leftarrow)$ via $\widetilde{C}$, then creates a 1-query $(1, k_1, u, v, \to)$. If abort does not occur, then this 1-query is indeed *new* because otherwise its adjacency to the newer query $((k_1, k_2), u, y, \leftarrow)$ would contradict Inv4. By the above, after step (1) and (2), if abort does not occur, then it holds $v \notin E_2[k_2]$ by Inv1, and the following holds by Inv2:

$$\forall k_1' \in \{0,1\}^\kappa \setminus \{k_1\}, v \notin E_1[k_1']^{-1}. \tag{1}$$

Moreover, as the PROCESSB3SUBTREE-call is assumed safe, $w$ is non-pebbled, i.e. $w \notin E_2[k_2]^{-1}$. Hence the "level 1" ADAPT-call (step (3)) does not cause abort, and creates a new adapted 2-query $(2, k_2, v, w, \perp)$.

$G_2$ then calls RECURSENEW2$(k_1, k_2, v, w)$, which would iterate for all $k_1' \in KSet_1 \setminus \{k_1\}$ and make a series of calls to layer-2 PROCESSTREE procedures. However, note that for each such $k_1'$, only if $v \in E_1[k_1']^{-1}$ could PROCESSCBSUBTREE and PROCESSDUALTREE be called. By this and (1), RECURSENEW2 only calls PROCESSB3SUBTREE. We argue that these PROCESSB3SUBTREE-calls are all safe. For this, let $x_1, \ldots, x_l$ be the non-pebbled children of $w$ in $T_{k_2,w}$,[20] and let the associated 3-queries be $(3, k_1^1, w, x_1), \ldots, (3, k_1^l, w, x_l)$. Note that it must be $k_1 \neq k_1^1 \neq \ldots \neq k_1^l$. Then for any $x_i$ among the $l$ children, $SubT(T_{k_2,w}, x_i)$ has no pebbled node: because $x$ is the unique pebbled leaf in $T_{k_2,w}$, and because the chain-completion phase only pebbles $w$. By this and the definition, right before the **forall** loop, $Li(k_2, x_i)$ consists of $SubT(T_{k_2,w}, x_i)$ and the pebbled node $w$, and has only one pebbled leaf. We then show that the state is kept till the call to PROCESSB3SUBTREE$(k_1^i, k_2, w, x_i, right)$. Note that for any $x_j$ such that the call PROCESSB3SUBTREE$(k_1^j, k_2, w, x_j, right)$ precedes the call PROCESSB3SUBTREE$(k_1^i, k_2, w, x_i, right)$ it holds $SubT(T_{k_2,w}, x_i) \cap SubT(T_{k_2,w}, x_j) = \emptyset$. Hence the processing of $Li(k_2, x_j)$ does not affect $Li(k_2, x_i)$. On the other hand, $v \notin E_1[k_1^i]^{-1}$ follows from (1) and $k_1 \neq k_1^1 \neq \ldots \neq k_1^l$, so that the call PROCESSB3SUBTREE$(k_1^i, k_2, w, x_i, right)$ is safe (if it is indeed made, say, $G_2$ does not abort until it makes this call). Thus all the $l$ PROCESSB3SUBTREE-calls are safe (the $l$ calls corresponding to $(k_1^1, k_2, w, x_1), \ldots, (k_1^l, k_2, w, x_l)$ will also be called *level 1* PROCESSB3SUBTREE-calls in this proof). These complete the analysis of Case 1.

*Case 2: the call is* PROCESSB3SUBTREE$(k_1, k_2, w, x, right)$. Let $v := E_2[k_2]^{-1}(w)$. Then the chain-completion phase has three steps (if abort does not occur):

(1) calls $\mathrm{E1}^{-1}$, which creates a new 1-query $(1, k_1, u, v, \leftarrow)$ (this query will be *new* because the PROCESSB3SUBTREE-call is assumed safe, cf. definition 2);

(2) queries $\widetilde{C}$, which creates a new C-query $((k_1, k_2), u, y, \to)$. This query will be *new* because otherwise its adjacency to the newer query $((k_1, k_2), u, y, \to)$ would contradict Inv4. Moreover, after this query is created, $y \notin E_4[k_2]^{-1}$ follows from Inv4, and the following holds due to Inv3:

$$\forall k_1' \in \{0,1\}^\kappa \setminus \{k_1\}, y \notin CTable[(k_1', k_2)]^{-1}. \tag{2}$$

(3) calls ADAPT. As the PROCESSB3SUBTREE-call is safe, $x \notin E_4[k_2]$, so that this ADAPT-call creates an adapted 4-query $(4, k_2, x, y, \perp)$ without abortion.

$G_2$ then calls RECURSENEW4$(k_1, k_2, x, y)$, which only makes a series of calls to PROCESSB3SUBTREE due to (2) (similarly to Case 1). The argument for their safeness is also similar to Case 1. In particular, assuming $x$ adjacent to $l$ 3-edges $(3, k_1^i, w_i, x)$ for $i = 1, \ldots, l$ besides $(3, k_1, w, x)$, then for each $w_i$, $x$ is the unique pebbled leaf in $Li(k_2, w_i)$ before the subsequent call to PROCESSB3SUBTREE$(k_1^i, k_2, w_i, x, left)$, and $y \notin CTable[(k_1^i, k_2)]^{-1}$ follows from (2) and $k_1 \neq k_1^1 \neq \ldots \neq k_1^l$. These complete the analysis of Case 2.

We then consider the statements. By the analysis above, if a PROCESSB3SUBTREE-call is safe, then we have:

(i) the level 1 ADAPT-call will not cause $G_2$ abort, and all the level 1 PROCESSB3SUBTREE-calls are safe. By this, (a) can be deduced by induction;

[20] The requirement "non-pebbled" excludes the node $x$.

(ii) the three variables $|KSet_1|$, $|KSet_2|$, and $|E_3|$ stay constant till the subsequent level 1 ProcessB3SubTree-calls (i.e. the three variables stay constant in the chain-completion phase). As all the subsequent ProcessB3-SubTree-calls are safe by (a), (b) is established. $\square$

The next lemma deals with safe calls to ProcessCBSubTree.

**Lemma 8.** *The following hold for a safe call to* ProcessCBSubTree$(k_1, k_2, y, v, left)$ *(or* ProcessCBSub-Tree$(k_1, k_2, y, v, right)$*, resp.):*

*(a) All the calls to layer-2* ProcessTree *procedures made in this call are safe, and $G_2$ never aborts due to calls to* Adapt *in this call;*

*(b) If this call returns without abortion, then after this call, $|KSet_1|$ and $|KSet_2|$ stay constant, while $|E_3|$ increases by $|T_{k_2,y}|$ ($|T_{k_2,v}|$, resp.). Moreover, each edge in $T_{k_2,y}$ ($T_{k_2,v}$, resp.) is a part of a completed path.*

*Proof.* Following the same line as Lemma 7, we analyze the flow first – which also consists of a *chain-completion phase* and a RecurseNew-call. Two possibilities are distinguished depending on the arguments:

*Case 1: the call is* ProcessCBSubTree$(k_1, k_2, y, v, left)$. Let $u := E_1[k_1]^{-1}(v)$ and $w := E_2[k_2](v)$. The chain-completion phase has two steps:

(1) calls E3, which creates a new 3-query $(3, k_1, w, x, \rightarrow)$ (as $w \notin E_3[k_1]$ due to the safe-ProcessCBSubTree-call assumption) and enlarges $|E_3|$ by 1. After this query is created, if abort does not occur, then $x \notin E_4[k_2]$ due to Inv1, and the following holds due to Inv2:

$$\forall k_1' \in \{0,1\}^\kappa \setminus \{k_1\}, x \notin E_3[k_1']^{-1}. \tag{3}$$

(2) calls Adapt, which creates a new adapted 4-query $(4, k_2, x, y, \perp)$ without abortion (as the ProcessCB-SubTree-call is safe, $y \notin E_4[k_2]^{-1}$).

Clearly, if abort never occurs, then the edge $(k_1, y, v)$ is a part of a completed path after this phase.

$G_2$ then calls RecurseNew4$(k_1, k_2, x, y)$, which would only make calls to ProcessCBSubTree due to (3): because ProcessB3SubTree and ProcessDualTree can only be called for $k_1' \neq k_1$ such that $x \in E_3[k_1']^{-1}$. Furthermore, these calls must be safe. To show this, let $v_1, \ldots, v_l$ be the non-pebbled children of $y$ in $T_{k_2,y}$, and let the associated edges be $(k_1^1, y, v_1), \ldots, (k_1^l, y, v_l)$ (note $k_1 \neq k_1^1 \neq \ldots \neq k_1^l$). Then for each $v_i$, the only pebbled leaf in $Li(k_2, v_i)$ before the subsequent call ProcessCBSubTree$(k_1^i, k_2, y, v_i, right)$ is $y$, and $x \notin E_3[k_1^i]^{-1}$ follows from (3) and $k_1 \neq k_1^1 \neq \ldots \neq k_1^l$. Moreover, since $T_{k_2,y}$ is complete, $Li(k_2, v_i)$ is also complete. Hence all the $l$ calls are safe. These complete the analysis of Case 1.

*Case 2: the call is* ProcessCBSubTree$(k_1, k_2, y, v, right)$. Let $x := E_4[k_2]^{-1}(y)$. The chain-completion phase has two steps: First, calls $\text{E3}^{-1}$ and creates a new 3-query $(3, k_1, w, x, \leftarrow)$ (thus enlarging $|E_3|$ by 1), after which $w \notin E_2[k_2]^{-1}$ due to Inv1 and the following holds due to Inv2 (if abort does not occur):

$$\forall k_1' \in \{0,1\}^\kappa \setminus \{k_1\}, w \notin E_3[k_1']; \tag{4}$$

Second, calls Adapt, which creates an adapted 2-query $(2, k_2, v, w, \perp)$ without abortion. Similarly to Case 1, after the chain-completion phase is completed without abortion, the edge $(k_1, y, v)$ is a part of a completed path.

$G_2$ then calls RecurseNew2$(k_1, k_2, v, w)$, which – similarly to Case 1 – would only call ProcessCBSub-Tree due to (4). To show the safeness of these calls, consider each $k_1^j$ such that $v \in E_1[k_1^j]^{-1}$ (excluding $k_1$). As $T_{k_2,v}$ is complete, let $u_j := E_1[k_1^j]^{-1}(v)$, then we have: (i) $u_j \in CTable[(k_1^j, k_2)]$, so that $y_j = CTable[(k_1^j, k_2)](u_j)$ is a child of $v$ in $T_{k_2,v}$, and the query to $\widetilde{C}$ made in RecurseNew2 would not lead to creating new C-queries; (ii) $SubT(T_{k_2,v}, y_j)$ is complete. Since $y$ is the unique pebbled node of $T_{k_2,v}$ and the chain-completion phase only pebbles $v$, $y_j$ must be non-pebbled. Gathering these and (4) yields the safeness of the calls. These complete the analysis of Case 2.

As to the statements, (a) is proved by an induction similar to Lemma 7 (a). We proceed to show (b). Assuming a non-aborting execution of ProcessCBSubTree$(k_1, k_2, v, y, left)$ and consider a ProcessCB-SubTree-call made during this process – for example, the call ProcessCBSubTree$(k_1^i, k_2, y, v_i, right)$ (cf.

the analysis of Case 1). Note that the arguments of this call indeed uniquely characterize an edge in $T_{k_2,y}$ (i.e. $(k_1^i, y, v_i)$). Moreover, it can be seen (from the code/the tree traversal algorithm and the analysis above) that two different ProcessCBSubTree-calls in this period cannot share the same arguments $(k_1, y, v)$. Hence there is a bijection between the ProcessCBSubTree-calls in this period (including the "mother" call ProcessCBSubTree$(k_1, k_2, v, y, left)$) and edges in $T_{k_2,y}$. The analysis of the flow of a safe ProcessCBSubTree-call shows that the edge characterized by its inputs is in a completed path after its chain-completion phase is completed. The above show that after the call ProcessCBSubTree$(k_1, k_2, v, y, left)$ returns, each edge in $T_{k_2,y}$ is a part of a completed path. Moreover, $|KSet_1|$ and $|KSet_2|$ clearly stay constant, while each ProcessCBSubTree-call enlarges $|E_3|$ by 1 during its chain-completion phase. As we have seen, there are $|T_{k_2,y}|$ ProcessCBSubTree-calls in this period (including ProcessCBSubTree$(k_1, k_2, v, y, left)$). Hence $|E_3|$ increases by $|T_{k_2,y}|$, and (b) holds for ProcessCBSubTree$(k_1, k_2, v, y, left)$. Similarly for ProcessCBSubTree$(k_1, k_2, v, y, right)$. These complete the proof. □

The last one deals with safe calls to ProcessDualTree.

**Lemma 9.** *The following hold for a safe call to* ProcessDualTree$(k_1, k_2, v, w, 2)$ *(*ProcessDualTree$(k_1, k_2, x, y, 4)$*, resp.):*

*(a) All the calls to layer-2* ProcessTree *procedures made in this call are safe, and $G_2$ never aborts due to calls to* Adapt *in this call;*
*(b) If this call returns without abortion, then after this call, $|KSet_1|$ and $|KSet_2|$ stay constant, while $|E_3|$ increases by at most $|T_{k_2,v}|$ ($|T_{k_2,y}|$, resp.). Moreover, each edge in $T_{k_2,y}$ ($T_{k_2,v}$, resp.) is a part of a completed path.*

*Proof.* There are two possibilities for the flow depending on the arguments:

*Case 1: the call is* ProcessDualTree$(k_1, k_2, v, w, 2)$*.* Let $x := E_3[k_1](w)$ and $y := E_4[k_2](x)$. The chain-completion phase only consists of a call to Adapt$(2, k_2, v, w)$, which creates a 2-query $(2, k_2, v, w, \perp)$ without abortion ($v \notin E_2[k_2] \wedge w \notin E_2[k_2]^{-1}$ as the ProcessDualTree-call is safe). It's clear that neither $|KSet_i|$ nor $|E_3|$ is modified. Additionally, note that the arguments of this ProcessDualTree-call uniquely characterize an edge in $T_{k_2,v}$, say, the edge $(k_1, y, v)$; and after the adaptation, $(k_1, y, v)$ is a part of a completed path.

