Revisiting Cascade Ciphers in Indifferentiability Setting

Chun Guo, Dongdai Lin*, and Meicheng Liu

State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences, Beijing 100093, China
{guochun,ddlin,liumeicheng}@iie.ac.cn

Abstract. Shannon defined an ideal ($\kappa$, $n$)-blockcipher as a secrecy system consisting of $2^n$ independent $n$-bit random permutations.
In this paper, we revisit the following question: in the ideal cipher model, can a cascade of several ideal ($\kappa$, $n$)-blockciphers realize 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. We give the first positive answer: for the cascade of independent ideal ($\kappa$, $n$)-blockciphers with two alternated independent keys, four stages are necessary and sufficient to realize an ideal (2$\kappa$, $n$)-blockcipher, in the sense of indifferentiability of Maurer et al. (TCC 2004). 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.
1 Introduction

A blockcipher with key length $\kappa$ and block size $n$ is a set of efficiently 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 $IC[\kappa, n]$.

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

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

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 [16,24] (this line of works will be recalled later). 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.

Our Main Question. In this paper, we consider cascade ciphers from such a different perspective—more clearly, in the indifferentiability framework of Maurer et al. [38]. Besides deepening the understanding of the power of cascade, it’s also related to 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 $2^\kappa$ $n$-bit permutations [46]; but Shannon did not provide additional knowledge of these permutations. Over the past decades, the question what set of permutations the cascade can perform received very few attention—we are only aware of the research of Even and Goldreich in 1980s, which proved that the (probably $2^\kappa$) different permutations realized by l-cascade are not independent: their behavior could be determined using only $l \cdot 2^n$ exhaustive experiments [21]. Since their negative result, a problem is whether the cascade of ideal $(\kappa, n)$-ciphers could “behave like” $2^\kappa$ independent random permutations for $l \geq 2$ (say, an ideal $(\kappa, n)$-cipher) and under which conditions could it be?

The experiments considered in [21] do not allow the adversary to access the underlying ciphers. However, we would like to consider the above problem in the ideal cipher model (ICM), as security proofs for cascade ciphers are typically made 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 [10,8]). 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 indifferentiability [38].
Note that cascade ciphers are idealized blockciphers; here we give a brief review on the indiﬀerentiability of idealized ciphers. Traditionally, blockciphers are used to ensure conﬁdentiality 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 [22] is not collision-intractable. Even if only used for encryption, merely-pseudorandom blockciphers may induce danger due to related-key attacks [6,4]. These motivated the line of works considering whether idealized blockciphers can be “as secure as” ideal ciphers—or indiﬀerentiable from ideal ciphers [3,34]. The indiﬀerentiability analysis of cascade follows this line in some sense.

Overall, motivated by the above discussion, our main question is:

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

Sub-Key Reuse, and Independent Underlying Ciphers. It has been observed by Lampe and Seurin that the cascade of two \( \text{IC}\,[\kappa, n] \) with two independent keys is not indiﬀerentiable from \( \text{IC}\,[2\kappa, n] \) [34]. Obviously, using \( l \) independent keys in \( l \)-cascade is like using a \( 2 \)-cascade; the case is a bit similar to the context of iterated Even-Mansour ciphers [3]. Therefore, the keys of the underlying ideal ciphers must be somehow reused.

To achieve key-reusing and key-length extension at the same time, 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 [7] we ﬁnd a chosen-key distinguisher on \( l \)-encryption with two alternated keys regardless of how large \( l \) is. Indeed, it is never surprising that multiple encryptions cannot 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 \( \text{CC}_l \) and let \( K = (k_1, k_2) \), then

\[
\text{CC}_l(K, x) = E_3(k_1, \ldots (E_2(k_1, E_1(k_1, x))))) \ldots
\]

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 paper thereby focuses on working out the value of \( l \) such that \( \text{CC}_l \) is indiﬀerentiable from \( \text{IC}\,[2\kappa, n] \).

Our Contribution. We give the ﬁrst positive answer to the main question: the 4-cascade \( \text{CC}_4 \) is indiﬀerentiable from \( \text{IC}\,[2\kappa, n] \). The security bound is \( O(q^8/2^n) \), and the query and time complexities of the simulator are \( O(q^4) \) and \( O(q^7) \) respectively. Besides, 2-cascade \( \text{CC}_2 \) and 3-cascade \( \text{CC}_3 \) are also considered—and are attacked, thus not indiﬀerentiable. Therefore, for cascades with alternating key schedule the number of stages \( j \) is tight.\(^1\)

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” [1]. Note that Coron et al. already provided a solution to key-length extension problem in indiﬀerentiability setting: they proved that the construction \( E(H(K), x) \) using a random oracle \( H \) and an ideal cipher \( E \) is indiﬀerentiable [13] (and the security bound \( O(q^2/2^n) \) 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 \( E(H(K), x) \) is essentially the same as \( E \) in secret-key setting.

\(^1\) The tightness does not necessarily hold for those with other key schedules. However, we have not found a schedule to cinch the positive result on 3-cascade, cf. Appendix A.
Technical Issues. Our proof is built on Andreeva et al.’s analysis of t-round Even-Mansour with Random oracle key schedule (EMR_t) [3]. We adapt their *tripwire paradigm* to build our indierentiability simulator. We also use their technique *explicitly bookkeeping* to simplify the language.\(^2\)

Compared to Andreeva et al.’s simulator for 5-round EMR, our simulator incorporates two modifications in order to correctly maintain chain completions in 4 stages. First, upon the distinguisher issuing a new query, our simulator would “patiently” find the most suitable “starting point” for the recursive 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” ideal primitive IC\(2\kappa, n\). Note that in indierentiability setting the simulator should *not* be able to access IC\(2\kappa, n\)’s entire history. To settle this, our simulator obtains some useful information about IC\(2\kappa, n\)’s history via the “check” procedure of Coron et al. [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 [14] and our fine-grained observations on our simulator and 4-cascade. For more details please jump ahead and see 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 [21,41]. As to security amplification, double-encryption was proved only slightly better than single-encryption [1], while triple-encryption was indeed better than single and double [5]. This line of researches was followed by a series of works [25,35,27], culminating with two recent ones of Dai et al. [16] and Gazi et al. [24]: the former proved tight security 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 ours 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-contruction [33], and was further generalized and analyzed in [26,35,27,24,31]. Additionally, the line of works on security amplification in the standard model was initiated by Luby and Rackoff [36] and followed by e.g. [43,37,40,39,47].

Finally, indierentiability of idealized blockciphers was initiated by Coron et al., with a proof for 14-round Feistel networks [14], which was later improved to 10 [18,15] and 8 rounds [17]. It was also extended to Even-Mansour [3,34,28] and confusion-diffusion networks [20].

Related Problems. As mentioned, for the same “tripwire configuration”, we succeed in finding a simulator-termination argument in our context of CC\(_1\), while Andreeva et al. did not do for EMR\(_4\) and turned to EMR\(_5\). It’s thus natural to ask if our termination argument could be adapted for EMR\(_4\). While there seems no obstacle, we have found a proof for the indierentiability of EMR\(_3\) [30] (which completely closes the gap between positive and negative sides in [2]). We therefore eschew the analysis of EMR\(_4\).

Organization. Section 2 serves conventions and definitions. Section 3 gives the attacks. Section 4 presents the main theorem as well as the key points of the proof. As the full proof for 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-abort, 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.

2 Preliminaries

Throughout the remaining, the \(\kappa\)-bit sub-keys are written in lower-case letters, i.e. \(k, k_1, k_2\), while the \(2\kappa\)-bit master key is interchangeably written in the capital letter \(K\) or the concatenation \((k_1, k_2)\). For simplicity,

\(^2\)One may get an illusion as follows: if we prove the indierentiability 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 \(P \circ IC_1[\kappa, n]\) is indierentiable from \(IC[\kappa, n]\), but as mentioned, \(IC_2[\kappa, n] \circ IC_1[\kappa, n]\) is *not* indierentiable from \(IC[2\kappa, n]\).
the notation $C$ refers to an ideal cipher $IC[2k,n]$, and the notation $E$ refers to a tuple of ideal ciphers $IC[k,n]$, say, $E = (E_1, \ldots, E_l)$, where the value of $l$ depends on the concrete context. We assume that the interfaces of $C$ are $C.C(K,z) := \{0,1\}^{2k} \times \{0,1\}^n \rightarrow \{0,1\}^n$, and $C.C^{-1}(K,z) := \{0,1\}^{2k} \times \{0,1\}^n \rightarrow \{0,1\}^n$, and the interfaces of $E$ are $E.E1(k,z) := \{0,1\}^{2k} \times \{0,1\}^n \rightarrow \{0,1\}^n$, $E.E1^{-1}(k,z)$, $\ldots$, $E.El(k,z)$, and $E.El^{-1}(k,z)$.

### The Cascade Cipher in Question

Given $l$ independent ideal ciphers $E = (E_1, \ldots, E_l)$ and following the above convention, the cascade cipher $CC_l^E$ considered in this work is formally written as follows (as depicted in Fig. 4):

$$CC_l^E((k_1, k_2), x) = E1(k_1, \ldots, E3(k_1, E2(k_2, E1(k_1, x))) \ldots),$$

$$(CC_l^E)^{-1}((k_1, k_2), y) = E1^{-1}(k_1, E2^{-1}(k_2, E3^{-1}(k_1, \ldots (E1^{-1}(k_1, y)) \ldots)),$$

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

#### Indifferentiability

Indifferentiability framework [38] addresses idealized constructions in settings where no underlying element (including building blocks and parameters) is secret. For concreteness, consider $CC_l^E$: a distinguisher $D^{CC_l^E,E}$ with oracle access to the cascade and the underlying ideal ciphers is trying to distinguish $CC_l^E$ from $C$. Then, a formal definition due to [34] is as follows.

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

$$|Pr[D^{CC_l^E,E} = 1] - Pr[D^{C,S^C} = 1]| \leq \varepsilon$$

Such a result means that $CC_l^E$ “behaves as” $C$, in the sense that $CC_l^E$ can safely replace $C$ whenever a moderate blow-up of the adversary’s time and memory requirements is acceptable, cf. [44,19] 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 functions [12] and permutations [14].

### 3 Attacks on $CC_2$, $CC_3$, and Multiple Encryption with Two Alernated Keys

Following the convention stipulated in Preliminaries (note an exception: for multiple encryption, the interfaces of $E$ are $E$ and $E^{-1}$), we present the attacks as follows. Note that the construction $CC_1$ is non-sense, as itself is an ideal cipher without any structure that can be studied.

#### 3.1 Distinguisher for 2-Cascade

The distinguisher $D$ for $CC_2^E$ works as follows:

(1) $D$ randomly chooses key differences $\Delta k_1, \Delta k_2 \in \{0,1\}^n \setminus \{0\}$ and $u \in \{0,1\}^n$;

(2) $D$ randomly chooses a master key $(k_1, k_2)$, and queries $w := C((k_1, k_2), u)$ and $w' := C((k_1, k_2 \oplus \Delta k_2), u)$;

(3) if $C^{-1}((k_1 \oplus \Delta k_1, k_2), w) = C^{-1}((k_1 \oplus \Delta k_1, k_2 \oplus \Delta k_2), w')$, $D$ outputs 1, otherwise outputs 0.

Denote by $v$ the unknown intermediate value $E1(k_1, u)$ (the value in gray in Fig. 1). Then clearly $C^{-1}((k_1 \oplus \Delta k_1, k_2), w) = E1^{-1}(k_1 \oplus \Delta k_1, v) = C^{-1}((k_1 \oplus \Delta k_1, k_2 \oplus \Delta k_2), w')$, as depicted in Fig. 1, and $D$ always outputs 1 when interacting with $CC_2^E$. Whereas when interacting with $IC[2k,n]$—or four 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 $2^{2n}$ 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 [21] that cascades of secret ideal ciphers with independent keys can be distinguished from longer-key ideal ciphers using exponential number of queries.
and "salvage" 3-cascade. However, we find distinguishers for a large class of such key schedules, cf. Appendix D.

3.3 Slide Attack on Multiple Encryption with Alternated Keys

The security parameter.

Fig. 1. Attack CC₂. \( k'_1 = k_1 \oplus \Delta_k, k'_2 = k_2 \oplus \Delta_k \). (Left) step 2: diverge at the 2nd stage; (Right) step 3: re-gather after inverse of the 2nd stage.

Fig. 2. Attack CC₃.

3.2 Distinguisher for 3-Cascade

The following equation is the basis of the distinguisher for \( CC_3 \) (cf. Fig. 2):

\[
CC^E_3((k'_1, k_2), E_1^{-1}(k'_1, E_1(k_1, u))) = E_3(k'_1, E_3^{-1}(k_1, CC^E_3((k_1, k_2), u))).
\]

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 \( v := E_1(k_1, u) \);
2. \( D \) randomly chooses \( k_2 \in \{0,1\}^\kappa \), and queries \( x := C((k_1, k_2), u) \);
3. \( D \) queries \( w := E_3^{-1}(k_1, x) \);
4. \( D \) randomly chooses \( k'_1 \in \{0,1\}^\kappa \setminus \{k_1\} \), and queries \( v' := E_1^{-1}(k'_1, v) \) and \( x' := E_3(k'_1, w) \);
5. \( D \) outputs 1 if \( x' = C((k'_1, k_2), u') \), and outputs 0 otherwise.

Clearly \( Pr[D^{CC_3,E} = 1] = 1 \). On the other hand, note that the value \( k_2 \) randomly chosen by \( D \) is unknown to \( \overline{S} \). Denote by HitK the event that \( \overline{S}^{CC} \) queried \( C \) on \((k'_1, k_2), u^* \) for some \( k'_1 \) and \( u^* \) during the execution \( D^{CC,\overline{S}^{CC}} \). Then, \( Pr[|x' = C((k'_1, k_2), u^*|] \leq Pr[|x' = C((k'_1, k_2), u') | \neq \text{HitK}] + Pr[|\text{HitK}] \leq 1/2^n + q_S/2^n \), so that \( D \)'s advantage is at least \( 1 - q_S/2^n - 1/2^n \). Typically, both \( \kappa \) and \( n \) are chosen to be polynomial functions of the security parameter. \( CC^E_3 \) is thereby not indistinguishable.

One may think appropriate key schedules could reduce the independence between the two halves \( k_1 \) and \( k_2 \) and "salvage" 3-cascade. However, we find distinguishers for a large class of such key schedules, cf. Appendix A. Thus achieving positive (yet non-trivial) indistinguishability 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 two alternate keys by \( ME^E_l \). Formally,

\[
ME^E_l((k_1, k_2), x) = E(k_1, \ldots (E(k_1, E(k_2, E(k_1, x)))) \ldots ),
\]

\[
(ME^E_l)^{-1}((k_1, k_2), y) = E^{-1}(k_1, E^{-1}(k_2, E^{-1}(k_1, \ldots (E^{-1}(k_1, y)))) \ldots ).
\]

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)

\[
ME^E_l((k_2, k_1), E(k_1, x)) = E(k_1, ME^E_l((k_1, k_2), x)).
\]

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 \( y := C((k_1, k_2), x) \);
2. \( D \) queries \( x' := E(k_1, x) \) and \( y' := E(k_1, y) \);
3. \( D \) outputs 1 if \( y' = C((k_2, k_1), x') \) and outputs 0 otherwise.
E-query and an i-query will sometimes be omitted, when it’s not of interest to the discussion at hand. Such tuples are called $D$ record $(i, k, z, z')$ on $(K, z)$.

However, as will be elaborated, our simulator utilizes the Check interface. Altogether, ProcessTree serves several instructive cases of interaction between the distinguisher $S$ and $4$. Informally describe a related (and hopefully easier to understand) simulator $T$. In this system, the ideal $(2\kappa, n)$-cipher accessed by $T$ offers an additional interface $CC(K, u, y)$ to $T$, which allows $T$ to know whether the query $(K, u, y)$ has appeared in the cipher’s history. To make a distinction from the normal ideal cipher, we denote this modified ideal cipher by $\hat{C}$. Note that the interface $CC$ is hidden from $D$, thus $\hat{C}$ has no difference with a normal ideal cipher in the view of $D$.

Remark 1. The intermediate scenario containing the modified ideal cipher $\hat{C}$ is motivated by Coron et al. [14]. However, as will be elaborated, our simulator utilizes the Check interface in a much more complicated way.

In the rest part, we first spend three subsections on presenting $T$: 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 $(\hat{C}, T)$, and introduces the “layer-2” ProcessTree procedures which “deal with” these cases; based on these underlying procedures, 4.3 describes how $T$ handles queries and “passes on the control” to the “right” ProcessTree procedure. Altogether, 4.2 and 4.3 form a bottom-up style overview of $T$. After these, 4.4 shows how to turn $T$ to $S$. Finally, 4.5 sketches the rest of the proof.

4.1 Simulator $T$: Basic Ideas

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

Informally speaking, $T$ internally keeps already answered queries: after $D$ querying $E_i(k, z) \to z'$, it keeps a record $(i, k, z', \to)$, where $i$ indicates the index, $(k, z')$ indicate the query and answer, and $\to$ indicates the query is a forward one; after $D$ querying $E_i^{-1}(k, z) \to z'$, it similarly keeps $(i, k, z', \to)$. The last coordinate will sometimes be omitted, when it’s not of interest to the discussion at hand. Such tuples are called $i$-queries, and an $E$-query refers to any $i$-query indifferently to the value of $i$. $T$ may call $E_i/E_i^{-1}$ 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 $T$ simply replies with the recorded answer; otherwise, $T$ may randomly sample an answer. For example, upon $D$ querying $E_1(k, z)$ such that no record of the form $(i, k, z, \cdot)$ pre-exists, $T$ randomly samples a value $z^*$ such that no record of the form $(i, k, \cdot, z')$ pre-exists,\(^3\) 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 [11] and make the randomness used by $T$ explicit through a tuple of four ideal ciphers $E = (E_1, E_2, E_3, E_4)$. This means if $T$ needs to assign a random answer $z'$ to $E_1(k, z)$, $T$ queries $E$ and set $z' := E.E_i(k, z)$.\(^4\) We denote by $T^C.E$ the simulator accessing $E$. However, for convenience, we keep saying “randomly sample” to refer to $T$’s such actions.

$T$ cannot answer all queries via randomly sampling, otherwise $T$ will behave as four ideal ciphers independent of $C$, whereas we need $(C, T)$ to behave as $(CC_4, E)$, i.e. answers from $T$ should depend on $C$. To this end, the basic idea is Coron et al.’s simulation via chain completion technique [14], which has been a routine for indistinguishability proof of idealized blockciphers.

