On the Indifferentiability of Key-Alternating Ciphers*

Elena Andreeva[1], Andrey Bogdanov[2], Yevgeniy Dodis[3], Bart Mennink[1], and John P. Steinberger[4]

[1] KU Leuven and iMinds, {elena.andreeva, bart.mennink}@esat.kuleuven.be
[2] Technical University of Denmark, a.bogdanov@mat.dtu.dk
[3] New York University, dodis@cs.nyu.edu
[4] Tsinghua University, jpsteinb@gmail.com

Abstract. The Advanced Encryption Standard (AES) is the most widely used block cipher. The high level structure of AES can be viewed as a (10-round) *key-alternating* cipher, where a t-round key-alternating cipher KA_t consists of a small number t of fixed permutations P_i on n bits, separated by key addition:

$$\mathrm{KA}_t(K, m) = k_t \oplus P_t(\ldots k_2 \oplus P_2(k_1 \oplus P_1(k_0 \oplus m))\ldots),$$

where $(k_0, \ldots, k_t)$ are obtained from the master key K using some key derivation function.
For $t = 1$, KA_1 collapses to the well-known Even-Mansour cipher, which is known to be *indistinguishable* from a (secret) random permutation, if P_1 is modeled as a (public) random permutation. In this work we seek for stronger security of key-alternating ciphers — *indifferentiability from an ideal cipher* — and ask the question under which conditions on the key derivation function and for how many rounds t is the key-alternating cipher KA_t indifferentiable from the ideal cipher, assuming $P_1, \ldots, P_t$ are (public) random permutations?
As our main result, we give an affirmative answer for $t = 5$, showing that the 5-*round key-alternating cipher* KA_5 *is indifferentiable from an ideal cipher*, assuming $P_1, \ldots, P_5$ are five independent random permutations, and the key derivation function sets all rounds keys $k_i = f(K)$, where $0 \le i \le 5$ and f is modeled as a random oracle. Moreover, when $|K| = |m|$, we show we can set $f(K) = P_0(K) \oplus K$, giving an n-bit block cipher with an n-bit key, making only six calls to n-bit permutations $P_0, P_1, P_2, P_3, P_4, P_5$.

Keywords. Even-Mansour, ideal cipher, key-alternating cipher, indifferentiability.

1 Introduction

Block Ciphers. A block cipher $E : \{0,1\}^\kappa \times \{0,1\}^n \to \{0,1\}^n$ takes a κ-bit key K and an n-bit input x and returns an n-bit output y. Moreover, for each key K the map $E(K, \cdot)$ must be a permutation, and come with an efficient inversion procedure $E^{-1}(K, \cdot)$. Block ciphers are central primitives in cryptography. Most importantly, they account for the bulk of data encryption and data authentication occurring in the field today, as well as play a critical role in the design of "cryptographic hash functions" [13,36,42,50].

Indistinguishability. The standard security notion for block ciphers is that of (computational) *indistinguishability* from a random permutation, which states that no computationally bounded distinguisher $\mathcal{D}$ can tell apart having oracle access to the block cipher $E(K, \cdot)$ or its inverse $E^{-1}(K, \cdot)$ for a *random key* K from having oracle access to a (single) truly random permutation P and its inverse P^{-1}. This security notion is relatively well understood in the theory community, and is known to be implied by the mere existence of one-way functions, through a relatively non-trivial path: from one-way functions to pseudorandom generators [38], to pseudorandom functions (PRFs) [32], to pseudorandom permutations (PRPs) [45], where the latter term is also a "theory synonym" for the "practical notion" of a block cipher. Among these celebrated results, we explicitly note the seminal work of Luby-Rackoff [45], who proved that four (independently keyed) rounds of the Feistel network $(L', R') = (R, f(K, R) \oplus L)$, also known as the "Luby-Rackoff construction", are enough to

* An extended abstract of this paper appears at CRYPTO 2013.

obtain a PRP $E((K_1, K_2, K_3, K_4), (L_0, R_0))$ on n-bit inputs/outputs from four $n/2$-to-$n/2$-bits PRFs $f(K_1, R_0), \ldots, f(K_4, R_3)$. In fact, modulo a few exceptions mentioned below, the Luby-Rackoff construction and its close relatives were the *only theoretically-analyzed* ways to build a block cipher.

Is Indistinguishability Enough? Despite this theoretical success, practical ciphers — including the current block cipher standard AES — are built using very different means. One obvious reason is that the theoretical feasibility results above are generally too inefficient to be of practical use (and, as one may argue, were not meant to be). However, a more subtle but equally important reason is that a practitioner — even the one who understands enough theory to know what a PRP is — would not think of a block cipher as a synonym of a PRP, but as something *much stronger*!

For example, the previous U.S. block cipher standard DES had the following so called "key complementary" property $E(\bar{K}, \bar{x}) = \overline{E(K, x)}$, where $\bar{y}$ stands for the bitwise complement of the string y. Although such an equality by itself does not contradict the PRP property, though effectively reducing the key space by a half, it was considered undesirable and typically used as an example of something that a "good" block cipher design should definitely avoid. Indeed, AES is not known to have any simple-to-express relations between its inputs/outputs on related keys. Generally speaking though, related-key attacks under more complex related-key relations (using nonlinear functions on the master key) for AES were identified and received a lot of attention in the cryptanalytic community several years ago [6,7,8], despite not attacking the standard PRP security. In fact, the recent biclique cryptanalysis of the full AES cipher [11] in the single-key setting implicitly uses the similarity of AES computation under related keys.

Indeed, one of the reasons that practical block ciphers are meant to have stronger-than-PRP properties is that various applications (e.g. [10,23,29,33,36,39,40,42,49,50,55]) critically rely on such "advanced properties", which are far and beyond the basic indistinguishability property. Perhaps the most important such example comes in the area of building good "hash functions", as many cryptographic hash functions, including the most extensively used SHA-1/2 and MD5 functions, use the famous block-cipher-based Davies-Meyer compression function $f(K, x) = E(K, x) \oplus x$ in their design.[5] This compression function f is widely believed to be collision-resistant (CR) if E is a "good-enough" block cipher (see more below), but this obviously does not follow from the basic PRP property. For example, modifying any good block cipher E to be the identity permutation on a single key K' clearly does not affect it PRP security much (since, w.h.p., a random key $K \neq K'$), but then $f(K', x) = x \oplus x = 0$ for all x, which is obviously not CR. While the example above seems artificial, we could instead use a natural and quite popular Even-Mansour (EM) [29] cipher $E(K, x) = P(K \oplus x) \oplus K$, where P is some "good-enough" public permutation. As we mention below, the EM cipher is known to be indistinguishable [29] assuming P is a public "random permutation", and, yet, the composed Davies-Meyer hash function $f(K, x) = E(K, x) \oplus x = P(K \oplus x) \oplus (K \oplus x)$ is certainly not CR, as any pair $(K, x) \neq (K', x')$ satisfying $K \oplus x = K' \oplus x'$ yields a collision.

Ideal Cipher Model. Motivated by these (and other) considerations, practitioners view a good block cipher as something much closer to an *ideal cipher* than a mere PRP, much like they view a good hash function much closer to a *random oracle* than a one-way (or collision-resistant) function. In other words, many important applications of block ciphers (sometimes implicitly) assume that E "behaves" like a family $\mathcal{IC}$ of 2^κ completely random and independent permutations $P_1, \ldots, P_{2^\kappa}$. More formally, an analysis in the ideal cipher model assumes that all parties, including the adversary, can make (a bounded number of) both encryption and decryption queries to the ideal block cipher $\mathcal{IC}$, for any given key K (not necessarily random!). Indeed, under such an idealistic assumption one can usually *prove* the security of most of the above mentioned applications of block ciphers [23,29,33,36,39,40,42,49,50,55],

[5] Where E is some particular block cipher; e.g., in the case if SHA-1/2, it was called SHACAL [34,35].

such as a simple and elegant proof that the Davies-Meyer compression function $f(K, x) = E(K, x) \oplus x$ is CR in the ideal cipher model (ICM) [55].

Of course, the ideal cipher model is ultimately a heuristic, and one can construct artificial schemes that are secure in the ICM, but insecure for any concrete block cipher [9]. Still, a proof in the ideal cipher model seems useful because it shows that a scheme is secure against generic attacks, that do not exploit specific weaknesses of the underlying block cipher. Even more important than potential applications, the ICM gives the block cipher designers a much "higher-than-PRP" *goal* that they should strive to achieve in their proposed designs, even though this goal is, theoretically-speaking, impossible to achieve. This raises an important question to the theory community if it is possible to offer some theoretical framework within which one might be able to evaluate the design of important block ciphers, such as AES, in terms of being "close" to an ideal cipher or, at least, resisting generic "structure-abusing" attacks.

Indifferentiability. One such framework is the so-called *indifferentiability* framework of Maurer et al. [46], popularized by Coron et al. [16] as a clean and elegant way to formally assess security of various idealized constructions of hash functions and block ciphers. Informally, given a construction of one (possibly) idealized primitive B (i.e., block cipher) from *another idealized* primitive A (i.e., random oracle), the indifferentiability framework allows one to formally argue the security of B in terms of (usually simpler) A. Thus, although one does not go all the way to building B from scratch, the indifferentiability proof illustrates the lack of "generic attacks" on B, and shows that any concrete attack must use something about the internals of any candidate implementation of A. Moreover, the indifferentiability framework comes with a powerful composition theorem [46] which means that most natural (see [51]) results shown secure in the "ideal-B" model can safely use the construction of B using A instead, and become secure in the "ideal-A" model.

For example, we already mentioned that the design of popular hash functions, such as SHA-1/2 and MD5, could be generically stated in terms of some underlying block cipher E. Using the indifferentiability framework, one can *formally* ask if the resulting hash function is indifferentiable from a random oracle if E is an ideal cipher. Interestingly, Coron et al. [16] showed a negative answer to this question. Moreover, this was not a quirk of the model, but came from a well-known (and serious) "extension" attack on the famous Merkle-Damgård domain extension [21,47]. Indeed, an attack on indifferentiability usually leads to a serious real-world attack for some applications, and, conversely, the security proof usually tells that the high-level design of a given primitive (in this case hash function) does not have structural weaknesses. Not surprisingly, all candidates for the recently concluded SHA-3 competition were strongly encouraged to come with a supporting indifferentiability proof in some model (as we will expand on shortly).

Random Oracle vs. Ideal Cipher. Fortunately, Coron et al. [16] also showed that several simple tweaks (e.g., truncating the output or doing prefix-free input encoding) make the resulting hash function construction indifferentiable from a random oracle. Aside from formally showing that the ICM model "implies" the random oracle model (ROM) in theory, these (and follow-up [3,14]) positive results showed that (close relatives of) *practically used* constructions are "secure" (in the sense of resisting *generic* attacks, as explained above).

From the perspective of this work, where we are trying to validate the design principle behind existing block ciphers, the opposite direction (of building an ideal cipher from a random oracle) is much more relevant. Quite interestingly, it happened to be significantly more challenging than building a PRP out of a PRF. Indeed, the most natural attempt is to use the already mentioned Feistel construction, that uses the given random oracles f to implement the required round functions.[6] However,

[6] The most natural modeling would give a *single* n-to-n-bit permutation from several $n/2$-to-$n/2$-bit random oracles. However, by prepending the *same* κ-bit key K to each such RO, one gets a candidate block cipher. We notice, though,

unlike the standard PRF-PRP case, where four rounds were already sufficient [45], in the indifferentiability setting even five rounds are provably insecure [15,16,25]. On a high-level, the key issue is that in the latter framework the distinguisher can have direct access to all the intermediate round functions, which was provably impossible in the more restricted indistinguishability framework. As a step towards overcoming this difficulty, Dodis and Puniya [25] considered a variant of the indifferentiability framework called "honest-but-curious" (HBC) indifferentiability, where the adversary can only query the global Feistel construction, and get all the intermediate results, but cannot directly query the round functions. In this model, which turns out to be *incomparable* to "standard" indifferentiability [15], they showed that the Feistel construction with a super-logarithmic number of rounds (with random oracle round functions) is HBC-indifferentiable from a fixed ideal permutation. The elegant work of Coron et al. [15] (and later Seurin [54]) conjectured and attempted a "standard" indifferentiability proof for the Feistel construction with six rounds. Unfortunately, while developing several important techniques, the proof contained some non-trivial flaws. Fortunately, this result was later fixed by Holenstein et al. [37], who succeeded in proving that a fourteen-round Feistel construction can be used to build an ideal cipher from a random oracle.

Key-Alternating Ciphers. Despite this great theoretical success showing the equivalence between the random oracle and the ideal cipher models, the above results of [15,37,54] only partially address our main motivation of theoretically studying the soundness of the design of *existing* block ciphers. In particular, we notice that (from a high level) there are two major design principles for block ciphers. The "old school" approach is indeed Feistel-based, with many prominent ciphers such as DES, Blowfish, Camellia, FEAL, Lucifer, and MARS. However, it appears that all such ciphers use *rather weak* (albeit non-trivial) round functions, and (in large part) get their security by using *many more* rounds than theoretically predicted. So, while the theoretical soundness of the Feistel network is important philosophically, it is unclear that random oracle modeling of the round functions is realistic.

In fact, we already mentioned a somewhat paradoxical fact: while, in theory, the random oracle model appears much more basic and minimal than the highly structured ideal cipher model (much like a one-way function is more basic than a one-way permutation), in practice, the implication appears to be *totally reversed*. In particular, in practice it appears much more accurate to say that hash functions (or "random oracles") are built from block ciphers (or "ideal ciphers") than the other way around. Indeed, in addition to the widely used SHA-1/2 and MD5 examples, other prominent block-cipher-based hash functions are recent SHA-3 finalists BLAKE [2] and Skein [30].

Perhaps most importantly for us, the current block cipher standard AES, as well as a few other "new school" ciphers (e.g., 3-Way, SHARK, Serpent, Present, and Square), are *not Feistel-based*. Instead, such ciphers are called *key-alternating ciphers*, and their design goes back to Daemen [18,19,20]. In general, a key-alternating cipher KA_t consists of a small number t of fixed permutations P_i on n bits, separated by key addition:

$$\mathrm{KA}_t(K, m) = k_t \oplus P_t(\ldots k_2 \oplus P_2(k_1 \oplus P_1(k_0 \oplus m))\ldots),$$

where the round keys $k_0, \ldots, k_t$ are derived from the master key K using some *key derivation* (aka "key schedule") function. For one round $t = 1$, the construction collapses to the well-known Even-Mansour (EM) [29] cipher. Interestingly, already in the standard "PRP indistinguishability" model, the analysis of the EM [29] (and more general key-alternating ciphers [12]) seems to require the modeling of P as a *random permutation* (but, on the other hand, does not require another computational assumption such as a PRF). With this idealized modeling, one can show that the Even-Mansour

that unlike the secret-key setting, it is (clearly) *not* secure to prepend several *independent* keys to each round function. We will come back to this important point when discussing the importance of key derivation in the indifferentiability proofs.

cipher is indistinguishable [29], and, in fact, its exact indistinguishablity security increases beyond the "birthday bound" as the number of round increases to 2 and above [12,28].

Our Main Question. Motivated by the above discussion, we ask the main question of our work:

> *Under which conditions on the key derivation function and for how many rounds t is the key-alternating cipher KA_t indifferentiable from the ideal cipher, assuming $P_1, \ldots, P_t$ are random permutations?*

As we mentioned, one motivation for this question comes from the actual design of the AES cipher, whose design principles we are trying to analyze. The second motivation comes from the importance of having the composition theorem guaranteed by the indifferentiability framework. Indeed, we already saw a natural example where using the Even-Mansour cipher to instantiate the classical Davies-Meyer compression function gave a totally insecure construction, despite the fact that the Davies-Meyer construction was known to be collision-resistant in the ideal cipher model [55], and the EM cipher indistinguishable in the random permutation model [29]. The reason for that is the fact that the EM cipher is easily seen to be not indifferentiable from an ideal cipher. In contrast, if we were to use a variant of the key alternating cipher which *is* provably indifferentiable, we would be *guaranteed* that the composed Davies-Meyer function remains collision-resistant (now, in the random permutation model).

The third motivation comes from the fact that the direct relationship between the random permutation (RP) model and the ideal cipher model is interesting in its own right. Although we know that these primitives are equivalent through the chain "IC $\Rightarrow$ RP (trivial) $\Rightarrow$ RO [24,26] $\Rightarrow$ IC [15,37]", a direct "RP $\Rightarrow$ IC" implication seems worthy of study in its own right (and was mentioned as an open problem in [17]).[7] More generally, we believe that the random permutation model (RPM) actually deserves its own place alongside the ROM and the ICM. The reason is that both the block cipher standard AES and the new SHA-3 standard Keccak [4] (as well as several other prominent SHA-3 finalists Grøstl [31] and JH [56]) are most cleanly described using a (constant number of) *permutation(s)*. The practical reason appears to be that it seems easier to ensure that the permutation design does not lose any entropy (unlike an ad-hoc hash function), or would not have some non-trivial relationship among different keys (unlike an ad-hoc block cipher). Thus, we find the indifferentiability analyses in the RPM very relevant both in theory and in practice. Not surprisingly, there has been an increased number of works as of late analyzing various constructions in the RPM [24,52,53,5,26,44,12].

Our Main Result. As our main result, we show the following theorem.

Theorem 1. *The 5-round key-alternative cipher KA_5 is indifferentiable from an ideal cipher, assuming $P_1, \ldots, P_5$ are five independent random permutations, and the key derivation function sets all rounds keys $k_i = f(K)$, where $0 \leq i \leq 5$ and f is modeled as a κ-to-n-bits random oracle.*

A more detailed statement appears in Theorem 3. In particular, our indifferentiability simulator has provable security $O(q^{10}/2^n)$, running time $O(q^3)$, and query complexity $O(q^2)$ to answer q queries made by the distinguisher. Although (most likely) far from optimal, our bounds are (unsurprisingly) much better than the $O(q^{16}/2^{n/2})$ and $O(q^8)$ provable bounds achieved by following the indirect "random-oracle route" [37].

We also show a simple attack illustrating that a one- or even two-round KA_t construction is never indifferentiable from the ideal cipher . This should be contrasted with the simpler indistinguishability setting, where the 1-round Even-Mansour construction is already secure [29]. Indeed, as was the case with Merkle-Damgård based hash function design and the "extension attack", the Davies-Meyer

[7] Indeed, our efficiency and security below are much better than following the indirect route through random oracle.

composition fiasco of the 1-round EM cipher demonstrated that this lack of indifferentiability indeed leads to a serious real-world attack on this cipher.