$G_2$ then calls RecurseNew2$(k_1, k_2, v, w)$. We proceed to argue that all the calls to layer-2 ProcessTree procedures made in RecurseNew2 are safe. Right after $(2, k_2, v, w, \perp)$ is created, for any child $x_i$ of $w$ in $T_{k_2,w}$ (excluding $x$), $x_i$ is non-pebbled; for any $k_1^j$ such that $v \in E_1[k_1^j]^{-1}$ (excluding $k_1$), since $T_{k_2,v}$ is complete, the node $y_j = CTable[(k_1^j, k_2)](E_1[k_1^j]^{-1}(v))$ is a child of $v$ in $T_{k_2,v}$ and must be non-pebbled (and the query to $\widetilde{C}$ would not create new C-queries). By the above, for any call to layer-2 ProcessTree procedures made in the subsequent **forall** loop, the requirement for safeness on the pebbling state is met right after $(2, k_2, v, w, \perp)$ is created, and will keep holding as processing several disjoint subtrees does not affect each other.

On the other hand, the other requirements are directly ensured by the RecurseNew2-call. More clearly, for any $k_1'$,

- only when $v \notin E_1[k_1']^{-1}$ and $w \in E_3[k_1']$ will RecurseNew2 make a call to ProcessB3SubTree$(k_1', k_2, w, E_3[k_1'](w), right)$, which is thereby safe;
- only when $w \notin E_3[k_1']$ and $v \in E_1[k_1']^{-1}$ (as $T_{k_2,v}$ is complete, this implies $E_1[k_1']^{-1}(v) \in CTable[(k_1', k_2)]$ and the completeness of $Li(k_2, y')$ for $y' := CTable[(k_1', k_2)](E_1[k_1']^{-1}(v)))$ will RecurseNew2 make a call to ProcessCBSubTree$(k_1', k_2, y', v, left)$, which is thereby safe;
- only when $w \in E_3[k_1'] \wedge v \in E_1[k_1']^{-1}$ will RecurseNew2 make a call to ProcessDualTree$(k_1', k_2, x', y', 4)$ (let $x' := E_3[k_1'](w)$; and similarly to the previous case, let $y' := CTable[(k_1', k_2)](E_1[k_1']^{-1}(v)))$, which is thereby safe. Similarly to the "mother" call ProcessDualTree$(k_1, k_2, v, w, 2)$, this call also uniquely characterize an edge $(k_1', y', v')$ in $T_{k_2,v}$.

These show the safeness of the layer-2 ProcessTree-calls made in the current ProcessDualTree-call and complete the analysis of Case 1.

*Case 2: the call is* ProcessDualTree$(k_1, k_2, x, y, 4)$*.* The argument is altogether symmetrical to the argument in Case 1, and is thereby omitted. As a summary of the key points, the Adapt-call made in the chain-completion phase does not cause abort due to the safeness assumption on the current ProcessDualTree-call, while the safeness of the calls to layer-2 ProcessTree procedures made in the subsequent call RecurseNew4$(k_1, k_2, x, y)$ is ensured by the RecurseNew4-call itself.

We then consider the statements. First, the analysis above shows that a safe ProcessDualTree-call makes a non-aborting Adapt-call as well as a series of safe calls to layer-2 ProcessTree procedures. By this and Lemmata 7 (a) and 8 (a), (a) is deduced through an induction.

As to (b), assuming a non-aborting execution of ProcessDualTree($k_1, k_2, v, w, 2$), and consider the subsequent calls to layer-2 ProcessTree procedures made in this period. As mentioned (cf. the analysis of Case 1), each subsequent call to ProcessDualTree uniquely characterize an edge in $T_{k_2,v}$. Meanwhile, in the proof of Lemma 8, we already argued that each subsequent ProcessCBSubTree-call also uniquely characterize an edge in $T_{k_2,v}$. Also, it can be seen from the code that two different calls among the set of all subsequent ProcessCBSubTree- and ProcessDualTree-calls in this period cannot characterize the same edge in $T_{k_2,v}$. Hence there is a bijection between the ProcessCBSubTree- and ProcessDualTree-calls in this period (including ProcessDualTree($k_1, k_2, v, w, 2$) itself) and the edges in $T_{k_2,v}$.

The analysis of the flow of a safe ProcessDualTree-call shows that the edge characterized by its arguments is a part of a completed path after the subsequent Adapt-call. The subsequent ProcessCBSubTree-calls have been proved safe, and the edges in the associated live tree are thereby in corresponding completed paths by Lemma 8 (b) (if abort does not occur). By all the above, each edge in $T_{k_2,v}$ is a part of a completed path if the call to ProcessDualTree($k_1, k_2, v, w, 2$) returns without abortion. For ProcessDualTree($k_1, k_2, x, y, 4$) the argument is similar.

We finally prove the claims on the sets to complete the proof. The claim on $|KSet_1|$ and $|KSet_2|$ simply follows from the analysis above and Lemmata 7 (b) and 8 (b). On the other hand, each subsequent ProcessCBSubTree-call enlarges $|E_3|$ by 1 during its chain-completion phase, while none of the subsequent calls to ProcessDualTree and ProcessB3SubTree "directly" enlarges $|E_3|$. As we have seen, there are $|T_{k_2,v}|$ calls to ProcessCBSubTree and ProcessCBSubTree in total in this period (including ProcessDualTree($k_1, k_2, v, w, 2$) itself). Hence $|E_3|$ increases by at most $|T_{k_2,v}|$. □

Around Layer-1 ProcessTree Procedures. As planed, this paragraph analyzes layer-1 ProcessTree procedures. Consider ProcessNonPebB3Tree and ProcessNonPebCBTree first. In such a call, a series of calls to ProcessB3SubTree and ProcessCBSubTree are made. The next two lemmata analyze the influence of such a bundle of layer-2 ProcessTree calls.

**Lemma 10.** *For a call to* ProcessNonPebB3Tree($k_2, w^*, left$) *(or* ProcessNonPebB3Tree($k_2, x^*, right$)*), if the tree $T_{k_2,w^*}$ ($T_{k_2,x^*}$, resp.) has no pebbled leaf, then the following hold:*

*(a) $G_2$ never aborts due to the subsequent calls to* Adapt*;*
*(b) $|KSet_2|$ increases by at most 1, while $|KSet_1|$ and $|E_3|$ stay constant.*

*Proof.* By inspection of the code, it can be seen that an execution of ProcessNonPebB3Tree first creates a new 2- or 4-query (thus enlarging $|KSet_2|$ by at most 1) and then makes several safe calls to ProcessB3SubTree. By this, the claims (a) and (b) immediately follow from Lemma 7.

To give a clearer analysis, let's consider the two possibilities:

*Case 1: the call is* ProcessNonPebB3Tree($k_2, w^*, left$). ProcessNonPebB3Tree starts by creating a new 2-query $(2, k_2, v^*, w^*, \leftarrow)$ via a call to RandomAssign($2, -, k_2, w^*$). If abort does not occur, then the following claim holds by Inv1:

$$\forall k_1 \in \{0,1\}^\kappa, v^* \notin E_1[k_1]^{-1}. \tag{5}$$

ProcessNonPebB3Tree then enters the **forall** loop. Assume that the children of $w^*$ in $T_{k_2,w^*}$ are $x_1, \ldots, x_l$ and the associated 3-edges are $(3, k_1^i, w^*, x_i)$ for $i = 1, \ldots, l$. Note $k_1^1 \neq \ldots \neq k_1^l$. Similarly to [ABD+13], throughout the remaining, we assume the $l$ children are ordered such that $k_1^1 < \ldots < k_1^l$, and assume that the loops of the type "**forall** $k \in KSet_j$" iterates from the smallest value of $k \in KSet_j$ to the largest one. Then, consider the point when the **forall** loop iterates with $k_1 = k_1^i$. At this point, the set of pebbled nodes in $T_{k_2,w^*}$ include the following ones:

- $w^*$, i.e. the parent of $x_i$ in $T_{k_2,w^*}$;
- all the nodes in $SubT(T_{k_2,w^*}, x_1), \ldots, SubT(T_{k_2,w^*}, x_{i-1})$.

Therefore $x_i$ is non-pebbled at this point and $w^*$ is the unique pebbled node in $Li(k_2, x_i)$. Moreover, due to (5) and $k_1^1 \neq \ldots \neq k_1^l$, it holds $v^* \notin E_1[k_1^i]^{-1}$, and hence the call to ProcessB3SubTree($k_1^i, k_2, w, x_i, right$) is safe, and the following hold by Lemma 7 (a) and (b):

– $G_2$ will not abort due to the ADAPT-calls made in PROCESSB3SUBTREE$(k_1^i, k_2, w, x_i, right)$;
– the call PROCESSB3SUBTREE$(k_1^i, k_2, w, x_i, right)$ will not enlarge the sets.

The above analysis indeed holds for $i = 1, \ldots, l$; as a consequence, $G_2$ never aborts due to ADAPT-calls, and (a) holds. Furthermore, by the above analysis, PROCESSNONPEBB3TREE does not enlarge $|KSet_1|$ nor $|E_3|$. On the other hand, the call RANDOMASSIGN$(2, -, k_2, w^*)$ at the beginning of PROCESSNONPEBB3TREE enlarges $|KSet_2|$ by at most 1, while the subsequent process does not affect $KSet_2$. These establish (b) and complete the analysis of Case 1.

*Case 2: the call is* PROCESSNONPEBB3TREE$(k_2, x^*, right)$. The case is symmetrical to Case 1. We recall the key points: PROCESSNONPEBB3TREE starts by creating a new 4-query $(4, k_2, x^*, y^*, \rightarrow)$, and the following holds by Inv4 (if abort does not occur):

$$\forall k_1 \in \{0,1\}^\kappa, y^* \notin CTable[(k_1, k_2)]^{-1}. \tag{6}$$

Then, for each 3-edge $(3, k_1^i, w_i, x^*)$, at the point when the subsequent **forall** loop iterates with $k_1 = k_1^i$, $w_i$ must be non-pebbled and $x^*$ is the unique pebbled node in $Li_{k_2,w_i}$. And $y^* \notin CTable[(k_1^i, k_2)]^{-1}$ due to (6) and $k_1^1 \neq \ldots \neq k_1^l$, so that there does not exist an edge $(k_1^i, y^*, \cdot)$ in $CB(k_2)$ and the call PROCESSB3SUBTREE$(k_1^i, k_2, w_i, x^*, left)$ is safe. Thus all the calls to PROCESSB3SUBTREE in PROCESSNONPEBB3TREE$(k_2, x^*, right)$ are safe and the claims hold by Lemma 7 and an analysis similar to Case 1. □

**Lemma 11.** *For a call to* PROCESSNONPEBCBTREE$(k_2, y^*, left)$ *(or* PROCESSNONPEBCBTREE$(k_2, v^*, right)$*), if the tree $T_{k_2,y^*}$ ($T_{k_2,v^*}$, resp.) is complete and has no pebbled leaf, then the following hold:*

*(a) $G_2$ never aborts due to the subsequent calls to* ADAPT*;*
*(b) If this call returns without abortion, then $|KSet_1|$ stays constant, $|KSet_2|$ increases by at most 1, and $|E_3|$ increases by $|T_{k_2,y^*}|$ ($|T_{k_2,v^*}|$, resp.); and each edge in $|T_{k_2,y^*}|$ ($|T_{k_2,v^*}|$, resp.) is a part of a completed path after this call returns.*

*Proof.* The analysis is very similar to Lemma 10: PROCESSNONPEBCBTREE first creates a new 2- or 4-query (which enlarges $|KSet_2|$ by at most 1) and then makes several safe calls to PROCESSCBSUBTREE, and the non-abortion claim and the claims on $|KSet_1|$ and $|KSet_2|$ follow from Lemma 8. As to $|E_3|$, note that by Lemma 8 (b), the increment of $|E_3|$ due to each PROCESSCBSUBTREE-call equals the size of the subtree processed by it. Hence the total increment equals the summation of the size of each subtree, which equals $|T_{k_2,y}|$.

For clearness, we recall the key observations in each possibility.

*Case 1: the call is* PROCESSNONPEBCBTREE$(k_2, y^*, left)$. PROCESSNONPEBCBTREE starts by creating a new 4-query $(4, k_2, x^*, y^*, \leftarrow)$. If abort does not occur, then the following holds by Inv1:

$$\forall k_1 \in \{0,1\}^\kappa, x^* \notin E_3[k_1]^{-1}. \tag{7}$$

Then, for each edge $(k_1^i, y^*, v_i)$, at the point when the **forall** loop in PROCESSNONPEBCBTREE iterates with $k_1 = k_1^i$, $v_i$ must be non-pebbled because the set of pebbled nodes in $T_{k_2,y^*}$ is $\{y^*\} \cup \bigcup_{j=1}^{i-1} \{\text{nodes in } SubT(T_{k_2,y^*}, v_j)\}$. Furthermore, $x^* \notin E_3[k_1^i]^{-1}$ due to (7) and $k_1^1 \neq \ldots \neq k_1^l$, and hence the call to PROCESSCBSUBTREE$(k_1^i, k_2, y^*, v_i, right)$ is safe. The analysis is applicable to all edges $(k_1^i, y^*, v_i)$ adjacent to $y^*$, hence all calls to PROCESSCBSUBTREE in PROCESSNONPEBCBTREE$(k_2, y^*, left)$ are safe, and the claims on non-abortion (due to ADAPT) and $|KSet_1|$ and $|KSet_2|$ follow from Lemma 8 and an analysis similar to Lemma 10. Finally, by Lemma 8, each call to PROCESSCBSUBTREE$(k_1^i, k_2, y^*, v_i, right)$ enlarges $|E_3|$ by $|SubT(T_{k_2,y^*}, v_i)| + 1$, and the increment is $|T_{k_2,y^*}|$ in total.