**Chain Completion—Tripwire Paradigm.** We stipulate some terminology first. A triple $((k_1, k_2), u, y)$ such that $C((k_1, k_2), u) \rightarrow y$ or $C^{-1}((k_1, k_2), y) \rightarrow u$ has appeared is called a $C$-query (such query may be made by $D$ or $T$, 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 $(CC_4, E)$, the answers given by $CC_4$ and $E$ always form such completed chains. E.g. if $D$ asks $u \xrightarrow{E.E_{k_1}^1} v \xrightarrow{E.E_{k_2}^2} w \xrightarrow{E.E_{k_3}^3} x \xrightarrow{E.E_{k_4}^4} y$, then it must hold $CC_4.C((k_1, k_2), u) = y$. This forms the intuition of chain-completion technique: to generate similar interactions, $T$ clearly needs to make its simulated answers form similar completed chains with $C$’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, $T$ takes such adjacent queries as “partial chains”, detects them, and pre-emptively completes them to completed chains. For example, after $D$ querying $E_1(k_1, u) \rightarrow v$ and $E_2(k_2, v) \rightarrow w$, $T$ detects two adjacent $(1, k_1, u, v)$ and $(2, k_2, v, w)$. A possible strategy for $T$ is to first internally call $E_3(k_1, w) \rightarrow x$ (with $x$ being a newly sampled random value) and create $(3, k_1, w, x, \rightarrow)$, then query $C.C^{-1}((k_1, k_2), u) \rightarrow y$. The second step ensures the existence of the $C$-query $((k_1, k_2), u, v)$. Now the only missing query of the chain $y - u - v - w - x$ is a 4-query $(4, k_2, x, y)$; $T$ therefore create such a record $(4, k_2, x, y, \bot)$ to adapt and “close the cycle”. The last coordinate $\bot$ of this record indicates that it’s a query created for adaptation, henceforth referred to as adapted queries.

The queries newly created during chain-completions (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. $T$ thus should devote to a recursive chain completion process; following [3], we call it a chain reaction.

If $T$ completes chains for every encountered adjacent pair, then it may complete infinitely many chains, which is not acceptable. However, if $T$ ignores too many adjacent pairs, then $D$ may be able to bypass $T$’s chain-detection conditions and “trap” $T$ in an over-constrained situation, e.g. trying to create two contradictory records $(4, k_2, x, y, \bot)$ and $(4, k_2, x, y', \bot)$ to adapt two chains. To ensure $T$ 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 by Andreeva et al. for EMR [3]. Thus we take the tripwire paradigm introduced by Andreeva et al.

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 $T$ 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 $T$ 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)\(^3\)

\(^3\) So that the simulated $i$-queries are consistent with a $(\kappa, n)$-blockcipher.

\(^4\) As argued by Andreeva et al. [3,11], using such explicit randomness is equivalent to lazily sampling enough randomness at the beginning of the experiment.
the C-query \(((k_1, k_2), u, y)\) exists in the history, i.e. \(\tilde{C}.\text{CHECK}((k_1, k_2), u, y)\) returns true. Vice versa for \((4, 1)\).
Whenever a tripwire is triggered (possibly by new adapted queries), \(T\) recurses to complete all the relevant chains.

For \(CC_4\), we adopt the following asymmetric configuration, cf. Fig. 4:

\[(1, 2), (3, 2), (3, 4), (1, 4).\]

This means only the simulated \(E_2\) and \(E_4\) serve as “beacons” for chain detection. As mentioned in Introduction, this configuration was designed for EMR_4, yet abandoned due to the lack of a termination argument [3]. Whereas in our context we succeed in this task and finally complete a proof for \(CC_4\).

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 \((1, 2), (3, 2), (3, 4), and (1, 4);\) note that the last one crosses \(\tilde{C}\).

The simulated \(E_2\) and \(E_4\) also facilitate adaptations, i.e. when completing chains, \(T\) always “closes cycles” by creating adapted 2- and 4-queries. In this way, the simulated \(E_1\) and \(E_3\) 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 \(\tilde{C}\) are random in the view of \(D\) and \(T\), thus each C-query that appears during the interaction also has at least one “random endpoint”. By the above, \(E_1\) along with \(\tilde{C}\) form the first “random zone”, while \(E_3\) acts as the second one, cf. Fig. 5. Reserving two never adapted stages is indeed inspired by [3], 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 \(T\) upon detecting new tripwires. As each query in the random zones has a random endpoint (cf. the previous subsection), these structures possess special properties, which enables \(T\) to find the “most appropriate starting point” of the chain reaction and enforce the best order in which paths are completed.\(^5\) 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 \((\tilde{C}, T)\) 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 these 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 \(\{0, 1\}^n\) 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

\(^5\) Cf. [17] for the importance of such chain-completion order (in another context).
3-query. A sequence of 3-queries thus may form directed paths in $B_3$, e.g. $w_1 \xrightarrow{E_3^{-1}k_1} x_1 \xrightarrow{E_3^{-1}k_2} w_2 \xrightarrow{E_3^{-1}} x_2$. If the answer of a later query collides with a pre-existing value, e.g. $E_3^{-1}(k_1^3, x_2) \rightarrow w_3 = w_1$, 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, with high probability (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 (e.g. see Case 1 below). $D$ may use different query sequences to create various trees and force $T$ to tackle; we serve four 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 possesses more novelties, we consider examples around it. Assuming four edges $(k_1^3, y_1, v_1)$, $(k_1^3, y_2, v_1)$, $(k_1^3, y_2, v_3)$, and $(k_1^3, y_2, v_3)$, and $D$ querying $E_2(k_2, v_1)$, cf. Fig. 6 (left). Suppose $T$ randomly samples $w_1$ as the answer and creates $(2, k_2, v_1, w_1, \rightarrow)$, a new 2-query. This leads $T$ to detecting two $(1, 2)$-tripwires due to $(2, k_2, v_1, w_1)$ and $(1, k_1^3, u_1, v_1)$ and $(1, k_1^3, u_2, v_1)$, completing the two associated chains one-by-one and creating two adapted 4-queries $(4, k_2, x_1, y_1, \bot)$ and $(4, k_2, x_2, y_2, \bot)$. $T$ then notices that $(4, k_2, x_2, y_2)$ and $(1, k_1^3, u_3, v_2)$ and $(1, k_1^3, u_4, v_3)$ 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 “leaves” $y_1$, $v_2$, and $v_3$. 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 [3]. 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 $v_3$ being its 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 $E_2(k_2, v_1)$. The keys $k_1^3$, 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 labels $y_1$, 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).](image)

**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 $E_2(k_2, v_1)$ which sets off new $(1, 2)$-tripwires, 

---

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

---

10
if \( T \) 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 would find \( v_3 \) already pebbled. This means \( T \) finds a chain \( x_2 \rightarrow y_2 \rightarrow u_4 \rightarrow v_3 \rightarrow w_3 \) that can only be adapted at \( E_3 \), cf. Fig. 7 (left). So \( T \) has to either admit a failure, or destroy the “randomness” of \( E_3 \)—which would significantly complicate the proof.

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. \cite{12} for \( \text{EMR}_2 \). 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 our \( T \) works in a more constrained setting of 4 stages and cannot follow this strategy.

\[ \text{Fig. 7. Figures for Case 2. The pre-existing query } (2, k_2, v_3, u_3) \text{ is indicated by the black bold undirected edge. (Left) the unsuccessful trial: } S \text{ has to adapt to } E_3; \text{ (Right) a more appropriate operation sequence leads to success.} \]

However, if \( T \) could be a bit more patient, then it would easily overcome the above dilemma: after \( T \) notices that the last query \( E_2(k_2, v_1) \) 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 \( Li(k_3, v_1) \) (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 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 \( E_2(k_2, v_1) \).

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

\textbf{Traversing in CB: FindPebLeafCB.} The access to the whole history of \( \hat{C} \) seems necessary for \( T \) to construct the graph \( CB(k_2) \) and then traverse it. However, \( T \) can only call \( \hat{C}.\text{Check} \) to verify the existence of certain C-queries. Thus we have to implement \text{FindPebLeafCB} based on \text{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 \( k_1 \) and \( v \), then the corresponding 1-query \((1, k_1, u, v)\) necessarily exists. Therefore, \( T \) could call \( \hat{C}.\text{Check}((k_1, k_2), u, y) \) 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 \( k_1 \) and \( v \), 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 \( \text{Check}((k_1, k_2), u, y) \). However, we observe that after “grasping” the 1-query \((1, k_1, u, v)\), \( T \) could simply call queries \( \hat{C}.\text{Check}((k_1, k_2), u) \rightarrow y \) 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 right-shore node \( v \), \( T \) 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 \text{FindPebLeafCB}.

\textbf{ProcessCBSubTree Procedure.} We first give some insights on \text{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 \( \text{ProcessCBSSubTree} \). More clearly, to start a chain reaction as above we let \( T \) make a call to \( \text{ProcessCBSSubTree}(k_1^4, k_2, y_2, v_3, \text{left}) \). 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 \text{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 live tree is in the left shore of \( CB(k_2) \). This \( \text{ProcessCBSSubTree} \)-call would have two steps:

1. complete the chain specified by its arguments. In the example the chain is \( y_2 - u_4 - v_3 - w_3 \), and \( \text{ProcessCBSSubTree} \) 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 \((4, k_2, x_2, y_2^3)\) for some \( y_2^3 \). 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. In consequence, \( D \) would know it is interacting with the simulated world. We thus need to prove that the chance of such an event is negligible. 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 to \( \text{ProcessCBSSubTree}(k_1^4, k_2, y_2, v_1, \text{right}) \) and \( \text{ProcessCBSSubTree}(k_1^4, k_2, y_2, v_2, \text{right}) \).

   Note that the involved non-pebbled nodes \( v_1 \) and \( v_2 \) are in the right shore, as specified by the fifth argument of these sub-calls.

The \( \text{ProcessCBSSubTree} \)-calls with \( \text{right} \) 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 wished.

![Fig. 8. Illustration for the recursive process. The white circulars indicate pebbled nodes, while the 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 \( Li(k_2, v_2) \). For cleaness, the edges of \( Li(k_2, v_1) \) are in magenta, while those of \( Li(k_2, v_2) \) are in blue.](image)

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 pretty easy to traverse in \( B_3 \), and this is implemented as a procedure \( \text{FindPebLeafB3} \). On the other hand, the chain reactions around \( B_3 \) is performed by a procedure \( \text{ProcessB3SubTree} \) via recursion (similarly to \( \text{ProcessCBSSubTree} \)).

**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 case, assuming the tree \( y_1 - v_1 - y_2 - (v_3, v_3) \) 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 \( v_3 \). Fig. 9 (left); if \( T \) starts from \( v_3 \) then it will fail when reaching \( y_1 \). 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 [3]: 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.

Case 4. Dual-tree: two trees are involved; and the procedure $\text{ProcessDualTree}$. 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 $E2(k_2, v) \rightarrow w$, then makes $E1^{-1}(k_1, v) \rightarrow u$, $C((k_1, k_2), u) \rightarrow y$, $E3(k_1, w) \rightarrow x$, 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. $E4(k_2, x)$ 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 $\text{ProcessDualTree}(k_1, k_2, x, y, 4)$ (the fifth argument indicates that the “missing query” is a 4-query, as mentioned), which creates the adapted query $(4, k_2, x, y, 1)$, cf. Fig. 10 (left).

After this $T$ (more precisely, the $\text{ProcessDualTree}$-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 $(3, k_1^2, w^1, x)$ in $B_3$, if the edge $(k_1^2, y, v^1)$ already exists in $CB(k_2)$, then along with $(4, k_2, x, y)$ they indeed form a 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 $\text{ProcessDualTree}(k_1^2, k_2, v^1, w^1, 2)$ 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 $\text{ProcessB3SubTree}(k_1^2, k_2, w^2, x, \text{left})$;

(iii) on the other hand, for an edge $(k_1^2, y, v^3)$ in $CB(k_2)$, if there does not exist any edge of the form $(3, k_1^2, \cdot, x)$ in $B_3$ (cf. the green subtree in Fig. 10 (right)), then $T$ makes a call to $\text{ProcessCBSBSubTree}(k_1^2, k_2, y, v^3, \text{right})$ to handle the live tree $Li(k_2, v^3)$.

The $\text{ProcessDualTree}$-calls with 2 as the fifth argument runs similarly. For elaboration consider the mentioned call to $\text{ProcessDualTree}(k_1^2, k_2, v^1, w^1, 2)$: it first creates $(2, k_2, v^1, w^1, \perp)$ and then makes several sub-calls similarly as $\text{ProcessDualTree}(k_1, k_2, x, y, 4)$. 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 $E4(k_2, x)$.

In the rest of the paper we would say “layer-2 $\text{ProcessTree}$-procedure” to indifferently refer to $\text{ProcessB3SubTree}$, $\text{ProcessCBSBSubTree}$, or $\text{ProcessDualTree}$. For the sake of page limits, we defer the technical details of these procedures to the pseudocode in 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 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 case the situation fits into, and call the right layer-2 ProcessTree procedure.

New Query E2^−1(k2, w) and E4(k2, x). Consider E2^−1(k2, w) first. Upon such a new query, if there is no pre-existing 3-query of the form (3, k1, w, ·), then no (3, 2)-tripwire would be set off, and T simply randomly samples v and creates a record (2, k2, v, w, ←). If there does pre-exist 3-queries (3, k1, w, ·), then T calls FindPebLeafB3 to traverse in Li(k2, w) (in B3) and check the number of leaves pebbled under k2, and reacts depending on the case:

1. If there is no pebbled leaf in Li(k2, 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, k2, v, w, ←) with randomly sampled v. Note that this makes w pebbled, so that for each pre-existing 3-query (3, k1, w, x), Li(k2, x) becomes a live tree with w as the unique pebbled leaf. Moreover, since v is newly sampled, w.h.p. v is not adjacent to any pre-existing edge in CB(k2). This means for each such (3, k1, w, x), Li(k2, x) is not “involved in” any dual-tree, thus T makes a call to ProcessB3SubTree(k1, k2, w, x, right) to “process” Li(k2, x) by recursion (as sketched before, page 12).

2. If there is exactly one pebbled leaf (denoted z^o), then FindPebLeafB3 returns the edge in Li(k2, x) that contains z^o. Briefly speaking, if T finds Li(k2, x) “involved in” a dual-tree, then it’s in Case 4 (page 13), and it calls ProcessDualTree; otherwise it’s in Case 2 (page 10), and it calls ProcessB3SubTree to deal with Li(k2, x) alone. For clearance, according to which shore z^o is in, we distinguish two cases:

   - z^o is in the left shore of B3. Rewriting z^o as w^o. Assume that the edge in Li(k2, x) that contains w^o is (3, k2^o, w^o, x^o), while the 2-query attached to w^o is (2, k2, v^o, w^o). Then:
     - if there’s no pre-existing 1-query of the form (1, k^o, v^o), then Li(k2, x^o) is not “involved in” any dual-tree. Thus it fits into Case 2, and T simply calls ProcessB3SubTree(k^o, k2, w^o, x^o, right) to handle Li(k2, x) alone.
     - if there’s a 1-query (1, k^o, u^o, v^o) for some u^o, then T makes a query C.C((k^o, k2), u^o) → y^o. At this point the (possibly new) node y^o would likely be non-pebbled, and thus the live tree Li(k2, y^o) makes sense. As Li(k2, y^o) and Li(k2, x^o) are linked by (2, k2, v^o, w^o), it now fits into Case 4. Therefore, T calls ProcessDualTree(k^o, k2, x^o, y^o, 4) to handle.

   - z^o is in the right shore of B3. Rewriting z^o as x^o, and assume that the edge in Li(k2, x) that contains x^o is (3, k2^o, w^o, x^o) while the 4-query attached to x^o is (4, k2, x^o, y^o). In this case, T calls Check((k^o, k2), u^o, y^o) for each pre-existing 1-query (1, k^o, u^o, v^o) to determine whether the edge (k^o, y^o, v^o) pre-exists in CB(k2). Depending on the result:
     - if none of the Check-calls return true, then there’s no edge of the form (k^o, y^o, ·) in CB(k2). Thus it fits into Case 2 and T calls ProcessB3SubTree(k^o, k2, w^o, x^o, left) to tackle.
     - otherwise it fits into Case 4, T finds the edge (k^o, y^o, v^o) and then calls ProcessDualTree(k^o, k2, v^o, w^o, 2) to deal with the dual-tree formed by Li(k2, v^o) and Li(k2, w^o).
(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-procedures return 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 $E_2^{-1}(k_2, v)$.

Upon a new forward 4-query $E_4(k_2, x)$, 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 FindPebLeafCB to traverse in $Li(k_2, x)$. The subsequent process is similar as that around new $E_2^{-1}(k_2, w)$ and is thus omitted.

New Query $E_4^{-1}(k_2, y)$ and $E_2(k_2, v)$. Consider a new query $E_4^{-1}(k_2, y)$ first, upon which $T$ first determines whether it sets off new (1, 4)-tripwires, by calling 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 ProcessCBSUBTree($k_1, k_2, v, y, right$) for each of them.

(2) If there is one pebbled leaf (denoted $z^*$), then FindPebLeafCB returns the edge in $Li(k_2, x)$ that contains $z^*$. 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 $z^*$ in the left shore of $CB(k_2)$, and rewrite $z^*$ as $y^*$ (for cleanness, the symmetrical case is omitted here). Assume that the edge in $Li(k_2, y)$ that contains $y^*$ is $(k_2^*, y^*, v^*)$ while the 4-query attached to $y^*$ is $(4, k_2^*, x^*, y^*)$. Then:

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

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

$T$ answers the original query $E_4^{-1}(k_2, y)$ with the recorded $x$ once the tree processing procedures return without abortion.

Finally, upon a new query $E_2(k_2, v)$, 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 $E_4^{-1}(k_2, y)$, thus omitted.

4.4 Obtaining $S$ from $T$

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

(i) $S^C$ has an additional procedure $S^C$.Check($K, u, y$), which makes a query to $C, C(K, u)$, and returns true if $C, C(K, u) = y$;
(ii) Each time $T^C$ calls $\tilde{C}, \text{Check}(K, u, y)$, $S^C$ calls $S^C, \text{Check}(K, u, y)$ instead.

The idea dates back to Coron et al. [14]. Briefly speaking, if $(K, u, y)$ really appeared in the history of $C$, then clearly $C, C(K, u) = y$ and $S^C, \text{Check}(K, u, y)$ returns true; otherwise, as long as Check is called polynomial times, it holds $Pr[C, C(K, u) = y] = O(\frac{1}{n^2})$, and thus w.h.p. $S^C, \text{Check}(K, u, y)$ does return false. Thus this modification is unlikely to cause essential difference. However, in contrast to $T$, the obtained $S$ is a “legal” simulator for $C_G$.