Finally, we give some justification of why we used 5 rounds, by attacking several "natural" simulators for the 4-round construction.

Importance of Key Derivation. Recall, in the secret-key indistinguishability case, the key derivation function was only there for the sake of minimizing the key length, and having $t+1$ independent keys $k_0, \ldots, k_t$ resulted in the best security analysis. Here, the key K is *public* and controlled by the attacker. In particular, it is trivial to see that having $t+1$ independent keys is like having a one-round construction (as then the attacker can simply fix all-but-one-keys k_i), which we know is trivially insecure. Thus, in the indifferentiability setting it is very important that the keys are somehow correlated (e.g., equal).

Another important property for the key derivation functions, at least if one wants to optimize the number of rounds, appears to be its *invertibility*. Very informally, this means that the only way to compute a valid round k_i is to "honestly compute" a key derivation function f on some key K first. In particular, in our analysis we use a random oracle as such a non-invertible key derivation function. We give some evidence of the importance of invertibility for understanding the indifferentiability-security of key-alternating ciphers by (1) critically using such non-invertibility in our analysis; and (2) showing several somewhat surprising attacks for the 3-round construction with certain natural "invertible" key schedules (e.g., all keys k_i equal to K for $\kappa = n$). We stress that our results do not preclude the use of invertible key schedules for a sufficiently large number of rounds (say, 10-12), but only indicate why having non-invertible key schedules is very helpful in specific analyses (such as ours) and also for avoiding specific attacks (such as our 3-round attacks). Indeed, subsequent to our work, Lampe and Seurin [43] showed that the 12-round key alternating cipher will all keys $k_i = K$ (for $\kappa = n$) is indeed indifferentiable from an ideal cipher, with security $O(q^{12}/2^n)$ and simulator query complexity $O(q^4)$ to answer q queries made by the distinguisher. Although using substantially more rounds and achieving noticeably looser exact security than this work, their result is closer to the actual design of the AES cipher, whose key schedule f is indeed easily invertible.

Instantiating the Key Derivation Function. Although we use a random oracle as a key derivation function (see above), in principle one can easily (and efficiently!) build the required random oracle from a random permutation [24,26], making the whole construction entirely permutation-based. For example, the most optimized "enhanced-CBC" construction from [24] will use only a single additional random permutation and make $\frac{2\kappa}{n} + O(1)$ calls to this permutation to build a κ-to-n-bit random oracle f.[8] Unsurprisingly, this instantiation will result in a cipher making a lot fewer calls to the random permutation (by a large constant factor) than following the indirect RP-to-RO-to-IC cycle.

Moreover, we can further optimize the most common case $\kappa = n$ as follows. First, [24] showed that $f(K) = P(K) \oplus P^{-1}(K)$ is $O(q^2/2^n)$-indifferentiable from an n-to-n-bit random oracle, which already results in a very efficient block cipher construction with 7 permutation calls. Second, by closely examining our proof, we observe that we do not need the full power of the random oracle f for key derivation. Instead, our proof only uses the "preimage awareness" [27] of the random oracle[9] and the fact that random oracle avoids certain simple combinatorial relations among different derived keys. In particular, we observe that the "unkeyed Davies-Meyer" function [24] $f(K) = P(K) \oplus K$ is enough for our analysis to go through. This gives the following result for building an n-bit ideal cipher with n-bit key, using only six random permutation calls.

[8] The indifferentiability security of this construction to handle q queries is "only" $O(q^4/2^n)$, but this is still much smaller than the bound in Theorem 3, and will not affect the final asymptotic security.

[9] Informally, at any point of time the simulator knows the list of all input-output pairs to f "known" by the distinguisher.

Theorem 2. *The following n-bit cipher with n-bit key is indifferentiable from an ideal cipher:*

$$E(K, m) = k \oplus P_5(k \oplus P_4(k \oplus P_3(k \oplus P_2(k \oplus P_1(k \oplus m))))),$$

where $k = P_0(K) \oplus K$ and $P_0, P_1, P_2, P_3, P_4, P_5$ are random permutations.

Overall, our results give the first theoretical evidence for the design soundness of key-alternative ciphers — including AES, 3-Way, SHARK, Serpent, Present, and Square — from the perspective of indifferentiability.[10]

Paper Organization. We establish some notations and define key-alternating ciphers in Section 2, where we also recall the definition of indifferentiability. In Section 3 we show differentiability attacks on KA_t constructions for $t = 1, 2, 3$. For $t = 1, 2$ the attacks apply to any key schedule, while our attacks for $t = 3$ presume specific key scheduling properties (in particular, invertibility). We present our main result on the indifferentiability of KA_5 with an RO key schedule in Section 4. Section 4.1 describes our simulator at a high level. The simulator's pseudocode itself, along with pseudocode for other security games, is found in Appendix B. Our choice of 5 rounds is also explained in Section 4.1, with some supporting attacks on various "natural" 3- or 4-round simulator candidates being catalogued in Appendix A. Section 4.2 highlights the main techniques in our indifferentiability proof; there, emphasis is placed on "what's novel" instead of on actually explaining the proof. A detailed overview of the proof appears in Section 4.3 with supporting material in Appendices C and D. Finally, we present the extension of our main result to key scheduling with an unkeyed Davies-Meyer function $f(K) = P_0(K) \oplus K$ (Theorem 2) in Appendix E.

2 Preliminaries

For a domain $\{0,1\}^m$ and a range $\{0,1\}^n$, a random oracle $\mathcal{R} : \{0,1\}^m \to \{0,1\}^n$ is a function drawn uniformly at random from the set of all possible functions that map m to n bits. For two sets $\{0,1\}^\kappa$ and $\{0,1\}^n$, an ideal cipher $\mathcal{IC} : \{0,1\}^\kappa \times \{0,1\}^n \to \{0,1\}^n$ is taken randomly from the set of all block ciphers with key space $\{0,1\}^\kappa$ and message and ciphertext space $\{0,1\}^n$. A random permutation $\pi : \{0,1\}^n \to \{0,1\}^n$ is a function drawn randomly from the set of all n-bit permutations.

Key-Alternating Ciphers. A key-alternating cipher KA_t consists of a small number t of fixed permutations P_i on n bits separated by key addition:

$$\mathrm{KA}_t(K, m) = k_t \oplus P_t(\ldots k_2 \oplus P_2(k_1 \oplus P_1(k_0 \oplus m))\ldots),$$

where the round keys $k_0, \ldots, k_t$ are derived from the master key K using some key schedule f: $(k_0, \ldots, k_t) = f(K)$. The notion of key-alternating ciphers itself goes back to Daemen [18,19,20] and was used in the design of AES. However, it was Knudsen [41] who proposed to instantiate multiple-round key-alternating ciphers with randomly drawn, fixed and public permutations (previously, a single-round key-alternating construction was proposed by Even-Mansour [29]).

Indifferentiability. We use the notion of indifferentiability [46,16] in our proofs to show that if a construction $\mathcal{C}^\mathcal{P}$ based on an ideal subcomponent $\mathcal{P}$ is indifferentiable from an ideal primitive $\mathcal{R}$, then $\mathcal{C}^\mathcal{P}$ can replace $\mathcal{R}$ in any system. As noticed in [51] the latter statement must be qualified with some fine print: since the adversary must eventually incorporate the simulator, the indifferentiability composition theorem only applies in settings where the adversary comes from a computational class that is able to "swallow" the simulator (e.g., the class of polynomial-time, polynomial-space algorithms); see [51,22] for more details on the limitations of indifferentiability.

[10] We also mention a complementary recent work of [48], who mainly looked at "weaker-than-indistinguishability" properties which can be proven about AES design.

Definition 1. *A Turing machine $\mathcal{C}$ with oracle access to an ideal primitive $\mathcal{P}$ is called $(t_D, t_S, q, \varepsilon)$-indifferentiable from an ideal primitive $\mathcal{R}$ if there exists a simulator $\mathcal{S}$ with oracle access to $\mathcal{R}$ and running in time t_S, such that for any distinguisher D running in time at most t_D and making at most q queries, it holds that:*

$$\mathrm{Adv}^{\mathrm{indif}}_{\mathcal{C},\mathcal{R},\mathcal{S}}(D) = \left|\Pr\left[D^{\mathcal{C}^{\mathcal{P}},\mathcal{P}} = 1\right] - \Pr\left[D^{\mathcal{R},\mathcal{S}^{\mathcal{R}}} = 1\right]\right| < \varepsilon.$$

Distinguisher D can query both its *left oracle* (either $\mathcal{C}$ or $\mathcal{R}$) and its *right oracle* (either $\mathcal{P}$ or $\mathcal{S}$). We refer to $\mathcal{C}^{\mathcal{P}}, \mathcal{P}$ as the *real world*, and to $\mathcal{R}, \mathcal{S}^{\mathcal{R}}$ as the *simulated world*.

3 Attacks on KA_t for $t \leq 3$

Let $P_1, P_2, \ldots, P_t$ be t randomly chosen permutations of KA_t, $f(K) = (k_0, \ldots, k_t)$ be the key derivation function and $\mathcal{IC}$ be an ideal cipher. Notice, depending on a particular attack below, f may or may not be idealized.

Let $\mathcal{S}$ be any simulator making at most $q_{\mathcal{S}}$ queries to its oracle $\mathcal{IC}$. In each of our attacks, we build a distinguisher D which will "fool" any such simulator $\mathcal{S}$ with non-negligible probability. Notice, D has access to $O = (\mathcal{C}; O_1, \ldots, O_t, O')$ where $O \in \left\{(\mathrm{KA}_t; P_1, \ldots, P_t, f), (\mathcal{IC}; \mathcal{S}_1, \ldots, \mathcal{S}_t, \mathcal{S}')\right\}$ and $O'/\mathcal{S}'$ is the oracle/simulator used to instantiate the key derivation function f (e.g., a fresh random oracle, or ideal permutation), if present. Of course, when f is non-idealized (e.g., identity function when $n = \kappa$), $O'/\mathcal{S}'$ are simply ignored.

We start with simple attacks for $t = 1, 2$ which work for any (even idealized) key derivation function f. We will then describe more specialized attacks for $t = 3$ which will place some restrictions on the type of key derivation function f.

3.1 Attack on KA_1 With Any Key Schedule

Notice, $\mathrm{KA}_1(K, x) = P_1(x \oplus k_0) \oplus k_1$, where $f(K) = (k_0, k_1)$. We claim that for any x, K and K', we have

$$\mathrm{KA}_1(K, x \oplus k_0) \oplus k_1 = \mathrm{KA}_1(K', x \oplus k_0') \oplus k_1'.$$

Now, we define a distinguisher D as follows. First, for two distinct keys K and K' it asks $O'(K) \to (k_0, k_1)$ and $O'(K') \to (k_0', k_1')$. Then, it selects a random $x \in \{0,1\}^n$ and computes $u = \mathcal{C}(K, x \oplus k_0) \oplus k_1$ and $v = \mathcal{C}(K', x \oplus k_0') \oplus k_1'$, and accepts if and only if the results are the same. In the "real world $\mathcal{C} = \mathrm{KA}_1$" we indeed have $u = v$ with probability 1. On the other hand, we claim that in the "ideal world $\mathcal{C} = \mathcal{IC}$", D accepts only with small probability. Denote by E the event that the simulator queried $\mathcal{IC}$ on input of $(K, x \oplus k_0)$. Then, $\Pr[u = v] \leq \Pr[u = v \mid \neg E] + \Pr[E]$, which is bounded by $1/(2^n - q_{\mathcal{S}} + 1) + q_{\mathcal{S}}/(2^n - q_{\mathcal{S}}) \leq (q_{\mathcal{S}} + 1)/(2^n - (q_{\mathcal{S}} + 1))$. Thus, D has an advantage of at least $1 - (q_{\mathcal{S}} + 1)/(2^n - (q_{\mathcal{S}} + 1))$.

3.2 Attack on KA_2 With Any Key Schedule

Notice, $\mathrm{KA}_2(K, x) = P_2(P_1(x \oplus k_0) \oplus k_1) \oplus k_2$, where $f(K) = (k_0, k_1, k_2)$. We claim that for any x, K and K', we have

$$\mathrm{KA}_2^{-1}(K', \mathrm{KA}_2(K, x \oplus k_0) \oplus k_2 \oplus k_2') \oplus k_0' = \mathrm{KA}_2^{-1}(K, \mathrm{KA}_2(K', x \oplus k_0') \oplus k_2' \oplus k_2) \oplus k_0.$$

To show this, we rewrite the right part as

$$P_1^{-1}(P_2^{-1}(P_2(P_1(x) \oplus k_1)) \oplus k_1') = P_1^{-1}(P_1(x) \oplus k_1 \oplus k_1')$$

and observe that the result is symmetric in k_1 and k_1' and, thus, in K and K', as claimed.

Now, we define a distinguisher D as follows. First, for two distinct keys K and K' it asks $O'(K) \to (k_0, k_1, k_2)$ and $O'(K') \to (k_0', k_1', k_2')$. Then, it selects a random $x \in \{0,1\}^n$ and computes $u = \mathcal{C}^{-1}(K', \mathcal{C}(K, x \oplus k_0) \oplus k_2 \oplus k_2') \oplus k_0'$ and $v = \mathcal{C}^{-1}(K, \mathcal{C}(K', x \oplus k_0') \oplus k_2' \oplus k_2) \oplus k_0$, and accepts if and only if the results are the same. In the "real world $\mathcal{C} = \mathrm{KA}_2$" we indeed have $u = v$ with probability 1. On the other hand, we claim that in the "ideal world $\mathcal{C} = \mathcal{IC}$", D accepts only with small probability.

Indeed, if we let $\pi(x) = \mathcal{IC}^{-1}(K', \mathcal{IC}(K, x \oplus k_0) \oplus k_2 \oplus k_2') \oplus k_0'$, then $\pi^{-1}(x) = \mathcal{IC}^{-1}(K, \mathcal{IC}(K', x \oplus k_0') \oplus k_2' \oplus k_2) \oplus k_0$, and $u = v$ if and only if $\pi(x) = \pi^{-1}(x)$. Denote by E_1 the event that the simulator queried $\mathcal{IC}$ on input of $(K, x \oplus k_0)$, and let E_2, E_3, E_4 be the events that $\pi(x) = x$, $\pi(y) = x$ and $\pi(x) = y$, respectively. Then,

$$\begin{aligned}
\Pr_\pi[\pi(x) = \pi^{-1}(x)] &\le \Pr_\pi[\pi(x) = \pi^{-1}(x) \mid \neg E_1] + \Pr[E_1] \\
&\le \Pr[E_2] + \Pr[E_3 \mid (E_4 \wedge y \neq x)] + \Pr[E_1] \\
&\le \frac{1}{2^n - q_{\mathcal{S}}} + \frac{1}{2^n - (q_{\mathcal{S}} + 1)} + \frac{q_{\mathcal{S}}}{2^n - q_{\mathcal{S}}} \le \frac{q_{\mathcal{S}} + 2}{2^n - (q_{\mathcal{S}} + 1)}.
\end{aligned}$$

Thus, D has an advantage of at least $1 - (q_{\mathcal{S}} + 2)/(2^n - (q_{\mathcal{S}} + 1))$.

3.3 Attacks on KA_3

To describe our specialized attacks on KA_3, we first define the property of the key derivation function f (which may or may not be idealized). We say that $f(K) = (k_0, k_1, k_2, k_3)$ is *invertible* if there exists an efficient procedure f_1^{-1} which, for any key k_1, outputs a key K such that $f(K) = (k_1, \cdot)$. Further, when $k_0 = k_1 = k_2 = k_3$, we say that f is *identical* everywhere.

Attack on Any Invertible f. This attack on KA_3 applies to arbitrary invertible key derivation functions that are not idealized, i.e., that can be freely computed by the distinguisher.

1. D queries O_1: $x_1 \to y_1$, $x_1' \to y_1'$ and $x_1'' \to y_1''$.
2. D selects a key $K \xleftarrow{\$} \{0,1\}^n$ at random and derives $(k_0, k_1, k_2, k_3) = f(K)$. Then, it computes a three-way collision $k_1 \oplus y_1 = k_1' \oplus y_1' = k_1'' \oplus y_1'' =: x_2$: by the strong invertibility of f, D computes K' and K'' from k_1' and k_1'', respectively, together with (k_0', k_2', k_3') and (k_0'', k_2'', k_3'').
3. D computes using forward queries to $\mathcal{C}$: $y_3 = \mathcal{C}(K, x_1 \oplus k_0) \oplus k_3$, $y_3' = \mathcal{C}(K', x_1' \oplus k_0') \oplus k_3'$ and $y_3'' = \mathcal{C}(K'', x_1'' \oplus k_0'') \oplus k_3''$.
4. D queries O_3^{-1}: $y_3 \to x_3$.
5. D chooses $b \xleftarrow{\$} \{0,1\}$. If $b = 0$, then D queries O_3^{-1} on $y_3' \to x_3'$, else queries O_3^{-1} on $y_3'' \to x_3''$.
6. D forward queries O_2: $x_2 \to y_2$.
7. If $y_2 = x_3 \oplus k_2 = x_3' \oplus k_2'$ when $b = 0$ and $y_2 = x_3 \oplus k_2 = x_3'' \oplus k_2''$ otherwise, D guesses the "real world" and otherwise the "simulated".

Now a successful simulator $\mathcal{S}$ should satisfy the equation $x_3' = (x_3 \oplus k_2' \oplus k_2)$ if $b = 0$ and $x_3'' = (x_3 \oplus k_2'' \oplus k_2)$ if $b = 1$. Notice that when the query $x_2 \to y_2$ is made by D, $\mathcal{S}$ can actually learn the corresponding key K by simply computing $y_1 \oplus x_2$. If then $\mathrm{KA}_3(K, x_1) = y_3$, $\mathcal{S}$ will output $(x_3 \oplus k_2) = y_2$. But a successful simulator should also be prepared to satisfy either $x_3' = (k_2' \oplus k_2 \oplus x_3)$ or $x_3'' = (k_2'' \oplus k_2 \oplus x_3)$ before the final query to P_2 is made (hence, only knowing the query tuples (x_1, y_1), (x_1', y_1'), (x_1'', y_1'') and (x_3, y_3)).