*Case 2: the call is* PROCESSNONPEBCBTREE$(k_2, v^*, right)$. PROCESSNONPEBCBTREE starts by creating a new 2-query $(2, k_2, v^*, w^*, \rightarrow)$, after which the following holds by Inv1 (if abort does not occur):

$$\forall k_1 \in \{0,1\}^\kappa, w^* \notin E_3[k_1]. \tag{8}$$

Then, for each $k_1^i$ such that $v^* \in E_1[k_1^i]^{-1}$, since $T_{k_2,v^*}$ is complete, the node $y_i = CTable[(k_1^i, k_2)](E_1[k_1^i]^{-1}(v^*))$ is a child of $v^*$ in $T_{k_2,v^*}$. Moreover, $y_i$ must be non-pebbled, and $w^* \notin E_3[k_1^i]$ due to (8) and $k_1^1 \neq \ldots \neq k_1^l$. Hence the call to PROCESSCBSUBTREE$(k_1^i, k_2, y_i, v^*, left)$ is safe, and the claims can be established similarly to Case 1. These complete the proof. □

An execution of PROCESSPEBB3TREE or PROCESSPEBCBTREE mostly consists of a single call to layer-2 PROCESSTREE procedures, and is thus relatively simpler. We thereby analyze them in the subsequent lemmata instead of giving separate analysis in this paragraph.

Around New 2- and 4-queries. Based on the lemmata above, we now analyze the influences of new 2- and 4-queries. In each case, the increments of $|KSet_i|$ and $|E_3|$ are presented, and it is proved that the absence of early-abortion implies non-abortion.

**Lemma 12.** *During $G_2$ processing a query* $\mathrm{E2}^{-1}$ *or* E4*, the following hold:*

*(a) $|KSet_1|$ stays constant, $|KSet_2|$ increases by at most 1, while $|E_3|$ increases by at most $|T_{CB}| \cdot |KSet_1|$ where $T_{CB}$ is the involved live tree in $CB(k_2)$.*[21]
*(b) If early-abortion does not occur, then $G_2$ does not abort.*
*(c) If abort does not occur, then each C-query newly created in this period is a part of a completed path after the process.*

*Proof.* By inspection of the code, it can be seen that the processes around $\mathrm{E2}^{-1}$ and E4 are almost the same. We thereby take a deep look into $\mathrm{E2}^{-1}$, and only sketch E4.

By construction, upon a query $\mathrm{E2}^{-1}(k_2, w^*)$, if $w^* \in E_2[k_2]^{-1}$, then $G_2$ answers with $E_2[k_2]^{-1}(w^*)$; if $w^* \notin E_2[k_2]^{-1} \wedge w^* \notin LS_3$, $G_2$ draws the random answer from $\mathbf{E}$. In these cases the statements clearly hold.

We thereby focus on the remaining (more complex) cases. If $G_2$ finds $w^* \notin E_2[k_2]^{-1}$ and $w^* \in LS_3$, then it calls FindPebLeafB3. By Lemma 5, $T_{k_2,w^*}$ has at most 1 pebbled leaf. Moreover, till now $Li(k_2, w^*)$ receives no modification, so that when FindPebLeafB3 is called, $Li(k_2, w^*)$ still equals $T_{k_2,w^*}$ and has at most 1 pebbled leaf. By this, $G_2$ would not abort due to $|OriginSet| > 1$.

We consider the flow in case of $|OriginSet| = 0$ first, which is simpler. It necessarily be that $Li(k_2, w^*)$ has no pebbled leaf before the earlier call to FindPebLeafB3; as FindPebLeafB3 does not modify $Li(k_2, w^*)$, $Li(k_2, w^*)$ has no pebbled leaf after FindPebLeafB3 returns. By construction, a call to ProcessNonPebB3Tree$(k_2, w^*, left)$ is then made, and (a) immediately follows from Lemma 10 (b). As to (b), first note that by Lemma 10 (a) and the analysis above, $G_2$ never aborts due to the conditions other than the early-abort ones (say, $G_2$ does not abort due to $|OriginSet| > 1$ nor adaptations). As we assume early-abortion absent, $G_2$ does not abort and (b) holds. Finally, it can be seen from the analysis in Lemma 10 that $\mathbf{T}$ only queries $\widetilde{C}$ during the chain-completion phase of the subsequent ProcessB3SubTree-calls, so that each newly-created C-query is a part of a completed path right after the corresponding Adapt-call. This claim keeps holding till the end of the current simulator cycle as nothing is overwritten, hence (c) holds – and all the claims hold in case of $|OriginSet| = 0$.

We then consider the case(s) of $|OriginSet| = 1$. In these cases, a call to ProcessPebB3Tree is made. Depending on the arguments to ProcessPebB3Tree and the concrete flow of ProcessPebB3Tree (depending on concrete conditions, ProcessPebB3Tree may call ProcessPebB3Tree or ProcessDualTree), the discussions are divided into four cases, as follows:

*Case 1: the call is* ProcessPebB3Tree$(k_1^\circ, k_2, w^\circ, x^\circ, left)$*, and* ProcessPebB3Tree *subsequently calls* ProcessB3SubTree*.* Then by construction, right before the call ProcessB3SubTree$(k_1^\circ, k_2, w^\circ, x^\circ, left)$ is made, it holds:

(i) $x^\circ$ is pebbled;
(ii) $w^\circ$ is non-pebbled, and $Li(k_2, w^\circ)$ has only one pebbled leaf (i.e. $x^\circ$; the reason is $Li(k_2, w^\circ) = T_{k_2,w^*}$);
(iii) For $y^\circ := E_4[k_2](x^\circ)$, there does not exist an edge $(k_1^\circ, y^\circ, v^\circ)$ in $CB(k_2)$ (as FindEdgeInCB$(k_1^\circ, k_2, y^\circ)$ returns $\perp$).

Therefore, the call ProcessB3SubTree$(k_1^\circ, k_2, w^\circ, x^\circ, left)$ is safe, and the statements can be established similarly as before. More clearly, (i) (a) holds by Lemma 7 (b) and the safeness of the ProcessB3SubTree-call; (ii) by Lemma 7 (a) and the analysis above, $G_2$ never aborts due to the conditions other than the early-abort ones. As early-abortion is absent, $G_2$ does not abort and (b) holds; (iii) by the analysis in Lemma 7, $\mathbf{T}$ only queries $\widetilde{C}$ during the chain-completion phase of the calls to ProcessB3SubTree, hence (c) holds.

[21] Formally speaking, the claim in (a) means: if ProcessDualTree$(k_1, k_2, v, w, 2)$ is subsequently called, then $|E_3|$ increases by at most $|T_{k_2,v}| \cdot |KSet_1|$; if ProcessDualTree$(k_1, k_2, x, y, 4)$ is called, then $|E_3|$ increases by at most $|T_{k_2,y}| \cdot |KSet_1|$; otherwise $|E_3|$ stays constant/increases by 0. We remark that the notations $T_{k_2,v}$ and $T_{k_2,y}$ refer to the live trees before the query $\mathrm{E2}^{-1}$/E4.

*Case 2: the call is* $\textsc{ProcessPebB3Tree}(k_1^\circ, k_2, w^\circ, x^\circ, left)$*, and* $\textsc{ProcessPebB3Tree}$ *subsequently calls* $\textsc{ProcessDualTree}$. In this case, it necessarily be that $\textsc{ProcessPebB3Tree}$ computes a value $v^\circ \neq \perp$ (thus calling $\textsc{ProcessDualTree}(k_1^\circ, k_2, v^\circ, w^\circ, 2)$ then). To establish the claims, we proceed to argue that this $\textsc{ProcessDualTree}$-call is safe.

As the starting point, we argue that $v^\circ$ is non-pebbled before the **forall** loop in $\textsc{ProcessPebB3Tree}$. Assuming otherwise, then as no query is newly created till the point before the **forall** loop, the following four queries necessarily existed before the query $\mathrm{E2}^{-1}(k_2, w^*)$: a 4-query $(4, k_2, x^\circ, y^\circ, d_4, n_4)$, a C-query $((k_1^\circ, k_2), u^\circ, y^\circ, d_C, n_C)$, a 1-query $(1, k_1^\circ, u^\circ, v^\circ, d_1, n_1)$, and a 2-query $(2, k_2, v^\circ, w^\circ, d_2, n_2)$. Then the four queries had to be in the same completed path, contradicting the assumption that $x^\circ$ was non-pebbled:

– if $n_C > n_1$, then $d_C$ must be $\to$ by Inv4, and further $n_4 > n_C$ by Inv4, so that the 4 queries are in the same completed path by Lemma 3;
– if $n_1 > n_C$, then $d_1 =\to$ and $n_2 > n_1$ and the queries are in the same completed path by Lemma 3.

Therefore, $y^\circ$ is a pebbled leaf of the tree $T_{k_2,v^\circ}$ (before the simulator cycle); as $T_{k_2,v^\circ}$ has at most one pebbled leaf (Lemma 5), $y^\circ$ is the unique pebbled node.

Then, depending on the flow of $\textsc{ProcessPebB3Tree}$, we've two subcases.

*Subcase 2.1:* $\textsc{FindPebLeafCB}$ *is not called.* Then it necessarily be that $v^\circ \notin E_1[k_1']^{-1}$ for any $k_1' \in KSet_1 \setminus \{k_1^\circ\}$, and $T_{k_2,v^\circ}$ consists of only one edge $(k_1^\circ, y^\circ, v^\circ)$. It's then an easy task to check that the subsequent call $\textsc{ProcessDualTree}(k_1^\circ, k_2, v^\circ, w^\circ, 2)$ is safe. By this, the three statements can be established by Lemma 9 and by an argument similar to that in Case 1.

*Subcase 2.2:* $\textsc{FindPebLeafCB}$ *is called.* Note that $y^\circ$ is the unique pebbled node in $T_{k_2,v^\circ}$. Moreover, the call $\textsc{FindPebLeafCB}(k_2, v^\circ, right)$ would not cause $G_2$ abort, as it can only cause early-abortion, which is assumed absent. Let $TX_{k_2,v^\circ}$ be the snapshot of the live tree $Li(k_2, v^\circ)$ standing right after $\textsc{FindPebLeafCB}(k_2, v^\circ, right)$ returns. Then the following hold by Lemma 6:

(i) $y^\circ$ remains the unique pebbled node in $TX_{k_2,v^\circ}$ (Lemma 6 (a));
(ii) $TX_{k_2,v^\circ}$ is complete (Lemma 6 (b));
(iii) $|TX_{k_2,v^\circ}| \leq |T_{k_2,v^\circ}| \cdot |KSet_1|$ (Lemma 6 (c). Note that here $|T_{k_2,v^\circ}| \geq 1$.);
(iv) Each C-query created in $\textsc{FindPebLeafCB}$ is a part of an edge in $TX_{k_2,v^\circ}$ (Lemma 6 (a));

As a consequence, the call $\textsc{ProcessDualTree}(k_1^\circ, k_2, v^\circ, w^\circ, 2)$ is safe, and the statements are proved as follows:

(i) (a) follows from Lemma 9 (b) and $|TX_{k_2,v^\circ}| \leq |T_{k_2,v^\circ}| \cdot |KSet_1|$;
(ii) (b) holds by Lemma 9 (a) and the analysis above;
(iii) Note that in this case, new C-queries may be created during the chain-completion phase of layer-2 $\textsc{ProcessTree}$-calls or during the call to $\textsc{FindPebLeafCB}$. We already argued (in the previous cases) that the former type of new C-queries are in completed paths. On the other hand, the latter type of new C-queries are in the edges of $TX_{k_2,v^\circ}$ (cf. the remark (iv) above); by Lemma 9 (b), each edge of $TX_{k_2,v^\circ}$ will be in a complected path. By this, each C-query newly created in $\textsc{FindPebLeafCB}$ is also in a completed path after the simulator cycle, and (c) is established.

These complete the analysis of Case 2.

*Case 3: the call is* $\textsc{ProcessPebB3Tree}(k_1^\circ, k_2, w^\circ, x^\circ, right)$*, and* $\textsc{ProcessPebB3Tree}$ *subsequently calls* $\textsc{ProcessB3SubTree}$. This case is very similar to Case 1: right before the call $\textsc{ProcessB3SubTree}(k_1^\circ, k_2, w^\circ, x^\circ, right)$ is made, it holds:

(i) $w^\circ$ is pebbled;
(ii) $x^\circ$ is non-pebbled, and $Li(k_2, x^\circ)$ has only one pebbled leaf, i.e. $w^\circ$;
(iii) For $v^\circ := E_2[k_2]^{-1}(w^\circ)$, it holds $v^\circ \notin E_1[k_1^\circ]^{-1}$.

Therefore, the call $\textsc{ProcessB3SubTree}(k_1^\circ, k_2, w^\circ, x^\circ, right)$ is safe, and the statements are established by an argument similar to Case 1.

*Case 4: the call is* $\textsc{ProcessPebB3Tree}(k_1^\circ, k_2, w^\circ, x^\circ, right)$*, and* $\textsc{ProcessPebB3Tree}$ *subsequently calls* $\textsc{ProcessDualTree}$. In this case, it necessarily be $v^\circ \in E_1[k_1^\circ]^{-1}$. Let $u^\circ := E_1[k_1^\circ]^{-1}(v^\circ)$. Then depending on the state of the history, the discussions are divided into three subcases.

*Subcase 4.1: $u^\circ \notin CTable[(k_1^\circ, k_2)]$ before the cycle.* In this case, ProcessPebB3Tree creates a new C-query $((k_1^\circ, k_2), u^\circ, y^\circ, \rightarrow)$ via querying $\widetilde{C}$. Right after this point, if early-abortion does not occur, then it holds $y^\circ \notin E_4[k_2]^{-1}/y^\circ$ is non-pebbled (by Inv4; we thereby let $T^*_{k_2,y^\circ} = Li(k_2, y^\circ)$ at this point) and (by Inv3)

$$\forall k_1' \in \{0,1\}^\kappa \setminus \{k_1^\circ\}, y^\circ \notin CTable[(k_1', k_2)]^{-1}, \tag{9}$$

By (9), FindEdgeInCB returns $\perp$ for any $k_1' \in KSet_1 \setminus \{k_1^\circ\}$, and FindPebLeafCB is never called. Consequently, $T^*_{k_2,y^\circ}$ consists of only one edge $(k_1^\circ, y^\circ, v^\circ)$ (with $v^\circ$ being pebbled), and the call ProcessDualTree$(k_1^\circ, k_2, v^\circ, w^\circ, 2)$ is safe. The statements are then proved as follows:

(i) The claims on $|KSet_1|$ and $|KSet_2|$ follow from Lemma 9 (b). As to the claim on $|E_3|$, note that by (9) and the fact that the edge $(k_1^\circ, y^\circ, v^\circ)$ is newly added to $CB(k_2)$, the live tree $Li(k_2, y^\circ)$ standing before this simulator cycle indeed has NO edge – i.e. $|T_{k_2,y^\circ}| = 0$. On the other hand, by (9) and by inspection of the code of ProcessDualTree , it can be seen that the call ProcessDualTree$(k_1^\circ, k_2, v^\circ, w^\circ, 2)$ only makes safe calls to ProcessB3Tree during the recursively-calling phase. By Lemma 7 (b), these calls do not enlarge $|E_3|$; hence $|E_3|$ does not increase in this subcase, which meets $|T_{k_2,y^\circ}| = 0$ and establishes (a).