Remark 2. Access to the entire history of $C$ seems necessary for $S^C$ 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 $T^C$ 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 $C$ and $S$, and by $G_3$ the real system formed by $CC_4$ and $E$. Then we need to show the following two claims for any computationally unbounded distinguisher $D$:

(i) $G_1(C, SC)$ and $G_3(CC^E, E)$ are indistinguishable for $D$.

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

The aforementioned intermediate system $G_2$ is formed by $\tilde{C}$ and $T$. As sketched in subsection 4.4, $G_1$ and $G_2$ are indistinguishable, if $T$ makes polynomial number of calls to $\tilde{C}$.Check. On the other hand, the indistinguishability of $G_2$ and $G_3$ is proved via a randomness mapping argument, which mainly requires proving $T$ always succeeds in adapting chains/never aborts due to adaptations. Thus the crux is to analyze $T$ (or $G_2$) to bound the complexity (i.e. termination argument) as well as the abort probability of $T$.

The next subsection introduces distinguisher which completes all chains, a standard approach to indistinguishability proofs for idealized blockciphers. The remaining three subsections sketches respectively the termination argument for $T$, calculating the abort probability for $T$, and the indistinguishability of systems. Moreover, the formal proof for $T$’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 $C((k_1, k_2), u) \rightarrow y$ or $C^{-1}((k_1, k_2), y) \rightarrow w$, $\overline{D}$ sequentially queries $E1(k_1, u) \rightarrow v$, $E2(k_2, v) \rightarrow w$, $E3(k_1, w) \rightarrow x$, and $E4(k_2, x) \rightarrow y$. 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 $\overline{D}^{G_1}$ 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;
- the number of distinct keys $k_1$ contained in $\overline{D}$’s queries is the same as $D$, i.e. at most $q$. The same bound holds for $k_2$.

These observations will help 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 $T$ 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 core 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. [34]). 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 $T$.

We then provide a more detailed overview as follows. First, $T$ itself never brings in “new” $k_1$ values. Thus $|KSet_1|$ equals the number of distinct keys $k_1$ contained in $\overline{D}$’s queries, which, as analyzed in the previous subsection, does not exceed $q$. Similarly, $|KSet_2| \leq q$.

We then show $|E_3| \leq 3q^2$. When a chain reaction is “burning through” a tree in $B_i$, 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 $T_{CB}^{i}$ the $i$-th involved live tree in $CB(k_2)$, by $|T_{CB}^{i}|$ the number of edges in $T_{CB}^{i}$, and by $Rv(T_{CB}^{i})$ the number of right-shore vertexes in $T_{CB}^{i}$. Then $T_{CB}^{i}$ corresponds to $T$ completing at most $|KSet_1| \cdot Rv(T_{CB}^{i})$ chains, which enlarges $|E_3|$ by at most $|KSet_1| \cdot Rv(T_{CB}^{i})$: essentially, this is because during the chain reaction, for each right-shore node $v$ in $T_{CB}^{i}$, $T$ may encounter at most $|KSet_1| \cdot (1, 2)$-tripwires, and complete the same amount of chains.

16
Chain reactions around $CB(k_2)$ may be the consequence of $D$ issuing 2- or 4-queries. For a live tree $T_{CB}$ involved by $D$ querying $E_2(k_2, v)$, it holds $Re(T_{CB}) \leq |T_{CB}| + 1$; for the other possibilities it holds $Re(T_{CB}) \leq |T_{CB}|$. As $D$ makes at most $q$ queries to $E_2$, chain reactions increase $|E_3|$ by at most $(q + \sum_i |T_{CB}^i|) \cdot |KSet_1|$. The edges in $T_{CB}$ are necessarily due to $D$ querying $\tilde{C}$ and thus $\sum_i |T_{CB}^i| \leq q$: 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 enlarges $|E_3|$ by at most $2q^2$. This plus the possible $q$-queries from $D$ yields $|E_3| \leq 3q^2$.

Now, as every completed chain corresponds to a unique pair of entries in $E_3$ and $KSet_2$, $T$ completes at most $|KSet_2| \cdot |E_3| \leq 3q^3$ chains. As a consequence, for $i = 1, 2, 4$, the number of $i$-queries internally created by $T$ is at most $3q^3$, which plus the $q$-queries from $D$ yields $|E_1|, |E_2|, |E_4| \leq 4q^3$. These further imply $T$ makes at most $|E_1| \cdot |E_4| \leq 16q^6$ distinct calls to $\tilde{C}.Check$.

**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 a new random value $z$ “appears” (either sampled by $T$ or given by $\tilde{C}$), if $z$ has appeared at some place in the history, then $G_2$ aborts. We call $G_2$’s abort due to these conditions early-abortions. With the bounds obtained before, the probability of early-abortions is upper bounded to $178q^6/2^n$.

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-abortions, 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. This is essentially due to the fact that each processed live tree has exactly one pebbled leaf, and the chain reaction “burns” from the pebbled leaf to the other nodes.

Thus by induction, all calls to layer-2 ProcessTree procedures are safe. This implies adaptations never cause $G_2$ abort, and $Pr[D^{G_2} \text{ aborts}]$ equals the probability of early-abortions, which has been proved at most $178q^6/2^n$.

**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 [14]: essentially, $D^{G_1}$ and $D^{G_2}$ 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 $D^{G_1}$ and $D^{G_2}$ deviate is at most $178q^6/2^n + 2 \cdot 16q^6/2^n \leq 210q^6/2^n$. This also shows that with probability at least $1 - 210q^6/2^n$, the query and time complexity of $S$ in $G_1$ do not exceed $8q^4$ and $O(q^7)$ respectively.

To prove $G_2$ and $G_3$ indistinguishable we take the randomness mapping argument [14]. In fact, we use an argument to link $G_1$, $G_2$, and $G_3$ together to bound $D$’s advantage in distinguishing $G_1$ and $G_3$ without appealing to the hybrid argument: $|Pr[D^{G_1} = 1] - Pr[D^{G_3} = 1]| \leq 219q^6/2^n$.

**Acknowledgements**

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

\footnote{This is because $E_2(k_2, v)$ itself identifies a right-shore node $v$.}
- 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 thank the committee of Eurocrypt 2017 for their invaluable suggestions. We also thank Yu Yu for his help during the past rebuttal phase.

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


27. Gazi, P.: Plain versus Randomized Cascading-Based Key-Length Extension for Block Ciphers. In: Canetti and Garay [9], pp. 551–570.


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 \(CC_3\) (subsection 3.2). To make a distinction, we denote this scheme by \(CC_3[k_1, k_2, k_1 \oplus k_2]\). 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 \(E_1^{-1}(k_1, v) \rightarrow u_1\) and \(E_1^{-1}(k'_1, v) \rightarrow u_2\);
3. Makes eight C-queries:
   \[
   u_1 \xrightarrow{C(k_1, k_2)} C^{-1}(k_1 \oplus k_2 \oplus k'_2, k_2) \xrightarrow{C(k_1 \oplus k_2 \oplus k'_2, k_2)} C^{-1}(k_1, k'_2) \rightarrow u_1',
   \]
   and
   \[
   u_1' \xrightarrow{C(k'_1, k_2)} C^{-1}(k'_1 \oplus k_2 \oplus k'_2, k_2) \xrightarrow{C(k_1 \oplus k_2 \oplus k'_2, k_2)} C^{-1}(k'_1, k'_2) \rightarrow u_2';
   \]
4. If \(E_1(k_1, u_1') = E_1(k'_1, u_2')\) then outputs 1, else outputs 0.

When interacting with \(CC_3[k_1, k_2, k_1 \oplus k_2]\), \(D\) 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. Therefore, no simulator could withstand this distinguisher.

![Fig. 11. Related-key Boomerang Structure in \(CC_3[k_1, k_2, k_1 \oplus k_2]\).](image1)

\[K_1 = \pi_1^{-1}(k_1^1, k_2^1)\]
\[\overline{K_1} = \pi_1^{-1}(\overline{k_1^1}, k_2^1)\]

![Fig. 12. Related-key Boomerang Structure in the general 3-cascade.](image2)
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 $k_1^1, k_2^1, \ldots, k_5^3$;
2. Randomly chooses 2 distinct round-keys $k_1^1$ and $k_1^3$ and $n$-bit value $v$;
3. Queries $E_1^{-1}(k_1^1, v) \rightarrow u_1$ and $E_1^{-1}(k_1^3, v) \rightarrow \overline{u}_1$;
4. For $i = 1, 2, 3, 4$, repeats the following operations:
   a. Computes $K_i \leftarrow \pi_1^{-1}(k_1^i, k_2^i)$, $K_i^2 \leftarrow \pi_1^{-1}(k_1^i, k_2^i)$, $(k_3^i, k_1^i) \leftarrow \pi_3(K_i)$, and $(\overline{k}_1^i, \overline{k}_1^i) \leftarrow \pi_3(\overline{K}_i)$;
   b. Computes $K_i^3 \leftarrow \pi_3^{-1}(k_1^{i+1}, k_1^i)$, $K_i^4 \leftarrow \pi_3^{-1}(k_1^{i+1}, \overline{k}_1^i)$, $(k_1^{i+1}_2, k_1^{i+1}) \leftarrow \pi_3(K_i^2)$, and $(\overline{k}_1^{i+1}, \overline{k}_1^{i+1}) \leftarrow \pi_3(\overline{K}_i^2)$;
   c. Queries $u_i \xrightarrow{c(K_i)} c^{-1}(K_i^2) \rightarrow u_{i+1}$ and $\overline{u}_i \xrightarrow{c(\overline{K}_i)} c^{-1}(\overline{K}_i^2) \rightarrow \overline{u}_{i+1}$;
5. Note that $u_5$ and $\overline{u}_5$ have been obtained. If $E_1^{-1}(\overline{k}_1^1, E_1(k_1^3, u_3)) = \overline{u}_5$ then outputs 1, and 0 otherwise.

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, $k_1^i, k_1^i, k_2^i$, and $\overline{k}_1^i$). 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 $z' := E.E_i(k, z)$. 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 $\{(k, z, z') : 3|dir, num| s.t. (i, k, z, z', dir, num) \in Queries\}$, 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 $E$, and indeed always define a partial cipher. However, since there exist adapted queries which are seldom consistent with $E$, the situation may be broken in the following two cases:

(i) When $S$ obtains a new random value $z$ from $E$, $z$ may collide with a value of an adapted query. For example, when $S$ obtains $w := E.E_2(k_2, v)$ and tries to create $(2, k_2, v, w, \leftarrow)$, there may already exists a record $(2, k_2, v', w, \leftarrow)$ for some $v' \neq v$;
(ii) When $S$ 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 $S$ 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 $\{z : 3z' z.s.t. (k, z, z') \in E_i\}$ respectively, and write $E_i[k](z)$ (for the unique $z'$ s.t. $(k, z', z') \in E_i$) for the (unique) corresponding $z'$ (resp.) when $z \in E_i[k]$ (resp.) Note that by the above mechanism, $S$ 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 $S$. We let $S$ maintain some additional sets. First is a set Completed which is used to keep a record of all the paths $S$ 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, $S$ would frequently check conditions of the form "$3k s.t. z \in E_i[k]/E_i[k]^{-1}$" for some $z$. To simplify the code, we let $S$ 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 $S$.

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

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

In detail, we let $\overline{C}$ maintain a set $CQueries$ to keep its query history. Once receiving a query $C(K, u)$, $\overline{C}$ takes $y := C.C(K, u)$ as the answer and adds a record $((k_1, k_2), u, y, \rightarrow, qnum)$ via a procedure $AddCQuery$—here $CQueries$ and $T$.Queries share the same counter $qnum$, which is made global in $G_2$; and upon a query $C^{-1}(K, y)$, $\overline{C}$ answers with $u := C.C^{-1}(K, y)$ and adds a record $((k_1, k_2), u, y, \leftarrow, qnum)$ via $AddCQuery$. As all the tuples in $CQueries$ are consistent with $C$, $CQueries$ always defines a partial blockcipher. The set $CQueries$ and $T$’s set Queries are used for the explicitly bookkeeping mechanism, which is lifted from $[2]$ to simplify the arguments.

To simplify the language we use notations similar to Queries: we write $CTable[K]$ for the set
\[ \{u : \exists y, \text{dir}, \text{num} \text{ s.t. } (K, u, y, \text{dir}, \text{num}) \in CQueries\}, \]
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), $T$’s eight sets $LS_1, \ldots, LS_4, RS_1, \ldots, RS_4$ are made global (thus accessible to $\overline{C}$’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 $\exists K \in \{0, 1\}^\kappa$ and $y : (K, u, y) \in CQueries$, while $LS_0$ keeps all the values $y$ satisfying $\exists K \in \{0, 1\}^\kappa$ 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 $T$ from $G_2$. When a line has a boxed variant next to it, $S$ uses the original code, whereas $T$ uses the boxed one. Additionally, the underlined red sentences only exist in $T$. 

Simulated System $G_1(C, S^C)$. No need to implement the ideal cipher $C$.

**Simulator** $S^{C,E}$.

**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 \( E \). The term “random assign” is from [34].

private procedure \( \text{RandomAssign}(i, \delta, k, z) \)
  if \( \delta = + \) then
    \( z' := E.Ei(k, z) \)
  else
    \( z' := E.Ei^{-1}(k, z) \)
  return \( E2(i, k, z', \rightarrow) \)

private procedure \( \text{AddQuery}(i, k, z', \rightarrow) \)
  \( \text{RandomAssign}(i, k, z', \rightarrow) \)

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

private procedure \( \text{AddQuery}(i, k, z', \rightarrow, \text{dir}) \)
  if \( \text{dir} \rightarrow \land z' \in (RS1 \cup \text{LS1}_{i-1 \mod 5}) \) then abort // Early-abort in \( G2 \).
  else if \( \text{dir} \leftarrow \land z \in (\text{LS1} \cup \text{RS1}_{i-1}) \) then abort // Early-abort in \( G2 \).

Queries := Queries \cup \{(i, k, z', \rightarrow, \text{dir}, qnum)\}

\( qnum := qnum + 1 \)

\( \text{RS1} := \text{LS1} \cup \{z\} \)

if \( i \in \{1, 3\} \) then \( KSet1 := KSet1 \cup \{k\} \)
else \( KSet2 := KSet2 \cup \{k\} \) // \( i \in \{2, 4\} \)

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

public procedure \( E1(k1, u) \)
  if \( u \not\in E1[k1] \) then
    \( \text{RandomAssign}(1, +, k1, u) \)
  return \( E1[k1](u) \)

public procedure \( E1^{-1}(k1, v) \)
  if \( v \not\in E1^{-1}[k1] \) then
    \( \text{RandomAssign}(1, -, k1, v) \)
  return \( E1^{-1}[k1](v) \)

// Interfaces for 2-queries: \( E2 \) considers live trees in \( CB(k2) \), while \( E2^{-1} \) considers live trees in \( B3 \).

public procedure \( E2(k2, x) \)
  if \( x \in E2[k2] \) then return \( E2[k2](x) \)
  if \( x \not\in \text{RS3} \) then // Lazy sampling
    return \( E2[k2](x) \)

OriginSet := \( \text{FindPebLeafCB}(k2, v, \text{right}) \)
// Traverse in \( \text{Li}(k2, v) \)
if \( \text{OriginSet} = \emptyset \)
  \( \text{ProcessNonPebCBTree}(k2, v, \text{right}) \)
else if \( |\text{OriginSet}| > 1 \) then abort
else
  \( (k1^0, y^0, v^0, \text{pos}) := \text{OriginSet} \)
  \( \text{ProcessPebCBTree}(k1^0, k2, y^0, v^0, \text{pos}) \)
  return \( E2[k2](v) \)

// Interfaces for 4-queries.

public procedure \( E4(k2, x) \)
  if \( x \in E4[k2] \) then return \( E4[k2](x) \)
  if \( x \not\in \text{RS4} \) then // Lazy sampling
    return \( E4[k2](x) \)

OriginSet := \( \text{FindPebLeafB3}(k2, x, \text{right}) \)
// Traverse in \( \text{Li}(k2, x) \)
if \( \text{OriginSet} = \emptyset \)
  \( \text{ProcessNonPebB3Tree}(k2, x, \text{right}) \)
else if \( |\text{OriginSet}| > 1 \) then abort
else
  \( (k1^0, y^0, v^0, \text{pos}) := \text{OriginSet} \)
  \( \text{ProcessPebB3Tree}(k1^0, k2, y^0, v0, x^0, \text{pos}) \)
  return \( E4[k2](x) \)
private procedure Exists14Tripwire(k₂, y)
  forall k₁ ∈ KSet₁ do
    if FindEdgeInCB(k₁, k₂, y) ≠ ⊥ then return true
  return false

private procedure FindEdgeInCB(k₁, k₂, y)
  forall u ∈ E₁[k₁] do
    if Check((k₁, k₂), u, y) = true then return E₁[k₁](u)
  return ⊥

public procedure Check((k₁, k₂), u, y) // This procedure only exists in G₁
  return C.C((k₁, k₂), u) = y
// The procedures on the trees: around trees in "dual-tree" (cf. page 13)
private procedure FindPebLeafCB(k₁, k₂, y)
  if pos = right then
    x := z
    if x ∉ E₃[k₁]⁻¹ then continue
    w := E₃[k₁]⁻¹(x)
    // If w is pebbled, then stop going deeper from x.
    if w ∉ E₂[k₂]⁻¹ then
      OriginSet := OriginSet ∪ {(k₁, w, x, right)}
    else
      SearchQueue._enqueue(z, pos)
    end if
    SearchQueue._enqueue(k₁, w, left)
  else
    // pos = left
    w := z
    if w ∉ E₃[k₁] then continue
    x := E₃[k₁](w)
    if x ∈ E₂[k₂] then
      OriginSet := OriginSet ∪ {(k₁, w, x, left)}
    else
      SearchQueue._enqueue(k₁, w, right)
    end if
  end if

private procedure ProcessNonPebB3Tree(k₂, z, pos)
  if pos = left then
    w := z
    RandomAssign(2, −, k₂, w)
    forall k₁ ∈ KSet₁ do
      if w ∉ E₃[k₁] then continue
      x := E₃[k₁](w)
      ProcessB3SubTree(k₁, k₂, w, x, right)
    end if
  else
    w := z
    RandomAssign(4, +, k₂, x)
    forall k₁ ∈ KSet₁ do
      if x ∉ E₃[k₁]⁻¹ then continue
      w := E₃[k₁]⁻¹(x)
      ProcessB3SubTree(k₁, k₂, w, x, left)
    end if
  end if

private procedure ProcessPebB3Tree(k₁, k₂, w, x, pos)
  if pos = left then
    y := E₄[k₂](x)
    Traversed := false
    v := FindEdgeInCB(k₁, k₂, y)
  end if
  if v = ⊥ then
    ProcessB3SubTree(k₁, k₂, w, x, left)
  return
  forall k’₁ ∈ KSet₁ \ {k₁} do