We demonstrate that D's advantage in at least 9 queries is lower bounded by $1/2 - q_{\mathcal{S}}/(2^n - q_{\mathcal{S}})$, namely the probability of $\mathcal{S}$ to succeed depends on whether of not $\mathcal{S}$ guesses correctly either the value of b or the key K' or K''. To compute the advantage of D, we argue that D succeeds except when its guess in the last step of the attack is wrong, namely $\mathcal{S}$ (after O_3^{-1} queries) has already prepared y_2, such as to satisfy $(x_3 \oplus k_2) = (x_3' \oplus k_2')$ and $(x_3 \oplus k_2) = (x_3'' \oplus k_2'')$. By construction, this condition is always satisfied in the "real world", and D only fails if this equation gets satisfied in the "simulated world".

Let "$\mathcal{S}$ succeeds" be the event that the outputs of $\mathcal{S}$ are compliant with the condition in step 7. Let $(\overline{x}_3, \overline{y}_3)$ be the query tuple queried in step 5: (x_3', y_3') or (x_3'', y_3''). Denote by E the event that $\mathcal{S}$, before the query $(\overline{x}_3, \overline{y}_3)$, queries $\mathcal{IC}$ on input of (K', x_1) or (K'', x_1''), namely identifies the correct key K' or K''. We then have

$$\mathrm{Adv}^{\mathrm{indif}}_{\mathrm{KA}_3,\mathcal{IC},\mathcal{S}}(D) \geq 1 - \Pr[\mathcal{S} \text{ succeeds}] \tag{1}$$

where the probability of "$\mathcal{S}$ succeeds" is lower bound by:

$$\Pr\left[\overline{x}_3 = x_3 \oplus k_2 \oplus \begin{cases} k_2', \text{ if b=0} \\ k_2'', \text{ if b=1} \end{cases} \middle| \neg E \right] - \Pr[E].$$

To compute the probability on E note that by that point the simulator only knows (x_1, y_1), (x_1', y_1') and (x_1'', y_1'') and the values $(k_1 \oplus k_1')$ and $(k_1 \oplus k_1'')$. To output correct (K', x_1') or (K'', x_1'') inputs to $\mathcal{IC}$, the simulator best strategy is to guess the correct key K' or K''. In $q_{\mathcal{S}}$ queries to $\mathcal{IC}$, this probability is at most $2q_{\mathcal{S}}/(2^n - q_{\mathcal{S}})$. Now, consider the first probability. Recall that the simulator may know the values $(k_1 \oplus k_1')$ and $(k_1 \oplus k_1'')$ by simply xor-ing y_1 values and deriving the corresponding differences. Assume w.l.o.g. that it also knows the values $(k_2 \oplus k_2')$ and $(k_2 \oplus k_2'')$ (for linear key deriving functions this could be the case). Now, regarding the inverse query $\overline{y}_3$, as $\neg E$ holds, from the simulator's point of view, this value could equally likely correspond to (x_1', y_1') or (x_1'', y_1''), and the best thing it could do to make $\overline{x}_3$ satisfied, is to simply guess b. Hence, this probability is bounded by 1/2. Thus, D has differentiating advantage in 9 queries of at least $1/2 - 2q_{\mathcal{S}}/(2^n - q_{\mathcal{S}})$.

Attack on Any Identical Invertible f. As opposed to our previous attack, this attack on KA_3 applies only to identical invertible key derivation functions (hence, where all round keys are the same), but can handle f's which can be idealized. For ease of presentation, we deviate from the previous notation, and simply write $k_i = O'(K_i)$ (for $i = 1, 2, 3, 4$), where then all round keys are k_i. The attack exhibits the following structural property of the KA_3 construction. For all input values x and distinct keys K_1 and K_2, there exist distinct K_3, K_4 with $K_3 \neq K_2$ and $K_4 \neq K_1$, such that the following holds:

$$\mathrm{KA}_3^{-1}(K_4, \mathrm{KA}_3(K_1, x \oplus k_1) \oplus k_1 \oplus k_4) \oplus k_4 = \mathrm{KA}_3^{-1}(K_3, \mathrm{KA}_3(K_2, x \oplus k_2) \oplus k_2 \oplus k_3) \oplus k_3. \tag{2}$$

This property (including definitions of K_3 and K_4 from K_1, K_2 and any plaintext $x = x_1$) is implicitly shown in the attack below, and is hard to satisfy for an ideal cipher. Formally:

1. D queries O_1 on some arbitrary x_1: $x_1 \to y_1$.
2. For two distinct, arbitrarily chosen keys K_1 and K_2, D queries $O'(K_1) \to k_1$ and $O'(K_2) \to k_2$.
3. D queries O_2: $x_2 = (y_1 \oplus k_1) \to y_2$ and $x_2' = (y_1 \oplus k_2) \to y_2'$; O_3: $x_3 = (y_2 \oplus k_1) \to y_3$ and $x_3' = y_2' \oplus k_2 \to y_3'$ (notice that D's objective is to compute two distinct values y_3 and y_3', namely two diverging paths connected only under the O_1 evaluation).
4. D sets $k_3 = y_2 \oplus x_3'$ and $k_4 = y_2' \oplus x_3$, and queries $O'^{-1}(k_3) \to K_3$ and $O'^{-1}(k_4) \to K_4$.
5. D computes using inverse queries to $\mathcal{C}^{-1}$: $x_1' = \mathcal{C}^{-1}(K_3, y_3' \oplus k_3) \oplus k_3$ and $x_1'' = \mathcal{C}^{-1}(K_4, y_3 \oplus k_4) \oplus k_4$.
6. If $x_1' = x_1''$, then D guesses the real world and otherwise the simulated.

Firstly, to show that $x_1' = x_1''$ in the "real world", note that by construction (steps 1-3) we have $(x_2 \oplus x_2' \oplus y_2 \oplus y_2' \oplus x_3 \oplus x_3') = 0$ since $(x_2 \oplus x_2') = (k_1 \oplus k_2)$ and $(y_2 \oplus x_3) \oplus (y_2' \oplus x_3') = (k_1 \oplus k_2)$.

Now, in the "real world", step 5 results in $x_1' = P_1^{-1}(x_2 \oplus (y_2 \oplus x_3'))$ and $x_1'' = P_1^{-1}(x_2' \oplus (y_2' \oplus x_3))$, which gives us $x_1' = x_1''$ (and hence shows the validity of (2)).

To bound the distinguishing advantage of D we use (1) where by "$\mathcal{S}$ succeeds" we denote the event that a successful simulator $\mathcal{S}$ finds, for given values x, K_1, K_2, a set of values $k_1, k_2, y_1, y_2, y_2', y_3, y_3', K_3, K_4$ such that (2) holds, where k_3, k_4 are as defined. For the specific adversarial choices of k_3, k_4 and x_3, x_3', this bound simplifies to:

$$\mathcal{IC}^{-1}(K_4, \mathcal{IC}(K_1, x \oplus k_1) \oplus y) \oplus y \oplus k_1 = \mathcal{IC}^{-1}(K_3, \mathcal{IC}(K_2, x \oplus k_2) \oplus y) \oplus y \oplus k_2,$$

where we write $y = y_2 \oplus y_2'$ and consider a simulator which tries to find, given x, K_1, K_2, values k_1, k_2, y, K_3, K_4 such that this equation holds. We use the wish list methodology of [1]. We consider the probability of a simulator to make a winning query to $\mathcal{IC}$. Consider (without loss of generality) any query with key input K_2 and any query with key input K_3 fixed. By construction, this also fixes the particular values k_2, y, K_3. The goal of the simulator is to make two queries matching the left hand side of the equation. If the last query is the inner one, consider any possible query for the outer $\mathcal{IC}$ evaluation: this one fixes the values k_1 and $\mathcal{IC}(K_1, x \oplus k_1)$, and hence the entire input and output of the inner $\mathcal{IC}$ evaluation. Therefore, the simulator succeeds in this case with probability at most $q_{\mathcal{S}}^3/(2^n - q_{\mathcal{S}})$. On the other hand, if the last query is the outer one, consider any possible query for the inner $\mathcal{IC}$ evaluation: this one fixes the values k_1 and $\mathcal{IC}(K_1, x \oplus k_1)$, and the simulator can make $q_{\mathcal{S}}$ queries (for various values of K_4) to obtain a proper outer $\mathcal{IC}$ evaluation. Therefore, the simulator succeeds with probability at most $q_{\mathcal{S}}^4/(2^n - q_{\mathcal{S}})$. By symmetry, we obtain the simulator succeeds with total success probability at most $(2q_{\mathcal{S}}^4 + 2q_{\mathcal{S}}^3)/(2^n - q_{\mathcal{S}})$. Thus, D has differentiating advantage in 7 queries of at least $1 - (2q_{\mathcal{S}}^4 + 2q_{\mathcal{S}}^3)/(2^n - q_{\mathcal{S}})$.

4 Indifferentiability of KA_5

In this section we discuss our main result, namely that KA_5 with an RO key schedule is indifferentiable from an ideal cipher. In the statement below, KA_5 stands for a 5-round key-alternating cipher implemented with round functions $P_1, \ldots, P_5$ and key scheduling function f, with the round functions, their inverses, and the key scheduling function all being available for oracle queries by the adversary (and thus, also, all being implemented as interfaces by the simulator).

Theorem 3. *Let $P_1, \ldots, P_5$ be independent random n-bit permutations, and f be a random κ-to-n-bits function. Let D be an arbitrary information-theoretic distinguisher that makes at most q queries. Then there exists a simulator $\mathcal{S}$ such that*

$$\mathrm{Adv}^{\mathrm{indif}}_{KA_5, \mathcal{IC}, \mathcal{S}}(D) \leq 320 \cdot 6^{10} \left(\frac{q^{10}}{2^n} + \frac{q^4}{2^n} \right) = O\left(\frac{q^{10}}{2^n} \right),$$

where $\mathcal{S}$ makes at most $2q^2$ queries to the ideal cipher $\mathcal{IC}$ and runs in time $O(q^3)$.

4.1 Simulator Overview

Our 5-round simulator $\mathcal{S}$ is given by the pseudocode in game G_1 (see Figures 5–8), and more precisely by the public functions f, P1, P1^{-1}, P2, P2^{-1}, ..., P5, P5^{-1} within G_1. Here f emulates the key scheduling random oracle, whereas P1, P1^{-1} emulate the random permutation P_1 and its inverse P_1^{-1}, and so on. Since the pseudocode of game G_1 is not easy to assimilate, a high-level description of our simulator is likely welcome. Furthermore, because the simulator is rather complex, we also try to argue the necessity of its complex behavior by discussing why some simpler classes of simulators might not work.

To describe the simulator-distinguisher interaction we use expressions such as "D makes the query $\mathrm{f}(K) \to k$" to mean that the distinguisher D queries f (which is implemented by the simulator) on input K, and receives answer k as a result. The set of values k for which the adversary has made a query of the form $\mathrm{f}(K) \to k$ for some $K \in \{0,1\}^\kappa$ is denoted $\mathcal{Z}$ (thus $\mathcal{Z}$ is a time-dependent set). If $\mathrm{f}(K) \to k$ then we also write K as "$f^{-1}(k)$"; here f and f^{-1} are internal tables maintained by the simulator to keep track of scheduled keys and their preimages (see procedure $\mathrm{f}(K)$ in Figure 5 for more details).

A triple (i, x, y) such that D has made the query $\mathrm{P}i(x) \to y$ or $\mathrm{P}i^{-1}(y) \to x$ is called an *i-query*, $i \in \{1, 2, 3, 4, 5\}$. Moreover, when the simulator "internally defines" a query $\mathrm{P}i(x) = y$, $\mathrm{P}i^{-1}(y) = x$ we also call the associated triple (i, x, y) an i-query, even though the adversary might not be aware of these values yet. (While this might seem a little informal, we emphasize that this section is, indeed, meant mainly as an informal overview.) A pair of queries (i, x_i, y_i), $(i+1, x_{i+1}, y_{i+1})$ such that $y_i \oplus k = x_{i+1}$ for some $k \in \mathcal{Z}$ is called *k-adjacent*. We also say that a pair of queries $(1, x_1, y_1)$, $(5, x_5, y_5)$ is *k-adjacent* if $k \in \mathcal{Z}$ and $\mathrm{E}(f^{-1}(k), x_1 \oplus k) = y_5 \oplus k$, where $\mathrm{E}(K, x)$ is the ideal cipher (and $\mathrm{E}^{-1}(K, y)$ its inverse). (Since $\mathcal{Z}$ is time-dependent, a previously non-adjacent pair of queries might become adjacent later on; of course, this is unlikely.) A sequence of queries

$$(1, x_1, y_1), (2, x_1, y_2), \ldots, (5, x_5, y_5)$$

for which there exists a $k \in \mathcal{Z}$ such that each adjacent pair is k-adjacent and such that the first and last queries are also k-adjacent is called a *completed k-path* or *completed k-chain.*

Consider first the simplest attack that a distinguisher D might carry out: D chooses a random $x \in \{0,1\}^n$ and a random $K \in \{0,1\}^\kappa$ (where $\{0,1\}^\kappa$ is the key space), queries $\mathrm{E}(K, x) \to y$ (to its left oracle), then queries $\mathrm{f}(K) \to k$, $\mathrm{P1}(x \oplus k) \to y_1$, $\mathrm{P2}(y_1 \oplus k) \to y_2$, $\mathrm{P3}(y_2 \oplus k) \to y_3$, ..., $\mathrm{P5}(y_4 \oplus k) \to y_5$ to the simulator, and finally checks that $y_5 \oplus k = y$. The simulator, having itself answered the query $\mathrm{f}(K)$, can already anticipate the distinguisher's attack when the query $\mathrm{P2}(y_1 \oplus k)$ is made, since it sees that a k-adjacency is about to be formed between a 1-query and a 2-query. At this point, a standard strategy would be for the simulator to pre-emptively[11] complete a k-chain by answering (say) the queries $\mathrm{P3}(y_2 \oplus k)$ and $\mathrm{P4}(y_3 \oplus k)$ randomly itself, and setting the value of $\mathrm{P5}(y_4 \oplus k)$ to $\mathrm{E}(f^{-1}(k), x) \oplus k$ by querying E.

The distinguisher might vary this attack by building a chain "from the right" (by choosing a random y and querying $\mathrm{P5}^{-1}(y \oplus k) \to x_5$, $\mathrm{P4}^{-1}(x_5 \oplus k) \to x_4$, etc) or by building a chain "from the inside" (e.g., by choosing a random x_3 and querying $\mathrm{P3}(x_3) \to y_3$, $\mathrm{P2}^{-1}(x_3 \oplus k)$, $\mathrm{P4}(y_3 \oplus k) \to y_4$, ...) or even by building a chain "from the left and right" simultaneously (the two sides meeting up somewhere in the middle). Given all these combinations, a natural strategy is to have the simulator complete chains whenever it detects *any* k-adjacency. We call this type of simulator *naïve.* The difficulty with the naïve simulator is that, as the path-completion strategy is applied recursively to queries created by the simulator itself, some uncontrollable chain reaction might occur that causes the simulator to create a superpolynomial number of queries, and, thus, lead to an unacceptable simulator running time and to an unacceptably watered-down security bound. Even if such a chain reaction cannot occur, the burden of showing so is on the prover's shoulders, which is not necessarily an easy task. We refer to the general problem of showing that runaway chain reactions do not occur as the problem of *simulator termination.*[12]

[11] Pre-emption is generally desirable in order for the simulator to avoid becoming "trapped" in an over-constrained situation.

[12] Naturally, since the simulator can only create finitely many different i-queries, the simulator is, in general, guaranteed to terminate. Thus "simulator termination" refers, more precisely, to the problem of showing that the simulator only creates polynomially many queries per adversarial query. We prefer the term "termination" to "efficiency" because it seems to more picturesquely capture the threat of an out-of-control chain reaction.

To overcome the naïve simulator's problematic termination, we modify the naïve simulator to be more restrained and to complete fewer chains. For this we use the "tripwire" concept. Informally, a tripwire is an ordered pair of the form $(i, i+1)$ or $(i+1, i)$ or $(1, 5)$ or $(5, 1)$ (for a 5-round cipher). "Installing a tripwire (i, j)" means the simulator will complete paths for k-adjacencies detected between positions i and j *and for which the j-query is made after the i-query.* (Thus, tripwires are "directed".) As long as no tripwires are triggered, the simulator does nothing; when a tripwire is triggered, the simulator completes the relevant chain(s), and recurses to complete chains for other potentially triggered tripwires, etc. The "naïve" simulator then corresponds to a tripwire simulator with all possible tripwires installed. The tripwire paradigm is essentially due to Coron et al. [15] even while the terminology is ours.

Restricting ourselves to the (fairly broad) class of tripwire simulators, conflicting goals emerge: to install enough tripwires so that the simulator cannot be attacked, while installing few enough tripwires (or in clever enough positions) that a termination argument can be made. Before presenting our own 5-round solution to this dilemma, we briefly justify our choice of five rounds.

Firstly, *no* tripwire simulator with 3 rounds is secure, since it turns out that the naïve 3-round simulator (i.e., with all possible tripwires) can already be attacked. Hence, regardless of termination issues, any 3-round tripwire simulator is insecure. Secondly, we focused on 4-round simulators with four tripwires, as proving termination for five or more tripwires seemed a daunting task. A particularly appealing simulator, here, is the 4-tripwire simulator

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

whose termination can easily be proved by modifying Holenstein et al. termination argument [37], itself adapted from an earlier termination argument of Seurin [54]. Unfortunately it turns out this simulator can be attacked, making it useless. This attack as well as the above-mentioned attack on the 3-round naïve simulator can be found in Appendix A, where some other attacks on tripwire simulators are also sketched.

Ultimately, the only 4-round, 4-tripwire simulator for which we didn't find an attack is the simulator with the (asymmetric) tripwire configuration

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

(and its symmetric counterpart). However, since we could not foresee a manageable termination argument for this simulator, we ultimately reverted to five rounds. Our 5-round simulator has tripwires

$$(2, 1), (2, 3), (4, 3), (4, 5)$$

(and no tripwires of the form $(1, 5)$ or $(5, 1)$), as sketched in Figure 1. This simulator has the advantage of having a clean (though combinatorially demanding) termination argument, and, as previously discussed, of having excellent efficiency and also better security than the state-of-the-art in "indifferentiable blockcipher" constructions.