(ii) (b) holds by Lemma 9 (a) and the analysis above;

(iii) In this case, besides the new queries created during the chain-completion phase of subsequent layer-2 ProcessTree-calls, the ProcessPebB3Tree-call itself creates a new C-query (i.e. $((k_1^\circ, k_2), u^\circ, y^\circ, \rightarrow)$). Similarly to subcase 2.2, the former type of new C-queries are in completed paths, while the C-query created by ProcessPebB3Tree is in the edge of $T^*_{k_2,y^\circ}$, which will be in a complected path by Lemma 9 (b). By this, (c) is established.

*Subcase 4.2: $u^\circ \in CTable[(k_1^\circ, k_2)]$ before the cycle, and* FindPebLeafCB *is not called.* Then it necessarily be that FindEdgeInCB returns $\perp$ for any $k_1' \in KSet_1 \setminus \{k_1^\circ\}$ (otherwise FindPebLeafCB would be called). Similarly to Case 2, $y^\circ = CTable[(k_1^\circ, k_2)](u^\circ)$ must be non-pebbled before the query $\mathrm{E2}^{-1}(k_2, w^*)$: otherwise either the 1-query $(1, k_1^\circ, u^\circ, v^\circ)$ or the 4-query which pebbles $y^\circ$ has an associated *num* value larger than the *num* value of the edge $(k_1^\circ, y^\circ, v^\circ)$ (due to Inv4), and the four queries are in the same completed path (by Lemma 3) and $x^\circ$ would have been pebbled. By this, right before ProcessDualTree is called, the tree $Li(k_2, y^\circ)$ consists of only one edge $(k_1^\circ, y^\circ, v^\circ)$ (with $v^\circ$ being pebbled). It's then clearly that the call ProcessDualTree$(k_1^\circ, k_2, v^\circ, w^\circ, 2)$ is safe, and the statements can be established similarly to subcase 2.2 (except that in this subcase, there's no new C-query due to FindPebLeafCB).

Indeed, the mere difference between subcase 4.2 and subcase 4.1 is that ProcessPebB3Tree does not create new C-queries in subcase 4.2.

*Subcase 4.3:* FindPebLeafCB *is called.* This case is very similar to subcase 2.2. First, $y^\circ$ must be non-pebbled before the current cycle, otherwise $x^\circ$ would have been pebbled (this argument is similar to subcase 4.2). By this, $v^\circ$ is the unique pebbled node in $T_{k_2,y^\circ}$. Let $TX_{k_2,y^\circ}$ be the snapshot of the live tree $Li(k_2, y^\circ)$ standing right after FindPebLeafCB$(k_2, y^\circ, left)$ returns (this call indeed returns without abortion because early-abortion is assumed absent). Then by Lemma 6 (a), $v^\circ$ remains the only pebbled node in $TX_{k_2,y^\circ}$; and by Lemma 6 (b), $TX_{k_2,y^\circ}$ is complete. As a consequence, the call ProcessDualTree$(k_1^\circ, k_2, x^\circ, y^\circ, 4)$ is safe, and the statements can be established similarly to subcase 2.2. These complete the analysis of the simulator cycles due to $\mathrm{E2}^{-1}$.

We then sketch the analysis of a cycle due to $\mathrm{E4}(k_2, x^*)$. If $T_{k_2,x^*}$ has no pebbled leaf, then the analysis is similar to the case of $\mathrm{E2}^{-1}(k_2, v^*)$ where $T_{k_2,v^*}$ has no pebbled leaf. If $T_{k_2,x^*}$ has pebbled leaves, then it has only one pebbled leaf by Lemma 5, and the subsequent cases are *the same as* those captured by Case 1-4 above. These complete the proof. □

We view the claim on $|E_3|$ as crucial. Since the proof of Lemma 12 is a bit long, in order to make it clearer, we illustrate the ideas around $|E_3|$ in Fig. 16.

**Lemma 13.** *During $G_2$ processing a query $\mathrm{E4}^{-1}(k_2, y^*)$, the following hold:*

*(a) $|KSet_1|$ stays constant, $|KSet_2|$ increases by at most 1, while $|E_3|$ increases by at most $|T_{k_2,y^*}| \cdot |KSet_1|$.*

*(b) If early-abortion does not occur, then $G_2$ does not abort.*

*(c) If abort does not occur, then each C-query newly created in this period is a part of a completed path after the process.*

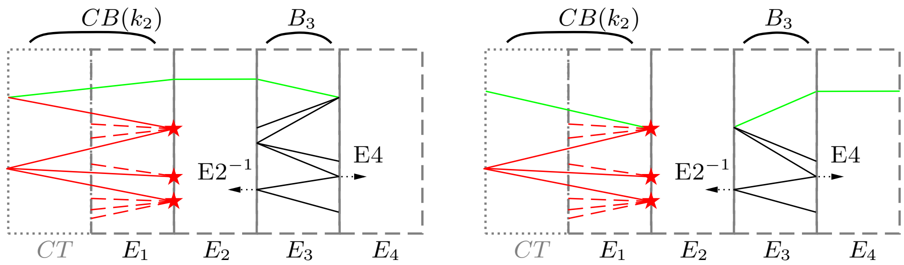


**Fig. 16.** For Lemma 12. If the query only involve an isolate tree in $B_3$, then $|E_3|$ stays constant, as argued in Lemmata 7 and 10. Thus if $|E_3|$ increases, the involved structure has to be a dual-tree. The left and right parts of this figure show the two different cases of dual-tree. In each case, the lime lines indicate the "single-query-missing" path. In ProcessDualTree, $G_2$ (first) completing this path would not enlarge $|E_3|$. The increments to $|E_3|$ are brought in by the red lines, i.e. the involved tree in $CB(k_2)$. In this tree, the dashed red lines indicate the 1-queries that are not already in edges before the cycle, i.e. 1-queries of the form $(1, k_1, u, v)$ with $u \notin ETable[(k_1, k_2)]$. For each involved right-shore node in $CB(k_2)$ (emphasized by red pentagrams), there are at most $|KSet_1|$ such 1-queries (because for $l$ distinct 1-queries of the form $(1, k_1^i, u^i, v)$ the associated $k_1^i$ must be $l$ distinct ones). It's purely geometrical to see that in each case, the number of right-shore nodes (i.e. red pentagrams) cannot exceed the number of involved edges (i.e. all the red lines and the lime line) in $CB(k_2)$. Therefore, $|E_3|$ increases by at most $|T_{CB}| \cdot |KSet_1|$.

*Proof.* Similarly to Lemma 12, we focus on a *new* query $\text{E4}^{-1}(k_2, y^*)$. If $T_{k_2,y^*} = \{y^*\}$, then for no $k_1$ does FindEdgeInCB return **true**, so that the call Exists14Tripwire$(k_2, y^*)$ (at the beginning of $\text{E4}^{-1}$) returns **false**, and $G_2$ simply calls RandomAssign to answer. In this case the statements clearly hold.

We then consider the remaining cases. If $G_2$ finds $y^* \notin E_4[k_2]^{-1}$ and a pre-existing edge $(k_1, y^*, v^*)$ in $CB(k_2)$ (via Exists14Tripwire), then it calls FindPebLeafCB$(k_2, y^*, left)$. At this point, $Li(k_2, y^*)$ still equals $T_{k_2,y^*}$, so that it has at most 1 pebbled leaf by Lemma 5. Let $TX_{k_2,y^*}$ be the snapshot of $Li(k_2, y^*)$ standing right after FindPebLeafCB$(k_2, y^*, left)$ returns (which would not cause abort due to the absence of early-abortion), then by Lemma 6, we have the following remarks (similarly to those mentioned in subcase 2.2 of Lemma 12): (1) the pebbling state of $TX_{k_2,y^*}$ is exactly the same as that of $T_{k_2,y^*}$; (2) $TX_{k_2,y^*}$ is complete; (3) $|TX_{k_2,y^*}| \leq |T_{k_2,y^*}| \cdot |KSet_1|$; (4) Each C-query created in FindPebLeafCB is a part of an edge in $TX_{k_2,y^*}$. By remark (1), $TX_{k_2,y^*}$ has at most 1 pebbled leaf, and hence $G_2$ will not abort due to $|OriginSet| > 1$.

We consider the case of $|OriginSet| = 0$ first. It necessarily be that $Li(k_2, y^*)$ has no pebbled leaf before the call to FindPebLeafCB; gathering this and remark (2) (mentioned before) establishes the safeness of the subsequent call to ProcessNonPebCBTree$(k_2, y^*, left)$. Then (a) and (b) follow from Lemma 11 and the analysis and remark (3). The argument for (c) is similar to subcase 2.2 of Lemma 12: the new C-queries due to FindPebLeafCB will be in the edges of $TX_{k_2,y^*}$ which will further be in completed paths, while those due to layer-2 ProcessTree-calls will clearly be in completed paths. Hence the claims hold for the case of $|OriginSet| = 0$.

We then consider the cases of $|OriginSet| = 1$. In these cases, a call to ProcessPebCBTree is made, and the discussions are divided into four cases depending on the concrete flow:

*Case 1: the call is* ProcessPebCBTree$(k_1^\circ, k_2, y^\circ, v^\circ, left)$*, and* ProcessPebCBTree *subsequently calls* ProcessCBSubTree. Then by construction, right before the call ProcessCBSubTree$(k_1^\circ, k_2, y^\circ, v^\circ, left)$ is made, it holds:

(i) $v^\circ$ is pebbled;
(ii) $y^\circ$ is non-pebbled; $Li(k_2, y^\circ)$ is complete and has only one pebbled leaf (since $Li(k_2, y^\circ) = TX_{k_2,y^*}$);
(iii) For $w^\circ := E_2[k_2](v^\circ)$, it holds $w^\circ \notin E_3[k_1^\circ]$.

Therefore, the call ProcessCBSubTree$(k_1^\circ, k_2, y^\circ, v^\circ, left)$ is safe, and the statements can be shown by Lemma 8 and an argument similar to the case of $|OriginSet| = 0$.

*Case 2: the call is* ProcessPebCBTree$(k_1^\circ, k_2, y^\circ, v^\circ, left)$*, and* ProcessPebB3Tree *subsequently calls* ProcessDualTree. In this case, it necessarily be $w^\circ \in E_3[k_1^\circ]$ ($w^\circ = E_2[k_2](v^\circ)$) and ProcessDualTree$(k_1^\circ, k_2, x^\circ, y^\circ, 4)$ is then called ($x^\circ = E_3[k_1^\circ](w^\circ)$). In order to utilize Lemma 9, we show the safeness of this ProcessDualTree-call. Similarly to Case 2 in Lemma 12, we first argue that $x^\circ$ is non-pebbled before the ProcessDualTree-call. Assume otherwise, then before the current simulator cycle, there already existed a 2-query $(2, k_2, v^\circ, w^\circ, d_2, n_2)$, a 3-query $(3, k_1^\circ, w^\circ, x^\circ, d_3, n_3)$, and a 4-query $(4, k_2, x^\circ, y^\circ, d_4, n_4)$ (these queries indeed existed before the cycle because they cannot be created by FindPebLeafCB). Then as $d_3 \in \{\leftarrow, \rightarrow\}$ (Inv0), either $n_4 > n_3$ or $n_2 > n_3$ due to Inv1, and the three queries had to be in the same completed path due to Inv5, contradicting the assumption that $y^\circ$ was non-pebbled.

Therefore, $w^\circ$ is a pebbled leaf of the tree $T_{k_2,x^\circ}$ (before the current simulator cycle); as $T_{k_2,x^\circ}$ has at most one pebbled leaf (Lemma 5), $w^\circ$ is the unique pebbled node. By this and the remarks above ((1) and (2)), the subsequent ProcessDualTree-call is safe. The statements thereby follow from Lemma 9 and an argument similar to the case of $|OriginSet| = 0$.

*Case 3: the call is* ProcessPebCBTree$(k_1^\circ, k_2, y^\circ, v^\circ, right)$*, and* ProcessPebCBTree *subsequently calls* ProcessCBSubTree. Symmetrically to Case 1, right before the call ProcessPebCBTree$(k_1^\circ, k_2, y^\circ, v^\circ, right)$ is made, it holds:

(i) $y^\circ$ is pebbled;
(ii) $v^\circ$ is non-pebbled; $Li(k_2, v^\circ)$ is complete and has only one pebbled leaf;
(iii) For $x^\circ := E_4[k_2]^{-1}(y^\circ)$, it holds $x^\circ \notin E_3[k_1^\circ]^{-1}$.

The subsequent call ProcessCBSubTree$(k_1^\circ, k_2, y^\circ, v^\circ, right)$ is thereby safe, and the proof of the statements is similar to Case 1 by symmetry.