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(k₂) 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 E₄⁻¹(k₂, y) or E₂(k₂, v) appears and ProcessDualTree is subsequently called). This strategy helps simplify some arguments.
if $\text{Traversed} = \text{true}$ then continue
if $v \notin E_1[k_1]^{-1}$ then continue
FindPebLeafCB($v_2, v, \text{right}$)
$\text{Traversed} := \text{true}$
ProcessDualTree($k_1, k_2, v, w, 2)$

else // $\text{pos} = \text{right}$
$v := E_2[k_2]^{-1}(w)$
if $v \notin E_1[k_1]^{-1}$ then
ProcessB3SubTree($k_1, k_2, w, x, \text{right}$)

// The procedures on the trees: around trees in $CB(k_2)$
private procedure FindPebLeafCB($k_2, z, \text{pos}$)
if $(\text{pos} = \text{left} \land z \in E_4[k_2]^{-1}) \lor (\text{pos} = \text{right} \land z \in E_2[k_2])$ then return $\emptyset$
$\text{OriginSet} := \emptyset$
$\text{SearchQueue}$.$\text{Enqueue}(\perp, z, \text{pos})$
while $\text{SearchQueue} \neq \emptyset$ do
forall $k_1 \in K\text{Sel}_1 \setminus \{\text{past}\}$ do
if $\text{pos} = \text{right}$ then
$v := z$
if $v \notin E_1[k_1]^{-1}$ then continue
$u := E_1[k_1]^{-1}(v)$
y := $C.(k_1, k_2, u) \quad y := \tilde{C}.((k_1, k_2), u)$
if $y \notin E_1[k_1]^{-1}$ then
$\text{OriginSet} := \text{OriginSet} \cup \{(k_1, y, v, \text{right})\}$
else
$\text{SearchQueue}.$$\text{Enqueue}(k_1, y, \text{left})$
else // $\text{pos} = \text{left}$
y := $z$
v := FindEdgeInCB($k_1, k_2, y$)
if $v = \perp$ then continue
if $v \in E_2[k_2]$ then
$\text{OriginSet} := \text{OriginSet} \cup \{(k_1, y, v, \text{left})\}$
else
$\text{SearchQueue}.$$\text{Enqueue}(k_1, v, \text{right})$
return $\text{OriginSet}$

private procedure ProcessNonPebCBTree($k_2, z, \text{pos}$)
if $\text{pos} = \text{left}$ then
$y := z$
RandomAssign(4, $-$, $k_2, y$)
forall $k_1 \in K\text{Sel}_1$ do
$v := \text{FindEdgeInCB}(k_1, k_2, y)$
if $v = \perp$ then continue
ProcessCBSSubTree($k_1, k_2, y, v, \text{right}$)

private procedure ProcessPebCBTree($k_1, k_2, y, v, \text{pos}$)
if $\text{pos} = \text{left}$ then
$w := E_2[k_2](v)$
if $w \in E_1[k_1]$ then
ProcessDualTree($k_1, k_2, E_3[k_1](w), y, 4$)
else
ProcessCBSSubTree($k_1, k_2, y, v, \text{pos}$)

// "Layer-2" ProcessTree procedures (cf. page 13)
return
$u := E_1[k_1]^{-1}(v)$
y := $C.(k_1, k_2, u) \quad y := \tilde{C}.((k_1, k_2), u)$
$\text{Traversed} := \text{false}$
forall $k'_1 \in K\text{Sel}_1 \setminus \{k_1\}$ do
if $\text{Traversed} = \text{true}$ then continue
$v' := \text{FindEdgeInCB}(k'_1, k_2, y)$
if $v' = \perp$ then continue
FindPebLeafCB($k_2, y, \text{left}$)
$\text{Traversed} := \text{true}$
ProcessDualTree($k_1, k_2, v, E_3[k_1]^{-1}(x), 2)$
else
ProcessCBSSubTree($k_1, k_2, y, v, \text{pos}$)

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 ProcessCBSSubTree 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 \texttt{ProcessB3SubTree} can be made inside such sub-call to \texttt{RecurseNew2} and \texttt{RecurseNew4} (cf. the proof of lemma 7). Similarly remarked for the code of \texttt{ProcessCBSubTree}.

```plaintext
private procedure ProcessB3SubTree(k1, k2, w, x, pos)
if pos = left then // A (3, 4)-tripwire.
y := E4[k2](x)
u := C.C−1((k1, k2), y) v := E1(k1, u)
Adapt(2, k2, v, w)
Completed := Completed ∪ \{(k1, k2, v, 2)\}
Completed := Completed ∪ \{(k1, k2, x, 4)\}
RecurseNew2(k1, k2, v, w)
else // pos = right; a (3, 2)-tripwire.
v := E7[k2](w)
u := E1−1(k1, v)
y := C.C((k1, k2), u)
Adapt(4, k2, x, y)
Completed := Completed ∪ \{(k1, k2, v, 2)\}
Completed := Completed ∪ \{(k1, k2, x, 4)\}
RecurseNew4(k1, k2, x, y)
private procedure ProcessDualTree(k1, k2, z, z’, i)
Adapt(i, k2, z, z’)
if i = 2 then
Completed := Completed ∪ \{(k1, k2, z, 2), (k1, k2, E3[k1](z’), 4)\}
RecurseNew2(k1, k2, z, z’)
else // i = 4
Completed := Completed ∪ \{(k1, k2, E2[k2]−1(E3[k1]−1(z)), 2), (k1, k2, 4)\}
RecurseNew4(k1, k2, z, z’)
// Deals with the triwires newly set off by a new 2-query (2, k2, v, w)
private procedure RecurseNew2(k1, k2, v, w)
forall k’1 ∈ KSet3 \ \{k1\} do
if v ∈ E1[k’1]−1 ∧ w ∈ E3[k’1] then
u’ := E1[k’1]−1(v)
y’ := C.C((k1, k2), u’) \quad \text{ProcessDualTree}(k1, k2, E3[k’1](w), y’, 4)
else if v ∈ E1[k’1]−1 ∧ w /∈ E3[k’1] then
u’ := E1[k’1]−1(v)
y’ := C.C((k1, k2), u’) \quad \text{ProcessCBSubTree}(k1, k2, y’, v, left)
else if v /∈ E1[k’1]−1 ∧ w ∈ E3[k’1] then
\text{ProcessCBSubTree}(k1, k2, w, E3[k’1](w), right)
// Deals with the triwires newly set off by a new 4-query (4, k2, x, y)
private procedure RecurseNew4(k1, k2, x, y)
forall k’1 ∈ KSet3 \ \{k1\} do
v’ := FindEdgeInCB(k1, k2, y)
if x ∈ E3[k’1]−1 ∧ v’ = ⊥ then
\text{ProcessDualTree}(k1, k2, v’, E3[k’1]−1(x), 2)
else if x ∈ E3[k’1]−1 ∧ v’ /∈ ⊥ then
\text{ProcessCBSubTree}(k1, k2, E3[k’1]−1(x), x, left)
else if x /∈ E3[k’1]−1 ∧ v’ = ⊥ then
\text{ProcessCBSubTree}(k1, k2, y’, right)
private procedure Adapt(i, k2, z, z’)
if z ∈ E1[k2] ∨ z’ ∈ E1[k2]−1 then abort
AddQuery(i, k2, z, z’, ⊥)
```

The second part of the code implements the intermediate system \(G_2\). Note that the involved simulator \(T\) has been implemented in the code above.
Intermediate System $G_2^{C^G, T^{C^G}}$

Global Variables

Sets $LS_0$, $RS_0$, and $CQueries$; all initially empty

Enhanced Ideal Cipher $C^{C^G}$

// Capture the early-abort conditions around new C-queries.

private procedure AddCQuery((k_1, k_2), u, y, dir)
    if $dir = \rightarrow \land y \in (LS_0 \cup RS_1)$ then abort // Early-abortion
    else if $u \in (RS_0 \cup LS_1)$ then abort // Early-abortion; $dir = \leftarrow$
    $CQueries := CQueries \cup \{(k_1, k_2), u, y, dir, qnum\}$
    $qnum := qnum + 1$
    $RS_0 := RS_0 \cup \{u\}$
    $LS_0 := LS_0 \cup \{y\}$
return

public procedure C((k_1, k_2), u)
    $y := C.C((k_1, k_2), u)$
    AddCQuery((k_1, k_2), u, y, →)
return $CTable[(k_1, k_2)](u)$

public procedure 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 indifferentiality 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 [2], which refers to the execution period from the point $D$ makes a query till the point $D$ 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 [2].

$Inv0$. 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 $(i+1, k', z', z'', \leftarrow, num)$ 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 $(i, k, z', z, dir, num)$ 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 $(K', u, y', dir, num')$. 27
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 \(((k_1, k_2), u, y, \text{dir}_C, \text{num}_C)\) and a 1-query \((1, k_1, u, v, \text{dir}_1, \text{num}_1)\) such that either: (i) \(\text{dir}_1 \leftarrow \) and \(\text{num}_1 > \text{num}_C\), or (ii) \(\text{dir}_C \leftarrow \) and \(\text{num}_C > \text{num}_1\);
- There does not exist a C-query \(((k_1, k_2), u, y, \text{dir}_C, \text{num}_C)\) and a 4-query \((4, k_2, x, y, \text{dir}_4, \text{num}_4)\) such that either: (i) \(\text{dir}_4 \rightarrow \) and \(\text{num}_4 > \text{num}_C\), or (ii) \(\text{dir}_C \rightarrow \) and \(\text{num}_C > \text{num}_4\).

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 \textit{Completed}:

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

**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 $RS$, before AddQuery is called on the \text{num}-th query. By this, AddQuery would abort and not add the \text{num}-th query to Queries. For Inv2 we wlog consider two queries \((i, k, z, z' \to, \text{num})\) and \((i, k', z'', z' \to, \text{num}')\). After \((i, k', z'', z' \to, \text{dir}, \text{num}')\) is created, \(z'\) must be added to $RS_i$, so the later query \((i, k, z, z' \to, \text{num})\) would cause AddQuery abort and would not be added to $\text{Queries}$. The proof of Inv3 and Inv4 follows the same line as Inv1 and Inv2 (differently: in some cases, the queries contradicting them would cause $\text{AddCQuery}$ 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 \((1, k_1, u, v, \text{dir}', \text{num}')\) and \((2, k_2, v, w, \text{dir}, \text{num})\) (i.e. \((i, j) = (1, 2)\)) first. Since \(\text{num} > \text{num}'\), it cannot be \(\text{dir} \leftarrow\) as otherwise contradicting Inv1. Hence \(\text{dir} \in \{\to, \perp\}\). By construction, such a 2-query may be created due to the following possibilities:

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

However, possibility (i) is not possible: for a call $E2(k_2, v)$ to call $\text{RandomAssign}$, 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 $\text{ProcessNonPebCBTree}$ creates the 2-query \((2, k_2, v, w, \to)\), it will check the set $E_1$ and find $v \in E_1[k_1]$ and call $\text{ProcessCBSUBTREE}(k_1, k_2, y, v, \text{left})$ for $y = C((k_1, k_2), u)$. By inspection of the code of $\text{ProcessCBSUBTREE}$, it can be seen once this call returns without abortion, $(k_1, k_2, v, 2) \in \text{Completed}$ 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-$2$ $\text{ProcessTree}$ procedures takes the first sub-key as an argument—to make a distinction, we denote by $k_1'$ the “first-key argument” to the layer-$2$ $\text{ProcessTree}$-call which creates $(2, k_2, v, w)$. Then:
  - If $k_1' = k_1$ and the $\text{ProcessC}$-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 $\text{ProcessB3SUBTREE}(k_1, k_2, x, \cdot)$, then $(k_1, k_2, x, 4) \in \text{Completed}$ 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 $k_1' \neq k_1$, then after the 2-query is created, the $\text{ProcessC}$-call would call $\text{RecurseNew2}$ and iterate for $k_1' \in K \setminus \{k_1\}$ and make a call to a layer-$2$ $\text{ProcessTree}$ procedure when it iterates with $k_1' = k_1$. The latter $\text{ProcessC}$-call indeed will be made due to $v \in E_1[k_1]'$ 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.

28
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 \(\tilde{C}.\text{Check}((k_1, k_2), u, y) = \text{true}\). The return value of \(\text{Check}\) 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 \(dir' \in \{ \leftarrow, \rightarrow \}\) due to Inv4. By construction, such a 4-query can only be created due to \(\text{ProcessNonPebCBTree}\) or layer-2 \(\text{ProcessTree}\)-calls (again, the query cannot be created due to \(E4^{-1}\), as the pre-existence of the 1- and C-query implies \(\text{Exists14Tripwire}(k_2, y)\) returning \(\text{true}\)). Similarly to the case of \((i,j) = (1,2)\), in each of the cases, the tuples are in \(\text{Completed}\) 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 \(k_2 \in \{0,1\}^8\), we use an edge-labeled graph \(B(k_2)\) to encode the information from \(\text{Queries}\) and \(\text{CQueries}\) 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 \(\{0,1\}^4\); note that \(B_3\) is independent from \(k_2\) as it is built from 3-queries. Further note that \((B_2(k_2), B_3,\) 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 \((3, k_1, w, x, dir, num) \in \text{Queries}\), 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 \(k_2 \in \{0,1\}^8\), \(CB(k_2)\) is constructed as follows: for every C-query \(((k_1, k_2), u, y, dir_C, num_C) \in \text{CQueries}\) and every 1-query \((1, k_1, u, v, dir_1, num_1) \in \text{Queries}\) (the two queries must share the same \(k_1\) and \(u\)), we construct an edge \((y, v)\) of label \(k_1\), direction \(dir_1\), 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 \(num_{y,v} > num_1\), 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 \(B_3(k_2)\): because only C-queries of the form \(((k_1, k_2), u, v)\) 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 \(((k_1, k_2), u, y, \rightarrow, num)\): 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) \(d_1 = \leftarrow\) (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 \(d \neq \leftarrow\).

We now formally prove the acyclic property of \(B_3\), which is almost the same as Lemma 12 of [2].

**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 \(k_2 \in \{0,1\}^8\), 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 \(\text{num}\) 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, \(\text{dir}_1,\text{num}\_1\)\), \((\(k_1,k_2\), u, v, \(\text{dir}_2,\text{num}\_2\)\), \((\(k_1',k_2\), u', v, \(\text{dir}_3,\text{num}\_3\)\), \((\(k_1',k_2\), u', y', \(\text{dir}_4,\text{num}\_4\)\). By Inv2, it necessarily holds \(\text{num}\_3 > \text{num}\_2\) and \(\text{dir}_3 = \leftarrow\); then by Inv4, it holds \(\text{num}\_4 > \text{num}\_3 > \text{num}\_2\). Since the edge formed by \((\(k_1,k_2\), u, y, \rightarrow, \text{num}\_2\)\) and \((\(k_1,k_2\), u, y, \text{dir}_1,\text{num}\_1\)\) is directed from \(y\) to \(v\), either \(\text{dir}_1 = \leftarrow \land \text{num}\_2 > \text{num}\_1\), or \(\text{dir}_1 = \rightarrow\) (which also implies \(\text{num}\_2 > \text{num}\_1\)). Therefore, the two \(\text{num}\) values associated to the two edges are \(\text{num}\_2\) and \(\text{num}\_4\) respectively, and \(\text{num}\_4 > \text{num}\_2\) and the claim on the \(\text{num}\) 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. \(\square\)

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

**Lemma 3.** For any \(k_2 \in \{0,1\}^\ast\), consider an edge \((k_1,y,v,\text{dir},\text{num})\) in \(CB(k_2)\). At any moment of a \(G_2\) execution such that Inv5 holds, if there exists a 2-query \((2,k_2,v,w,\text{dir}_3,\text{num}_3)\) (4-query \((4,k_4,x,y,\text{dir}_4,\text{num}_4)\), resp.) such that \(\text{num}_3 > \text{num}\), then the queries of the edge \((k_1,y,v,\text{dir},\text{num})\) 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 \((k_1,y,v,\text{dir},\text{num})\) are \((\(k_1,k_2\), u, y, \text{dir}_C,\text{num}_C)\) and \((1,k_1,u,v,\text{dir}_1,\text{num}_1)\). If there exists a 2-query \((2,k_2,v,w,\text{dir}_2,\text{num}_2)\) satisfying all the assumptions, then it necessarily holds \(\text{num}_3 > \text{Max}\{\text{num}_C,\text{num}_1\} \geq \text{num}_1\); if there is a 4-query \((4,k_4,x,y,\text{dir}_4,\text{num}_4)\) satisfying all the assumptions, then: (i) \(\text{num}_3 > \text{Max}\{\text{num}_C,\text{num}_1\} \geq \text{num}_1\); (ii) due to the existence of \((\(k_1,k_2\), u, y, \text{dir}_C,\text{num}_C)\), \(\text{C.Check}(\(k_1,k_2\), u, y)\) returns true. In both cases, the claim holds by Inv5. \(\square\)

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 \(B_2\) and the right shore of \(CB(k_2)\) with existing 2-queries \((2,k_2,v,w,\text{dir},\text{num})\), and linking the nodes in the left shore of \(CB(k_2)\) and the right shore of \(B_3\) with existing 4-queries \((4,k_2,x,y,\text{dir},\text{num})\). 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 \(k_2 \in \{0,1\}^\ast\), \(B(k_2)\) has four “shores” \(\{0,1\}\) numbered 2, 3, 4, and 5:

- 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 \((i,k_2,z,z',\text{dir},\text{num})\) becomes a (possibly directed) unlabeled \(i\)-edge from node \(z\) in shore \(i\) to node \(z'\) in shore \(i+1\), with direction consistent with \(\text{dir}\).

Note that unlike the graph \(B\) used in [2] (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 [2], 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.\(^8\) For a pebbled node \(z\) in shore 2 or 5 (3 or 4, resp.), we also say that \(z\) is a pebbled node of \(CB(k_2)\) \((B_3,\text{resp.})\). Similarly to [2], 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 \(k_2 \in \{0,1\}^\ast\), 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,

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

\(^8\) They are numbered from 2 to 5 because they consist of inputs to \(E_2\) (\(v\)), inputs to \(E_4\) (\(w\)), inputs to \(E_4\) (\(z\)) respectively, and \(y \in CTable[\mathbf{K}]^{-1}\).  