Some more high-level description of the 5-round simulator. We have already mentioned that our 5-round simulator has tripwires

$$(2, 1), (2, 3), (4, 3), (4, 5).$$

To complete the simulator's description it (mainly) remains to describe how the simulator completes chains, once a tripwire is triggered, since there is some degree of freedom as to which i-query is "adapted" to fit E, etc. Quickly and informally, when a newly created 1-query or 3-query triggers

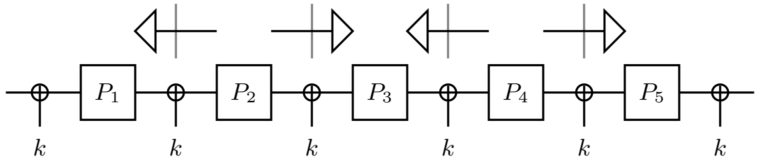

Fig. 1: Tripwire positions for our 5-round simulator. A directed arrow from column P_i to column P_j indicates a tripwire (i, j). The tripwires are $(2, 1)$, $(2, 3)$, $(4, 3)$ and $(4, 5)$.

respectively the $(2, 1)$ or $(2, 3)$ tripwire, the relevant path(s) that are completed have their 5-query adapted to fit E. (We note the same query may trigger the completion of several new paths.) Symmetrically, when a newly created 3-query or 5-query triggers a $(4, 3)$ or $(4, 5)$ tripwire, the completed paths have their 1-query adapted to fit E. We note that new 2-queries and 4-queries can never trigger a tripwire, due to the tripwire structure. Moreover, 2- and 4-queries are never adapted, and always have at least one "random endpoint". The latter property turns out to be crucial for various arguments in the proof. It also makes the implementation of the procedures P2(), $\mathrm{P2}^{-1}()$, P4() and $\mathrm{P4}^{-1}()$ particularly simple, since these do nothing else than lazy sample and return.

The above "quick and informal" summary of the path-completion process is over-simplified because 3-queries can also, in specific situations, be adapted to complete a path. To gain some preliminary intuition about 3-queries, consider a distinguisher D that chooses values x and K and then makes the queries $\mathrm{f}(K) \to k$, $\mathrm{P1}(x \oplus k) \to y_1$, $\mathrm{P2}(y_1 \oplus k) \to y_2$, $\mathrm{E}(K, x) \to y$, $\mathrm{P5}^{-1}(y \oplus k) \to x_5$ and $\mathrm{P4}^{-1}(x_5 \oplus k) \to x_4$. So far, no tripwires have been triggered, but the adversary already knows (e.g., in the real world) that $\mathrm{P3}(y_2 \oplus k) = x_4 \oplus k$, even while the simulator has not yet defined anything internally about P3. Typically, such a situation where the adversary "already knows" something the simulator doesn't are dangerous for the simulator and can lead to attacks; in this case, it turns out the distinguisher cannot use this private knowledge to fool the simulator. It does mean, however, that the simulator needs to be on the lookout for such "pre-defined" 3-queries whenever it answers queries to P3(), $\mathrm{P3}^{-1}()$ or, more generally, whenever it makes a new 3-query internally.

In fact the code used by the simulator to answer 3-queries is altogether rather cautious and sophisticated, even slightly more so than the previous discussion might suggest. To gain further insight into the simulator's handling of 3-queries, consider a distinguisher D' that similarly chooses values x and K and then makes the queries $\mathrm{f}(K) \to k$, $\mathrm{P1}(x \oplus k) \to y_1$, $\mathrm{P2}(y_1 \oplus k) \to y_2$, $\mathrm{E}(K, x) \to y$ and $\mathrm{P5}^{-1}(y \oplus k) \to x_5$. (So D' makes all the same queries as the distinguisher D above except for the final query $\mathrm{P4}^{-1}(x_5 \oplus k)$, which is *not* made by D'.) At this point, the value $\mathrm{P3}(y_2 \oplus k)$ is not yet pre-defined by E and by the previous queries, since the query $\mathrm{P4}^{-1}(x_5 \oplus k)$ hasn't been made; if D' queries $\mathrm{P3}(y_2 \oplus k) \to y_3$, the simulator might conceivably sample y_3 randomly, and later use the freedom afforded by the missing P4 query to adapt the chain. If the simulator did this, however, the simulator would create a "non-random" 4-query (i.e., a 4-query that doesn't have at least one non-adapted, "random endpoint"), which would wreak havoc within the proof. Instead, when faced with the query $\mathrm{P3}(y_2 \oplus k)$, the simulator detects the situation above and starts by making the "missing" query $\mathrm{P4}^{-1}(x_5 \oplus k) \to x_4$ internally, thus giving the P4-query its required "random endpoint" (at x_4), and finally adapts $\mathrm{P3}(y_2 \oplus k)$ to $x_4 \oplus k$. It so turns out that, with high probability, the simulator is never caught trying to adapt P3() to two different values in this way.

We summarize below how the simulator handles the main types of queries. (This summary is only meant as a high-level guide and should not be viewed as the simulator specification, which is game G_1 in Figures 5–8.) The simulator performs the same actions when it makes queries internally. The sets

LeftQueue and *RightQueue* mentioned below are two queues of queries maintained by the simulator for the purpose of tripwire detection. When a new i-query is created, $i \in \{1, 3\}$, that the simulator believes might set off the $(2, 1)$ or $(2, 3)$ tripwire, the simulator puts this i-query into *LeftQueue*, to be checked later; similarly for $i \in \{3, 5\}$, the simulator puts a newly created i-query into *RightQueue* if it believes this new query might set off a $(4, 3)$ or $(4, 5)$ tripwire. (The same 3-query might end up in both *LeftQueue* and *RightQueue*.) As evidenced by the procedure EmptyQueue() in Fig. 7, *LeftQueue* and *RightQueue* are emptied sequentially and separately, which we choose to do mostly because it offers conceptual advantages within the proof.

THE SIMULATOR IN PSEUDOCODE. The global variables present in game G_1 are tables P_1, P_1^{-1}, P_2, P_2^{-1}, $\ldots$, P_5, P_5^{-1} as well as a table $ETable[K]$ for each $K \in \{0,1\}^\kappa$, the (partially defined) keying function f and its inverse f^{-1}, the set of subkeys $\mathcal{Z}$ (being the image of f), the FIFO queues *LeftQueue* and *RightQueue*, the sets *LeftFreezer* and *RightFreezer*, as well as "bookkeeping sets" $Queries$ and $KeyQueries$ that we discuss below (the set $EQueries$ is only used in game G_2). Finally, there are two global variables *qnum* and *Eqnum*; these are "query counters" (initially set to 0) that measure, respectively, the size of the set $Queries$ and the number of distinct queries to E or E^{-1} made internally by the simulator; for the purpose of counting the latter, an additional table $TallyETable$, similar to $ETable$, is used by the function TallyEQuery (Figure 8). All global variables are persistent, in the sense that they maintain their state from one distinguisher query to the next. By $k\mathcal{Z}$, we denote the k-fold direct sum $\mathcal{Z} \oplus \cdots \oplus \mathcal{Z}$ of $\mathcal{Z}$.

The tables $P_1, P_1^{-1}, P_2, P_2^{-1}, \ldots, P_5, P_5^{-1}$ are maintained by the simulator and correspond to the five permutations. Here $P_1(x) = y, P_1^{-1}(y) = x$ if x maps to y under the first permutation, etc; initially, every entry in every P_i is $\perp$, and the simulator fills in entries as the game progresses. Consistency between P_i and P_i^{-1} is always maintained: we have $P_i(x) = y$ if and only if $P_i^{-1}(y) = x$, for all $x, y \in \{0,1\}^n$; moreover, entries are never overwritten. We let $\mathrm{domain}(P_i) = \{x : P_i(x) \neq \perp\}$, $\mathrm{range}(P_i) = \{y : P_i^{-1}(y) \neq \perp\}$. Each pair of tables $ETable[K]$, $ETable[K]^{-1}$ is similarly maintained by the ideal cipher E.

Our simulator can *abort* ("**abort**"). When the simulator aborts, we assume that the distinguisher D is consequently notified, and that D can then return its own output bit. (Since the "real world" never aborts, there should be little doubt about D's opinion, in this case, but D can return as it wants.)

Importantly, the simulator and the cipher E take *explicit random tapes* as randomness sources. The simulator's random tapes are tables $p_1, \ldots, p_5$ and r_f. Here p_i is actually two tables $p_i(\rightarrow, \cdot)$ and $p_i(\leftarrow, \cdot)$ defining a uniform random permutation; i.e., for every $x, y \in \{0,1\}^n$ we have $p_i(\rightarrow, x) = y$ if and only if $p_i(\leftarrow, y) = x$, and p_i is selected uniformly at random from all pairs of tables $p_i(\rightarrow, \cdot)$, $p_i(\leftarrow, \cdot)$ with this property. Likewise, the random tape p_E for E's use consists of a different random permutation $p_E[K]$ for each $K \in \{0,1\}^\kappa$ encoded as two tables $p_E[K](\rightarrow, \cdot)$, $p_E[K](\leftarrow, \cdot)$. The table r_f simply holds uniform random n-bit values: $r_f(K)$ is uniform at random in $\{0,1\}^n$ for each $K \in \{0,1\}^\kappa$.

The simulator uses the table p_i when it wants to "lazy sample", say, $P_i(x)$; instead of doing the random sampling on its own by a call such as "$P_i(x) \xleftarrow{\$} \{0,1\}^n \backslash \mathrm{range}(P_i)$", the simulator will call the function ReadTape$(P_i, x, p_i(\rightarrow, \cdot))$. (Concerning the implementation of ReadTape: we assume that $(P_i^{-1})^{-1} = P_i$, etc.) If all of the previously defined entries in P_i have been sampled via calls to ReadTape, the effect of calling ReadTape is identical to the instruction $P_i(x) \xleftarrow{\$} \{0,1\}^n \backslash \mathrm{range}(P_i)$, i.e., the outcomes are identically distributed. But if some entries of P_i have been adapted (i.e., not set via lazy sampling), then ReadTape might abort if, through bad luck, it hits an adapted value. Hence the use of ReadTape is not quite equivalent to pure lazy sampling, though the difference is fairly slight. Moreover, this issue does not arise when E uses ReadTape, because E never adapts its queries. (In the previous discussion we omitted mentioning ReadTape and the use of "explicit randomness" by the simulator, and we pretended that the simulator did pure lazy sampling; this is not quite the case, but it matters little for that discussion.)

We also note that while the random tapes represent an "unreasonably large" amount of at-hand randomness, the simulator (and ideal cipher E) can simulate access to such random tapes via lazy sampling. For example, the simulator just keeps a partially defined copy of p_i "in its head" for which it does (true) lazy sampling whenever it needs to read a new entry. Hence, access to this type of randomness can also be efficiently simulated, and we are not "cheating" by giving the simulator access to such random tapes.

Whenever the simulator defines a new entry in P_i it creates a "bookkeeping record" of this new entry in the set *Queries*. More precisely, a new entry $P_i(x) = y$, $P_i^{-1}(y) = x$ is recorded as a tuple (i, x, y, dir, num) added to the set *Queries*, where $dir \in \{\leftarrow, \rightarrow, \perp\}$ is the "direction" of the query and where *num* is the previous value of *qnum*, incremented by one (we call this the "query number" of the new tuple). Here $dir = \rightarrow$ if the new P_i entry is created by a call of the type $\text{ReadTape}(P_i, x, p_i(\rightarrow, \cdot))$, and $dir = \leftarrow$ if it is created by a call of the type $\text{ReadTape}(P_i^{-1}, y, p_i(\leftarrow, \cdot))$. In all other cases, $dir = \perp$. Queries with $dir = \perp$ are called *adapted*. The set *KeyQueries* is similarly maintained, but is simpler (see the function 'AddKeyQuery' in Figure 8). We note that *KeyQueries* and *Queries* both share the same "query counter" *qnum*, so that elements of these lists are totally ordered by their last coordinates.

We sometimes omit the last coordinate or last two coordinates of a query (i, x, y, dir, num)—writing simply (i, x, y, dir) or (i, x, y)—when these coordinates aren't of interest to the discussion at hand.

4.2 Proof Techniques

A self-contained overview of the indifferentiability proof appears in Section 4.3. Here we mention only the "main highlights"(with emphasis on novelties).

Our indifferentiability proof uses a fairly short sequence of games, with only four games in all. The first game implements the "simulated world" while the last game implements the "real world". A novel feature of our proof is that we use no "bad events" to bound the distinguishability of adjacent games. In places where a "bad event" flag might traditionally be used, our code simply aborts instead.

In a little more detail, the second game is identical to the first game except that it contains even more abort conditions than the first game. (Some of these new abort conditions involve the simulator "illegally" examining queries made by the adversary to the ideal cipher, which is why these abort conditions cannot be incorporated in the original simulated world.) Since the fourth game never aborts, and since the second game is identical to the first except that it sometimes aborts when the first game doesn't, it suffices to upper bound the distinguishability of the second and fourth games.

The third game changes only the implementation of the ideal cipher, which is no longer "ideal" in the third game, but is indeed implemented as a key-alternating cipher, where the key-alternating cipher uses the same random tapes (i.e., permutations and key scheduling function) as the simulator uses. The transition from the second to third game is the crucial transition, and to be perfectly formal we use the nice "randomness mapping" technique of Holenstein et al. [37]. This technique links the executions of two games with explicitly given random coins by exhibiting a partial bijective function from the set of random tapes in one game to the set of random tapes in the other that preserves game behavior, as viewed by the distinguisher. Our randomness mapping argument presents some novelties, however. In particular, we observe that it is sufficient for executions that are paired up by the randomness map (one execution in the second game, one execution in the third game) to have *very similar probabilities of occurring* in each world as opposed to *exactly equal probabilities of occurring*. This natural relaxation allows us to handle lazy permutation sampling without the complicated workarounds of Holenstein et al. (in particular without use of a "two-sided random function") and considerably simplifies the whole argument. Our randomness mapping argument also introduces the idea of *random footprints* and of *execution trees*, of potential independent interest.

The transition from the third game to the fourth game is rather straightforward, as one can show (somewhat similarly to the first and second games) that the third and fourth games proceed identically on identical random tapes except for the possibility that the third game might abort while the fourth game (which is the real world) never aborts. For this transition it thus suffices to upper bound the probability of the third game aborting, which is easy to do once we have already proved that the second game aborts with small probability (establishing the latter is necessary for the randomness mapping argument) and by the previously established similarity of the second and third games.

Another "syntactic novelty" in our proof, besides the fact that we eschew bad events in favor of abort conditions, is that our simulator maintains explicit "bookkeeping" data structures in addition to its other data structures. The bookkeeping data structures keep track, among others, of the order and "direction" (for permutation queries) of queries internally defined by the simulator. There are two main advantages here: (1) having an unambiguous timeline and description of events within the data structures themselves, which clarifies arguments within the proof; (2) the fact that the "bookkeeping copy" is only updated with new information after a series of checks have been made (if one of these checks isn't passed, the simulator aborts) which implies that various "good invariants" (postulated about the bookkeeping data structures instead of about the primary data structures) can be shown to hold *unconditionally* at *any point in any execution*. Having such "unconditional invariants" considerably simplifies the language in the proof, which is a very non-negligible gain. One could theoretically achieve the same effect using only primary data structures, but then one cannot, for example, include an instruction that simultaneously reads from a random tape and updates the primary data structure with the read value, since the value might momentarily corrupt an invariant (even if this is caught and abort occurs soon after, the invariant no longer holds unconditionally at all points in time). One would need, instead, to check the value after reading it from the random tape before using it. By contrast, being able to immediately use a random value and then check its goodness only a few lines later, when it comes time to update the bookkeeping data structure, produces much more readable code.

Another standard concern of indifferentiability proofs is the issue of simulator termination, already mentioned in the previous subsection. For more details on the termination argument we refer to the outline in Section 4.3 (the termination argument itself appears in Appendix C).

4.3 Proof Overview

In this section we give the backbone of the proof of Theorem 3, our main result: the indifferentiability of 5-round key-alternating cipher with RO-scheduled subkeys. Some supporting lemmas are found in Appendices C and D.

The simulator $\mathcal{S}$ referred to in the statement of Theorem 3 is, of course, the simulator outlined in Section 4.1, and formally given by the game G_1 in Figures 5–8.

We start by noting that Theorem 3 actually consists of three separate claims: (i) the indistinguishability of the real key-alternating cipher and of its simulated counterpart; (ii) the fact that $\mathcal{S}$ never makes more than $2q^2$ queries to the ideal cipher $\mathcal{IC}$ (renamed as the functions E, E^{-1} in game G_1) when interacting with a q-query distinguisher D; (iii) the fact that that $\mathcal{S}$'s total running time[13] is $O(q^3)$ with probability 1. Proofs of (ii) and (iii) can be found in lemmas 9 and 10 at the end of the section. The rest of our discussion is devoted to (i).

[13] For simplicity, every pseudocode instruction is assumed to take unit time. Other models of running time might introduce additional factors of order $O(n)$.

Our indistinguishability argument uses a sequence of four games. Each game is an environment in which the distinguisher D can be run. We start by briefly describing the four games:

• Game G_1 (Figures 5–8 in Appendix B) is "the simulated world". The distinguisher's left oracle E, E^{-1} implements an ideal cipher, while the distinguisher's right oracle, consisting of eleven distinct interfaces f, P1, $P1^{-1}$, …, P5, $P5^{-1}$ implements our 5-round simulator as discussed in Section 4.1.

• Game G_2 (Figures 5–8 in Appendix B) makes some modest changes to game G_1. Essentially, a number of abort conditions are added to the simulator, and some abort conditions are also added to E/E^{-1}. G_2 uses the same random tapes as G_1, and two executions of G_1 and G_2, for the same random tapes and the same distinguisher queries, will proceed identically except for the possibility that G_2 might abort when G_1 does not. We note that one of the abort conditions added to the simulator in game G_2 (in the function FreezeLeftValues(), Figure 6) "illegally" examines the private tables $ETable[\cdot]$ maintained by the cipher E, and also that E now examines the simulator's own tables from within the function AddEQuery; however, since this is an intermediate game and not the simulated world, these idiosyncrasies are of no import.

• Game G_3 (Figure 9 in Appendix B) changes the procedures $E(K,\cdot)$, $E^{-1}(K,\cdot)$ to directly use the table r_f (in order to compute the key schedule k of K) and the permutation tables $p_1,\ldots,p_5$ to compute its answers, treating these tables as the cipher's underlying permutations. Moreover, a second (shallow) change occurs in game G_3, in that the random tables $p_1,\ldots,p_5$ are actually renamed as $q_1,\ldots,q_5$, in order to facilitate future comparison between games G_2 and G_3. However, the simulator-related procedures of game G_3 are not rewritten to reflect this change, since this would more or less be a waste of paper. To summarize: game G_3 is obtained by changing p_i everywhere to q_i in game G_2, and by replacing the procedures E and E^{-1} of Figure 5 with those of Figure 9.