*Case 4: the call is* ProcessPebCBTree$(k_1^\circ, k_2, y^\circ, v^\circ, right)$*, and* ProcessPebCBTree *subsequently calls* ProcessDualTree. In this case, it must be $x^\circ \in E_3[k_1^\circ]^{-1}$ $(x^\circ = E_4[k_2]^{-1}(y^\circ))$. Let $w^\circ := E_3[k_1^\circ]^{-1}(x^\circ)$. By an argument similar to Case 2, $w^\circ$ must be non-pebbled, and $x^\circ$ is the unique pebbled node in $T_{k_2,w^\circ}$. Hence the subsequent ProcessDualTree-call is safe and the proof of the statements follows the same line as Case 2. These complete the proof. ◻

**Lemma 14.** *During $G_2$ processing a query* $\mathrm{E2}(k_2, v^*)$*, the following hold:*

*(a) $|KSet_1|$ stays constant, $|KSet_2|$ increases by at most 1, while $|E_3|$ increases by at most $Max\{|T_{k_2,v^*}|, 1\} \cdot |KSet_1|$.*
*(b) If early-abortion does not occur, then $G_2$ does not abort.*
*(c) If abort does not occur, then each C-query newly created in this period is a part of a completed path after the process.*

*Proof.* If the query is new while $v^* \notin LS_1$ then $G_2$ simply calls RandomAssign to answer and the statements clearly hold. Otherwise, by construction, there is to be a call to FindPebLeafCB$(k_2, v^*)$. Let $TX_{k_2,v^*}$ be the snapshot of $Li(k_2, v^*)$ standing right after FindPebLeafCB returns. Then by Lemma 6: (1) the pebbled nodes of $T_{k_2,v^*}$ are also the pebbled nodes of $TX_{k_2,v^*}$; (2) $TX_{k_2,v^*}$ is complete; (3) $|TX_{k_2,v^*}| \leq \mathrm{Max}\{|T_{k_2,v^*}|, 1\} \cdot |KSet_1|$; (4) Each C-query created in FindPebLeafCB is a part of an edge in $TX_{k_2,v^*}$ – indeed, observation (3) is the *mere* difference between the cycles due to $\mathrm{E2}(k_2, v^*)$ and the cycles due to $\mathrm{E4}^{-1}(k_2, y^*)$ (cf. Lemma 13). The analysis follows the same line as Lemma 13 and has no novelty except for replacing $|T_{k_2,y^*}| \cdot |KSet_1|$ (as in Lemma 13 (a)) by $\mathrm{Max}\{|T_{k_2,v^*}|, 1\} \cdot |KSet_1|$. ◻

We similarly illustrate the ideas around $|E_3|$ in Fig. 17.

**Bounding the Complexity.** With the analysis in the previous subsubsection, we now bound the complexity of $\mathbf{T}$ in $G_2$. They are captured by a series of lemmata as follows. The first lemma presents the core idea: the C-queries made by $\mathbf{T}$ do not "essentially" enlarge the size of any live tree in $CB(k_2)$ for any $k_2$.

**Lemma 15.** *At the end of each simulator cycle, as long as $G_2$ does not abort, it holds:*

*(a) Each C-query made by $\mathbf{T}$ (till that point) is a part of a completed path, and has an associated 4-tuple $(k_1, k_2, x, 4)$ in Completed.*
*(b) $\sum_{k_2 \in \{0,1\}^\kappa} |CB(k_2)| \leq q$, where $|CB(k_2)|$ is the total number of edges in live trees in $CB(k_2)$.*

*Proof.* Consider (a) first. We note that during the simulator cycles due to $\overline{D}$ issuing a 1- or 3-query, $\mathbf{T}$ does not make any C-queries; whereas during the cycles due to $\overline{D}$ issuing a 2- or 4-query, the claims are ensured by Lemmata 12-14. On the other hand, the existence of the 4-tuple could be verified by revisiting the proof of Lemmata 12-14.

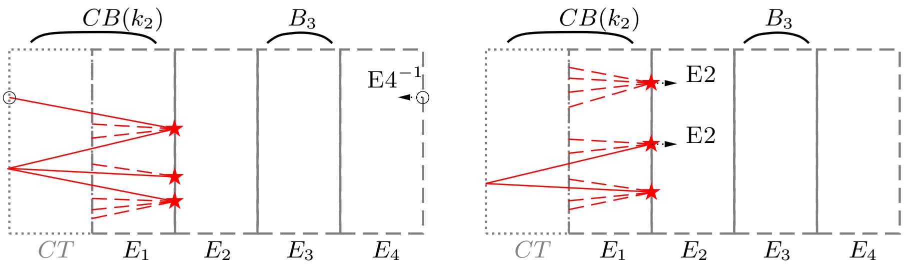


**Fig. 17.** For Lemmata 13 and 14. These queries themselves are already "anchored" at trees in $CB(k_2)$, thus they can "directly" bring increments to $|E_3|$ (i.e. not necessarily via dual-trees). (Left) the case of $\overline{D}$ querying $E4^{-1}$. Clearly, the number of right-shore nodes/red pentagrams cannot exceed the number of involved edges/red lines in $CB(k_2)$, and thus $|E_3|$ increases by at most $|T_{CB}| \cdot |KSet_1|$. (Right) the case of $\overline{D}$ querying E2. The upper structure depicts the extreme case of $|T_{CB}| = 0$, while the lower structure depicts the general case. In the general case, the number of right-shore nodes cannot exceed the number of edges either. Whereas in extreme case, the number of right-shore nodes equals that of edges plus 1. Therefore, generally speaking, $|E_3|$ increases by at most $\mathrm{Max}\{|T_{k_2,v^*}|, 1\} \cdot |KSet_1|$.

(b) is indeed a corollary of (a). We note that for any $k_2$, each C-query in *CQueries* gives rise to at most one edge in $CB(k_2)$. The associated C-queries of two edges in $CB(k_2)$ and $CB(k_2')$ ($k_2 \neq k_2'$) cannot be the same, as the associated $k_2$ values deviate. Furthermore, note that an edge in a live tree has at least one non-pebbled endpoint, so that its associated C-query cannot have been created by $\mathbf{T}$ (as otherwise the query is in a completed path and the two endpoints of the edge are both pebbled). As $\overline{D}$ creates at most $q$ C-queries, (b) holds. □

The second lemma bounds $|KSet_1|$ and $|KSet_2|$.

**Lemma 16.** *At the end of* $\overline{D}^{G_2}$, $|KSet_1| \leq q$, $|KSet_2| \leq q$.

*Proof.* For $|KSet_1|$, consider each query of the *original distinguisher* $D$. If the query from $D$ is a 1- or 3-query, then the same query will appear in $\overline{D}^{G_2}$, which enlarges $|KSet_1|$ by at most 1. If the query is a 2- or 4-query, then the same query will appear in $\overline{D}^{G_2}$, which does not enlarge $|KSet_1|$ by Lemmata 12-14. Finally, if the query is a C-query, then four E-queries will appear in $\overline{D}^{G_2}$. Among the four queries, the 2- and the 4-query do not enlarge $|KSet_1|$. The 1- and the 3-query may enlarge $|KSet_1|$; but they share the same associated $k_1$ value, thereby only enlarge $|KSet_1|$ by at most 1. By the above, each query from $D$ leads to $|KSet_1|$ increasing by at most 1. As $D$ makes at most $q$ queries, $|KSet_1| \leq q$. The argument for $|KSet_2| \leq q$ is similar: $|KSet_2|$ stays constant for a 1- or 3-query of $D$, while increases by at most 1 for a 2-, 4-, or C-query of $D$ (by Lemmata 12-14, and by an analysis similar to the above). □

The third lemma bounds $|E_3|$.

**Lemma 17.** *At the end of* $\overline{D}^{G_2}$, $|E_3| \leq 3q^2$.

*Proof.* By construction, $|E_3|$ can only be enlarged in the following three cases:

- $\overline{D}$ directly makes a 3-query, the number of which is at most $q$ (cf. page 16);
- $\overline{D}$ makes a 2-query. It's further divided into two subcases:
  - the query is $E2(k_2, v)$: by Lemma 14 (a), $|E_3|$ increases by at most $\mathrm{Max}\{|T_{k_2,v}|, 1\} \cdot |KSet_1| \leq (|T_{k_2,v}| + 1) \cdot |KSet_1|$. Furthermore, if $|T_{k_2,v}| \geq 1$, then by Lemma 15 (a), the edges are necessarily formed by C-queries due to $\overline{D}$;
  - the query is $E2^{-1}(k_2, w)$: assume that before this query, the live tree in $CB(k_2)$ that is connected to $T_{k_2,w}$ is $T_{CB}$. Then by Lemma 12 (a), $|E_3|$ increases by at most $|T_{CB}| \cdot |KSet_1|$. Similarly, the edges in $T_{CB}$ are necessarily formed by C-queries due to $\overline{D}$;
- $\overline{D}$ makes a 4-query. It's also divided into two subcases:
  - the query is $E4(k_2, x)$: the case is similar to the case of $E2^{-1}(k_2, w)$. Assume that before this query, the live tree in $CB(k_2)$ that is connected to $T_{k_2,x}$ is $T_{CB}$. Then by Lemma 12 (a), $|E_3|$ increases by at most $|T_{CB}| \cdot |KSet_1|$, and the edges in $T_{CB}$ are necessarily due to $\overline{D}$ querying $\widetilde{C}$;
  - the query is $E4^{-1}(k_2, y)$: by Lemma 13 (a), $|E_3|$ increases by at most $|T_{k_2,y}| \cdot |KSet_1|$. Also, the edges of $T_{k_2,y}$ are necessarily formed by C-queries due to $\overline{D}$;

By the above, let $T_{CB,i}$ be the live tree in $CB(k_2)$ involved in $G_2$ processing the $i$-th 2-query from $\overline{D}$ ($i \le q$), then $|E_3|$ increases by at most $\sum_i(|T_{CB,i}|+1)\cdot|KSet_1| \le \sum_i |T_{CB,i}|\cdot|KSet_1| + q\cdot|KSet_1|$ in total; let $T'_{CB,j}$ be the live tree in $CB(k_2)$ involved in $G_2$ processing the $j$-th 4-query from $\overline{D}$, then $|E_3|$ increases by at most $\sum_j |T'_{CB,j}|\cdot|KSet_1|$. By the above and the fact that $\overline{D}$ makes at most $q$ queries to $\widetilde{C}$, $\sum_i |T_{CB,i}|+\sum_j |T'_{CB,j}| \le q$; by Lemma 16, $|KSet_1| \le q$. Therefore the bound in total is $|E_3| \le q+(q+\sum_i |T_{CB,i}|+\sum_j |T'_{CB,j}|)\cdot|KSet_1| \le 3q^2$. ◻

$|Completed|$ is finite: **T** completes at most $3q^3$ chains.

**Lemma 18.** *At the end of $\overline{D}^{G_2}$, there are at most $3q^3$ tuples of the form $(k_1, k_2, x, 4)$ in Completed. The same bound holds for $(k_1, k_2, v, 2)$.*

*Proof.* Consider $(k_1, k_2, x, 4)$ first. By Lemma 1, for each $(k_1, k_2, x, 4) \in Completed$, it holds $x \in E_4[k_2]$, $x \in E_3[k_1]^{-1}$, $k_1 \in KSet_1$ and $k_2 \in KSet_2$. Then the bound $10q^3$ follows from $|KSet_2| \le q$ and $|E_3| \le 3q^2$ (Lemmata 16 and 17). Similarly for $(k_1, k_2, v, 2)$: it holds $v \in E_2[k_2]$, $w \in E_3[k_1]$ for $w := E_2[k_2](v)$, and $k_i \in KSet_i$, and the number of such tuples is thereby at most $|KSet_2|\cdot|E_3| \le 3q^3$. ◻

The next lemma bounds the rest variables around **T**.

**Lemma 19.** *At the end of $\overline{D}^{G_2}$, $|E_2| \le 4q^3$, $|E_4| \le 4q^3$, $|E_1| \le 4q^3$, and* **T** *makes at most $16q^6$ distinct calls to $\widetilde{C}$.*Check.

*Proof.* Consider $|E_2|$ first. **T** may create a new 2-query with $dir \ne \perp$ during E2, E2$^{-1}$, ProcessNonPebCBTree, and ProcessNonPebB3Tree. But by inspection of the execution, such 2-queries are indeed due to $\overline{D}$'s 2-queries, and should be "billed" to $\overline{D}$. As $\overline{D}$ makes at most $q$ 2-queries, the number of 2-queries with $dir \ne \perp$ is at most $q$. On the other hand, **T** only creates adapted 2-queries in calls to layer-2 ProcessTree procedures, and once an adapted 2-query $(2, k_2, v, w, \perp)$ is created, a tuple $(k_1, k_2, v, 2)$ will be added to *Completed* (for some $k_1$). It's clear that two different 2-queries $(2, k_2, v, w, \perp)$ and $(2, k'_2, v', w', \perp)$ cannot correspond to the same tuple $(k_1, k_2, v, 2)$, so that the number of adapted 2-queries is at most $3q^3$ by Lemma 18. The bound in total is thereby $q + 3q^3 \le 4q^3$. The argument for $|E_4|$ is similar. The argument for $|E_1|$ is even simpler: the number of 1-queries created during E1 and E1$^{-1}$ is at most $q$ as they are attributed to $\overline{D}$'s queries, while the number of 1-queries created during layer-2 ProcessTree-calls is at most $3q^3$.

By the above, the number of *distinct* calls to $\widetilde{C}$.Check is easily bounded to $|E_1|\cdot|E_4| \le 16q^6$. ◻

The next lemma bounds $|CQueries|$.

**Lemma 20.** *At the end of a non-aborting execution $\overline{D}^{G_2}$, there is a bijection between the 4-tuples $(k_1, k_2, x, 4) \in$ Completed and the C-queries $((k_1, k_2), u, y) \in CQueries$. As a corollary, $|CQueries| \le 3q^3$ for* any $G_2$ *execution.*

*Proof.* The proof will establish two goals: first, associating a unique C-query $((k_1, k_2), u, y) \in CQueries$ to each 4-tuple $(k_1, k_2, x, 4) \in Completed$; second, associating a unique 4-tuple $(k_1, k_2, x, 4) \in Completed$ to each C-query $((k_1, k_2), u, y) \in CQueries$. The first goal is easier: by Lemma 1, for $(k_1, k_2, x, 4) \in Completed$, there is a corresponding completed path and the associated C-query is the one in this path. Furthermore, the same C-query cannot be associated to two different 4-tuples $(k_1, k_2, x, 4)$ and $(k'_1, k'_2, x', 4)$. This is indeed obvious: for $(k_1, k_2, x, 4)$ and $(k'_1, k'_2, x', 4)$, if $k_1 \ne k'_1$ or $k_2 \ne k'_2$, then the two C-tuples associated to them clearly cannot be the same; if $k_1 = k'_1$ and $k_2 = k'_2$, then it must be $x \ne x'$, and as the sets *Queries* and *CQueries* define partial blockciphers, the two C-queries $((k_1, k_2), u', y')$ and $((k_1, k_2), u', y')$ associated to the two tuples cannot meet $u = u'$.