\(^9\) Note that in simulator overview (page 9), the notion pebbling is informally specified under certain \(k_2\) for the sake of simplicity.
(b) for an edge \((k_1, y, v, \text{dir}, \text{num})\) in \(CB(k_2)\), if \(\text{dir} = \rightarrow\) and \(v\) is adjacent to a 2-query, then \(y\) is adjacent to a 4-query; if \(\text{dir} = \leftarrow\) 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 \((4, k_2, x, y, \text{dir}', \text{num}')\). Then it has to be \(\text{num}' > \text{num}\), as otherwise contradicting Inv1; then as Inv5 holds, the two queries \((3, k_1, v, x, \text{dir}, \text{num})\) and \((4, k_2, x, y, \text{dir}', \text{num}')\) 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 \(((k_1, k_2), u, y, \text{dir}, n_1)\) 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 \(n_1 = n_2\). Due to Inv1, it has to be \(n_2 > n_1\); hence \(n_2 > \text{num}\) 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 \(k_2 \in \{0, 1\}^n\), 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 \(B_3(z)\) \((CB(k_2), \text{resp.})\) the connected component containing \(z\) in \(B_3\) \((CB(k_2), \text{resp.})\). Then at any point in a \(G_2\) execution and for any non-pebbled node \(z\) of \(B_3\) \((CB(k_2), \text{resp.})\), define the live tree anchored at \(z\) in \(B_3\) \((CB(k_2), \text{resp.})\) (denoted \(Li(k_2, z)\)) as the tree obtained by “dangling” \(B_3(z)\) \((CB(k_2, z), \text{resp.})\) 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 \(B_3\) \((CB(k_2), \text{resp.})\) is obtained from \(B_3(z)\) \((CB(k_2, z), \text{resp.})\) 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 \(z' \in B_3(z)\) \((z' \in CB(k_2, z), \text{resp.})\) \((z' \neq z)\), 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 [2]. 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-abort 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 \(k_2 \in \{0, 1\}^n\), 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 \(B_3(z)\) \((CB(k_2, z), \text{resp.})\). 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 \(|Li(k_2, z)|\).

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 \(SubT(T, z) = T\).

Note that this deviates from the analogue terminology in [2, page 44]: the term “size” in [2] 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 G2 execution and for any k2 ∈ {0, 1}*, a live tree Li in CB(k2) is complete, if for any non-pebbled right-shore node v in Li and any k1, ∃(1, k1, u, v) ∈ Queries ⇒ ∃((k1, k2), u, y) ∈ CQueries. In other words, for no pair (k1, v) does the following hold:

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

The motivations behind this definition are two-fold: first, we observe that during G2 processing a complete tree Li, the number of newly-created 3-queries is at most \(|Li|\); second, a call to FindPebLeafCB(k2, z, pos) would turn the live tree Li(k2, 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(k2, 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 [2], this notation is used to avoid ambiguity, as the tree Li(k2, z) changes during G2 processing.

Lemma 6. Right after a call to FindPebLeafCB(k2, z, pos) returns without abortion, the following hold:

(a) Each C-query created in this call is a part of an edge in Li(k2, z); and all the nodes “newly added”\(^{11}\) to Li(k2, z) are non-pebbled leaves;

(b) Li(k2, z) is complete;

(c) if pos = left then \(|Li(k_2,z)| \leq |T_{k_2,z}| \cdot |KSet_1|; otherwise (pos = right) \(|Li(k_2,z)| \leq \max(|T_{k_2,z}|, 1) \cdot |KSet_1|; \)

Proof. We start by reminding that in this lemma, the notation \(T_{k_2,z}\) is the snapshot of Li(k2, 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.\(^{12}\) 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(k2, 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.\(^{13}\) We thereby assume that the children of y in \(T_{k_2,y}\) are \(v_1, \ldots, v_l\), and the associated edges are \((k_1^i, y, v_i), \ldots, (k_1^i, y, v_l)\). 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 \((k_1^i, v_i, \text{right})\) is pushed into SearchQueue;
- if \(v_i\) has been pebbled, then \((k_1^i, k_2, y, v_i, \text{left})\) 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(k2, z).

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 \(y_{1i}, \ldots, y_{si}\), with associated edges \((k_1^{i1}, y_{1i}, v_i), \ldots, (k_1^{is}, y_{si}, v_i)\);
(ii) there are \(\bar{y}\) 1-queries \((1, k_1^{ii}, u_{1i}, v_i) (j = 1, \ldots, \bar{y})\) such that \(u_{1i} \notin CTable[(k_1^{i1}, k_2)]\). Clearly \(k_1^{i1} \neq k_1^{11} \neq \ldots \neq k_1^{is} \neq \bar{y}^{-1} \neq \ldots \neq k_1^{i1}\) and \(s + \bar{y} + 1 \leq |KSet_1|\).

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 \(k_1 = \text{past}(=k_1^i)\).

---

\(^{11}\) Say, the nodes that were not in \(T_{k_2,z}\) but are in Li(k2, z). It should be reminded that by the assumption of the lemma, Li(k2, z) refers to the live tree anchored at z right after the FindPebLeafCB-call.

\(^{12}\) 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.

\(^{13}\) 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 \( \tilde{C} \) does not create any new C-queries. \( \text{FindPebLeafCB} \) simply obtains the level 3 node \( y_i^j \) and pushes it into either \( \text{SearchQueue} \) or \( \text{OriginSet} \) depending on its pebbling state (as previously pushing level 2 nodes).

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

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 \( \text{Li}_{k_2,y} \):

- at most \( |\text{KSet}| \) edges in \( \text{Li}_{k_2,y} \) are adjacent to \( v_i \);
- for any \( k_1, v_i \in \text{Queries} \Rightarrow \exists ((k_1, k_2), u, y) \in C-queries \).

Later when the newly added node \( y_i^j \) is popped from \( \text{SearchQueue} \), the subsequent process is similar to the above process to \( \text{Li}_{k_2,y} \) which has been described. Subsequently, level 4 nodes in \( \text{Li}_{k_2,y} \) are processed to the level 2 nodes as described, and have similar effects. The above are interleavingly repeated till all the nodes in \( \text{Li}_{k_2,y} \) are visited.

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

Case 2: the call is \( \text{FindPebLeafCB}(k_2,v,\text{right}) \). When \( |\text{Li}_{k_2,v}| \geq 1 \), then it can be checked that the process has no essential difference with Case 1, and the statements hold. In particular, when \( |\text{Li}_{k_2,v}| \geq 1 \), \( \text{Li}_{k_2,v} \) has at least one left-shore node, and hence the number of right-shore nodes in \( \text{Li}_{k_2,v} \) is at most \( |\text{Li}_{k_2,v}| \) and \( |\text{Li}(k_2,v)| \leq |\text{Li}_{k_2,v}| \cdot |\text{KSet}| \) by the assumption \( |\text{Li}_{k_2,v}| = |v| \). Assume that there are \( l \) pre-existing 1-queries \((1,k_1^i,u_i,v_i), \ldots, (1,k_l^i,u_l,v_l)\) (note that \( \forall i \in \{1, \ldots, l\}, u_i \notin \text{CTable}(k_1^i,k_2) \) by the assumption \( |\text{Li}_{k_2,v}| = |v| \)). Then for \( i = 1, \ldots, l \), \( \text{FindPebLeafCB} \) obtains \( u_i \) by accessing \( E_{i,j}^{-1} \), and then queries \( \tilde{C} \), which results in creating a new C-query \((k_1^i,k_2,u_i,v_i)\) and adding an edge \((k_1^i,y_i^j,v)\) to \( \text{Li}_{k_2,v} \). Right after the C-query is created, \( \forall k_i \in [0, 1]^{v_i} \setminus \{k_i^{i,j}\}, y_i^j \notin \text{CTable}(k_i^{i,j}, k_2) \) immediately holds due to Inv3, so that \( y_i \) turns out to be a leaf of \( \text{Li}_{k_2,v} \) and \( y_i^j \notin E_{i,j}^{k_2} \) 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 \( \text{FindPebLeafCB} \). In this case, \( \text{FindPebLeafCB} \) exactly makes \( l \) C-queries and adds \( l \) leaves into \( \text{Li}(k_2,v) \), and (a), (b) clearly hold. On the other hand, in this case, \( \text{Li}_{k_2,v} \) has one right-shore node and \( |\text{Li}_{k_2,v}| = 0 \), hence (c) \((|\text{Li}(k_2,v)|) \leq |\text{KSet}| \leq \max(\{|\text{Li}_{k_2,v}|,1\} \cdot |\text{KSet}| \) when \( \text{pos} = \text{right} \)) holds. \( \square \)

As a corollary of Lemma 6 (a), the number and positions of pebbled nodes in \( \text{Li}_{k_2,z} \) are both invariant after \( \text{Li}_{k_2,z} \) is “extended” by \( \text{FindPebLeafCB} \).

Inside a Simulator Cycle. Based on the above observations, this subsection analyzes the process and effects of simulator cycles. Recall that a simulator cycle is triggered by a query from \( \mathcal{D} \), 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 \( \mathcal{D} \) making a query which has been in the history is clearly not interesting. Whereas according to the strategy (cf. page 14) and the code, in the following is made two cases (of \( \mathcal{D} \) making a new query), the subsequent simulator cycle only consists querying \( \mathcal{E} \) to obtain the random answer, and the corresponding set is simply enlarged by 1:

(a) \( \mathcal{D} \) makes a new 1- or 3-query;
(b) \( \mathcal{D} \) 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 clarity, 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 $\mathcal{D}$ 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 $\text{FindPebLeafB3}$ or $\text{FindPebLeafCB}$;
2. calls a procedure among $\text{ProcessNonPebB3Tree}$, $\text{ProcessPebB3Tree}$, $\text{ProcessNonPebCBTree}$, and $\text{ProcessPebCBTree}$—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)—$\text{ProcessB3SubTree}$, $\text{ProcessCBSSubTree}$, and $\text{ProcessDualTree}$—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 PROCESS TREE PROCEDURES. The analysis starts with formally defining safe ProcessTree-calls. The term “safe” is due to [34].

Definition 2. A call to $\text{ProcessB3SubTree}$ is safe if depending on its arguments, the following hold right before the call is made:

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

(ii) when the arguments to the call are $(k_1, k_2, w, x, \text{right})$, let $y := E_2[k_2]^{-1}(w)$, then: (a) $w$ is the unique pebbled leaf of $\text{Li}(k_2, x)$, and (b) $v / \in E_1[k_1]^{-1}$ (cf. Fig. 13 (left)).

Definition 3. A call to $\text{ProcessCBSSubTree}$ is safe if depending on its arguments, the following hold right before the call is made:

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

(ii) when the arguments to the call are $(k_1, k_2, y, v, \text{right})$, let $x := E_4[k_2]^{-1}(y)$, then: (a) $y$ is the unique pebbled leaf of $\text{Li}(k_2, v)$, and (b) $x \not\in E_3[k_1]^{-1}$, and (c) $\text{Li}(k_2, v)$ is complete (cf. Fig. 14 (right)).

Definition 4. A call to $\text{ProcessDualTree}$ 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) $u$ (resp.) is the unique pebbled leaf of $\text{Li}(k_2, v)$ ($\text{Li}(k_2, w)$, resp.), and (c) $\text{Li}(k_2, v)$ 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) $w$ (resp.) is the unique pebbled leaf of $\text{Li}(k_2, x)$ ($\text{Li}(k_2, y)$, resp.), and (c) $\text{Li}(k_2, y)$ is complete (cf. Fig. 15 (right)).

Recall from the previous subsection that the notation $T_{k_2,z}$ refers to the snapshot of the live tree $\text{Li}(k_2, z)$ 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 $\text{ProcessB3SubTree}$.

Lemma 7. The following hold for a safe call to $\text{ProcessB3SubTree}$:

(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 $\text{ProcessB3SubTree}$. For clearness, we divide the flow into two steps: first is a chain-completion phase, then is a call to $\text{RecursNew}$. Depending on the arguments, there are two possibilities:

14 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.

15 Formally speaking, either $y /\in CTable[(k_1, k_2)]^{-1}$ (Fig. 13 (right-upper)), or $CTable[(k_1, k_2)]^{-1}(y) /\in E_1[k_1]$ (Fig. 13 (right-lower)).
Fig. 13. (Left) a safe call to \texttt{ProcessB3SubTree}(k_1, k_2, w, x, right); (Right) a safe call to \texttt{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 \texttt{ProcessTree} calls.

Fig. 14. (Left) a safe call to \texttt{ProcessCBSUBSubTree}(k_1, k_2, y, v, left); (Right) a safe call to \texttt{ProcessCBSUBSubTree}(k_1, k_2, y, v, right).

Fig. 15. (Left) a safe call to \texttt{ProcessDualTree}(k_1, k_2, v, w, 2); (Right) a safe call to \texttt{ProcessDualTree}(k_1, k_2, x, y, 4).
Case 1: the call is \text{ProcessB3SubTree}(k_1, k_2, w, x, \text{left}). Let \( y := E_4[k_2](x) \). Then the chain-completion phase consists of three steps: (1) queries \( \hat{C} \); (2) calls \( E_1 \); (3) calls \text{Adapt}—for ease of reference, this call will be called level 1 \text{Adapt}-call in this proof. We argue that a new 1-query \((1, k_1, u, v, \rightarrow)\) must be created during step (2). To this end, recall the definition of safe \text{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 \((1, k_1, u, v, \rightarrow)\); in the former case, \( G_2 \) first creates a new \( C \)-query \((k_1, k_2), u, y, \rightarrow) \) via \( \hat{C} \), then creates a 1-query \((1, k_1, u, v, \rightarrow)\). If abort does not occur, then this 1-query is indeed new because otherwise its adjacency to the newer query \((k_1, k_2), u, y, \rightarrow)\) 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:

\[
\forall k_1' \in \{0, 1\}^n \setminus \{k_1\}, v \notin E_4[k_1']^{-1}.
\]

Moreover, as the \text{ProcessB3SubTree}-call is assumed safe, \( w \) is non-pebbled, i.e. \( w \notin E_3[k_2]^{-1} \). Hence the “level 1” \text{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 \text{RECURSENEW2}(k_1, k_2, v, w) \text{, which would iterate for all } k_1' \in KSet_1 \setminus \{k_1\} \text{ and make a series of calls to layer-2 \text{PROCESS TREE}} \text{ procedures. However, note that for each such } k_1', \text{ only if } v \in E_1[k_1']^{-1} \text{ could \text{PROCESSCBSUBTREE} and \text{PROCESSDUALTREE} be called. By this and (1), \text{RECURSENEW2} only calls \text{PROCESSB3SUBTREE}. We argue that these \text{PROCESSB3SUBTREE}-calls are all safe. For this, let } x_1, \ldots, x_l \text{ be the non-pebbled children of } w \text{ in } T_{k_2, w, 1} \text{, and let the associated 3-queries be } (3, k_1', w, x_1), \ldots, (3, k_1', w, x_l). \text{ Note that it must be } k_1 \neq k_1' \neq \ldots \neq k_1. \text{ Then for any } x_i \text{ among the children, Sub}(T_{k_2, w, x_i}) \text{ has no pebbled node: because } x \text{ is the unique pebbled leaf in } T_{k_2, w, n} \text{, and because the chain-completion phase only pebbles } w. \text{ By this and the definition, right before the } \text{forall} \text{ loop, } Li(k_2, x_i) \text{ consists of Sub}(T_{k_2, w, x_i}) \text{ and the pebbled node } w, \text{ and has only one pebbled leaf. We then show that the state is kept till the call to \text{PROCESSB3SUBTREE}(k_1, k_2, w, x_i, \text{right}). Note that for any } x_j \text{ such that the call \text{PROCESSB3SUBTREE}(k_1', k_2, w, x_j, \text{right}) \text{ precedes the call \text{PROCESSB3SUBTREE}(k_1', k_2, w, x_i, \text{right}) \text{ it holds Sub}(T_{k_2, w, x_i}) \cap \text{Sub}(T_{k_2, w, x_j}) = \emptyset). \text{ Hence the processing of } Li(k_2, x_j) \text{ does not affect } Li(k_2, x_i). \text{ On the other hand, } v \notin E_1[k_1']^{-1} \text{ follows from (1) and } k_1 \neq k_1' \neq \ldots \neq k_1. \text{ so that the call \text{PROCESSB3SUBTREE}(k_1', k_2, w, x_i, \text{right}) \text{ is safe (if it is indeed made, say, } G_2 \text{ does not abort until it makes this call). Thus all the } l \text{ \text{PROCESSB3SUBTREE}-calls are safe (the } l \text{ calls corresponding to } (k_1', k_2, w, x_1), \ldots, (k_1', k_2, w, x_l) \text{ will also be called level 1 \text{PROCESSB3SUBTREE}-calls in this proof). These complete the analysis of Case 1.}

Case 2: the call is \text{ProcessB3SubTree}(k_1, k_2, w, x, \text{right}). Let \( v := E_2[k_2]^{-1}(w) \). Then the chain-completion phase has three steps (if abort does not occur):

(1) calls \( E_1^{-1} \), which creates a new L-query \((1, k_1, u, v, \rightarrow)\) (this query will be new because the \text{PROCESSB3SUBTREE}-call is assumed safe, cf. definition 2);

(2) queries \( \hat{C} \), which creates a new \( C \)-query \((k_1, k_2), u, y, \rightarrow\). This query will be new because otherwise its adjacency to the newer query \((k_1, k_2), u, y, \rightarrow\) 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:

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

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

\( G_2 \) then calls \text{RECURSENEW4}(k_1, k_2, x, y), which only makes a series of calls to \text{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 \((3, k_i, w_i, x)\) 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 \text{PROCESSB3SUBTree}(k_1', k_2, w_i, x, \text{left}) and \( y \notin CTable[(k_1', k_2)]^{-1} \) follows from (2) and \( k_1 \neq k_1' \neq \ldots \neq k_1' \). These complete the analysis of Case 2.

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

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

---

16 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 PROCESSB3SubTree-calls are safe by (a), (b) is established. \qed

The next lemma deals with safe calls to PROCESSCBSUBTree.