• Game G_4 (Figure 9 in Appendix B) is the "real world": the simulator directly answers queries using r_f and the permutation tables $q_1,\ldots,q_5$, while E/E^{-1} are unchanged from game G_3.

Proving indifferentiability amounts to showing that games G_1 and G_4 are indistinguishable. For this, it turns out to be helpful if we first "normalize" the distinguisher D. More precisely, we assume (i) that D is deterministic and always outputs either 1 or 0, (ii) that D outputs 1 if the system aborts, and (iii) that D *completes all paths*, meaning that for every query $E(K,x) \to y$ or $E^{-1}(K,y) \to x$ made by D, D eventually makes the (possibly redundant queries)

$$f(K) \to k,\ P1(x \oplus k) \to y_1,\ P2(y_1 \oplus k) \to y_2,\ \ldots,\ P5(y_4 \oplus k) \to \ldots$$

in this order, unless it is prevented from doing so because the system has aborted. (Presumably, the output of the query $P5(y_4 \oplus k)$ will be $y \oplus k$, but whether this is the case does not concern the definition of a path-completing distinguisher.) Points (i) and (ii) are obviously without loss of generality, since G_4 never aborts. Point (iii) is also without loss of generality as long as we give D a few extra queries (or, to be precise, a factor 6 more queries), since D is free to ignore the information that it gathers while path-completing. In more detail, we first prove indifferentiability with respect to a "normalized" (i.e., deterministic, path-completing, etc) q-query distinguisher D, and then deduce our main theorem via a straightforward reduction (with a factor 6 loss in the number of queries).

Thus let D denote a fixed, q-query deterministic distinguisher D that completes all paths. Notations such as

$$D^{G_2} = 1 \qquad \text{and} \qquad D^{G_2}(\alpha) = 1$$

indicate that D outputs 1 after interacting with G_2, but the second notation explicitly mentions the random tape $\alpha = (r_f, p_1, \ldots, p_5, p_E)$ on which the game is run. It is sufficient and necessary to upper bound

$$\Pr[D^{G_1} = 1] - \Pr[D^{G_4} = 1]$$

where the probabilities are computed over the explicit random tapes in each game (and *only* over these random tapes, since D is deterministic). Since game G_2 only introduces additional abort conditions from G_1 and since D outputs 1 when the game aborts, we have

$$\Pr[D^{G_2} = 1] \geq \Pr[D^{G_1} = 1]$$

and so it suffices to upper bound

$$\Pr[D^{G_2} = 1] - \Pr[D^{G_4} = 1].$$

For the latter, we apply a standard hybrid argument by upper bounding

$$\Pr[D^{G_2} = 1] - \Pr[D^{G_3} = 1] \tag{3}$$

and

$$\Pr[D^{G_3} = 1] - \Pr[D^{G_4} = 1] \tag{4}$$

separately.

The crux of the proof to upper bound (3), i.e., the game transition from G_2 to G_3, as the transition from G_3 to G_4 turns out to be much less problematic (cf. Lemma 8 below). To upper bound the transition from G_2 to G_3 we essentially use a randomness mapping argument *à la* Holenstein et al. [37]. It seems worthwhile to first give a high-level overview of the randomness mapping argument, which requires a few more definitions.

Randomness mapping (high-level overview). An *execution* consists of the start-to-finish interaction of D with either G_2 or G_3, including all internal actions performed by the simulator (i.e., performed by the game[14]). Since D is fixed and deterministic, we note that every tuple of random tapes in G_2 determines a unique G_2-execution and likewise every tuple of random tapes in G_3 determines a unique G_3-execution. On the other hand, two different tuples of (say) G_2 random tapes might give rise to the same G_2 execution since certain portions of the random tapes might not be read, and thus not affect the execution. (The execution includes everything read from the random tapes but not the random tapes themselves.) If $\alpha = (r_f, p_1, \ldots, p_5, p_E)$ is a G_2 random tuple, the *footprint* of α consists of that portion of the random tapes actually read during the execution $D^{G_2}(\alpha)$. (This somewhat hand-wavy definition is more carefully restated below.) We make a similar definition for G_3. By definition, then, there is a bijection between the set of possible G_2 executions and the set of different G_2 footprints, and similarly there is a bijection between the set of possible G_3 executions and the set of different G_3 footprints. We say a G_2 footprint is *good* if G_2 does not abort on that execution, and likewise a G_3 footprint is *good* if G_3 does not abort on the corresponding execution.

One can observe that not all footprints have the same probability of occurring, even if D is somehow normalized to always make the same number of queries (which we are not even assuming); for example, if D only makes queries to the key scheduling function f on a certain execution, that execution's probability will be a power of $(1/2^n)$, which will not be the case for a generic execution.

In a nutshell, the randomness mapping argument upper bounds $\Pr[D^{G_2} = 1] - \Pr[D^{G3} = 1]$ by exhibiting a bijection τ between the set of good G_2 footprints and good G_3 footprints such that (i) τ preserves the output of D (indeed, τ maps G_2 executions to G_3 executions that look exactly the same from D's viewpoint); (ii) τ maps executions of G_2 to executions of G_3 of nearly equal probability. For the randomness mapping to be effective, one obviously needs, thirdly, the set of good G_2 footprints to represent most of the probability mass of all G_2 footprints (since the domain of τ is limited to good

[14] In G_2 and G_3 we use "simulator" and "game" interchangeably. This choice of terminology would make less sense for G_1, since the functions E/E^{-1} are obviously not part of the "original" simulator.

footprints), which is exactly the same as saying that one needs the probability of abortion to be low in G_2. One of the main sub-goals of the proof, thus, is to show that G_2 aborts with low probability.

We note that our randomness mapping argument has some significant differences with that of Holenstein et al., from a technical standpoint. Most notably, Holenstein et al.'s randomness map has the property that an execution and its image under the map have *exactly the same probability* instead of *nearly equal probability*. To achieve this Holenstein et al. need, in particular, to use random tapes in which each n-bit block is uniformly random and independent from other n-bit blocks in the tape. Such a random tape cannot represent a random permutation which means that Holenstein et al. have to jump through various hoops in order to work with random permutations, thus significantly complicating their proof. We believe that relaxing the requirements on the randomness map constitutes a more efficient and natural approach. A second difference is that Holenstein et al. reason directly about the "original" probability space consisting of all possible random tapes instead of reasoning about the "induced" probability space consisting of all possible footprints (which we found to be more convenient).

On the other hand, the idea of working with a distinguisher D that completes all paths is lifted directly from [37]. Without such a distinguisher, defining the randomness map seems much harder. (Indeed, finding a clean-cut proof that does not require a distinguisher that completes all paths would be a significant technical innovation for such proofs.)

G_2 ABORTION AND TERMINATION ARGUMENT. As noted in the outline above, proving that G_2 aborts with small probability is a central component of the proof. The following lemma, which upper bounds this probability, is proved in Appendix C:

Lemma 1. *The probability of abortion is at most* $160q^{10}/2^n$ *in* G_2*. I.e.,*

$$\Pr[(r_f, p_1, \ldots, p_5, p_E) \text{ are good for } G_2] \geq 1 - \frac{160q^{10}}{2^n}$$

for any (q-query, deterministic) distinguisher D*.*

Because G_2 automatically aborts if the simulator makes too many internal queries (see the function AddQuery in Fig. 7) proving Lemma 1 subsumes the termination issue, i.e., to prove Lemma 1 we in particular need to prove that "out-of-control" chain reactions either don't occur or else occur with low probability in game G_2. (While proving out-of-control chain reactions occur only with low probability would be sufficient, we show in Appendix C that such chain reactions don't occur at all. Indeed, G_2 pre-emptively aborts when any anomalies occur that might eventually lead to such reactions.) While termination is thus the domain of Lemma 1 and of Appendix C we include a (very) high-level outline of the termination argument here for the curious reader's sake. We can emphasize at the outset that our termination argument bears no relation at all to Holenstein et al.'s, as our termination argument relies on fine-grained combinatorial observations that are tailored to our simulator and to the setting of key-alternating ciphers.

Let us first recall how a (not out-of-control) "chain reaction" might occur in the first place. Say a distinguisher D makes the queries $\mathrm{f}(K_1) \to k_1$, $\mathrm{f}(K_2) \to k_2$, $\mathrm{f}(K_3) \to k_3$, $\mathrm{f}(K_4) \to k_4$, then chooses a value $x_2^1 \in \{0,1\}^n$ and makes the queries $\mathrm{P2}(x_2^1) \to y_2^1$, $\mathrm{P2}^{-1}(y_2^2) \to x_2^2$ where $y_2^2 := y_2^1 \oplus k_1 \oplus k_2$, $\mathrm{P2}(x_2^3) \to y_2^3$ where $x_2^3 := x_2^2 \oplus k_2 \oplus k_3$, $\mathrm{P2}^{-1}(y_2^4) \to x_2^4$ where $y_2^4 := y_2^3 \oplus k_3 \oplus k_2$. Let $y_1^1 := x_2^1 \oplus k_1$, $x_3^{12} := y_2^1 \oplus k_1 = y_2^2 \oplus k_2$, $y_1^{23} := x_2^2 \oplus k_2 = x_2^3 \oplus k_3$, $x_3^{34} := y_2^3 \oplus k_3 = y_2^4 \oplus k_4$ and $y_1^4 := x_2^4 \oplus k_4$. If the distinguisher queries $\mathrm{P1}^{-1}(y_1^1)$ at this point the simulator will create a new 1-query $(1, x_1^1, y_1^1, \leftarrow)$ and put this query on *LeftQueue*; the query is immediately popped from *LeftQueue*, and the simulator detects that y_1^1 is k_1-adjacent to the pre-existing 2-query $(2, x_2^1, y_2^1, \to)$, so the simulator completes a path of subkey k_1 passing through the points y_1^1 and x_3^{12}, and in particular creates a new 3-query

$(3, x_3^{12}, y_3^{12}, \rightarrow)$ in the process that is put on *LeftQueue* (in the form of the pair $(3^-, x_3^{12})$); when this 3-query is popped from *LeftQueue*, a k_2-adjacency with the pre-existing 2-query $(2, x_2^2, y_2^2)$ is uncovered, leading to the completion of a path of label k_2 going through y_1^{23}, and so on. In all, the simulator creates paths of labels k_1, k_2, k_3 and k_4 before *LeftQueue* is finally emptied.

In our proof, the 2-queries $(2, x_2^1, y_2^1), \ldots, (2, x_2^4, y_2^4)$ together with the scheduled keys $k_1, \ldots, k_4 \in \mathcal{Z}$ gives rise to a connected component of 4 edges in a bipartite graph called B_2. (In fact, the way B_2 is defined, the edges have endpoints y_1^1, x_3^{12}, y_1^{23}, x_3^{34}, y_1^4 and are labeled by the values $k_1, \ldots, k_4$. Each 2-query gives rise to $|\mathcal{Z}|$ different parallel edges of B_2.) A similar graph B_4 is defined with respect to 4-queries. Essentially, the graphs B_2 and B_4 are defined such that chain reactions occur within connected components of B_2 and B_4, as in the case above with B_2.

The key observation we make is that while a chain reaction is "burning through" a connected component of B_2, no edges are added to B_2 (thus not enlarging the size of the maximum connected component of B_2) while the edges that are added to B_4 are disconnected[15] from previous connected components of B_4 by design; for example, when the new 3-query $(3, x_3^{12}, y_3^{12}, \rightarrow)$ is created above, the set of values $y_3^{12} \oplus 4\mathcal{Z}$ is disjoint from the set of left endpoints of edges in B_4 with high probability because the query has direction $\rightarrow$ (and if it *isn't* disjoint the simulator actually aborts; cf. the first "forall" loop of procedure QueryChecks in Fig. 8). In a nutshell, the modifications made to B_4 while a component of B_2 is being processed are "random and therefore harmless", and vice-versa.

To be a little more precise we succeed in showing that the size of the largest maximum connected component (of a certain "unprocessed" type) in the disjoint union of B_2 and B_4 grows by at most 2 every time the simulator answers an adversarial query. Thus the size of the largest component in B_2, B_4 grows no faster than linearly in the number of queries, which cinches the termination argument.

Of course, the termination argument is not quite as simple as sketched above. In particular, the four-query example that we used doesn't represent the full complexity of cases that can occur. E.g., the distinguisher might first make a 3-query and then build large connected components in both B_2 and B_4 on either side of this 3-query without triggering any tripwires before "lighting the fuse" and causing the components on both sides to be processed. Or the adversary might make matching 1- and 5-queries (i.e., via an E-query), attach a B_2 component to the 1-query and a B_4 component to the 5-query without triggering any tripwires, and then cause both these components to be processed by making a single query. All possible cases are analyzed in Appendix C, and we refer in particular to the proof of Lemma 18 for a detailed and leisurely walkthrough. A salient and important feature of all cases, however, is that at most one pre-existing component of B_2 and one pre-existing component of B_4 is affected per distinguisher query.

The transition from G_2 to G_3, and the rest of the proof. Having given outlines of the proof's main elements above (randomness mapping and termination) we continue with more details, in particular upper bounding the transition from G_2 to G_3 and the transition from G_3 to G_4. We assume Lemma 1 which is proved in Appendix C.

A tuple of random coins

$$(r_f, p_1, \ldots, p_5, p_E)$$

for G_2 is called a G_2-*tuple*, whereas a tuple of random coins

$$(r_f, q_1, \ldots, q_5)$$

for G_3 is, likewise, called a G_3-tuple. We let R_2 be the set of all G_2-tuples, and R_3 be the set of all G_3-tuples. Elements in $R_2 \cup R_3$ are simply called *random tuples*.

[15] This oversimplifies. In specific situations one or two edges may be added to unprocessed components of B_4. See Appendix C for more details.

We say a G_2-tuple

$$(r_f, p_1, \ldots, p_5, p_E) \in R_2$$

is *good* (with respect to our fixed distinguisher D) if the simulator doesn't abort for this tuple. In this case, we also say that the interaction of D with G_2 constitutes a *good execution.* We let $R_2^{\text{good}} \subseteq R_2$ be the set of all good G_2-tuples. We emphasize that R_2^{good} depends on D, even if this dependency is not reflected in the notation.

We can view the permutation tables $p_1, \ldots, p_5, \{p_E[K] : K \in \{0,1\}^\kappa\}, q_1, \ldots, q_5$ that are present in random tuples as perfect matchings from $\{0,1\}^n$ to $\{0,1\}^n$. The *restriction* of such a matching (permutation) p is obtained by removing a subset of the edges (possibly empty); this can still be encoded as a table p by setting relevant entries $p(\to, x)$ and $p(\leftarrow, y)$ to $\perp$. Likewise, a *restriction* of r_f is obtained by setting certain entries of r_f to $\perp$. Finally, the *restriction* of a G_2-tuple $(r_f, p_1, \ldots, p_5, p_E)$ is obtained by taking arbitrary restrictions of each coordinate (for p_E, we mean taking a different restriction of $p_E[K]$ for each $K \in \{0,1\}^\kappa$). We call the result a *partial* G_2*-tuple* or a *restricted* G_2*-tuple*. We let R_2^{partial} be the set of all partial G_2-tuples, and we define R_3^{partial} similarly. (In particular, $R_j \subseteq R_j^{\text{partial}}$ for $j = 2, 3$.)

Running G_2 on an element of R_2^{partial} doesn't necessarily make sense, since G_2 might access a $\perp$-value. But if G_2 doesn't access any such entries, then the execution, obviously, is well-defined. Similar remarks hold for G_3 and for R_3^{partial}.

We say an edge (x, y) of a permutation p of a random tuple is *examined* during an execution of the relevant game (either G_2 or G_3, and for our fixed distinguisher D) if the value $p(\to, x)$ or $p(\leftarrow, y)$ is ever read; likewise we say an entry $r_f(x)$ is *examined* during an execution, if it is read. The *footprint* of a random tuple is the partial tuple obtained by keeping only the edges examined during the execution associated to that tuple, and by likewise keeping only the entries of r_f examined during that tuple. Thus a footprint for G_2 is an element of R_2^{partial}, and a footprint for G_3 is an element of R_3^{partial}.

Since the only source of randomness is the coins $(r_f, p_1, \ldots, p_5, p_E)$ in G_2 and $(r_f, q_1, \ldots, q_5)$ in G_3, we note that an execution's footprint contains all the information necessary to re-create the execution. Thus, there is a bijective correspondence between the set of possible executions and the set of possible footprints. (Still, always, for our fixed D.)

We let $R_j^{\text{foot}} \subseteq R_j^{\text{partial}}$ denote the set of G_j-footprints and let $R_j^{\text{good-f}} \subseteq R_j^{\text{foot}}$ be the set footprints of good executions in G_j for $j = 2$, 3. (Note that $R_j^{\text{good-f}} \not\subseteq R_j^{\text{good}}$ due to the fact that $R_j^{\text{good-f}}$ contains partial tuples, whereas R_j^{good}, by definition, contains full tuples.)

The *execution tree* of a game (say G_2) groups all possible executions with distinguisher D as follows: each time a value is read from one of the random tables $(r_f, p_1, \ldots, p_5, p_E)$, the tree forks into as many branches as there are possible answers. (So for example, when a value is read from r_f, this corresponds to a 2^n-wise fork; if a value is read from p_1 (in either direction) when two values have already been read from p_1, this gives rise to a $(2^n - 2)$-wise fork in the execution tree, and so on.) The leaves of the tree correspond to aborted states or to points at which D decides to return. (Since in any case D returns 1 after abortion, all leaves, in effect, correspond to return points of D.) We label the leaves by the return value of D. (We emphasize that the nodes in the execution tree do *not* correspond to queries made by D; every node in the execution tree corresponds, instead, to random tape accesses. Indeed, with D being deterministic, these accesses are the only "source of branching" when D interacts with G_2.) The execution tree of G_3 is defined similarly.

The possible executions of G_2 are in bijective correspondence with the leaves in the execution tree of G_2 (ditto for G_3), and, by the same token, in bijective correspondence with the different possible footprints for G_2 (ditto for G_3). Moreover, the probability that a leaf v is reached (with probability taken over the uniform choice of $(r_f, \ldots, p_E) \in R_2$ in G_2 and over the uniform choice of

$(r_f, \ldots, q_5) \in R_3$ in G_3) is

$$\prod_u \frac{1}{\deg(u)}$$

where the product is taken over all nodes u from the root of the tree to v, and where $\deg(u)$ is the degree of u (as is easy to see). We also note that the depth of a leaf v (i.e., the path length from v to the root) is the total number of permutation edges plus the number of r_f-entries in the footprint associated to v.