To establish the second goal, we first show the existence of such associated 4-tuples. For the C-queries created due to **T** querying $\widetilde{C}$.C, the existence of the associated 4-tuples has been shown by Lemma 15 (a). For the C-queries created due to $\overline{D}$ querying $\widetilde{C}$.C, we have the following reasoning: as $\overline{D}$ completes all chains (cf. page 16), right after some simulator cycle in $\overline{D}^{G_2}$, there exists four queries $(1, k_1, u, v, dir_1, num_1)$, $(2, k_2, v, w, dir_2, num_2)$, $(3, k_1, w, x, dir_3, num_3)$, $(4, k_2, x, y', dir_4, num_4)$. With respect to the four queries, we have: $dir_3 = \rightarrow$ or $\leftarrow$ due to Inv0; so that by Inv1, either $num_2 > num_3$ or $num_4 > num_3$; then by Inv5, the 2- or 4-query and the 3-query are in the same completed path, and the associated 4-tuple is in *Completed*. The above show the existence of the associated 4-tuples.

We then proceed to show the uniqueness, say, a 4-tuple $(k_1, k_2, x, 4)$ cannot be associated to two different C-queries $((k_1, k_2), u, y)$ and $((k_1', k_2'), u', y')$. This is also obvious: for $((k_1, k_2), u, y)$ and $((k_1', k_2'), u', y')$, if $k_1 \neq k_1'$ or $k_2 \neq k_2'$, then the two 4-tuples associated to them clearly cannot be the same; if $k_1 = k_1'$ and $k_2 = k_2'$, then as $CTable$ defines a part of the ideal cipher $\mathbf{C}$, it must be $u \neq u' \wedge y \neq y'$ and $E_4[k_2]^{-1}(y) \neq E_4[k_2]^{-1}(y')$. These complete the main proof.

As to the corollary, the above shows $|CQueries| = |\{(k_1, k_2, x, 4)\}| \leq 3q^3$ for non-aborting $G_2$ executions. The variable $|CQueries|$ obtained in aborted $G_2$ executions can only be smaller, and hence the corollary holds. $\square$

The last lemma bounds the running time of $\mathbf{T}$.

**Lemma 21.** *In $\overline{D}^{G_2}$, the simulator $\mathbf{T}$ runs in time $O(q^7)$.*

*Proof.* As $\mathbf{T}$ receives $O(q)$ queries, there are $O(q)$ simulator cycles. We bound the time of a single cycle to $O(q^6)$ to complete the proof. Consider a simulator cycle. By inspection of the strategy (cf. page 13) and the code, we observe that the *longest possible* cycle due to $\mathrm{E4}^{-1}$ or E2 consists of the following two steps:

$$\textsc{FindPebLeafCB} \rightarrow \textsc{ProcessNonPebCBTree}/\textsc{ProcessPebCBTree}.$$

The second step ProcessNonPebCBTree or ProcessPebCBTree consists of a series of calls to layer-2 ProcessTree procedures.

On the other hand, the *longest possible* cycle due to E4 or $\mathrm{E2}^{-1}$ consists of the following two steps:

$$\textsc{FindPebLeafB3} \rightarrow \textsc{ProcessPebB3Tree},$$

and the second step ProcessPebB3Tree is further divided into two steps: first calling FindPebLeafCB, then making a series of calls to layer-2 ProcessTree procedures. The second possibility has one more step than the first one. We thereby focus on the second possibility, and bound the running time of each step as follows:

First, denote by $t_1$ the time cost of FindPebLeafB3. Then $t_1 = O(q^4)$: because by the code, the time should be $|SearchQueue| \cdot |KSet_1|$, and by Lemmata 16 and 17, $|SearchQueue| \leq |E_3| = O(q^3)$ and $|KSet_1| = O(q)$.

Second, denote by $t_2$ the time cost from the point ProcessPebB3Tree is started till the point the layer-2 ProcessTree procedure (either ProcessB3SubTree or ProcessDualTree) is called. Then $t_2 = O(q^5)$. For this, consider the process of ProcessPebB3Tree in the case $pos = left$ first, and consider the call to FindEdgeInCB (which loops for all $u \in E_1[k_1^\circ]$) and the loop right after this call (which loops for all $k_1 \in KSet_1$). The overall running time is clearly $O(|E_1|) = O(q^3)$ (Lemma 19) if FindPebLeafCB is never executed inside the latter loop. Due to the tag $Traversed$, FindPebLeafCB is executed *at most once*, so that when $pos = left$, we have $t_2 = O(q^3) + t^*$, where $t^*$ is the running time of FindPebLeafCB. By the code, $t^* = |SearchQueue| \cdot O(|E_1|)$. Furthermore, the size of a live tree (in $CB(k_2)$ for the corresponding $k_2$) extended by FindPebLeafCB is $O(q^2)$ (by Lemmata 15 (b) and 6 (c) and $|KSet_1| = O(q)$), hence $t_2 = O(q^5)$. In case of $pos = right$, the argument is similar: on one hand, the process lying between the point ProcessPebB3Tree is started and the point the layer-2 ProcessTree procedure is called costs $O(q^3)$ time; on the other hand, FindPebLeafCB is executed *at most once* due to the tag $Traversed$. Therefore, in this case, the overall time cost $t_2$ is also $O(q^5)$.

We finally show that the subsequent series of layer-2 ProcessTree calls runs in $t_3 = O(q^6)$ in total. As the size of a live tree in $CB(k_2)$ (extended by FindPebLeafCB) is $O(q^2)$ while the size of one in $B_3$ is $O(q^3)$, there are $O(q^3)$ calls to layer-2 ProcessTree procedures. By inspection of layer-2 ProcessTree procedures and noting $|KSet_1|, |KSet_2| \leq q$ and $|E_1| = O(q^3)$, the time cost of a layer-2 ProcessTree-call is dominated by the subsequent call to RecurseNew, which is $O(q^3)$. By this, the time cost $t_3$ is $O(q^6)$, and the time cost of a single cycle is obtained through $t_1 + t_2 + t_3 = O(q^6)$. $\square$

**Bounding the Abort Probability of $G_2$.** For the distinguisher $\overline{D}$ (cf. page 16) and a random tuple $(\mathbf{C}, \mathbf{E})$, the following propositions analyze the abort probability of $G_2$ (during $\overline{D}^{G_2(\mathbf{C},\mathbf{E})}$) due to each possibility.

**Proposition 3.** *The probability that $G_2$ aborts inside a call to* AddCQuery *is at most $\frac{21q^6}{2^n - 3q^3}$.*

*Proof.* By Lemma 19, it holds $RS_4 \leq |E_4| \leq 4q^3$ and $LS_1 \leq |E_1| \leq 4q^3$; by Lemma 20, it holds $LS_0 \leq |CQueries| \leq 3q^3$ and $RS_0 \leq |CQueries| \leq 3q^3$. So a single call to AddCQuery induces abortion with probability at most $\frac{4q^3+3q^3}{2^n-3q^3}$ regardless of the value of the involved parameter $dir$, and the probability in total is at most $\frac{21q^6}{2^n-3q^3}$. $\square$

**Proposition 4.** *The probability that $G_2$ aborts inside a call to* AddQuery *is at most* $\frac{32q^6}{2^n-4q^3} + \frac{21q^5}{2^n-3q^2} + \frac{8q^4}{2^n-q} + \frac{7q^4}{2^n-q}$.

*Proof.* Consider calls to AddQuery$(1, k_1, u, v, dir)$ first. Note $\mathrm{Max}\{|RS_0| + |LS_1|, |RS_1| + |LS_2|\} \leq 8q^3$ by Lemmata 19 and 20. So a single call to AddQuery$(1, k_1, u, v, dir)$ induces abortion with probability at most $\frac{8q^3}{2^n-4q^3}$ regardless of $dir$, and the bound in total is $\frac{32q^6}{2^n-4q^3}$. Similar argument establish the bound $\frac{21q^5}{2^n-3q^2}$ for calls to AddQuery$(3, k_1, w, x, dir)$.

Then, consider calls to AddQuery$(2, k_2, v, w, dir)$ with $dir = \leftarrow$ or $\rightarrow$. As noted in the proof of Lemma 19, the number of such calls is at most $q$ (rather than $|E_2|$): this means that $\mathbf{E}_2$ is queried at most $q$ times during $\overline{D}^{G_2}$. Furthermore, $\mathrm{Max}\{|RS_1| + |LS_2|, |RS_2| + |LS_3|\} \leq 8q^3$ by Lemmata 19 and 20, so calls to AddQuery$(2, k_2, v, w, dir)$ induce abortion with probability at most $q \cdot \frac{8q^3}{2^n-q} \leq \frac{8q^4}{2^n-q}$ in total. For calls to AddQuery$(4, k_2, x, y, dir)$ the case is similar although the values deviate, and the bound in total is $\frac{7q^4}{2^n-q}$. $\square$

**Proposition 5.** *The probability that $G_2$ aborts inside calls to* E2, E2$^{-1}$, E4, *and* E4$^{-1}$ *(excluding all subcalls) is 0.*

*Proof.* By design, this lemma indeed focuses on the case where $G_2$ finds more than one pebbled node in the involved live tree. If $G_2$ did not abort at some earlier point (due to early-abort conditions), then this type of abortion is not possible by Lemmata 12 (b), 13 (b), and 14 (b). $\square$

**Proposition 6.** *The probability that $G_2$ aborts inside calls to* Adapt *is 0.*

*Proof.* By Lemmata 12-14 and follows the same line as Proposition 5. $\square$

The propositions above together yield the overall abort probability of $G_2$.

**Lemma 22.** *The overall probability that $G_2$ aborts is at most* $\frac{178q^6}{2^n}$.

*Proof.* By Propositions 3-6 above, assuming $4q^3 < 2^n/2$, then the bound is

$$\frac{21q^6}{2^n - 3q^3} + \frac{32q^6}{2^n - 4q^3} + \frac{21q^5}{2^n - 3q^2} + \frac{8q^4}{2^n - q} + \frac{7q^4}{2^n - q} \leq \frac{178q^6}{2^n}.$$

Note that the abortions inside RandomAssign while outside the subsequent AddQuery (say, the abortions due to the two conditions $z' \in E_i[k]^{-1}$ and $z' \in E_i[k]$) are not included in these propositions. The reason is that the two conditions have been covered by the early-abort conditions in the subsequent AddQuery, and the probability has been accounted in the probability of those in AddQuery (Proposition 4). For clearness, consider the case of a call to RandomAssign$(1, +, k_1, u)$. Let $Sh := RS_1 \cup LS_2$. Then by noting that $v \in Sh \Rightarrow v \in E_1[k_1]^{-1}$, it holds

$$\begin{aligned} &Pr[G_2 \text{ aborts in RandomAssign or AddQuery}] \\ =&Pr[v \in E_1[k_1]^{-1} \wedge v \in Sh] + Pr[v \notin E_1[k_1]^{-1} \wedge v \in Sh] \\ =&Pr[v \in Sh] = Pr[G_2 \text{ aborts in AddQuery}]. \end{aligned}$$

The other possibilities are similar. $\square$

# D Formal Proof for Transition

Since we now have got a thorough understanding of $G_2$ from the previous section, we can complete the proofs for the transitions.

**$G_1$ and $G_2$ Behave the same: Around Check.** This subsubsection gives the transition from $G_1$ to $G_2$. As mentioned (cf. page 17), the central issue is the procedure Check, and the argument follows the idea initiated by Coron et al. [CHK+14]. As we avoid the two-sided random function used in [CHK+14], our argument is closer to the corresponding part in [GL15b]. More clearly, we first specify a bad event $\mathsf{BadCheck1}$ in $G_1$, which captures the possible differences brought in by Check. We then formally prove that for a tuple $(\mathbf{C}, \mathbf{E})$ such that $\overline{D}^{G_2(\mathbf{C},\mathbf{E})}$ does not abort (such tuples would be called *good $G_2$-tuples*), if $\mathsf{BadCheck1}$ does not happen during the $G_1$ execution $\overline{D}^{G_1(\mathbf{C},\mathbf{E})}$, then $G_1$ and $G_2$ have the same behaviors when running on this tuple.

The Event BadCheck1. Recall that the return value of a call $\widetilde{C}$.Check in $G_2$ depends on the history of $\widetilde{C}$ which has a polynomial size, whereas the return value of $\mathbf{S}$.Check in $G_1$ is completely up to $\mathbf{C}$ which indeed has an exponential size. To capture this difference, we use an event $\mathsf{BadCheck1}$:[22] for a tuple of random primitives $(\mathbf{C}, \mathbf{E})$, $\mathsf{BadCheck1}$ happens during the execution $\overline{D}^{G_1(\mathbf{C},\mathbf{S}^{\mathbf{C},\mathbf{E}})}$ if $\exists(K, u, y)$ s.t. all the following hold:

(i) $\mathbf{S}^{\mathbf{C},\mathbf{E}}$ makes a call to Check$(K, u, y)$;
(ii) $\mathbf{C}.\mathrm{C}(K, u) = y$;
(iii) Before the call in (i), at no point outside the Check-calls was $\mathbf{C}.\mathrm{C}(K, u)$ or $\mathbf{C}.\mathrm{C}^{-1}(K, y)$ called.

Behaviors of $G_1$ and $G_2$. To formally capture the behaviors of the two systems, consider the *transcripts* of queries and (random) answers appeared in the two systems, where the queries include C, $\mathrm{C}^{-1}$, E$i$, E$i^{-1}$ $(i = 1, 2, 3, 4)$, and Check; but (in $\overline{D}^{G_1}$) the queries to C made inside Check are not included. More clearly, the following queries are included in such transcripts:

(i) in $\overline{D}^{G_2}$: all the $\widetilde{C}$.C, $\widetilde{C}.\mathrm{C}^{-1}$, $\widetilde{C}$.Check, $\mathbf{E}.\mathrm{E}i$, and $\mathbf{E}.\mathrm{E}i^{-1}$ queries issued by $\overline{D}$ and $\mathbf{T}$;
(ii) in $\overline{D}^{G_1}$: all the $\mathbf{S}$.Check, $\mathbf{E}.\mathrm{E}i$, and $\mathbf{E}.\mathrm{E}i^{-1}$ queries issued by $\overline{D}$ and $\mathbf{S}$, all the queries $\mathbf{C}$.C and $\mathbf{C}.\mathrm{C}^{-1}$ issued by $\overline{D}$, and all the queries $\mathbf{C}$.C and $\mathbf{C}.\mathrm{C}^{-1}$ issued by $\mathbf{S}$ outside the Check procedure.