**Lemma 8.** The following hold for a safe call to PROCESSCBSUBTree$(k_1, k_2, y, v, \text{left})$ (or PROCESSCBSUBTree$(k_1, k_2, y, v, \text{right})$, 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$(k_1, k_2, y, v, \text{left})$.** 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:

$$\forall k'_1 \in \{0,1\}^* \setminus \{k_1\}, x \notin E_3[k'_1]^{-1}. \quad (3)$$

2. calls Adapt, which creates a new adapted 4-query $(4, k_2, x, y, \bot)$ without abortion (as the PROCESSCBSUBTree-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, and 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 $(k_1, y, v_1), \ldots, (k_1, y, v_l)$ (note $k_1 \neq k'_1 \neq \ldots \neq k'_1$). Then for each $v_i$, the only pebbled leaf in $Li(k_2, v_i)$ before the subsequent call PROCESSCBSUBTree$(k_1, k_2, y, v_i, \text{right})$ is $y$, and $x \notin E_3[k'_1]^{-1}$ follows from (3) and $k'_1 \neq k'_1 \neq \ldots \neq k'_1$. 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$(k_1, k_2, y, v, \text{right})$.** Let $x := E_4[k_2]^{-1}(y)$. The chain-completion phase has two steps: First, calls E3$^{-1}$ and creates a new 3-query $(3, k_1, x, w, \leftarrow)$ (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):

$$\forall k'_1 \in \{0,1\}^* \setminus \{k_1\}, w \notin E_3[k'_1]; \quad (4)$$

Second, calls Adapt, which creates an adapted 2-query $(2, k_2, v, w, \bot)$ 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 PROCESSCBSUBTree due to (4). To show the safeness of these calls, consider each $k'_1$ such that $v \in E_1[k'_1]^{-1}$ (excluding $k_1$). As $T_{k_2,v}$ is complete, let $u_j := E_1[k'_1]^{-1}(v)$, then we have: (i) $u_j \in CTable([k'_1, k_2])$, so that $y_j = CTable([k'_1, k_2])(u_j)$ is a child of $v$ in $T_{k_2,v}$, and the query to $\overline{C}$ 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$(k_1, k_2, v, y, \text{left})$ and consider a PROCESSCBSUBTree-call made during this process—for example, the call PROCESSCBSUBTree$(k'_1, k_2, y, v, \text{right})$ (cf. 37
the analysis of Case 1). Note that the arguments of this call indeed uniquely characterize an edge in $T_{k_2, y}$ (i.e. $(k'_1, y, v_1)$). Moreover, it can be seen (from the code/the tree traversal algorithm and the analysis above) that two different $\text{ProcessCBSubTree}$-calls in this period cannot share the same arguments $(k_1, y, v)$. Hence there is a bijection between the $\text{ProcessCBSubTree}$-calls in this period (including the “mother” call $\text{ProcessCBSubTree}(k_1, k_2, y, left)$) and edges in $T_{k_2, y}$. The analysis of the flow of a safe $\text{ProcessCBSubTree}$-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 $\text{ProcessCBSubTree}(k_1, k_2, y, left)$ 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 $\text{ProcessCBSubTree}$-call enlarges $|E_3|$ by 1 during its chain-completion phase. As we have seen, there are $|T_{k_2, y}|$ $\text{ProcessCBSubTree}$-calls in this period (including $\text{ProcessCBSubTree}(k_1, k_2, y, left)$). Hence $|E_3|$ increases by $|T_{k_2, y}|$, and (b) holds for $\text{ProcessCBSubTree}(k_1, k_2, v, y, left)$. Similarly for $\text{ProcessCBSubTree}(k_1, k_2, v, y, right)$. These complete the proof. 

The last one deals with safe calls to $\text{ProcessDualTree}$.

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

(a) All the calls to layer-2 $\text{ProcessTree}$ procedures made in this call are safe, and $G_2$ never aborts due to calls to $\text{Adapt}$ 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 $|T_{k_2, y}|$ ($|T_{k_2, y}|$, resp.). Moreover, each edge in $T_{k_2, y}$ ($T_{k_2, y}$, 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 $\text{ProcessDualTree}(k_1, k_2, v, w, 2)$. Let $x := E_3[k_1][w]$ and $y := E_4[k_2][x]$. The chain-completion phase only consists of a call to $\text{Adapt}(2, k_2, v, w)$, which creates a 2-query $(2, k_2, v, w, \perp)$ without abortion $(v \notin E_3[k_2] \land w \notin E_2[k_2]^{-1}$ as the $\text{ProcessDualTree}$-call is safe). It’s clear that neither $|KSet_1|$ nor $|E_3|$ is modified. Additionally, note that the arguments of this $\text{ProcessDualTree}$-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 $\text{RecurseNew2}(k_1, k_2, v, w)$. We proceed to argue that all the calls to layer-2 $\text{ProcessTree}$ procedures made in $\text{RecurseNew2}$ are safe. Right after $(2, k_2, v, w, \perp)$ is created, for any child $x_i$ of $w$ in $T_{k_2, v}$ (excluding $x$), $x_i$ is non-pebbled; for any $k'_1$ such that $v \in E_1[k'_1]^{-1}$ (excluding $k_1$), since $T_{k_2, v}$ is complete, the node $y_j = \text{CTable}([k'_1, k_2])(E_1[k'_1]^{-1}(v))$ is a child of $v$ in $T_{k_2, v}$ and must be non-pebbled (and the query to $\hat{C}$ would not create new C-queries). By the above, for any call to layer-2 $\text{ProcessTree}$ procedures made in the subsequent for all 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 $\text{RecurseNew2}$-call. More clearly, for any $k'_1$,

- only when $v \notin E_1[k'_1]^{-1}$ and $w \in E_3[k'_1][w]$, $\text{RecurseNew2}$ make a call to $\text{ProcessB3SubTree}(k'_1, k_2, w, E_3[k'_1][w], right)$, 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 $E_1[k'_1]^{-1}(v) \in \text{CTable}([k'_1, k_2])$ and the completeness of $\text{Li}(k_2, y)$ for $y' := \text{CTable}([k'_1, k_2])(E_1[k'_1]^{-1}(v))$) $\text{RecurseNew2}$ make a call to $\text{ProcessCBSubTree}(k'_1, k_2, y', v, left)$, which is thereby safe;
- only when $w \in E_3[k'_1][w] \cup v \in E_1[k'_1]^{-1}$ will $\text{RecurseNew2}$ make a call to $\text{ProcessDualTree}(k'_1, k_2, x', y', 4)$ (let $x' := E_3[k'_1][w]$; and similarly to the previous case, let $y' := \text{CTable}([k'_1, k_2])(E_1[k'_1]^{-1}(v)))$, which is thereby safe. Similarly to the “mother” call $\text{ProcessDualTree}(k_1, k_2, v, w, 2)$, this call also uniquely characterize an edge $(k'_1, y', v')$ in $T_{k_2, v}$.

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

**Case 2:** the call is $\text{ProcessDualTree}(k_1, k_2, x, y, 4)$. The argument is altogether symmetrical to the argument in Case 1, and is thereby omitted. As a summary of the key points, the $\text{Adapt}$-call made in the chain-completion phase does not cause abort due to the safeness assumption on the current $\text{ProcessDualTree}$-call, while the safeness of the calls to layer-2 $\text{ProcessTree}$ procedures made in the subsequent call $\text{RecurseNew4}(k_1, k_2, x, y)$ is ensured by the $\text{RecurseNew4}$-call itself.
We then consider the statements. First, the analysis above shows that a safe \textit{ProcessDualTree}-call makes a non-aborting \textit{Adapt}-call as well as a series of safe calls to layer-2 \textit{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 \textit{ProcessDualTree}(k_1, k_2, v, w, 2), and consider the subsequent calls to layer-2 \textit{ProcessTree} procedures made in this period. As mentioned (cf. the analysis of Case 1), each subsequent call to \textit{ProcessDualTree} uniquely characterize an edge in \(T_{k_2,v}\). Meanwhile, in the proof of Lemma 8, we already argued that each subsequent \textit{ProcessCBSSubTree}-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 \textit{ProcessCBSSubTree} and \textit{ProcessDualTree}-calls in this period cannot characterize the same edge in \(T_{k_2,v}\). Hence there is a bijection between the \textit{ProcessCBSSubTree}- and \textit{ProcessDualTree}-calls in this period (including \textit{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 \textit{ProcessDualTree}-call shows that the edge characterized by its arguments is a part of a completed path after the subsequent \textit{Adapt}-call. The subsequent \textit{ProcessCBSSubTree}-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 \textit{ProcessDualTree}(k_1, k_2, v, w, 2) returns without abortion. For \textit{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 \textit{ProcessCBSSubTree}-call enlarges \(|E_3|\) by 1 during its chain-completion phase, while none of the subsequent calls to \textit{ProcessDualTree} and \textit{ProcessCBSSubTree} "directly" enlarges \(|E_3|\). As we have seen, there are \(|T_{k_2,v}|\) calls to \textit{ProcessCBSSubTree} and \textit{ProcessCBSSubTree} in total in this period (including \textit{ProcessDualTree}(k_1, k_2, v, w, 2) itself). Hence \(|E_3|\) increases by at most \(|T_{k_2,v}|\). \(\square\)

AROUND LAYER-1 \textit{ProcessTree} PROCEDURES. As planed, this paragraph analyzes layer-1 \textit{ProcessTree} procedures. Consider \textit{ProcessNonPebB3Tree} and \textit{ProcessNonPebCBTree} first. In such a call, a series of calls to \textit{ProcessBSSubTree} and \textit{ProcessBSSubTree} are made. The next two lemmata analyze the influence of such a bundle of layer-2 \textit{ProcessTree} calls.

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

(a) \(G_2\) never aborts due to the subsequent calls to \textit{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 \textit{ProcessNonPebB3Tree} first creates a new 2- or 4-query (thus enlarging \(|KSet_2|\) by at most 1) and then makes several safe calls to \textit{ProcessCBSSubTree}. 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 \textit{ProcessNonPebB3Tree}(k_2, w^*, left). \textit{ProcessNonPebB3Tree} starts by creating a new 2-query \((2, k_2, v^*, w^*, \leftarrow)\) via a call to \textit{RandomAssign}(2, –, k_2, w^*). If abort does not occur, then the following claim holds by Inv1:

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

\textit{ProcessNonPebB3Tree} then enters the \textbf{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 \((3, k_1^1, w^*, x_i)\) for \(i = 1, \ldots, l\). Note \(k_1^1 \neq \ldots \neq k_1^l\). Similarly to [2], 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 “\textbf{forall} \ \ell \in KSet_j” iterates from the smallest value of \(k \in KSet_j\) to the largest one. Then, consider the point when the \textbf{forall} loop iterates with \(k_1 = k_1^1\). At this point, the set of pebbled nodes in \(T_{k_2,w^*}\) include the following ones:

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

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 \(v^* \notin E_1[k_1^1]^{-1}\), and hence the call to \textit{ProcessBSSubTree}(k_1^1, k_2, w, x_i, right) is safe, and the following hold by Lemma 7 (a) and (b):
Then, for each 3-edge \((i, v, \rightarrow)\), after this call returns.

For a call to ProcessCBSubTree \((i, v, \rightarrow)\) and an analysis similar to Lemma 10, the analysis of Case 1.

Moreover, by the above analysis, ProcessNonPebB3Tree does not enlarge \(|KSet_1|\) nor \(|E_3|\). On the other hand, the call RANDOMASSIGN\((2, \rightarrow, k_2, \rightarrow, w^*)\) 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\((k_2, x^*, \rightarrow)\).** 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):

\[
\forall k_1 \in \{0, 1\}^*, y^* \notin \text{CTable}[E_3[k_1]]^{-1}. \tag{6}
\]

Then, for each 3-edge \((3, k_1, w, x^*)\), at the point when the subsequent forall loop iterates with \(k_1 = k_1^1, w_1\) must be non-pebbled and \(x^*\) is the unique pebbled node in \(L_{w_1}\). And \(y^* \notin \text{CTable}[E_3[k_1]]^{-1}\) due to (6) and \(k_1^1 \neq \ldots \neq k_1^i\), so that there does not exist an edge \((k_1^i, y^*, \cdot)\) in \(CB(k_2)\) and the call ProcessB3SubTree\((k_1^i, k_2, w, x^*, \leftarrow)\) is safe. Thus all the calls to ProcessB3SubTree in ProcessNonPebB3Tree\((k_2, x^*, \rightarrow)\) are safe and the claims hold by Lemma 7 and an analysis similar to Case 1. □

**Lemma 11.** For a call to ProcessNonPebCBTree\((k_2, y^*, \leftarrow)\) (or ProcessNonPebCBTree\((k_2, v^*, \rightarrow)\)), if the tree \(T_{k_2, y^*}\) \((T_{k_2, v^*}, \text{resp.})\) is complete and has no pebbled leaf, then the following hold:

(a) \(G_2\) never aborts due to the subsequent calls to \(\text{Adapt}\);

(b) If this call returns without abortion, then \(|KSet_1|\) stays constant, \(|KSet_2|\) increases by at most 1, and \(|E_3|\) increases by \(|T_{k_2, y^*}| \cup |T_{k_2, v^*}|\), resp.; and each edge in \(|T_{k_2, y^*}| \cup |T_{k_2, v^*}|\), 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-abort 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 clarity, we recall the key observations in each possibility.

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

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

Then, for each edge \((k_1, y^*, v_i)\), 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, \(x^* \notin E_3[k_1]^{-1}\) due to (7) and \(k_1^1 \neq \ldots \neq k_1^i\), and hence the call to ProcessCBSUBTree\((k_1^i, k_2, y^*, v_i, \rightarrow)\) 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\((k_2, y^*, \leftarrow)\) are safe, and the claims on non-abort (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\((k_1^i, k_2, y^*, v_i, \leftarrow)\) 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\((k_2, v^*, \rightarrow)\).** 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):

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

Then, for each \(k_1^i\) such that \(v^* \in E_3[k_1^i]^{-1}\), since \(T_{k_2, v^*}\) is complete, the node \(y_i = \text{CTable}(E_3[k_1^i])^{-1}(v^*)\) 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 \(k_1^i \neq \ldots \neq k_1^i\). Hence the call to ProcessCBSUBTree\((k_1^i, k_2, y_i, v^*, \leftarrow)\) 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 $|KSet_1|$ 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 $E_2^{-1}$ or $E_4$, 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 $CB(k_2)$.

(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 $E_2^{-1}$ and $E_4$ are almost the same. We thereby take a deep look into $E_2^{-1}$, and only sketch $E_4$.

By construction, upon a query $E_2^{-1}(k_2, w^*)$, if $w^* \in E_2[k_2]^{-1}$, then $G_2$ answers with $E_2[k_2]^{-1}(w^*)$; if $w^* \notin E_2[k_2]^{-1}$, then $G_2$ draws the random answer from $E$. 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 $\text{FindPebLeafB3}$. 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 $\text{FindPebLeafB3}$ 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 $\text{FindPebLeafB3}$; as $\text{FindPebLeafB3}$ does not modify $Li(k_2, w^*)$, $Li(k_2, w^*)$ has no pebbled leaf after $\text{FindPebLeafB3}$ returns. By construction, a call to $\text{ProcessNonPebB3Tree}(k_2, w^*, left)$ 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 $T$ only queries $\hat{C}$ during the chain-completion phase of the subsequent $\text{ProcessB3SubTree}$-calls, so that each newly-created C-query is a part of a completed path right after the corresponding $\text{Adapt}$-call. This claim holds keeping 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 $\text{ProcessPebB3Tree}$ is made. Depending on the arguments to $\text{ProcessPebB3Tree}$ and the concrete flow of $\text{ProcessPebB3Tree}$ (depending on concrete conditions, $\text{ProcessPebB3Tree}$ may call $\text{ProcessPebB3Tree}$ or $\text{ProcessDualTree}$), the discussions are divided into four cases, as follows:

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

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

Therefore, the call $\text{ProcessB3SubTree}(k_2^*, k_2, w^*, x^*, left)$ 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 $\text{ProcessB3SubTree}$-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, $T$ only queries $\hat{C}$ during the chain-completion phase of the calls to $\text{ProcessB3SubTree}$, hence (c) holds.

---

17 Formally speaking, the claim in (a) means: if $\text{ProcessDualTree}(k_1, k_2, v, w, 2)$ is subsequently called, then $|E_3|$ increases by at most $|T_{k_2,w}| \cdot |KSet_1|$; if $\text{ProcessDualTree}(k_1, k_2, x, y, 4)$ 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 $E_2^{-1}/E_4$. 

41
Case 2: the call is ProcessPebB3Tree\((k_1^*, k_2, w^o, x^o, \text{left})\), and ProcessPebB3Tree subsequently calls ProcessDualTree. In this case, it necessarily be that ProcessPebB3Tree computes a value \(v^\circ \neq \bot\) (thus calling ProcessDualTree\((k_1^*, k_2, v^o, w^o, 2)\) then). To establish the claims, we proceed to argue that this ProcessDualTree-call is safe.

As the starting point, we argue that \(v^o\) is non-pebbled before the forall loop in 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 \(E2^{-1}(k_2, w^o)\): a 4-query \((k_1^*, k_2, x^o, y^o, d_4, n_4)\), a C-query \((k_1^*, k_2, u^o, d_C, n_C)\), a 1-query \((k_1^*, u^o, v^o, d_1, n_1)\), and a 2-query \((k_2, k_2, v^o, d_2, n_2)\). Then the four queries had to be in the same completed path, contradicting the assumption that \(x^o\) was non-pebbled:

- if \(n_C > n_1\), then \(d_C\) must be \(\rightarrow\) 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 \(d_1 = \rightarrow\) and \(n_2 > n_1\) and the queries are in the same completed path by Lemma 3.

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

Then, depending on the flow of ProcessPebB3Tree, we’ve two subcases.