We also note that the probability of a leaf can be easily computed from the leaf's footprint, without having to retrace the execution associated to the footprint. More precisely, if we notate by $|p|$ the number of edges in a partial matching p and if we notate by $|r_f|$ the number of non-$\perp$ entries in r_f, then the probability of reaching a leaf v with footprint $(r_f, p_1, \ldots, p_5, p_E) \in R_2^{\text{partial}}$ in the G_2 execution tree is

$$\left(\frac{1}{2^n}\right)^{|r_f|} \left(\prod_{i=1}^{5} \frac{(2^n - |p_i|)!}{2^n!}\right) \left(\prod_{K \in \{0,1\}^\kappa} \frac{(2^n - |p_E[K]|)!}{2^n!}\right).$$

A similar (but simpler) computation holds for leaves of G_3.

Let $\Pr_{\mathrm{G}_2}[v]$ denote the probability of reaching a leaf v of the G_2 execution tree, and likewise define $\Pr_{\mathrm{G}_3}[v]$ to be the probability of reaching a leaf v in the G_3 execution tree. Let $\mathrm{lf}_1(\mathrm{G}_2)$ be the set of 1-leaves in the G_2 execution tree, and likewise for G_3. Then

$$\Pr[D^{\mathrm{G}_2} = 1] = \sum_{v \in \mathrm{lf}_1(\mathrm{G}_2)} \Pr_{\mathrm{G}_2}[v]$$

and a similar equation holds for G_3.

As already outlined, the basic idea of the randomness mapping argument is to exhibit an injective mapping τ from the leaves of the G_2 execution tree to the leaves of the G_3 execution tree such that (i) the domain of τ is all (leaves associated to) non-aborted G_2 executions (and, in particular, "almost all" leaves are in the domain of τ, as measured by their total probability mass), (ii) τ maps 1-leaves to 1-leaves and 0-leaves to 0-leaves, and (iii) $Pr_{\mathrm{G}_2}[v]$ is very close to $\Pr_{\mathrm{G}_3}[\tau(v)]$, for all v in the domain of τ. Details follow.

We start by defining a map

$$\tau : R_2^{\text{good}} \to R_3^{\text{partial}}.$$

In a second step, we will show that the "restriction" of τ to the footprint of a G_2-tuple $(r_f, \ldots, p_E)$ makes sense (so that τ can be defined on the non-aborted leaves of the G_2 execution tree, i.e., can be defined on $R_2^{\text{good-f}}$), and that elements in the image of τ are in fact in $R_3^{\text{good-f}}$, so that the range of τ consists of leaves of the G_3 execution tree.

Let $\alpha := (r_f, p_1, \ldots, p_5, p_E) \in R_2$. Then $\tau(\alpha)$ is simply defined according to the tuple $(f, P_1, \ldots, P_5)$, as it stands at the end of the execution of D^{G_2} with randomness α. More precisely, if $(r_f^*, q_1, \ldots, q_5)$ is the image of $\tau(\alpha)$, then $r_f^*(K) = f(K)$ for all $K \in \{0,1\}^\kappa$, $q_i(\rightarrow, x) = P_i(x)$ for all $x \in \{0,1\}^n$, and $q_i(\leftarrow, y) = P_i^{-1}(y)$ for all $y \in \{0,1\}^n$. (In this case, we also say that q_i is a *copy* of P_i.) Clearly, $\tau(\beta)$ is also well-defined for any footprint β, and in particular for any $\beta \in R_2^{\text{good-f}}$.

The next proposition (variants of which are also proved in Appendix C) makes a few basic observations that are helpful, in particular, for the subsequent lemma.

Proposition 1. *Consider a non-aborting execution of G_2. Let (j, x, y, dir, num) be a query (i.e., an element of Queries) with $j = 2$ or $j = 4$. Then $dir \in \{\leftarrow, \rightarrow\}$. Moreover, if $(i, x_i, y_i, dir_i, num_i)$,*

$(i+1, x_{i+1}, y_{i+1}, dir_{i+1}, num_{i+1})$ *are k-adjacent queries for some* $k \in \mathcal{Z}$*, with* (K, k, num_k) *the corresponding element of KeyQueries, then (i)* $num_k < \max(num_i, num_{i+1})$*, and (ii)* $dir_i = \rightarrow$ *implies* $num_i < num_{i+1}$ *whereas* $dir_{i+1} = \leftarrow$ *implies* $num_i > num_{i+1}$*.*

Proof. The first assertion is a consequence of the fact that 2-queries and 4-queries are always lazy-sampled (see the code of the procedures P2() and P4()). Point (i) of the second assertion is a direct consequence of the abort condition in the second for-loop of the procedure KeyQueryChecks in Figure 8 while point (ii) is a direct consequence of point (i) and of the checks that occur in the first two for-loops of the procedure QueryChecks (still Figure 8), since $\mathcal{Z} \subseteq 5\mathcal{Z}$. □

Lemma 2. *Assume that* D *is a distinguisher that completes all chains, and let* $P_1, \ldots, P_5$*, Queries, EQueries and* f *be the global variables in* G_2 *at the end of a non-aborting (i.e., "good") execution* D^{G_2}*. Then for all* $(K, x, y) \in$ *EQueries we have* $f(K) \neq \perp$ *and there exists a corresponding completed path in Queries, i.e., there exists a 5-tuple*

$$(1, x_1, y_1), (2, x_2, y_2), (3, x_3, y_3), (4, x_4, y_4), (5, x_5, y_5)$$

in Queries such that $x_1 = x \oplus f(K)$*,* $y_5 = y \oplus f(K)$ *and such that* $y_i \oplus f(K) = x_{i+1}$ *for* $i = 1, 2, 3, 4$*.*

Proof. Let $(K, x, y, dir) \in \mathit{EQueries}$ at the end of the G_2 execution (with $dir \in \{\leftarrow, \rightarrow\}$). Then either D queried $\mathrm{E}(K, x)$ or $\mathrm{E}^{-1}(K, y)$ at some point, or else the simulator made one of these queries from within EmptyQueue(), ForcedP3() or CompletedPath() (since these are the only functions in which the simulator queries E). If D queried $\mathrm{E}(K, x)$ or $\mathrm{E}^{-1}(K, y)$, then since D completes all chains D also queries $f(K)$ at some point and there exist five queries

$$(1, x_1, y_1, dir_1, num_1), (2, x_2, y_2, dir_2, num_2), \ldots, (5, x_5, y_5, dir_5, num_5)$$

in *Queries* such that $x_1 = k \oplus x_1$ and such that $y_i \oplus k = x_{i+1}$ for $i \in \{1, 2, 3, 4\}$. We next argue that $y_5 \oplus k = y$.

To see this, note that $dir_2 \in \{\leftarrow, \rightarrow\}$ by Proposition 1. Say first that $dir_2 = \leftarrow$. Then, again by Proposition 1, we have that $num_1 > num_2$, that $f(K)$ is scheduled before query num_1 is created, and that $dir_1 \neq \rightarrow$. When query num_1 is created, therefore, it goes onto *LeftQueue*, and when it is popped from *LeftQueue* we already have $f(K) \in \mathcal{Z}$ and $y_1 \oplus f(K) = x_2 \in \mathrm{domain}(P_2)$; the simulator will therefore complete a path, at this point, for the pair of adjacent queries $(1, x_1, y_1)$, $(2, x_2, y_2)$, with respect to the cipher key K; since the simulator aborts if it cannot complete the path successfully (and since the simulator hasn't aborted), the path must therefore consist of queries $num_1, \ldots, num_5$ above (obviously, one can "uniquely follow" a path from left to right), and we must have $y_5 \oplus f(K) = \mathit{ETable}[K](x)$, i.e., $y_5 \oplus f(K) = y$. In the second case, if $dir_2 = \rightarrow$, then we can similarly argue that $num_3 > num_2$, that $f(K)$ was scheduled before num_3 was created, and that $dir_3 \neq \leftarrow$, so that $(3^-, x_3)$ was placed on $\mathit{LeftQueue}$ when query num_3 was created, and so that when this pair is popped from $\mathit{LeftQueue}$ the simulator detects the adjacency $y_2 \oplus f(K) = x_3$, and completes a path for queries num_2 and num_3, which must be the path above.

Next, we consider the case when the simulator made either the query $\mathrm{E}(K, x)$ or $\mathrm{E}^{-1}(K, y)$ (but the distinguisher did not make such queries). If the simulator makes (say) the query $\mathrm{E}(K, x)$ in EmptyQueue(), then obviously there exists a corresponding completed path in *Queries* since the simulator is in the process of completing this path, and since the simulator doesn't abort on this execution, by assumption. Now say the simulator makes the query $\mathrm{E}(K, x)$ or $\mathrm{E}^{-1}(K, y)$ in ForcedP3(i, z)— say, with $i = 3^-$, $x_3 := z$. For this query to be made, there must (already) exist queries $(1, x_1, y_1)$, $(2, x_2, y_2) \in$ *Queries* such that $x = x_1 \oplus f(K)$ and $y_1 \oplus f(K) = x_2$, $y_2 \oplus f(K) = x_3$, and $f(K)$ must already be scheduled. If ForcedP3 finds that $\mathrm{E}(K,x) \oplus f(K) \in \mathrm{range}(P_5)$ then it completes the corresponding path up to the 3-query $(3, x_3, y_3) := (3, x_3, x_4 \oplus f(K))$, which is left for PrivateP3() to add (and thus complete the path). Otherwise, if $\mathrm{E}(K,x) \oplus f(K) \notin \mathrm{range}(P_5)$, then ForcedP3 returns $\perp$, but PrivateP3() will add $(3^-, x_3)$ to *LeftQueue*, and when this pair is subsequently popped from *LeftQueue* in EmptyQueue(), the corresponding path will be completed because $f(K)$ is already scheduled and $(2, x_2, y_2)$ is already in *Queries*. (Calls to ForcedP3(i,z) with $i = 3^+$ are argued symmetrically.) Finally, for queries to $\mathrm{E}/\mathrm{E}^{-1}$ made by the simulator in the function CompletedPath(), the argument is clear, because CompletedPath() aborts unless the $\mathrm{E}/\mathrm{E}^{-1}$ query matches the existing path. □

Lemma 3. *$\tau : R_2^{\text{good-f}} \to R_3^{\text{partial}}$ is one-to-one, and the image under τ of a good G_2-footprint is a good G_3-footprint of equal label (0 or 1).*

Proof. Let $\alpha = (r_f, \ldots, p_E) \in R_2^{\text{good-f}}$ and let $\beta = \tau(\alpha) = (r'_f, q_1, \ldots, q_5) \in R_3^{\text{partial}}$. Since α is a footprint, the f-entries defined at the end of the execution $D^{\mathrm{G}_2}(\alpha)$ are exactly the non-$\perp$ entries in r_f. Hence, by definition of τ, $r'_f = r_f$ and $\beta = (r_f, q_1, \ldots, q_5)$ where $q_1, \ldots, q_5$ are copies of the tables $P_1, \ldots, P_5$ at the end of the execution $D^{\mathrm{G}_2}(\alpha)$.

We consider parallel executions of $D^{\mathrm{G}_2}(\alpha)$ and $D^{\mathrm{G}_3}(\beta)$. We note that the only difference between G_2 and G_3, besides the renaming of the random tapes $p_1, \ldots, p_5$ and $q_1, \ldots, q_5$, lies in the implementation of the procedures E, E^{-1}. Thus, when E and E^{-1} are treated as black boxes, the only differences that may arise in the parallel executions of $D^{\mathrm{G}_2}(\alpha)$ and $D^{\mathrm{G}_3}(\beta)$ (if any) must be due to one of: (i) different answers returned by the "black boxes" E or E^{-1}; (ii) differences in the tables $p_1, \ldots, p_5$ and $q_1, \ldots, q_5$ (since r_f is the same for α and β, and since p_E is only used within $\mathrm{E}/\mathrm{E}^{-1}$).

In particular, to show that the two parallel executions don't "diverge", it is sufficient to show: (a) that queries to E and E^{-1} are always answered identically in both executions, and (b) that whenever an assignment of the type $y := P_i(x) \leftarrow p_i(\rightarrow, x)$, $P_i^{-1}(y) \leftarrow x$ or of the type $x := P_i^{-1}(y) \leftarrow p_i(\leftarrow, y)$, $P_i(x) \leftarrow y$ occurs in G_2, then we have $q_i(\rightarrow, x) = p_i(\rightarrow, x)$ and (necessarily) $q_i(\leftarrow, y) = p_i(\leftarrow, y)$ in G_3. (Indeed, (b) is sufficient since such assignments are the only ways in which the tables $p_1, \ldots, p_5$, $q_1, \ldots, q_5$ are used in either game.)

We prove (a) and (b) by induction on the number of steps having occurred in the parallel execution, where calls to E and E^{-1} are treated in a black-box fashion in order to keep the number of steps comparable. Firstly, (b) is clear, since values in P_i are never overwritten and since values in P_i transfer over to q_i by definition of τ. For (a), we note that Lemma 2 implies that at the end of the execution of $D^{\mathrm{G}_2}(\alpha)$, there exists a unique 5-chain in $P_1, \ldots, P_5$ for every (distinct[16] query to $\mathrm{E}/\mathrm{E}^{-1}$ made by D or by the simulator such that the 5-chain is consistent with the $\mathrm{E}/\mathrm{E}^{-1}$-query. Since $q_1, \ldots, q_5$ are the copies of $P_1, \ldots, P_5$ at the end of the execution $D^{\mathrm{G}_2}(\alpha)$, this directly implies (by construction of E in G_3) that queries to E and E^{-1} in G_3 are answered the same as in G_2. Hence the two parallel executions $D^{\mathrm{G}_2}(\alpha)$ and $D^{\mathrm{G}_3}(\beta)$ are identical when E and E^{-1} are treated as black boxes.

We next argue that β is the footprint of some good leaf in the G_3 execution tree. By the above the simulator does not abort in $D^{\mathrm{G}_3}(\beta)$ (because α is good) so it suffices to argue that all edges of $q_1, \ldots, q_5$ and that all non-$\perp$ entries of r_f are examined during the execution $D^{\mathrm{G}_3}(\beta)$. For r_f this is clear, since all these entries are examined during the execution $D^{\mathrm{G}_2}(\alpha)$. If an edge in q_i corresponds to a non-adapted value of P_i, i.e. to a value of P_i "downloaded" from p_i, then this q_i-edge is clearly examined during the execution $D^{\mathrm{G}_3}(\beta)$, because the corresponding "download" also occurs in the execution $D^{\mathrm{G}_3}(\beta)$. Otherwise, an edge in q_i corresponds to an adapted value of P_i in G_2; by direct inspection of G_2 it is obvious that all adapted edges are on some completed path associated to a previous E or E^{-1} query made by the simulator (and potentially, before that, by the distinguisher); since this E or E^{-1} query will also be made in G_3, the q_i-edge is again, therefore, examined. Thus β

[16] More exactly, for every element of $EQueries$, or for every pair (K, x) such that $ETable[K](x) \neq \perp$.

is the footprint of some good G_3 leaf. Obviously, given that D receives identical answers in $G_2(\alpha)$ and in $G_3(\beta)$, the α-leaf in G_2 has the same label as the β-leaf in G_3.

It finally remains to show that τ is one-to-one on $R_2^{\text{good-f}}$. For this we argue by contradiction. Let α and β be as above, and assume there exists $\alpha^* \in R_2^{\text{good-f}}$, $\alpha^* = (r_f^*, p_1^*, \ldots, p_5^*, p_E^*) \neq \alpha$, such that $\tau(\alpha^*) = \beta = (r_f, q_1, \ldots, q_5)$. Then $r_f^* = r_f$, and because the executions $D^{G_2}(\alpha^*)$ and $D^{G_2}(\alpha)$ are parallel up to the internals of E/E^{-1} queries, since both these executions are parallel to $D^{G_3}(\beta)$. The latter in particular implies that the set of entries of $p_1, \ldots, p_5$ and $p_1^*, \ldots, p_5^*$ that are examined during the two executions are identical (and identically-valued) (no lookups to p_i's occur in E or E^{-1}) so that $p_i = p_i^*$ for $i = 1, \ldots, 5$. However, because the queries made to E and E^{-1} are identical and identically-answered in the two executions $D^{G_2}(\alpha)$ and $D^{G_2}(\alpha^*)$, we also have $p_E = p_E^*$ (indeed, p_E can be reconstructed from the answers to E/E^{-1}-queries), contradicting $\alpha \neq \alpha^*$. □

The next lemma (somewhat similar to Lemma 2) establishes that queries made to E/E^{-1} during a good G_2-execution are in one-to-one correspondence with "adapted" i-queries. Its rather technical proof can be found in Appendix D.

Lemma 4. *Assume that D is a distinguisher that completes all chains, and let $P_1, \ldots, P_5$, Queries, EQueries and f be the global variables in G_2 at the end of a non-aborting (i.e., "good") execution D^{G_2}. Then the number of "adapted" queries in $P_1, \ldots, P_5$ equals the number of distinct queries made to E/E^{-1} by the simulator and the distinguisher combined. More precisely, we have*

$$|\{(i, x, y, dir) \in \mathit{Queries} : dir = \perp\}| = |\mathit{EQueries}|.$$

From here, the randomness mapping argument is rather easy to complete.