With the notions above, the next lemma claims that for a good $G_2$-tuple, if $\mathsf{BadCheck1}$ does not occur in $\overline{D}^{G_1(\mathbf{C},\mathbf{E})}$ for sufficiently many Check calls, then the transcripts of the two executions $\overline{D}^{G_1(\mathbf{C},\mathbf{E})}$ and $\overline{D}^{G_2(\mathbf{C},\mathbf{E})}$ will be the same, and $\overline{D}$ thereby gives the same output. Additionally, the probability is overwhelming.

**Lemma 23.** *Consider two executions $\overline{D}^{G_1(\mathbf{C},\mathbf{E})}$ and $\overline{D}^{G_2(\mathbf{C},\mathbf{E})}$ on a good $G_2$-tuple $(\mathbf{C}, \mathbf{E})$. Assume that there are $t$ calls to $\widetilde{C}$.Check during $\overline{D}^{G_2(\mathbf{C},\mathbf{E})}$. We have the following two claims:*

*(a) If $\mathsf{BadCheck1}$ does not happen in the first $t$ calls to $\mathbf{S}$.Check during $\overline{D}^{G_1(\mathbf{C},\mathbf{E})}$, then the transcripts (defined as above) of $\overline{D}^{G_1(\mathbf{C},\mathbf{E})}$ and $\overline{D}^{G_2(\mathbf{C},\mathbf{E})}$ are the same, and $\overline{D}$ gives the same output in the two executions.*
*(b) The assumption in (a) holds with probability at least $1 - 32q^6/2^n$.*

*Proof.* We first prove (a) by an induction. Assume that the answers to the previous queries equal correspondingly, and consider the next query. As both $\overline{D}$ and $\mathbf{S}/\mathbf{T}$ are deterministic, the next query will be the same. If the query is to $\mathbf{C}.\mathrm{C}/\mathbf{C}.\mathrm{C}^{-1}$ or $\widetilde{C}.\mathrm{C}/\widetilde{C}.\mathrm{C}^{-1}$, then the answers in the two executions will clearly be the same as both of them are due to the same ideal cipher $\mathbf{C}$. Ditto for the case of a query to $\mathbf{E}$. If the query is to Check, then the answers are the same by assumption of $\neg\mathsf{BadCheck1}$. By construction, $\mathbf{S}$ and $\mathbf{T}$ will then proceed to the same sequence of operations until the next query, or either $\mathbf{S}$ or $\mathbf{T}$ aborts.

We then show that neither $\mathbf{S}$ nor $\mathbf{T}$ aborts. As the tuple $(\mathbf{C}, \mathbf{E})$ in question is a good $G_2$-tuple, $\mathbf{T}$ clearly never aborts. Since we have showed that the transcripts obtained till this point are the same, each time $\mathbf{S}$ is to check an abort condition, $\mathbf{T}$ is to perform exactly the same check operation, so that $\mathbf{T}$'s non-abortion implies $\mathbf{S}$'s non-abortion. By this, $\overline{D}^{G_1(\mathbf{C},\mathbf{E})}$ and $\overline{D}^{G_2(\mathbf{C},\mathbf{E})}$ proceed to the next query and the proof proceed by induction.

The above show that the transcripts of the two executions are the same. This means that $\overline{D}$ obtains the same queries and answers in the two executions, so that $\overline{D}$ outputs the same as it is deterministic.

We then consider (b). Assume that there are $t'$ *distinct* calls to $\widetilde{C}$.Check during $\overline{D}^{G_2(\mathbf{C},\mathbf{E})}$. We calculate $Pr[\neg\mathsf{BadCheck1}]$ by the follow process: consider a call Check$(K, u, y)$ in $\overline{D}^{G_2(\mathbf{C},\mathbf{E})}$ at some point, and assume that the transcript obtained so far in $\overline{D}^{G_1(\mathbf{C},\mathbf{E})}$ and $\overline{D}^{G_2(\mathbf{C},\mathbf{E})}$ are the same. Then, by the discussions above, there is a corresponding call Check$(K, u, y)$ in $\overline{D}^{G_1(\mathbf{C},\mathbf{E})}$. Depending on the state we have the following discussions:

[22] The number 1 indicates that the event is defined for $G_1$.

– if this call appears for the first time (i.e. $\textsc{Check}(K,u,y)$ was never made before this point), then by the definition of $\mathsf{BadCheck1}$, the probability that $\mathsf{BadCheck1}$ happens with respect to this call is at most $1/(2^n - q^*)$, where $q^*$ is the total number of queries received by $\mathbf{C}$ in $\overline{D}^{G_1}$;
– if the call $\textsc{Check}(K,u,y)$ has appeared before this point, then it further consists of two subcases:
  - during the period between $\textsc{Check}(K,u,y)$ was first made and the current point, either $\mathrm{C}(K,u)$ or $\mathrm{C}^{-1}(K,y)$ was issued at some point outside $\textsc{Check}$. Then $\mathsf{BadCheck1}$ does not happen with respect to the later $\textsc{Check}(K,u,y)$ call, as it does not meet the requirements;
  - opposite to the previous subcase: neither $\mathrm{C}(K,u)$ nor $\mathrm{C}^{-1}(K,y)$ was issued (outside $\textsc{Check}$) during the period. Then $\mathsf{BadCheck1}$ does not happen with respect to the later $\textsc{Check}(K,u,y)$ call, as it did not happen with respect to the first $\textsc{Check}(K,u,y)$ call.

By the above, it already suffices to sum over the $t'$ *distinct* $\textsc{Check}$-calls. As $t' \le 16q^6$ by Lemma 19, the overall probability that $\mathsf{BadCheck1}$ occurs is at most $16q^6/(2^n - q^*)$. Assuming $q^* < 2^n/2$, then we get the bound $32q^6/2^n$. □

A good $G_2$-tuple $(\mathbf{C},\mathbf{E})$ is a *good $G_1$-tuple* if $\mathsf{BadCheck1}$ does not happen during $\overline{D}^{G_1(\mathbf{C},\mathbf{S}^{\mathbf{C},\mathbf{E}})}$. As a corollary of Lemma 23 (b), the probability that a random tuple $(\mathbf{C},\mathbf{E})$ is a good $G_1$-tuple is at least $1 - (\frac{178q^6}{2^n} + \frac{32q^6}{2^n}) \ge 1 - \frac{210q^6}{2^n}$.

**Efficiency of S.** Gathering Lemma 23 and the bounds on $\mathbf{T}$ (Lemmata 16-20) yields the bounds on the complexity of $\mathbf{S}$ (in $G_1$).

**Lemma 24.** *During a $G_1$ execution $\overline{D}^{G_1(\mathbf{C},\mathbf{S}^{\mathbf{C},\mathbf{E}})}$, with probability at least $1 - 210q^6/2^n$, $\mathbf{S}$ issues no more than $8q^4$ queries to $\mathbf{C}$ (assuming $\mathbf{S}$ avoids redundant queries), and runs in time $O(q^7)$.*

*Proof.* By Lemma 23, with probability at least $1-210q^6/2^n$ (the probability that a randomly chosen tuple $(\mathbf{C},\mathbf{E})$ is a good $G_1$-tuple), the transcripts in $\overline{D}^{G_1(\mathbf{C},\mathbf{S}^{\mathbf{C},\mathbf{E}})}$ and $\overline{D}^{G_2(\widetilde{C}^{\mathbf{C}},\mathbf{T}^{\widetilde{C}^{\mathbf{C}},\mathbf{E}})}$ are the same. Hence in $\overline{D}^{G_1(\mathbf{C},\mathbf{S}^{\mathbf{C},\mathbf{E}})}$, the bounds given in Lemma 16-20 hold with probability at least $1 - 210q^6/2^n$. As $\mathbf{S}$ issues at most $|E_1| \cdot |KSet_2| + |E_4| \cdot |KSet_1|$ queries to $\mathbf{E}$, we have the bound $8q^4$.

The running time $O(q^7)$ directly follows from Lemma 21. □

***$G_2$* and *$G_3$* Behave the same: Randomness Mapping.** This part is proved by a randomness mapping argument of [CHK+14], while the formalism is similar to [CS15]. As the beginning, with respect to $\overline{D}$, we borrow some terminology from [ABD+13] and [CS15].

First, recall the notion *good $G_1$-tuple*: $\alpha = (\mathbf{C},\mathbf{E})$ is a good $G_1$-tuple if (i) it is a good $G_2$-tuple, and (ii) $\textsc{BadCheck1}$ does not occur during $\overline{D}^{G_1(\mathbf{C},\mathbf{E})}$. Second, denote by $\mathcal{R}$ the set of all possible tuples of sets $ET$ (of $\mathbf{T}$) standing at the end of $G_2$ executions when running with good $G_1$-tuples. For a good $G_1$-tuple $\alpha = (\mathbf{C},\mathbf{E})$ and a tuple of sets $ET \in \mathcal{R}$, if the sets of $\mathbf{T}$ standing at the end of $\overline{D}^{G_2(\alpha)}$ define exactly the same values as $ET$ (i.e. if $ET'$ are the sets of $\overline{D}^{G_2(\alpha)}$, then $\forall (i,k,z), E_i[k](z) = E'_i[k](z)$), then write $\overline{D}^{G_2(\alpha)} \to ET$. Third, consider a set-tuple $ET = (E_1, E_2, E_3, E_4) \in \mathcal{R}$. For a tuple of ideal ciphers $\mathbf{E}$, if for any $(i,k,z)$ such that $z \in E_i[k]$ it holds $\mathbf{E}.\mathrm{E}i(z) = E_i[k](z)$ (note this implies that for any $(k,z')$ such that $z' \in E_i[k]^{-1}$ it holds $\mathbf{E}.\mathrm{E}i^{-1}(z') = E_i[k]^{-1}(z')$), then $\mathbf{E}$ *coincides with $ET$*, and denoted $\mathbf{E} \cong ET$.

Then the following lemma claims that the results of 4-cascade computed from $ET$ are the same as the answers given by $\widetilde{C}$. This lemma is a bit similar to Lemma 2 in [ABD+13, page 24].

**Lemma 25.** *Consider a good $G_1$-tuple $(\mathbf{C},\mathbf{E})$. At the end of the $G_2$ execution $\overline{D}^{G_2(\mathbf{C},\mathbf{E})}$, for any C-query $((k_1,k_2),u,y)$ in CQueries, there are four queries (for some $v$, $w$, and $x$) in Queries as follows:*

$$(1,k_1,u,v),(2,k_2,v,w),(3,k_1,w,x),(4,k_2,x,y).$$

*Proof.* This is a corollary of Lemma 1 (*each tuple in Completed corresponds to a completed path*) and Lemma 20 (*note that the 4-tuple $(k_1,k_2,x,4)$ associated to $((k_1,k_2),u,y)$ indeed characterizes the corresponding completed path*).

The number of adapted queries (queries with $dir = \perp$) equals the number of $\widetilde{C}^{\mathbf{C}}$'s queries to $\mathbf{C}$. To show this, we need a helper proposition.

**Proposition 7.** *Consider a good $G_1$-tuple $(\mathbf{C}, \mathbf{E})$. During $\overline{D}^{G_2(\mathbf{C},\mathbf{E})}$, it holds:*

*(a) Two tuples $(k_1, k_2, x, 4)$ and $(k_1', k_2', x', 4)$ computed in two different* safe *calls to layer-2* ProcessTree *procedures cannot be the same;*
*(b) All the calls to layer-2* ProcessTree *procedures are safe.*

*Proof.* For (a), assume otherwise, then we show that the later call (to layer-2 ProcessTree procedure) cannot be safe to establish a contradiction. Further assume that the 4-tuple computed in the earlier call is $(k_1, k_2, x, 4)$. By construction, right after this call adapts, it holds $(k_1, k_2, x, 4) \in \mathit{Completed}$, which implies the existence of the following completed path:

$$((k_1, k_2), u, y), (1, k_1, u, v), (2, k_2, v, w), (3, k_1, w, x), (4, k_2, x, y).$$

This in particular means that all of the four nodes $v$, $w$, $x$, and $y$ are pebbled after the earlier layer-2 ProcessTree-call adapts. By this, the later layer-2 ProcessTree-call cannot be safe regardless of its concrete type.

The claim (b) follows from the analysis in Lemmata 12-14. ☐

**Lemma 26.** *Consider a good $G_1$-tuple $(\mathbf{C}, \mathbf{E})$. At the end of the $G_2$ execution $\overline{D}^{G_2(\mathbf{C},\mathbf{E})}$, it holds*

$$|\{(i, k, z, z', dir) \in \mathit{Queries} : dir = \perp\}| = |\mathit{CQueries}|.$$

*Proof.* We exhibit a bijective mapping between the adapted queries and the C-queries, through the following chain:

$$\begin{aligned} &\text{C-queries } ((k_1, k_2), u, y) \in \mathit{CQueries} && \leftrightarrow \text{ 4-tuples } (k_1, k_2, x, 4) \in \mathit{Completed} \\ \leftrightarrow\ &\text{layer-2 ProcessTree-calls} && \leftrightarrow \text{ calls to Adapt/adapted queries} \end{aligned}$$

In this chain, each $\leftrightarrow$ denotes a bijection. The first bijection in this chain has been proved by Lemma 20. We proceed to prove the remaining two.

For the second bijection, note that for each layer-2 ProcessTree-call we could associate a 4-tuple $(k_1, k_2, x, 4) \in \mathit{Completed}$ (i.e. the tuple added to $\mathit{Completed}$ during its chain-completion phase); for each $(k_1, k_2, x, 4) \in \mathit{Completed}$ we could associate a unique layer-2 ProcessTree-call (i.e. the call which adds it to $\mathit{Completed}$). Moreover, by Proposition 7, two different layer-2 ProcessTree-calls in $\overline{D}^{G_2(\mathbf{C},\mathbf{E})}$ would not lead to the same 4-tuple $(k_1, k_2, x, 4)$. These establish the second bijection.