Subcase 2.1: FindPebLeafCB is not called. Then it necessarily be that \(v^o \notin E_1[k_1^*]^{-1}\) for any \(k_1^* \in KSet_1 \setminus \{k_1\}\), and \(T_{k_2, v^o}\) consists of only one edge \((k_1^*, y^o, v^o)\). It’s then an easy task to check that the subsequent call ProcessDualTree\((k_1^*, k_2, v^o, w^o, 2)\) 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: FindPebLeafCB is called. Note that \(y^o\) is the unique pebbled node in \(T_{k_2, v^o}\). Moreover, the call FindPebLeafCB\((k_2, v^o, \text{right})\) would not cause \(G_2\) abort, as it can only cause early-abort, which is assumed absent. Let \(TX_{k_2, v^o}\) be the snapshot of the live tree \(Li(k_2, v^o)\) standing right after FindPebLeafCB\((k_2, v^o, \text{right})\) returns. Then the following hold by Lemma 6:

(i) \(y^o\) remains the unique pebbled node in \(TX_{k_2, v^o}\) (Lemma 6 (a));
(ii) \(TX_{k_2, v^o}\) is complete (Lemma 6 (b));
(iii) \(|TX_{k_2, v^o}| \leq |T_{k_2, v^o}| \cdot |KSet_1|\) (Lemma 6 (c)). Note that here \(|T_{k_2, v^o}| \geq 1\);
(iv) Each C-query created in FindPebLeafCB \((k_2, v^o, \text{right})\) returns. Then the following hold by Lemma 6:

As a consequence, the call ProcessDualTree\((k_1^*, k_2, v^o, w^o, 2)\) is safe, and the statements are proved as follows:

(i) \((a)\) follows from Lemma 9 (b) and \(|TX_{k_2, v^o}| \leq |T_{k_2, v^o}| \cdot |KSet_1|\);
(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 ProcessTree-calls or during the call to 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 \(TX_{k_2, v^o}\) (cf. the remark (iv) above): by Lemma 9 (b), each edge of \(TX_{k_2, v^o}\) will be in a completed path. By this, each C-query newly created in 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 ProcessPebB3Tree\((k_1^*, k_2, w^o, x^o, \text{right})\), and ProcessPebB3Tree subsequently calls ProcessB3SubTree. This case is very similar to Case 1: right before the call ProcessB3SubTree\((k_1^*, k_2, w^o, x^o, \text{right})\) is made, it holds:

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

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

Case 4: the call is ProcessPebB3Tree\((k_1^*, k_2, w^o, x^o, \text{right})\), and ProcessPebB3Tree subsequently calls ProcessDualTree. In this case, it necessarily be \(v^o \in E_1[k_1^*]^{-1}\). Let \(w^o := E_1[k_1^*]^{-1}(v^o)\). Then depending on the state of the history, the discussions are divided into three subcases.
Subcase 4.1: \( u^o \notin CTable([k_1^2, k_2]) \) before the cycle. In this case, \( \text{ProcessPebB3Tree} \) creates a new C-query \( ((k_1^2, k_2), u^o, y^o, \rightarrow) \) via querying \( C \). Right after this point, if early-abortion does not occur, then it holds \( y^o \notin E_4[k_2]^{-1}/y^o \) is non-pebbled (by Inv4; we thereby let \( T_{k_2, y^o} = \text{Li}(k_2, y^o) \) at this point) and (by Inv3)

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

By (9), \( \text{FindEdgeInCB} \) returns \( \perp \) for any \( k_1' \in \text{KSet}_1 \setminus \{k_1^2\} \), and \( \text{FindPebLeafCB} \) is never called. Consequently, \( T_{k_2, y^o} \) consists of only one edge \( (k_1^2, y^o, v^o) \) (with \( v^o \) being pebbled), and the call \( \text{ProcessDualTree}(k_1^2, k_2, v^o, w^o, 2) \) is safe. The statements are then proved as follows:

(i) The claims on \( |\text{KSet}_1| \) and \( |\text{KSet}_2| \) follow from Lemma 9 (b). As to the claim on \( |E_3| \), note that by (9) and the fact that the edge \( (k_1^2, y^o, v^o) \) is newly added to \( C(B(k_2)) \), the live tree \( \text{Li}(k_2, y^o) \) standing before this simulator cycle indeed has NO edge—i.e. \( |T_{k_2, y^o}| = 0 \). On the other hand, by (9) and by inspection of the code of \( \text{ProcessDualTree} \), it can be seen that the call \( \text{ProcessDualTree}(k_1^2, k_2, v^o, w^o, 2) \) only makes safe calls to \( \text{ProcessPebB3Tree} \) 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 \( |T_{k_2, y^o}| = 0 \) 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 \( \text{ProcessTree} \)-calls, the \( \text{ProcessPebB3Tree} \)-call itself creates a new C-query (i.e. \( ((k_2^2, k_2), u^o, y^o, \rightarrow) \)). Similarly to subcase 2.2, the former type of new C-queries are in completed paths, while the C-query created by \( \text{ProcessPebB3Tree} \) is in the edge of \( T_{k_2, y^o} \), which will be in a completed path by Lemma 9 (b). By this, (e) is established.

Subcase 4.2: \( u^o \in CTable([k_1^2, k_2]) \) before the cycle, and \( \text{FindPebLeafCB} \) is not called. Then it necessarily be that \( \text{FindEdgeInCB} \) returns \( \perp \) for any \( k_1' \in \text{KSet}_1 \setminus \{k_1^2\} \) (otherwise \( \text{FindPebLeafCB} \) would be called). Similarly to Case 2, \( y^o = CTable([k_1^2, k_2])(u^o) \) must be non-pebbled before the query \( E_2^{-1}(k_2, w^o) \): otherwise either the 1-query \( (1, k_2^2, u^o, v^o) \) or the 4-query which pebbles \( y^o \) has an associated \( \text{num} \) value larger than the \( \text{num} \) value of the edge \( (k_1^2, y^o, v^o) \) (due to Inv4), and the four queries are in the same completed path (by Lemma 3) and \( x^o \) would have been pebbled. By this, right before \( \text{ProcessDualTree} \) is called, the tree \( \text{Li}(k_2, y^o) \) consists of only one edge \( (k_1^2, y^o, v^o) \) (with \( v^o \) being pebbled). It’s then clearly that the call \( \text{ProcessDualTree}(k_1^2, k_2, v^o, w^o, 2) \) 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 \( \text{FindPebLeafCB} \)).

Indeed, the mere difference between subcase 4.2 and subcase 4.1 is that \( \text{ProcessPebB3Tree} \) does not create new C-queries in subcase 4.2.

Subcase 4.3: \( \text{FindPebLeafCB} \) is called. This case is very similar to subcase 2.2. First, \( y^o \) must be non-pebbled before the current cycle, otherwise \( x^o \) would have been pebbled (this argument is similar to subcase 4.2). By this, \( v^o \) is the unique pebbled node in \( T_{k_2, y^o} \). Let \( TX_{k_2, y^o} \) be the snapshot of the live tree \( \text{Li}(k_2, y^o) \) standing right after \( \text{FindPebLeafCB}(k_2, y^o, \text{le,flt}) \) returns (this call indeed returns without abortion because early-abortion is assumed absent). Then by Lemma 6 (a), \( v^o \) remains the only pebbled node in \( TX_{k_2, y^o} \); and by Lemma 6 (b), \( TX_{k_2, y^o} \) is complete. As a consequence, the call \( \text{ProcessDualTree}(k_1^2, k_2, v^o, w^o, 4) \) is safe, and the statements can be established similarly to subcase 2.2. These complete the analysis of the simulator cycles due to \( E_2^{-1} \).

We then sketch the analysis of a cycle due to \( E_4(k_2, x^* \}) \). If \( T_{k_2, x^*} \) has no pebbled leaf, then the analysis is similar to the case of \( E_2^{-1}(k_2, v^*) \) 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 \( E_4^{-1}(k_2, y^*) \), the following hold:

(a) \( |\text{KSet}_1| \) stays constant, \( |\text{KSet}_2| \) increases by at most 1, while \( |E_3| \) increases by at most \( |T_{k_2, y^*}| \cdot |\text{KSet}_1| \).
(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. Similarly to Lemma 12, we focus on a new query $E_{4,1}^{-1}(k_2, y^*)$. If $T_{k_2, y^*} = \{y^*\}$, then for no $k_1$ does \texttt{FindEdgeINC}B return \texttt{true}, so that the call \texttt{Exists14Tripwire}(k_2, y^*) (at the beginning of $E_{4,1}^{-1}$) returns false, and $G_2$ simply calls \texttt{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 \texttt{Exists14Tripwire}), then it calls \texttt{FindPebLeafCB}(k_2, y^*, \texttt{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 \texttt{FindPebLeafCB}(k_2, y^*, \texttt{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) $|TX_{k_2, y^*}| \leq |T_{k_2, y^*}| - |KSet_1|$; (4) Each C-query created in \texttt{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 \texttt{FindPebLeafCB}; gathering this and remark (2) (mentioned before) establishes the safeness of the subsequent call to \texttt{ProcessNonPebCBTree}(k_2, y^*, \texttt{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 \texttt{FindPebLeafCB} will be in the edges of $TX_{k_2, y^*}$ which will further be in completed paths, while those due to layer-2 \texttt{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 \texttt{ProcessPebCBTree} is made, and the discussions are divided into four cases depending on the concrete flow:

**Case 1:** the call is \texttt{ProcessPebCBTree}(k^o_1, k_2, y^*, v^o, \texttt{left}), and \texttt{ProcessPebCBTree} subsequently calls \texttt{ProcessCBS}B\texttt{SubTree}. Then by construction, right before the call \texttt{ProcessCBS}B\texttt{SubTree}(k^o_1, k_2, y^*, v^o, \texttt{left}) is made, it holds:

(i) $v^o$ is pebbled;
(ii) $y^*$ is non-pebbled; $Li(k_2, y^*)$ is complete and has only one pebbled leaf (since $Li(k_2, y^*) = TX_{k_2, y^*}$);
(iii) For $w^o := E_2[k_2](v^o)$, it holds $w^o \notin E_3[k^o_1]$.

Therefore, the call \texttt{ProcessCBS}B\texttt{SubTree}(k^o_1, k_2, y^*, v^o, \texttt{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 \texttt{ProcessPebCBTree}(k^o_1, k_2, y^*, v^o, \texttt{left}), and \texttt{ProcessPebB3Tree} subsequently calls \texttt{ProcessDualT}ree. In this case, it necessarily be $w^o \in E_3[k^o_1]$ ($w^o = E_3[k_2](v^o)$) and \texttt{ProcessDualT}ree($k^o_1, k_2, x^o, y^*, 4$) is then called ($x^o = E_3[k^o_1](w^o)$). In order to utilize Lemma 9, we show the safeness of this \texttt{ProcessDualT}ree-call. Similarly to Case 2 in Lemma 12, we first argue that $x^o$ is non-pebbled before the \texttt{ProcessDualT}ree-call. Assume otherwise, then before the current simulator cycle, there already existed
a 2-query \((2, k_2, v^o, w^o, d_2, n_2)\), a 3-query \((3, k_1^2, w^o, x^o, d_3, n_3)\), and a 4-query \((4, k_2, x^o, y^o, d_4, n_4)\) (these queries indeed existed before the cycle because they cannot be created by \text{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, \(w^o\) is a pebbled leaf of the tree \(T_{k_2,x}\) (before the current simulator cycle); as \(T_{k_2,x}\) has at most one pebbled leaf (Lemma 5), \(w^o\) is the unique pebbled node. By this and the remarks above ((1) and (2)), the subsequent \text{ProcessDualTree}-call is safe. The statements thereby follow from Lemma 9 and an argument similar to the case of \(|\text{OriginSet}| = 0\).

**Case 3:** the call is \text{ProcessPebCBTree\(k_1^1, k_2, y^o, v^o, \text{right}\)}, and \text{ProcessPebCBTree} subsequently calls \text{ProcessCBSSubTree}. Symmetrically to Case 1, right before the call \text{ProcessPebCBTree\(k_1^1, k_2, y^o, v^o, \text{right}\)} is made, it holds:

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

The subsequent call \text{ProcessCBSSubTree\(k_1^1, k_2, y^o, v^o, \text{right}\)} is thereby safe, and the proof of the statements is similar to Case 1 by symmetry.

**Case 4:** the call is \text{ProcessPebCBTree\(k_1^0, k_2, y^o, v^o, \text{right}\)}, and \text{ProcessPebCBTree} subsequently calls \text{ProcessDualTree}. In this case, it must be \(x^o \in E_3[k_1^1]^{-1}(x^o = E_4[k_2]^{-1}(y^o))\). Let \(w^o := E_3[k_1^1]^{-1}(x^o)\). By an argument similar to Case 2, \(w^o\) must be non-pebbled, and \(x^o\) is the unique pebbled node in \(T_{k_2,w^o}\). Hence the subsequent \text{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 \(E2(k_2, v^*)\), the following hold:

(a) \(|K\text{Set}_2| \) stays constant, \(|K\text{Set}_3| \) increases by at most 1, while \(|E_3| \) increases by at most \(\text{Max}\{ |T_{k_2,v^s}|, 1 \} \cdot |K\text{Set}_1|\).

(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 \text{RandomAssign} to answer and the statements clearly hold. Otherwise, by construction, there is to be a call to \text{FindPebLeafCB}(k_2, v^*). Let \(TX_{k_2,v^*}\) be the snapshot of \(L_i(k_2, v^*)\) standing right after \text{FindPebLeafCB} returns. Then by Lemma 6: (1) the pebbled nodes of \(TX_{k_2,v^*}\) are also the pebbled nodes of \(TX_{k_2,v^*}\); (2) \(TX_{k_2,v^*}\) is complete; (3) \(|TX_{k_2,v^*}| \leq \text{Max}\{ |T_{k_2,v^s}|, 1 \} \cdot |K\text{Set}_1|\); (4) Each C-query created in \text{FindPebLeafCB} is a part of an edge in \(TX_{k_2,v^*}\); indeed, observation (3) is the mere difference between the cycles due to \(E2(k_2, v^*)\) and the cycles due to \(E4^{-1}(k_2, y^*)\) (cf. Lemma 13). The analysis follows the same line as Lemma 13 and has no novelty except for replacing \(|T_{k_2,v^s}| \cdot |K\text{Set}_1|\) (as in Lemma 13 (a)) by \(\text{Max}\{ |T_{k_2,v^s}|, 1 \} \cdot |K\text{Set}_1|\).

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 \(T\) 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 \(T\) 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 \(T\) (till that point) is a part of a completed path, and has an associated 4-tuple \((k_1, k_2, x, 4)\) in \text{Completed}.

(b) \(\sum_{k_2 \in \{0,1\}} |CB(k_2)| \leq q\), 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, \(T\) 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.
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 Lemma 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 Lemma 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 $\max\{|T_{k_2,v}|, 1\} \cdot |KSet_1| \leq (|T_{k_2,v}| + 1) \cdot |KSet_1|$. 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 $\overline{C}$;
  - 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 $\overline{D}$.

Fig. 17. For Lemma 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, $|E_3|$ increases by at most $\max\{|T_{k_2,y}|, 1\} \cdot |KSet_1|$.

(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 $T$ (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. □
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 $\overline{\mathcal{D}}$ ($i \leq q$), then $|E_3|$ increases by at most $\sum_{i}(|T_{CB,i}| + 1) \cdot |KSet_1| \leq \sum_{i} |T_{CB,i}| \cdot |KSet_1| + q \cdot |KSet_1|$ 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 $\overline{\mathcal{D}}$, then $|E_3|$ increases by at most $\sum_{j} |T_{CB,j}| \cdot |KSet_1|$. By the above and the fact that $\overline{\mathcal{D}}$ makes at most $q$ queries to $\overline{\mathcal{C}}$, $\sum_{i} |T_{CB,i}| + \sum_{j} |T_{CB,j}| \leq q$; by Lemma 16, $|KSet_1| \leq q$. Therefore the bound in total is $|E_3| \leq q + (q + \sum_{i} |T_{CB,i}| + \sum_{j} |T_{CB,j}|) \cdot |KSet_1| \leq 3q^2$. \hfill \Box

$[\text{Completed}]$ is finite; $\mathbf{T}$ completes at most $3q^3$ chains.

**Lemma 18.** At the end of $D^{G_2}$, there are at most $3q^3$ tuples of the form $(k_1, k_2, x, 4)$ in $\text{Completed}$. 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 $(k_1, k_2, x, 4) \in \text{Completed}$, it holds $x \in E_3[k_2]$, $x \in E_3[k_1]^{-1}$, $k_1 \in KSet_1$ and $k_2 \in KSet_2$. Then the bound $10q^3$ follows from $|KSet_2| \leq q$ and $|E_3| \leq 3q^2$ (Lemmata 16 and 17). Similarly for $(k_1, k_2, v, 2)$: it holds $v \in E_3[k_2]$, $w \in E_3[k_1]$ for $w := E_2[k_2](v)$, and $k_1 \in KSet_1$, and the number of such tuples is thereby at most $|KSet_2| \cdot |E_3| \leq 3q^3$. \hfill \Box

The next lemma bounds the rest variables around $\mathbf{T}$.

**Lemma 19.** At the end of $D^{G_2}$, $|E_2| \leq 4q^3$, $|E_4| \leq 4q^3$, $|E_1| \leq 4q^3$, and $\mathbf{T}$ makes at most $16q^9$ distinct calls to $\text{C.Check}$.

**Proof.** Consider $|E_2|$ first. $\mathbf{T}$ may create a new 2-query with $\text{dir} \neq \perp$ during E2, E2$^{-1}$, PROCESSNonPebCBTree, and PROCESSNonPebB3Tree. But by inspection of the execution, such 2-queries are indeed due to $\overline{\mathcal{D}}$’s 2-queries, and should be “billed” to $\overline{\mathcal{D}}$. As $\overline{\mathcal{D}}$ makes at most $q$ 2-queries, the number of 2-queries with $\text{dir} \neq \perp$ is at most $q$. On the other hand, $\mathbf{T}$ only creates adapted 2-queries in calls to layer-2 PROCESSTREE procedures, and once an adapted 2-query $(k_1, k_2, v, w, \perp)$ is created, a tuple $(k_2, k_2, v, 2)$ will be added to $\text{Completed}$ (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 $q + 3q^3 \leq 4q^3$. 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 $\overline{\mathcal{D}}$’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 $\text{C.Check}$ is easily bounded to $|E_1| \cdot |E_4| \leq 16q^6$. \hfill \Box

The next lemma bounds $|CQueries|$.

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

**Proof.** The proof will establish two goals: first, associating a unique $\text{C}$-query $((k_1, k_2), u, y) \in CQueries$ to each 4-tuple $(k_1, k_2, x, 4) \in \text{Completed}$; second, associating a unique 4-tuple $(k_1, k_2, x, 4) \in \text{Completed}$ to each $\mathbf{C}$-query $((k_1, k_2), u, y) \in CQueries$. The first goal is easier: by Lemma 1, for $(k_1, k_2, x, 4) \in \text{Completed}$, there is a corresponding completed path and the associated $\mathbf{C}$-query is the one in this path. Furthermore, the same $\mathbf{C}$-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 $k_1 \neq k'_1$ or $k_2 \neq k'_2$, 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 $x \neq x'$, 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 $\mathbf{C}$-queries created due to $\mathbf{T}$ querying $\overline{\mathbf{C}}$, the existence of the associated 4-tuples has been shown by Lemma 15 (a). For the $\mathbf{C}$-queries created due to $\overline{\mathbf{D}}$ querying $\overline{\mathbf{C}}$, we have the following reasoning: as $\overline{\mathcal{D}}$ completes all chains (cf. page 16), right after some simulator cycle in $\overline{D}^{G_2}$, there exists four queries $(1, k, u, v, \text{dir}_1, \text{num}_1)$, $(2, k_2, v, w, \text{dir}_2, \text{num}_2)$, $(3, k_1, w, x, \text{dir}_3, \text{num}_3)$, $(4, k_2, x, y', \text{dir}_4, \text{num}_4)$. With respect to the four queries, we have: $\text{dir}_3 \rightarrow \text{r}$ due to Inv0; so that by Inv1, either $\text{num}_2 > \text{num}_3$ or $\text{num}_4 > \text{num}_3$; 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 \(C\), it must be \(u \neq u' \wedge y \neq y'\) and \(E_1[k_2]^{-1}(y) \neq E_4[k_2]^{-1}(y')\). 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.

The last lemma bounds the running time of \(T\).

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

**Proof.** As \(T\) 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 14) and the code, we observe that the *longest possible* cycle due to \(E_4^{-1}\) or \(E_2\) consists of the following two steps:

\[
\text{FindPebLeafCB} \rightarrow \text{ProcessNonPebCBTree/ProcessPebCBTree}.
\]

The second step \(\text{ProcessNonPebCBTree} or \text{ProcessPebCBTree}\) consists of a series of calls to layer-2 \(\text{ProcessTree}\) procedures.