Lemma 5. *τ maps good-execution leaves at depth t in the G_2 execution tree to leaves at depth t in the G_3 execution tree (via their associated footprints).*

Proof. Let $\alpha = (r_f, p_1, \ldots, p_5, p_E) \in R_2^{\text{good-f}}$ be the G_2 footprint of some good execution, and let $\beta = \tau(\alpha) = (r_f, q_1, \ldots, q_5)$. Then the depth of α in the G_2 execution tree is easily seen to be

$$|r_f| + |p_1| + |p_2| + |p_3| + |p_4| + |p_5| + \sum_{K \in \{0,1\}^\kappa} |p_E[K]|$$

where $|r_f|$ is the number of non-$\perp$ entries in r_f and where $|p|$ is the number of edges in a partial matching ("partial permutation") p. Likewise, the depth of β in the G_3 execution tree is

$$|r_f| + |q_1| + |q_2| + |q_3| + |q_4| + |q_5|.$$

Since $(q_1, \ldots, q_5)$ are the copies of the tables $(P_1, \ldots, P_5)$ at the end of the G_2 execution $D^{G_2}(\alpha)$, it is therefore sufficient and necessary to show that

$$|P_1| + \cdots + |P_5| = |p_1| + \cdots + |p_5| + \sum_{K \in \{0,1\}^\kappa} |p_E[K]| \tag{5}$$

at the end of the execution $D^{G_2}(\alpha)$. Obviously, every edge in p_i is also in P_i, since edges read from p_i are placed into P_i and never modified (and since α is a footprint, every edge in p_i is read). And the edges in $P_1, \ldots, P_5$ that are not copied from $p_1, \ldots, p_5$ are, precisely, adapted queries. The number of these adapted queries being equal to $\sum_{K \in \{0,1\}^\kappa} |p_E[K]|$ by Lemma 4, this establishes (5), as desired. □

Lemma 6. *Let u and v be leaves of equal depth in, respectively, the G_2 and G_3 execution trees (relative to a fixed q-query distinguisher D). Then*

$$\frac{\Pr_{G_3}[v]}{\Pr_{G_2}[u]} \geq 1 - \frac{81q^4}{2^n}.$$

Proof. By Corollary 3 (Appendix C) any G_2 execution reaches depth at most

$$|r_f| + |p_1| + |p_2| + |p_3| + |p_3| + |p_4| + |p_5| + \sum_{K \in \{0,1\}^\kappa} |p_E[K]| < 8q^2 + q \leq 9q^2,$$

It is easy to see that the arity of a node at depth d in either the G_2 or G_3 execution tree lies in the interval $[2^n - d, 2^n]$. It follows that

$$\frac{\Pr_{\mathrm{G}_3}[v]}{\Pr_{\mathrm{G}_2}[u]} \geq \left(\frac{2^n - 9q^2}{2^n}\right)^{9q^2} \geq 1 - \frac{81q^4}{2^n}$$

as claimed. ◻

Lemma 7. *We have*

$$\Pr[D^{G_2} = 1] - \Pr[D^{G_3} = 1] \leq 160q^{10}/2^n + 81q^4/2^n.$$

Proof. By Lemma 1 we have

$$\begin{aligned}\Pr[D^{\mathrm{G}_2} = 1] &\leq \Pr[D^{\mathrm{G}_2} = 1 \wedge \neg\text{abort}] + \Pr_{\mathrm{G}_2}[\text{abort}] \\ &\leq \Pr[D^{\mathrm{G}_2} = 1 \wedge \neg\text{abort}] + 160q^{10}/2^n.\end{aligned}$$

On the other hand, by Lemmas 3, 5 and 6, we have

$$\Pr[D^{\mathrm{G}_3} = 1] \geq \Pr[D^{\mathrm{G}_2} = 1 \wedge \neg\text{abort}](1 - 81q^4/2^n).$$

Thus

$$\begin{aligned}\Pr[D^{\mathrm{G}_2} = 1] - \Pr[D^{\mathrm{G}_3} = 1] &\leq 160q^{10}/2^n + \Pr[D^{\mathrm{G}_2} = 1 \wedge \neg\text{abort}]81q^4/2^n \\ &\leq 160q^{10}/2^n + 81q^4/2^n\end{aligned}$$

as claimed. ◻

Lemma 8. *We have*

$$\Pr[D^{G_3} = 1] - \Pr[D^{G_4} = 1] \leq 160q^{10}/2^n + 81q^4/2^n.$$

Proof. We first argue that when the distinguisher makes a query to one of the P_i's (or their inverses) in G_3, the answer returned is the corresponding entry in q_i. For this we prove, more generally and by induction on the number of adapted P_i queries created by the simulator, that whenever an entry of P_i is adapted, it is adapted to the corresponding entry of q_i. The latter is clear from the fact that every adapted query is adapted to fit a query to E (possibly E^{-1}) that is itself computed using the relevant entry of q_i; and while the simulator is itself looking up the four other queries in the path using the tables P_i, that might themselves contain previously adapted queries (instead of values directly downloaded from the q_i's), the P_i's hold the same values as the q_i's by the induction hypothesis. Hence the claim.

It follows that Games G_3 and G_4 proceed identically for identical random inputs unless G_3 aborts (G_4 never aborts). Thus, the distinguisher's advantage is upper bounded by the probability that G_3 aborts. By Lemmas 3, 5 and 6, and also by Lemma 19 (Appendix C), we have

$$\begin{aligned}\Pr_{G_3}[\neg\text{abort}] &\geq (1 - 81q^4/2^n)\Pr_{G_2}[\neg\text{abort}]\\ &\geq (1 - 81q^4/2^n)(1 - 160q^{10}/2^n)\\ &\geq 1 - 81q^4/2^n - 160q^{10}/2^n\end{aligned}$$

from which the claim follows. $\square$

Corollary 1. *Let D be a deterministic q-query information-theoretic that completes all paths. Then D has advantage at most*

$$320q^{10}/2^n + 162q^4/2^n \leq 320(q^{10}/2^n + q^4/2^n)$$

at distinguishing games G_1 and G_4.

Proof. This follows directly by Lemmas 7 and 8, as well as by our initial observation that $\Pr[D^{G_2} = 1] \geq \Pr[D^{G_1} = 1]$. $\square$

Corollary 2. *Let D be an arbitrary q-query information-theoretic distinguisher. Then D has advantage at most*

$$320 \cdot 6^{10}(q^{10}/2^n + q^4/2^n)$$

at distinguishing games G_1 and G_4.

Proof. This follows from the previous corollary by fixing the coins of D to their best possible (i.e., to the value that maximizes $\Pr[D^{G_1} = 1] - \Pr[D^{G_4} = 1]$) and by noting that any (arbitrary) q-query deterministic distinguisher D, there exists a $6q$-query deterministic distinguisher D of equal advantage that completes all paths. $\square$

Theorem 3 follows as a corollary of Corollary 2 and of the last two lemmas below, that bound respectively the simulator's query-complexity and running time.

Lemma 9. *The simulator $\mathcal{S}$ makes at most $2q^2$ (distinct) queries to E or E^{-1} in any execution of G_1, assuming interaction with a q-query distinguisher D.*

Proof. Every query to E or E^{-1} in G_1 is followed by a call to TallyEQuery, which specifically enforces the desired upper bound via the query counter $EQnum$. $\square$

Lemma 10. *The simulator $\mathcal{S}$ has total running time $O(q^3)$ in game G_1, assuming interaction with a q-query distinguisher D.*

Proof. Note the most onerous "forall" loops (that iterate, e.g., over $\mathcal{Z} \times \mathcal{Z}$) included in G_2 are excluded in G_1. Also note that $\mathcal{S}$ automatically aborts if $|Queries|$ ever grows more than $6q^2$. From there, it's simply a matter of checking that the simulator's running time is $O(|\mathcal{Z}| \cdot |Queries|)$, which is easy to do by inspection. $\square$

References

1. Armknecht, F., Fleischmann, E., Krause, M., Lee, J., Stam, M., Steinberger, J.: The preimage security of double-block-length compression functions. In: Advances in Cryptology - ASIACRYPT 2011. Lecture Notes in Computer Science, vol. 7073, pp. 233–251. Springer, Heidelberg (2011)
2. Aumasson, J., Henzen, L., Meier, W., Phan, R.: SHA-3 proposal BLAKE (2010), submission to NIST's SHA-3 competition
3. Bellare, M., Ristenpart, T.: Multi-Property-Preserving Hash Domain Extension and the EMD Transform. In: ASIACRYPT 2006. LNCS, vol. 4284, pp. 299–314. Springer-Verlag, Berlin (2006)
4. Bertoni, G., Daemen, J., Peeters, M., Assche, G.: The KECCAK sponge function family (2011), submission to NIST's SHA-3 competition
5. Bertoni, G., Daemen, J., Peeters, M., Van Assche, G.: On the Indifferentiability of the Sponge Construction. In: EUROCRYPT 2008. LNCS, vol. 4965, pp. 181–197. Springer-Verlag, Berlin (2008)
6. Biryukov, A., Dunkelman, O., Keller, N., Khovratovich, D., Shamir, A.: Key Recovery Attacks of Practical Complexity on AES-256 Variants with up to 10 Rounds. In: Gilbert, H. (ed.) EUROCRYPT. LNCS, vol. 6110, pp. 299–319. Springer (2010)
7. Biryukov, A., Khovratovich, D.: Related-Key Cryptanalysis of the Full AES-192 and AES-256. In: Matsui, M. (ed.) ASIACRYPT. LNCS, vol. 5912, pp. 1–18. Springer (2009)
8. Biryukov, A., Khovratovich, D., Nikolic, I.: Distinguisher and Related-Key Attack on the Full AES-256. In: Halevi, S. (ed.) CRYPTO. LNCS, vol. 5677, pp. 231–249. Springer (2009)
9. Black, J.: The Ideal-Cipher Model, Revisited: An Uninstantiable Blockcipher-Based Hash Function. In: FSE 2006. LNCS, vol. 4047, pp. 328–340. Springer-Verlag, Berlin (2006)
10. Black, J., Rogaway, P., Shrimpton, T.: Black-Box Analysis of the Block-Cipher-Based Hash-Function Constructions from PGV. In: CRYPTO 2002. LNCS, vol. 2442, pp. 320–335. Springer-Verlag, Berlin (2002)
11. Bogdanov, A., Khovratovich, D., Rechberger, C.: Biclique Cryptanalysis of the Full AES. In: Lee, D.H., Wang, X. (eds.) ASIACRYPT. LNCS, vol. 7073, pp. 344–371. Springer (2011)
12. Bogdanov, A., Knudsen, L.R., Leander, G., Standaert, F.X., Steinberger, J., Tischhauser, E.: Key-Alternating Ciphers in a Provable Setting: Encryption Using a Small Number of Public Permutations. In: EUROCRYPT 2012. LNCS, vol. 7237, pp. 45–62. Springer-Verlag, Berlin (2012)
13. Brachtl, B., Coppersmith, D., Hyden, M., Matyas, S., Meyer, C., Oseas, J., Pilpel, S., Schilling, M.: Data authentication using modification detection codes based on a public one-way encryption function (March 1990), U.S.Patent No 4.908.861
14. Chang, D., Lee, S., Nandi, M., Yung, M.: Indifferentiable Security Analysis of Popular Hash Functions with Prefix-Free Padding. In: ASIACRYPT 2006. LNCS, vol. 4284, pp. 283–298. Springer-Verlag, Berlin (2006)
15. Coron, J.S., Patarin, J., Seurin, Y.: The Random Oracle Model and the Ideal Cipher Model Are Equivalent. In: CRYPTO 2008. LNCS, vol. 5157, pp. 1–20. Springer-Verlag, Berlin (2008)
16. Coron, J.S., Dodis, Y., Malinaud, C., Puniya, P.: Merkle-Damgård Revisited: How to Construct a Hash Function. In: CRYPTO 2005. LNCS, vol. 3621, pp. 430–448. Springer-Verlag, Berlin (2005)
17. Coron, J.S., Dodis, Y., Mandal, A., Seurin, Y.: A Domain Extender for the Ideal Cipher. In: TCC 2010. LNCS, vol. 5978, pp. 273–289. Springer-Verlag, Berlin (2010)
18. Daemen, J., Govaerts, R., Vandewalle, J.: Correlation Matrices. In: Preneel, B. (ed.) FSE. LNCS, vol. 1008, pp. 275–285. Springer (1994)
19. Daemen, J., Rijmen, V.: The Wide Trail Design Strategy. In: Honary, B. (ed.) IMA Int. Conf. LNCS, vol. 2260, pp. 222–238. Springer (2001)
20. Daemen, J., Rijmen, V.: The Design of Rijndael: AES - The Advanced Encryption Standard. Springer (2002)
21. Damgård, I.: A Design Principle for Hash Functions. In: CRYPTO '89. LNCS, vol. 435, pp. 416–427. Springer-Verlag, Berlin (1990)
22. Demay, G., Gaži, P., Hirt, M., , Maurer, U.: Resource-Restricted Indifferentiability. In: Johansson, T., Nguyen, P.Q. (eds.) EUROCRYPT. Lecture Notes in Computer Science, vol. 7881, pp. 664–683. Springer (2013)
23. Desai, A.: The Security of All-or-Nothing Encryption: Protecting against Exhaustive Key Search. In: CRYPTO 2000. LNCS, vol. 1880, pp. 359–375. Springer-Verlag, Berlin (2000)
24. Dodis, Y., Pietrzak, K., Puniya, P.: A New Mode of Operation for Block Ciphers and Length-Preserving MACs. In: EUROCRYPT. pp. 198–219 (2008)
25. Dodis, Y., Puniya, P.: On the Relation Between the Ideal Cipher and the Random Oracle Models. In: TCC 2006. LNCS, vol. 3876, pp. 184–206. Springer-Verlag, Berlin (2006)
26. Dodis, Y., Reyzin, L., Rivest, R., Shen, E.: Indifferentiability of Permutation-Based Compression Functions and Tree-Based Modes of Operation, with Applications to MD6. In: Fast Software Encryption 2009. LNCS, vol. 5665, pp. 104–121. Springer, Heidelberg (2009)
27. Dodis, Y., Ristenpart, T., Shrimpton, T.: Salvaging Merkle-Damgård for Practical Applications. In: EUROCRYPT 2009. LNCS, vol. 5479, pp. 371–388. Springer-Verlag, Berlin (2009)

28. Dunkelman, O., Keller, N., Shamir, A.: Minimalism in Cryptography: The Even-Mansour Scheme Revisited. In: EUROCRYPT 2012. LNCS, Springer-Verlag, Berlin (2012)
29. Even, S., Mansour, Y.: A Construction of a Cipher from a Single Pseudorandom Permutation. In: ASIACRYPT '91. LNCS, vol. 739, pp. 201–224. Springer-Verlag, Berlin (1991)
30. Ferguson, N., Lucks, S., Schneier, B., Whiting, D., Bellare, M., Kohno, T., Callas, J., Walker, J.: The Skein Hash Function Family (2010), submission to NIST's SHA-3 competition
31. Gauravaram, P., Knudsen, L., Matusiewicz, K., Mendel, F., Rechberger, C., Schläffer, M., Thomsen, S.: Grøstl – a SHA-3 candidate (2011), submission to NIST's SHA-3 competition
32. Goldreich, O., Goldwasser, S., Micali, S.: How to Construct Random Functions. In: 25th Annual Symposium on Foundations of Computer Science, FOCS. pp. 464–479. IEEE Computer Society, West Palm Beach, Florida, USA (1984)
33. Granboulan, L.: Short Signatures in the Random Oracle Model. In: ASIACRYPT 2002. LNCS, vol. 2501, pp. 364–378. Springer-Verlag, Berlin (2002)
34. Handschuh, H., Naccache, D.: SHACAL. Submission to the NESSIE project (2000)
35. Handschuh, H., Naccache, D.: SHACAL : A Family of Block Ciphers. Submission to the NESSIE project (2002)
36. Hirose, S.: Some Plausible Constructions of Double-Block-Length Hash Functions. In: FSE 2006. LNCS, vol. 4047, pp. 210–225. Springer-Verlag, Berlin (2006)
37. Holenstein, T., Künzler, R., Tessaro, S.: The equivalence of the random oracle model and the ideal cipher model, revisited. In: ACM Symposium on Theory of Computing, STOC. pp. 89–98. ACM, San Jose, CA, USA (2011)
38. Impagliazzo, R., Levin, L.A., Luby, M.: Pseudo-random Generation from one-way functions. In: ACM Symposium on Theory of Computing, STOC. pp. 12–24. ACM, Seattle, Washington, USA (1989)
39. Jonsson, J.: An OAEP Variant With a Tight Security Proof. Cryptology ePrint Archive, Report 2002/034 (2002)
40. Kilian, J., Rogaway, P.: How to Protect DES against Exhaustive Key Search (An Analysis of DESX). Journal of Cryptology 14(1), 17–35 (2001)
41. Knudsen, L.: Block Ciphers - The Basics (May 2011), eCRYPT II Summer School on Design and Security of Cryptographic Algorithms and Devices, Invited talk
42. Lai, X., Massey, J.: Hash Function Based on Block Ciphers. In: EUROCRYPT '92. LNCS, vol. 658, pp. 55–70. Springer-Verlag, Berlin (1992)
43. Lampe, R., Seurin, Y.: How to Construct an Ideal Cipher from a Small Set of Public Permutations. Cryptology ePrint Archive, Report 2013/255 (2013)
44. Lee, J., Hong, D.: Collision Resistance of the JH Hash Function. IEEE Transactions on Information Theory 58(3), 1992–1995 (2012)
45. Luby, M., Rackoff, C.: How to construct pseudorandom permutations from pseudorandom functions. SIAM Journal of Computing 17(2), 373–386 (1988)
46. Maurer, U., Renner, R., Holenstein, C.: Indifferentiability, Impossibility Results on Reductions, and Applications to the Random Oracle Methodology. In: TCC 2004. LNCS, vol. 2951, pp. 21–39. Springer-Verlag, Berlin (2004)
47. Merkle, R.: One Way Hash Functions and DES. In: CRYPTO '89. LNCS, vol. 435, pp. 428–446. Springer-Verlag, Berlin (1990)
48. Miles, E., Viola, E.: Substitution-Permutation Networks, Pseudorandom Functions, and Natural Proofs. In: Safavi-Naini, R., Canetti, R. (eds.) CRYPTO. Lecture Notes in Computer Science, vol. 7417, pp. 68–85. Springer (2012)
49. Phan, D.H., Pointcheval, D.: Chosen-Ciphertext Security without Redundancy. In: ASIACRYPT 2003. LNCS, vol. 2894, pp. 1–18. Springer-Verlag, Berlin (2003)
50. Preneel, B., Govaerts, R., Vandewalle, J.: Hash Functions Based on Block Ciphers: A Synthetic Approach. In: CRYPTO '93. LNCS, vol. 773, pp. 368–378. Springer-Verlag, Berlin (1993)
51. Ristenpart, T., Shacham, H., Shrimpton, T.: Careful with Composition: Limitations of the Indifferentiability Framework. In: EUROCRYPT 2011. LNCS, vol. 6632, pp. 487–506. Springer-Verlag, Berlin (2011)
52. Rogaway, P., Steinberger, J.P.: Constructing Cryptographic Hash Functions from Fixed-Key Blockciphers. In: Wagner, D. (ed.) CRYPTO. Lecture Notes in Computer Science, vol. 5157, pp. 433–450. Springer (2008)
53. Rogaway, P., Steinberger, J.P.: Security/Efficiency Tradeoffs for Permutation-Based Hashing. In: EUROCRYPT. pp. 220–236 (2008)
54. Seurin, Y.: Primitives et protocoles cryptographiques à sécurité prouvée. Ph.D. thesis, Université de Versailles Saint-Quentin-en-Yvelines, France (2009)
55. Winternitz, R.S.: A Secure One-Way Hash Function Built from DES. In: IEEE Symposium on Security and Privacy. pp. 88–90. IEEE Computer Society (1984)
56. Wu, H.: The Hash Function JH (2011), submission to NIST's SHA-3 competition

A Some Attacks on Tripwire Simulators

We recall that a simulator has *tripwire* (i, j) if it attempts to complete a chain whenever an adjacency with respect to some previously scheduled key is detected between two queries at positions i and

j, with the j-query coming later, with completions being made recursively. (Here i and j are either adjacent or at opposite ends of the chain; see Section 4.1.) The "naïve" simulator is the simulator with all possible tripwires.