We follow the same line as above to establish the third bijection: an Adapt-call could be associated to each layer-2 ProcessTree-call; and a unique layer-2 ProcessTree-call could be associated to each Adapt-call. Furthermore, the same call to Adapt cannot be made in two different layer-2 ProcessTree-calls, otherwise $G_2$ would abort during the later one. These complete the proof.[23] ☐

If we use the values in the sets of a good $G_2$ execution as the randomness source of a $G_3$ execution, then the transcripts of queries and answers of $\overline{D}$ in these two executions are the same.

**Lemma 27.** *Let $\alpha = (\mathbf{C}, \mathbf{E})$ be a good $G_1$-tuple, and denote by $ET$ the sets of $\mathbf{T}$ standing at the end of $\overline{D}^{G_2(\alpha)}$. Then for any tuple $\mathbf{E}'$ such that $\mathbf{E}' \cong ET$, the transcripts of queries and answers of $\overline{D}$ in the two executions $\overline{D}^{G_2(\alpha)}$ and $\overline{D}^{G_3(\mathbf{E}')}$ are the same; and $\overline{D}^{G_2(\alpha)} = \overline{D}^{G_3(\mathbf{E}')}$.*

*Proof.* We use an induction similar to Lemma 23. Assume that the transcripts of $\overline{D}$ in the two executions are the same up to some point, and consider the next query. As $\overline{D}$ is deterministic, $\overline{D}$'s next queries in the two executions are the same. We prove that $\overline{D}$ obtains the same answer. For this we consider the two possibilities:

[23] The proof appears much simpler than the analogue in [ABD+13] (Lemma 4 in page 26, the proof of which takes 4 pages) despite the closeness of the overall paradigms. The reason is: in [ABD+13], adapted queries are not only created in "chain-completion phases" (which was captured by CompletePath procedures), but also in PrivateP3. Some of our bijections thereby cannot be established within a few words in the scenarios of [ABD+13].

(i) the query is to E$i$/E$i^{-1}$: then the answers are the same, since the answer obtained in $\overline{D}^{G_2(\alpha)}$ equals the value in $ET$, and $\mathbf{E}'$ *coincides with* $ET$;

(ii) the query is to C/C$^{-1}$: then due to Lemma 25 and the fact that $\mathbf{E}' \cong ET$, the answers obtained in $\overline{D}^{G_2(\alpha)}$ and $\overline{D}^{G_3(\mathbf{E}')}$ are the same.

Therefore, the answers are the same, and the two transcripts of $\overline{D}$ are the same as the induction proceeds. Since $\overline{D}$ is deterministic, $\overline{D}$ gives the same output in the two executions. □

For any $ET \in \mathcal{R}$, the probabilities of the following two events are close:

(i) a $G_2$ execution with a random tuple $(\mathbf{C}, \mathbf{E})$ generates $ET$;

(ii) a random tuple $\mathbf{E}$ coincides with $ET$.

**Lemma 28.** *For any $ET \in \mathcal{R}$, it holds*

$$\frac{Pr_{\mathbf{E}}[\mathbf{E} \cong ET]}{Pr_{\mathbf{C},\mathbf{E}}[\overline{D}^{G_2(\mathbf{C},\mathbf{E})} \to ET]} \geq 1 - \frac{9q^6}{2^n}.$$

*Proof.* Let $ET = (E_1, E_2, E_3, E_4)$. Then

$$Pr_{\mathbf{E}}[\mathbf{E} \cong ET] = \prod_{i=1}^{4} \prod_{k \in \{0,1\}^\kappa} \prod_{j=0}^{|E_i[k]|-1} \frac{1}{2^n - j}.$$

To calculate $Pr[\overline{D}^{G_2(\mathbf{C},\mathbf{E})} \to ET]$, consider a good $G_1$-tuple $\alpha' = (\mathbf{C}', \mathbf{E}')$ which satisfies $\overline{D}^{G_2(\mathbf{C}',\mathbf{E}')} \to ET$. It can be shown that $\overline{D}^{G_2(\mathbf{C},\mathbf{E})} \to ET$ if and only if the transcripts (cf. page 50) of $\overline{D}^{G_2(\mathbf{C},\mathbf{E})}$ and $\overline{D}^{G_2(\mathbf{C}',\mathbf{E}')}$ are the same (by an induction similar to that of Lemma 27) – also, the random values used during $\overline{D}^{G_2(\mathbf{C},\mathbf{E})}$ are exactly the same as those used during $\overline{D}^{G_2(\mathbf{C}',\mathbf{E}')}$. Assume that during $\overline{D}^{G_2(\mathbf{C},\mathbf{E})}$, for each $k \in \{0,1\}^\kappa$, there are $|\widetilde{E_i}[k]|$ entries in $E_i[k]$ that are defined by RandomAssign. By construction, it clearly holds $|\widetilde{E_i}[k]| = |E_i[k]|$ when $i = 1, 3$. Let $v = |CQueries|$. Then each random answer from $\widetilde{C}^{\mathbf{C}}$ is uniformly picked from a pool of size *at least* $2^n - v$, and it holds

$$Pr[\overline{D}^{G_2(\mathbf{C},\mathbf{E})} \to ET] \leq \left(\prod_{i=1,3} \prod_{k \in \{0,1\}^\kappa} \prod_{j=0}^{|E_i[k]|-1} \frac{1}{2^n - j}\right) \cdot \left(\prod_{i=2,4} \prod_{k \in \{0,1\}^\kappa} \prod_{j=0}^{|\widetilde{E_i}[k]|-1} \frac{1}{2^n - j}\right) \cdot \left(\frac{1}{2^n - v}\right)^v.$$

By Lemma 26, it holds $\sum_{k \in \{0,1\}^\kappa} |E_2[k]| + \sum_{k \in \{0,1\}^\kappa} |E_4[k]| - \sum_{k \in \{0,1\}^\kappa} |\widetilde{E_2}[k]| - \sum_{k \in \{0,1\}^\kappa} |\widetilde{E_4}[k]| = v$. Furthermore $v \leq 3q^3$ by Lemma 20, hence

$$\frac{Pr_{\mathbf{E}}[\mathbf{E} \cong ET]}{Pr_{\mathbf{C},\mathbf{C}}[\overline{D}^{G_2(\mathbf{C},\mathbf{E})} \to ET]} \geq \frac{\prod_{i=2,4} \prod_{k \in \{0,1\}^\kappa} \prod_{j=0}^{|E_i[k]|-1} \frac{1}{2^n - j}}{\left(\prod_{i=2,4} \prod_{k \in \{0,1\}^\kappa} \prod_{j=0}^{|\widetilde{E_i}[k]|-1} \frac{1}{2^n - j}\right) \cdot \left(\frac{1}{2^n - v}\right)^v} \geq \frac{\left(\frac{1}{2^n}\right)^v}{\left(\frac{1}{2^n - v}\right)^v} \geq 1 - \frac{v^2}{2^n} \geq 1 - \frac{9q^6}{2^n}.$$

as claimed. □

An implication of Lemma 27 is that the good $G_2$ executions can be partitioned with respect to the sets generated by them: for any $ET \in \mathcal{R}$ and any two tuples $(\mathbf{C}, \mathbf{E})$ and $(\mathbf{C}', \mathbf{E}')$, once $\overline{D}^{G_2(\mathbf{C},\mathbf{E})} \to ET$ and $\overline{D}^{G_2(\mathbf{C}',\mathbf{E}')} \to ET$, then $\overline{D}^{G_2(\mathbf{C},\mathbf{E})} = \overline{D}^{G_2(\mathbf{C}',\mathbf{E}')}$. With this in mind, let $\Theta_1$ be the subset of $\mathcal{R}$ such that for any tuple $(\mathbf{C}, \mathbf{E})$ such that $\overline{D}^{G_2(\mathbf{C},\mathbf{E})} \to ET \in \Theta_1$ it holds $\overline{D}^{G_2(\mathbf{C},\mathbf{E})} = 1$. Then the following inequality holds. Its interpretation is that the $G_3$ executions in which $\overline{D}$ outputs 1 can be partitioned with respect to the member of $\Theta_1$ without any "repeat count".

**Lemma 29.** $Pr_{\mathbf{E}}[\overline{D}^{G_3(\mathbf{E})} = 1] \geq \sum_{ET \in \Theta_1} Pr_{\mathbf{E}}[\mathbf{E} \cong ET]$.

*Proof.* We show that for any tuple $\mathbf{E}^*$, there exists at most one $ET \in \mathcal{R}$ s.t. $\mathbf{E}^* \cong ET$. Assume otherwise, i.e. $\exists ET' \in \mathcal{R}$ s.t. $ET \neq ET' \wedge \mathbf{E}^* \cong ET \wedge \mathbf{E}^* \cong ET'$. Assume that for two good tuples $\alpha = (\mathbf{C}, \mathbf{E})$ and $\alpha' = (\mathbf{C}', \mathbf{E}')$, it holds $\overline{D}^{G_2(\alpha)} \to ET$ and $\overline{D}^{G_2(\alpha')} \to ET'$. Then we show that the transcripts (cf. page 50) of the two executions $\overline{D}^{G_2(\alpha)}$ and $\overline{D}^{G_2(\alpha')}$ are the same, so that the two set-tuples $ET$ and $ET'$ should be the same, which is a contradiction. This is proved by an induction similar to Lemma 27: assume the transcripts obtained so far are the same and consider the next query:

(i) the query is to $\mathbf{E}/\mathbf{E}'$: the answers are the same as $\mathbf{E}.\mathrm{E}i(k,z) = E_i[k](z) = \mathbf{E}^*.\mathrm{E}i(k,z) = E_i'[k](z) = \mathbf{E}'.\mathrm{E}i(k,z)$ and $\mathbf{E}.\mathrm{E}i^{-1}(k,z) = E_i[k]^{-1}(z) = \mathbf{E}^*.\mathrm{E}i^{-1}(k,z) = E_i'[k]^{-1}(z) = \mathbf{E}'.\mathrm{E}i^{-1}(k,z)$;
(ii) the query is to $\widetilde{C}^{\mathbf{C}}/\widetilde{C}^{\mathbf{C}'}$: then by Lemma 25, the answers are the same;
(iii) the query is to Check: as the transcripts obtained so far are equal, the contents in *CTable* in the two executions are also the same, so that the answers to Check are the same.

The above establish that for any tuple $\mathbf{E}^*$, there exists at most one $ET \in \mathcal{R}$ s.t. $\mathbf{E}^* \cong ET$. After this, we have

$$Pr_{\mathbf{E}}[\overline{D}^{G_3(\mathbf{E})} = 1] \geq Pr_{\mathbf{E}}[\overline{D}^{G_3(\mathbf{E})} = 1 \wedge \exists ET \in \mathcal{R} \text{ s.t. } \mathbf{E} \cong ET] = \sum_{ET \in \Theta_1} Pr_{\mathbf{E}}[\mathbf{E} \cong ET] \text{ (by Lemma 27)}$$

as claimed. ◻

**Transition from $G_1$ to $G_3$: Linking the Three Systems.** With all the discussions and lemmata above, we complete the transition from $G_1$ to $G_3$. Note that we directly transit from $G_1$ to $G_3$. This "direct" transition argument allows avoiding counting $Pr[\overline{D}^{G_2(\mathbf{C},\mathbf{E})}$ aborts] twice.

**Lemma 30.** $|Pr_{\mathbf{E}}[\overline{D}^{G_3(CC_4^{\mathbf{E}},\mathbf{E})} = 1] - Pr_{\mathbf{C},\mathbf{E}}[\overline{D}^{G_2(\widetilde{C}^{\mathbf{C}},\mathbf{T}^{\widetilde{C}^{\mathbf{C}}},\mathbf{E})} = 1]| \leq \frac{219q^6}{2^n}$.

*Proof.* *Wlog* assume that $Pr_{\mathbf{C},\mathbf{E}}[\overline{D}^{G_1(\mathbf{C},\mathbf{E})} = 1] \geq Pr_{\mathbf{E}}[\overline{D}^{G_3(\mathbf{E})} = 1]$, then

$$\begin{aligned}
&|Pr_{\mathbf{E}}[\overline{D}^{G_3(\mathbf{E})} = 1] - Pr_{\mathbf{C},\mathbf{E}}[\overline{D}^{G_1(\mathbf{C},\mathbf{E})} = 1]| \\
\leq &\underbrace{Pr_{\mathbf{C},\mathbf{E}}[(\mathbf{C},\mathbf{E}) \text{ is not a good } G_1\text{-tuple}]}_{\leq \frac{210q^6}{2^n} \text{ (cf. page 51)}} \\
&+ \underbrace{Pr_{\mathbf{C},\mathbf{E}}[(\mathbf{C},\mathbf{E}) \text{ is a good } G_1\text{-tuple} \wedge \overline{D}^{G_1(\mathbf{C},\mathbf{E})} = 1]}_{= Pr_{\mathbf{C},\mathbf{E}}[(\mathbf{C},\mathbf{E}) \text{ is a good } G_1\text{-tuple} \wedge \overline{D}^{G_2(\mathbf{C},\mathbf{E})} = 1] \text{ (by Lemma 23)}} - Pr_{\mathbf{E}}[\overline{D}^{G_3(\mathbf{E})} = 1] \\
\leq &\frac{210q^6}{2^n} + \sum_{ET \in \Theta_1} \left(Pr_{\mathbf{C},\mathbf{E}}[\overline{D}^{G_2(\mathbf{C},\mathbf{E})} \to ET] - Pr_{\mathbf{E}}[\mathbf{E} \cong ET]\right) \text{ (by Lemma 29)} \\
\leq &\frac{210q^6}{2^n} + \sum_{ET \in \Theta_1} \frac{9q^6}{2^n} \cdot Pr_{\mathbf{C},\mathbf{E}}[\overline{D}^{G_2(\mathbf{C},\mathbf{E})} \to ET] \text{ (by Lemma 28)} \leq \frac{219q^6}{2^n}
\end{aligned}$$

as claimed. ◻