On the other hand, the *longest possible* cycle due to \(E_4\) or \(E_2^{-1}\) consists of the following two steps:

\[
\text{FindPebLeafB3} \rightarrow \text{ProcessPebB3Tree},
\]

and the second step \(\text{ProcessPebB3Tree}\) is further divided into two steps: first calling \(\text{FindPebLeafCB}\), then making a series of calls to layer-2 \(\text{ProcessTree}\) 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 \(\text{FindPebLeafB3}\). 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 \(\text{ProcessPebB3Tree}\) is started till the point the layer-2 \(\text{ProcessTree}\) procedure (either \(\text{ProcessB3SubTree}\) or \(\text{ProcessDualTree}\)) is called. Then \(t_2 = O(q^5)\). For this, consider the process of \(\text{ProcessPebB3Tree}\) in the case \(pos = left\) first, and consider the call to \(\text{FindEdgeInCB}\) (which loops for all \(u \in E_1[k_2]\)) 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 \(\text{FindPebLeafCB}\) is never executed inside the latter loop. Due to the tag \(Traversed\), \(\text{FindPebLeafCB}\) is executed at most once, so that when \(pos = left\), we have \(t_2 = O(q^5) + t^*\), where \(t^*\) is the running time of \(\text{FindPebLeafCB}\). 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 \(\text{FindPebLeafCB}\)) 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 \(\text{ProcessPebB3Tree}\) is started and the point the layer-2 \(\text{ProcessTree}\) procedure is called costs \(O(q^3)\) time; on the other hand, \(\text{FindPebLeafCB}\) 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 \(\text{ProcessTree}\) calls runs in \(t_3 = O(q^6)\) in total. As the size of a live tree in \(CB(k_2)\) (extended by \(\text{FindPebLeafCB}\)) 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 \(\text{ProcessTree}\) procedures. By inspection of layer-2 \(\text{ProcessTree}\) procedures and noting \(|KSet_1|, |KSet_2| \leq q\) and \(|E_1| = O(q^3)\), the time cost of a layer-2 \(\text{ProcessTree}\)-call is dominated by the subsequent call to \(\text{RecurseNew}\), which is \(O(q^5)\). By this, the time cost \(t_3 = O(q^6)\), and the time cost of a single cycle is obtained through \(t_1 + t_2 + t_3 = O(q^9)\). □

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

**Proposition 3.** The probability that \(G_2\) aborts inside a call to \(\text{AddCQuery}\) is at most \(\frac{21q^6}{2^{n-3q^2}}\).
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 \( LS_0 \leq |C\text{Queries}| \leq 3q^3 \) and \( RS_0 \leq |C\text{Queries}| \leq 3q^3 \). So a single call to \( \text{ADDQUERY} \) induces abortion with probability at most \( \frac{8q^4}{2^n-3q^3} \) regardless of the value of the involved parameter \( \text{dir} \), and the probability in total is at most \( \frac{21q^6}{2^n-3q^3} \). \( \Box \)

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

Proof. Consider calls to \( \text{ADDQUERY}(1, k_1, u, v, \text{dir}) \) first. Note \( \max\{|RS_0|, |LS_1|, |RS_1|, |LS_2|\} \leq 8q^3 \) by Lemmata 19 and 20. So a single call to \( \text{ADDQUERY}(1, k_1, u, v, \text{dir}) \) induces abortion with probability at most \( \frac{8q^3}{2^n-4q^3} \), regardless of \( \text{dir} \), and the bound in total is \( \frac{32q^6}{2^n-4q^3} \). Similar argument establish the bound \( \frac{21q^7}{2^n-3q^3} \) for calls to \( \text{ADDQUERY}(3, k_1, w, x, \text{dir}) \).

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

**Proposition 5.** The probability that \( G_2 \) aborts inside calls to \( E_2, E_2^{-1}, E_4, \) and \( E_4^{-1} \) (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). \( \Box \)

**Proposition 6.** The probability that \( G_2 \) aborts inside calls to \( \text{ADAPT} \) is \( 0 \).

Proof. By Lemmata 12-14 and follows the same line as Proposition 5. \( \Box \)

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

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

Note that the aborts inside \( \text{RANDOMASSIGN} \) while outside the subsequent \( \text{ADDQUERY} \) (say, the aborts 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 \( \text{ADDQUERY} \), and the probability has been accounted in the probability of those in \( \text{ADDQUERY} \) (Proposition 4). For clearness, consider the case of a call to \( \text{RANDOMASSIGN}(1, +, k_1, u) \). Let \( S_h := RS_1 \cup LS_2 \). Then by noting that \( v \in S_h \Rightarrow v \in E_1[k_1]^{-1} \), it holds

\[
\Pr[G_2 \text{ aborts in } \text{RANDOMASSIGN} \text{ or } \text{ADDQUERY}] = \Pr[v \in E_1[k_1]^{-1} \land v \in S_h] + \Pr[v \notin E_1[k_1]^{-1} \land v \in S_h] = \Pr[v \in S_h] = \Pr[G_2 \text{ aborts in } \text{ADDQUERY}].
\]

The other possibilities are similar. \( \Box \)

### 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. [14]. As we avoid the two-sided random function used in [14], our argument is closer to the corresponding part in [29]. More clearly, we first specify a bad event BadCheck1 in $G_1$, which captures the possible differences brought in by Check. We then formally prove that for a tuple $(C,E)$ such that $\overline{D}^{G_2}_{(C,E)}$ does not abort (such tuples would be called good $G_2$-tuples), if BadCheck1 does not happen during the $G_1$ execution $\overline{D}^{G_1}_{(C,E)}$, 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 $\tilde{C}.\text{Check}$ in $G_2$ depends on the history of $\tilde{C}$ which has a polynomial size, whereas the return value of $S.\text{Check}$ in $G_1$ is completely up to $C$ which indeed has an exponential size. To capture this difference, we use an event BadCheck1: for a tuple of random primitives $(C,E)$, BadCheck1 happens during the execution $\overline{D}^{G_1}_{(C,S^2_{C,E})}$ if $\exists (K,u,y)$ s.t. all the following hold:

(i) $S^{C,E}$ makes a call to $\text{Check}(K,u,y)$;
(ii) $C.C(K,u)=y$;
(iii) Before the call in (i), at no point outside the $\text{Check}$-calls was $C.C(K,u)$ or $C.C^{-1}(K,y)$ 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 $C$, $C^{-1}$, $E_i$, $E_i^{-1}$ ($i = 1, 2, 3, 4$), 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 $\tilde{C}.C$, $\tilde{C}.C^{-1}$, $\tilde{C}.\text{Check}$, $E.Ei$, and $E.Ei^{-1}$ queries issued by $\overline{D}$ and $T$;
(ii) in $\overline{D}^{G_1}$: all the $S.\text{Check}$, $E.Ei$, and $E.Ei^{-1}$ queries issued by $\overline{D}$ and $S$, all the queries $C.C$ and $C.C^{-1}$ issued by $\overline{D}$, and all the queries $C.C$ and $C.C^{-1}$ issued by $S$ outside the Check procedure.

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

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

(a) If BadCheck1 does not happen in the first $t$ calls to $S.\text{Check}$ during $\overline{D}^{G_1}_{(C,E)}$, then the transcripts (defined as above) of $\overline{D}^{G_1}_{(C,E)}$ and $\overline{D}^{G_2}_{(C,E)}$ are the same, and $\overline{D}$ gives the same output in the two executions.

(b) The assumption in (a) holds with probability at least $1 - 32q^n/2^n$.

**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 $S/T$ are deterministic, the next query will be the same. If the query is to $C.C/C.C^{-1}$ or $\tilde{C}.C/\tilde{C}.C^{-1}$, then the answers in the two executions will clearly be the same as both of them are due to the same ideal cipher $C$. Ditto for the case of a query to $E$. If the query is to Check, then the answers are the same by assumption of $\neg$BadCheck1. By construction, $S$ and $T$ will then proceed to the same sequence of operations until the next query, or either $S$ or $T$ aborts.

We then show that neither $S$ nor $T$ aborts. As the tuple $(C,E)$ in question is a good $G_2$-tuple, $T$ clearly never aborts. Since we have showed that the transcripts obtained till this point are the same, each time $S$ is to check an abort condition, $T$ is to perform exactly the same check operation, so that $T$'s non-abortion implies $S$'s non-abort.

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

18 The number 1 indicates that the event is defined for $G_1$. 

---

18 The number 1 indicates that the event is defined for $G_1$. 

---
if this call appears for the first time (i.e. \(\text{Check}(K, u, y)\) was never made before this point), then by the definition of \(\text{BadCheck1}\), the probability that \(\text{BadCheck1}\) happens with respect to this call is at most \(1/(2^n - q^*)\), where \(q^*\) is the total number of queries received by \(C\) in \(D^{(i)}\); 
if the call \(\text{Check}(K, u, y)\) has appeared before this point, then it further consists of two subcases:

- during the period between \(\text{Check}(K, u, y)\) was first made and the current point, either \(C(K, u)\) or \(C^{-1}(K, y)\) was issued at some point outside \(\text{Check}\). Then \(\text{BadCheck1}\) does not happen with respect to the later \(\text{Check}(K, u, y)\) call, as it does not meet the requirements;
- opposite to the previous subcase: neither \(C(K, u)\) nor \(C^{-1}(K, y)\) was issued (outside \(\text{Check}\)) during the period. Then \(\text{BadCheck1}\) does not happen with respect to the later \(\text{Check}(K, u, y)\) call, as it did not happen with respect to the first \(\text{Check}(K, u, y)\) call.

By the above, it already suffices to sum over the \(t'\) distinct \(\text{Check}\)-calls. As \(t' \leq 16q^6\) by Lemma 19, the overall probability that \(\text{BadCheck1}\) 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 \((C, E)\) is a good \(G_1\)-tuple if \(\text{BadCheck1}\) does not happen during \(D^{G_1}(C, S^{C,E})\). As a corollary of Lemma 23 (b), the probability that a random tuple \((C, E)\) is a good \(G_1\)-tuple is at least \(1 - (\frac{178q^6}{2^n} + \frac{32q^6}{2^n}) \geq 1 - \frac{210q^6}{2^n}\).

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

**Lemma 24.** During a \(G_1\) execution \(D^{G_1}(C, S^{C,E})\), with probability at least \(1 - 210q^6/2^n\), \(S\) issues no more than \(8q^4\) queries to \(C\) (assuming \(S\) 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 \((C, E)\) is a good \(G_1\)-tuple), the transcripts in \(D^{G_1}(C, S^{C,E})\) and \(D^{G_2}(C^{C,T}C^{C,E})\) are the same. Hence in \(D^{G_1}(C, S^{C,E})\), the bounds given in Lemma 16-20 hold with probability at least \(1 - 210q^6/2^n\). As \(S\) issues at most \(|E_1| \cdot |KSet_2| + |E_1| \cdot |KSet_1|\) queries to \(E\), 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 [14], while the formalism is similar to [11]. As the beginning, with respect to \(D\), we borrow some terminology from [2] and [11].

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

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

**Lemma 25.** Consider a good \(G_1\)-tuple \((C, E)\). At the end of the \(G_2\) execution \(D^{G_2}(C, E)\), for any \(C\)-query \(((k_1, k_2), u, y)\) in \(C\text{Queries}\), there are four queries (for some \(v, w,\) and \(x\)) in \(\text{Queries}\) 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 \(\text{Completed corresponds to a completed path}\) 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 \( \text{dir} = \bot \)) equals the number of \( \tilde{C}^C \)'s queries to \( C \). To show this, we need a helper proposition.

**Proposition 7.** Consider a good \( G_1 \)-tuple \((C, E)\). During \( \overrightarrow{D}^{G_2(C, E)} \), 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 \((k_1, k_2, x, 4) \in \text{Completed} \), 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. \( \square \)

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

\[|\{(i, k, z, z', \text{dir}) \in \text{Queries} : \text{dir} = \bot \}| = |C\text{Queries}|.\]

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

- C-queries \((k_1, k_2), u, y) \in C\text{Queries} \leftrightarrow 4\text{-tuples} \((k_1, k_2, x, 4) \in \text{Completed}\)
- \leftrightarrow \text{layer-2 ProcessTree-calls} \leftrightarrow \text{calls to Adapt/adapter queries}

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 \((k_1, k_2, x, 4) \in \text{Completed}\) (i.e. the tuple added to Completed during its chain-completion phase): for each \((k_1, k_2, x, 4) \in \text{Completed}\) we could associate a unique layer-2 ProcessTree-call (i.e. the call which adds it to Completed).

Moreover, by Proposition 7, two different layer-2 ProcessTree-calls in \( \overrightarrow{D}^{G_2(C, E)} \) 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.\(^{19}\) \( \square \)

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 \( \overrightarrow{D} \) in these two executions are the same.

**Lemma 27.** Let \( \alpha = (C, E) \) be a good \( G_1 \)-tuple, and denote by \( ET \) the sets of \( T \) standing at the end of \( \overrightarrow{D}^{G_2(\alpha)} \).

Then for any tuple \( E' \) such that \( E' \equiv ET \), the transcripts of queries and answers of \( \overrightarrow{D} \) in the two executions \( \overrightarrow{D}^{G_2(\alpha)} \) and \( \overrightarrow{D}^{G_2(E')} \) are the same; and \( \overrightarrow{D}^{G_2(\alpha)} = \overrightarrow{D}^{G_2(E')} \).

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

\(^{19}\) The proof appears much simpler than the analogue in [2] (Lemma 4 in page 26, the proof of which takes 4 pages) despite the closeness of the overall paradigms. The reason is: in [2], 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 [2].
(i) the query is to $E_i/E_i^{-1}$; then the answers are the same, since the answer obtained in $\overline{D}^G_{2}(a)$ equals the value in $ET$, and $E'$ coincides with $ET$;
(ii) the query is to $C/C^{-1}$; then due to Lemma 25 and the fact that $E^\prime \cong ET$, the answers obtained in $\overline{D}^G_{2}(a)$ and $\overline{D}^G_{2}(E')$ 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. \qed

For any $ET \in \mathcal{R}$, the probabilities of the following two events are close:
(i) a $G_2$ execution with a random tuple $(C, E)$ generates $ET$;
(ii) a random tuple $E$ coincides with $ET$.

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

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

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

$$Pr_{E} [E \cong ET] = \prod_{i=1,3} \prod_{k \in \{0,1\}^n} \prod_{j=0}^{[E_i[k]]-1} \prod_{j=0}^{2^n-j} \frac{1}{2}.$$  

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

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

By Lemma 26, it holds $\sum_{k \in \{0,1\}^n} |\overline{E}_2[k]| + \sum_{k \in \{0,1\}^n} |\overline{E}_4[k]| - \sum_{k \in \{0,1\}^n} |\overline{E}_2[k]| - \sum_{k \in \{0,1\}^n} |\overline{E}_4[k]| = v$. Furthermore $v \leq 3q^3$ by Lemma 20, hence

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

as claimed. \qed

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 $(C, E)$ and $(C', E')$, once $\overline{D}^G_{2}(C,E) \rightarrow ET$ and $\overline{D}^G_{2}(C',E') \rightarrow ET$, then $\overline{D}^G_{2}(C,E) \equiv \overline{D}^G_{2}(C',E')$. With this in mind, let $\Theta_1$ be the subset of $\mathcal{R}$ such that for any tuple $(C, E)$ such that $\overline{D}^G_{2}(C,E) \rightarrow ET \in \Theta_1$ it holds $\overline{D}^G_{2}(C,E) = 1$. 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. $Pr_{E} [\overline{D}^G_{3}(E) = 1] \geq \sum_{ET \in \Theta_1} Pr_{E} [E \cong ET].$
Proof. We show that for any tuple $E^*$, there exists at most one $ET \in \mathcal{R}$ s.t. $E^* \cong ET$. Assume otherwise, i.e. $\exists ET' \in \mathcal{R}$ s.t. $ET \not= ET' \land E^* \cong ET \land E^* \cong ET'$. Assume that for two good tuples $\alpha = (C,E)$ and $\alpha' = (C',E')$, it holds $D^{G_2(\alpha)} \rightarrow ET$ and $D^{G_2(\alpha')} \rightarrow ET'$. Then we show that the transcripts (cf. page 50) of the two executions $D^{G_2(\alpha)}$ and $D^{G_2(\alpha')}$ 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 $ET$: the answers are the same as $E_i E_i(k,z) = E_i'[k][z] = E_i E_i^{-1}(k,z) = E_i'[k]^{-1}(z) = E^* E_i E_i^{-1}(k,z) = E^* E_i'[k][z]$;
(ii) the query is to $C$: then by Lemma 25, the answers are the same;
(iii) the query is to $C$: then by Lemma 25, the answers are the same.

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

$$P_{ET}[D^{G_3(E)} = 1] \geq P_{ET}[D^{G_3(E)} = 1 \land \exists ET \in \mathcal{R} E \cong ET] = \sum_{ET \in \mathcal{T}_1} P_{ET}[E \cong ET] \ (by \ Lemma \ 27)$$

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 $P_{ET[D^{G_3(E)} \not= ET]}$ twice.

Lemma 30. $|P_{ET}[D^{G_3(CC^*,E)} = 1] - P_{TC,E}[D^{G_3(CC^*,TCC^*,E)} = 1]| \leq \frac{219f^6}{2^n}$.

Proof. Wlog assume that $P_{TC,E}[D^{G_3(E)} = 1] \geq P_{ET}[D^{G_3(E)} = 1]$, then

$$|P_{ET}[D^{G_3(E)} = 1] - P_{TC,E}[D^{G_3(E)} = 1]| \leq 219f^6 \ (cf. \ page \ 51)$$

$$+ \frac{P_{TC,E}[(C,E) \text{ is a good } G_1 \text{-tuple } \land D^{G_1(C,E)} = 1]}{P_{TC,E}[(C,E) \text{ is a good } G_1 \text{-tuple } \land D^{G_2(C,E)} = 1]} - P_{ET}[D^{G_3(E)} = 1]$$

$$\leq \frac{210f^6}{2^n} + \sum_{ET \in \mathcal{T}_1} (P_{TC,E}[D^{G_2(C,E)} \rightarrow ET] - P_{ET}[E \cong ET]) \ (by \ Lemma \ 29)$$

$$\leq \frac{210f^6}{2^n} + \sum_{ET \in \mathcal{T}_1} \frac{9f^6}{2^n} \cdot P_{TC,E}[D^{G_2(C,E)} \rightarrow ET] \ (by \ Lemma \ 28) \leq \frac{219f^6}{2^n}$$

as claimed. □