In this appendix we outline some relevant attacks on 3- and 4-round tripwire simulators, where these attacks are valid regardless of the strategy applied by the simulator to complete chains.

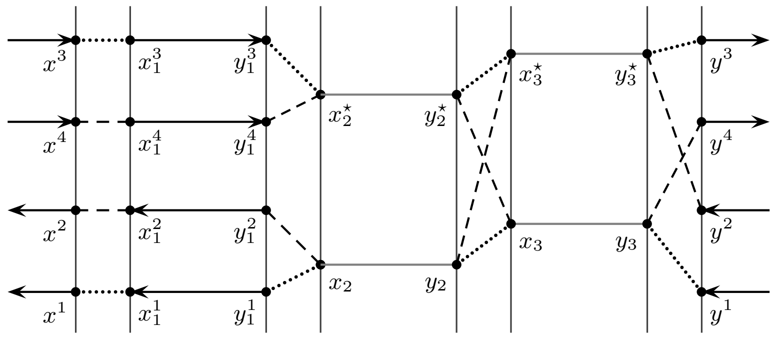

Fig. 2: Attack on the 3-round naïve simulator. Points x and y joined by a dotted line satisfy the relation $x \oplus y = k_1$, whereas points x, y joined by a dashed line satisfy the relation $x \oplus y = k_2$. Moreover, $\mathrm{E}(K_{(i \bmod 2)}, x^i) = y^i$ for $i = 1, 2, 3, 4$.

Attack on the naïve 3-round simulator. The distinguisher D chooses values $x_2 \in \{0,1\}^n$, $K_1, K_2 \in \{0,1\}^\kappa$, $K_1 \neq K_2$, and queries $\mathrm{f}(K_1) \to k_1$, $\mathrm{f}(K_2) \to k_2$. Then D queries $\mathrm{P1}^{-1}(x_2 \oplus k_1) \to x_1^1$, $\mathrm{P1}^{-1}(x_2 \oplus k_2) \to x_1^2$, $\mathrm{E}(K_1, x_1^1 \oplus k_1) \to y^1$, $\mathrm{E}(K_2, x_1^2 \oplus k_2) \to y^2$, $\mathrm{E}^{-1}(K_2, y^1 \oplus k_1 \oplus k_2) \to x^4$, $\mathrm{E}^{-1}(K_1, y^2 \oplus k_1 \oplus k_2) \to x^3$, $\mathrm{P1}(x^3 \oplus k_1) \to y_1^3$, $\mathrm{P1}(x^4 \oplus k_2) \to y_1^4$. We prove below that in the "real world", $y_1^3 \oplus k_1 = y_1^4 \oplus k_2$ with probability 1 (see Figure 2). However, so far the simulator has only answered queries to f, P1 and $\mathrm{P1}^{-1}$, and so has not yet put any thought into its answers. Therefore, the naïve simulator's answers will be inconsistent with the real world with overwhelming probability.

Now to prove the claim above, consider the real world (when D interacts with the actual cipher and its components). Let $y_3^1 = y^1 \oplus k_1$, $y_3^2 = y^2 \oplus k_2$, $x_3^1 = \mathrm{P3}^{-1}(y_3^1)$, $x_3^2 = \mathrm{P3}^{-1}(y_3^2)$, $y_2 = \mathrm{P2}(x_2)$. It is easy to check that $y_2 \oplus k_1 = x_3^1$ and $y_2 \oplus k_2 = x_3^2$. Let $y_2^\star = x_3^1 \oplus k_2 = y_2 \oplus k_1 \oplus k_2 = x_3^2 \oplus k_1$. Then it is also easy to check that $\mathrm{P2}^{-1}(y_2^\star) = y_1^1 \oplus k_1 = y_1^2 \oplus k_2$, whence the claim.

(The above attack, in fact, also works if the four subkeys are scheduled by *independent* random oracles, as can easily be checked.)

Before delving into the details of our 5-round tripwire simulator, we further justify our use of 5 rounds by attacking a few promising-looking 4-round tripwire simulators (which are, therefore, not promising at all in the end). The first of these 4-round simulators is especially relevant because it has an easy termination argument, quite similar to the termination argument for Holenstein et al.'s 14-round simulator [37] and originally developed by Seurin [54] for another simulator (but which is, in our case, ultimately useless because the simulator is insecure):

Attack on the 4-Round Tripwire Simulator With Tripwires $(1,4)$, $(4,1)$, $(2,3)$, $(3,2)$: (See Figure 3.) The distinguisher D chooses $x_3 \in \{0,1\}^n$ and values $K_1, K_2 \in \{0,1\}^\kappa$, $K_1 \neq K_2$. Then D makes the queries $\mathrm{f}(K_1) \to k_1$, $\mathrm{f}(K_2) \to k_2$, $\mathrm{P2}^{-1}(x_3 \oplus k_1) \to x_2^1$, $\mathrm{P2}^{-1}(x_3 \oplus k_2) \to x_2^2$, $\mathrm{P1}^{-1}(x_2^1 \oplus k_1) \to$

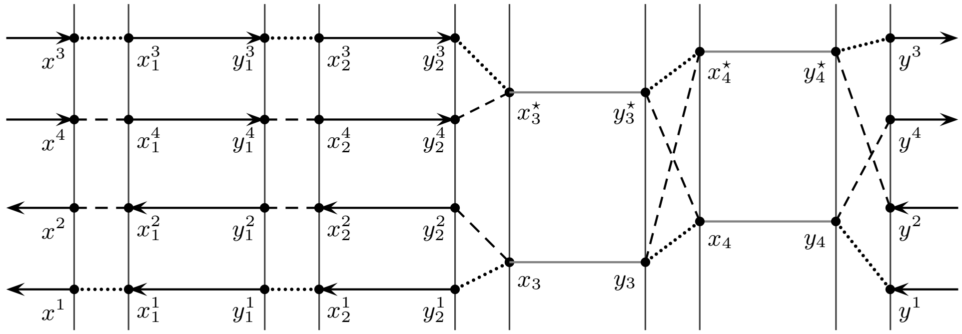

Fig. 3: Attack on the 4-round simulator with tripwires $(1,4)$, $(4,1)$, $(2,3)$, $(3,2)$. Points x and y joined by a dotted line satisfy the relation $x \oplus y = k_1$, whereas points x, y joined by a dashed line satisfy the relation $x \oplus y = k_2$. Moreover, $\mathrm{E}(K_{(i \bmod 2)}, x^i) = y^i$ for $i = 1,2,3,4$.

x_1^1, $\mathrm{P1}^{-1}(x_2^2 \oplus k_2) \to x_1^2$, $\mathrm{E}(K_1, x_1^1 \oplus k_1) \to y^1$, $\mathrm{E}(K_2, x_1^2 \oplus k_2) \to y^2$, $\mathrm{E}^{-1}(K_2, y^1 \oplus k_1 \oplus k_2) \to x^4$, $\mathrm{E}^{-1}(K_1, y^2 \oplus k_2 \oplus k_1) \to x^3$, $\mathrm{P1}(x^3 \oplus k_1) \to y_1^3$, $\mathrm{P1}(x^4 \oplus k_2) \to y_1^4$, $\mathrm{P2}(y_1^3 \oplus k_1) \to y_2^3$, $\mathrm{P2}(y_1^4 \oplus k_2) \to y_2^4$. Finally D checks whether $y_2^3 \oplus k_1 = y_2^4 \oplus k_2$. We prove below that in the real world, this is the case with probability 1; however, no tripwires have been triggered yet, so the simulator has put no thought into its query answers yet, and so the simulator's answers will be inconsistent with the real world with overwhelming probability.

We now prove that $y_2^3 \oplus k_1 = y_2^4 \oplus k_2$ in the real world. Keeping all variables as defined above, further let $y_3 = \mathrm{P3}(x_3)$, $x_4 = y_3 \oplus k_1$, $y_3^\star = x_4 \oplus k_2$, $x_4^\star = y_3^\star \oplus k_1$. Let $y_4 = \mathrm{P4}(y_4)$ and $y_4^\star = \mathrm{P4}(x_4^\star)$. It is easy to check that $y_4 = y^1 \oplus k_1 = y^4 \oplus k_2$ and (because $x_4^\star = y_3 \oplus k_2$) that $y_4^\star = y^2 \oplus k_2 = y^3 \oplus k_1$, where $y^3 = y^2 \oplus k_2 \oplus k_1$ and $y^4 = y^1 \oplus k_1 \oplus k_2$. Thus $y_3^\star = x_4^\star \oplus k_1 = \mathrm{P4}^{-1}(y^3 \oplus k_1) \oplus k_1$ on the one hand, while $y_3^\star = x_4 \oplus k_2 = \mathrm{P4}^{-1}(y^4 \oplus k_2) \oplus k_2$ on the other hand. From this it is easy to see that $y_2^3 \oplus k_1 = \mathrm{P3}^{-1}(y_3^\star)$ and that $y_2^4 \oplus k_2 = \mathrm{P3}^{-1}(y_3^\star)$, completing the claim.

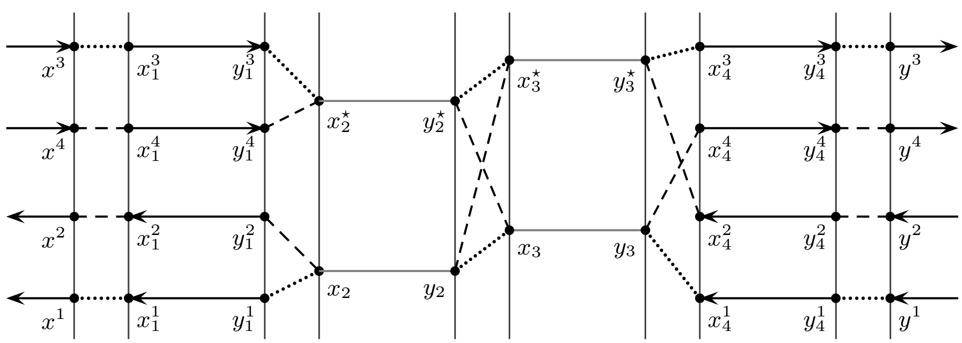

Fig. 4: Attack on the 4-round simulator with tripwires $(1,2)$, $(2,1)$, $(3,4)$, $(4,3)$. Points x and y joined by a dotted line satisfy the relation $x \oplus y = k_1$, whereas points x, y joined by a dashed line satisfy the relation $x \oplus y = k_2$. Moreover, $\mathrm{E}(K_{(i \bmod 2)}, x^i) = y^i$ for $i = 1,2,3,4$.

Attack on the 4-Round Tripwire Simulator With Tripwires $(1,2)$, $(2,1)$, $(3,4)$, $(4,3)$: (See Figure 4.) This is, essentially, the previous attack rotated cyclically to the left by one round. The distinguisher D chooses values x_2 and K_1, K_2, $K_1 \neq K_2$. Then D makes the queries $\mathrm{f}(K_1) \to k_1$, $\mathrm{f}(K_2) \to k_2$, $\mathrm{P1}^{-1}(x_2 \oplus k_1) \to x_1^1$, $\mathrm{P1}^{-1}(x_2 \oplus k_2) \to x_1^2$, $\mathrm{E}(K_1, x_1^1 \oplus k_1) \to y^1$, $\mathrm{E}(K_2, x_1^2 \oplus k_2) \to y^2$, $\mathrm{P4}^{-1}(y^1 \oplus k_1) \to x_4^1$, $\mathrm{P4}^{-1}(y^2 \oplus k_2) \to x_4^2$, $\mathrm{P4}(x_4^1 \oplus k_1 \oplus k_2) \to y_4^4$, $\mathrm{P4}(x_4^2 \oplus k_2 \oplus k_1) \to y_4^3$, $\mathrm{E}^{-1}(K_2, y_4^4 \oplus k_2) \to x^4$, $\mathrm{E}^{-1}(K_1, y_4^3 \oplus k_1) \to x^3$, $\mathrm{P1}(x^4 \oplus k_2) \to y_1^4$, $\mathrm{P1}(x^3 \oplus k_1) \to y_1^3$. Finally, the distinguisher checks that $y_1^4 \oplus k_2 = y_1^3 \oplus k_1$. No tripwires have been set off, and it is easy to check the attack's validity by similar observations as in the last attack (details omitted).

Attack on the 4-Round Tripwire Simulator With Tripwires $(1,2)$, $(2,3)$, $(3,2)$, $(4,3)$: D chooses values x_3 and K_1, K_2, $K_1 \neq K_2$. Then D makes the queries $\mathrm{f}(K_1) \to k_1$, $\mathrm{f}(K_2) \to k_2$, $\mathrm{P2}^{-1}(x_3 \oplus k_1) \to x_2^1$, $\mathrm{P2}^{-1}(x_3 \oplus k_2) \to x_2^2$, $\mathrm{P1}^{-1}(x_2^1 \oplus k_1) \to x_1^1$, $\mathrm{P1}^{-1}(x_2^2 \oplus k_2) \to x_1^2$, $\mathrm{E}(K_1, x_1^1 \oplus k_1) \to y^1$, $\mathrm{E}(K_2, x_1^2 \oplus k_2) \to y^2$, $\mathrm{P4}^{-1}(y^1 \oplus k_1) \to x_4^1$, $\mathrm{P4}^{-1}(y^2 \oplus k_2) \to x_4^2$. Finally, D checks that $x_4^1 \oplus k_1 = x_4^2 \oplus k_2$. No tripwires have been set off, and the attack's validity is easy to check.

Attack on the 4-Round Tripwire Simulator With Tripwires $(2,1)$, $(2,3)$, $(3,2)$, $(3,4)$: D chooses values x_2 and K_1, K_2, $K_1 \neq K_2$. Then D makes the queries $\mathrm{f}(K_1) \to k_1$, $\mathrm{f}(K_2) \to k_2$, $\mathrm{P1}^{-1}(x_2 \oplus k_1) \to x_1^1$, $\mathrm{P1}^{-1}(x_2 \oplus k_2) \to x_1^2$, $\mathrm{E}(K_1, x_1^1 \oplus k_1) \to y^1$, $\mathrm{E}(K_2, x_1^2 \oplus k_2) \to y^2$, $\mathrm{P4}^{-1}(y^1 \oplus k_1) \to x_4^1$, $\mathrm{P4}^{-1}(y^2 \oplus k_2) \to x_4^2$, $\mathrm{P3}^{-1}(x_4^1 \oplus k_1) \to x_3^1$, $\mathrm{P3}^{-1}(x_4^2 \oplus k_2) \to x_3^2$. Finally, D checks that $x_4^1 \oplus k_1 = x_4^2 \oplus k_2$. No tripwires have been set off, and the attack's validity is again easy to check.

Except for the 4-round tripwire simulator with tripwires

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

and its symmetric counterpart, other 4-round, 4-tripwire simulators can easily be attacked by variants of the attacks above (but we omit the case analysis).

B Security Games

See Figures 5–9.

C The Abort Probability of game $\mathbf{G_2}$

The main goal of this appendix is the proof of Lemma 1 (stated in Section 4.3) upper bounding the probability of abortion in game G_2. We start with some basic definitions and notations (some of which are recalled from Section 4). All definitions in this section refer to game G_2.

A *simulator cycle* consists of the execution period starting from when an adversary makes a query to when the adversary receives an answer. (The answer being, possibly, an abort message.) We also count queries to $\mathrm{E}/\mathrm{E}^{-1}$ as "simulator cycles". Hence in G_2 the term "simulator" is somewhat liberally interpreted as "the code implementing the adversary's oracle".

A tuple (i, x, y, dir, num) of the set *Queries* maintained by the simulator is called an *i-query*. We abbreviate i-queries to their first three or four coordinates as convenient. An i-query with $dir = \rightarrow$ (resp. $dir = \leftarrow$) is called *random at the right* (resp., *random at the left*). An i-query with $dir = \perp$ is called *adapted*. A query (i, x, y, dir, num) is *later* than a query (i', x', y', dir', num') if $num > num'$; or *earlier* if $num < num'$.

We recall that $\mathcal{Z}$ is the set of scheduled keys (the image of the table f). Two queries (i, x, y), $(i+1, x', y')$ are *adjacent* if $y \oplus x' \in \mathcal{Z}$; we also say two such queries are *k-adjacent* to mean that $y \oplus x' = k \in \mathcal{Z}$.

```
Game G1 G2

Random tapes: p_1, ..., p_5, {p_E[K] : K ∈ {0,1}^κ}, r_f

private procedure ReadTape(Table, x, p)
    y ← p(x)
    if (Table(x) ≠ ⊥) then abort
    if (Table^{-1}(y) ≠ ⊥) then abort
    Table(x) ← y
    Table^{-1}(y) ← x
    return y

public procedure E(K, x)
    if (ETable[K](x) ≠ ⊥) return ETable[K](x)
    y ← ReadTape(ETable[K], x, p_E[K](→, ·))
    AddEQuery(K, x, y, →) // G2
    return y

public procedure E^{-1}(K, y)
    if (ETable[K]^{-1}(y) ≠ ⊥) return ETable[K]^{-1}(y)
    x ← ReadTape(ETable[K]^{-1}, y, p_E[K](←, ·))
    AddEQuery(K, x, y, ←) // G2
    return x

public procedure f(K)
    if f(K) ≠ ⊥ return f(K)
    k ← r_f(K)
    KeyQueryChecks(k) // G2
    Z ← Z ∪ k
    f(K) ← k
    f^{-1}(k) ← K
    KeyQueries ← KeyQueries ∪ {(K, k, ++qnum)}
    if (qnum > 6q^2 + q) then abort
    return f(K)

public procedure P1(x)
    if (P_1(x) ≠ ⊥) return P_1(x)
    y ← ReadTape(P_1, x, p_1(→, ·))
    AddQuery(1, x, y, →)
    return P_1(x)

public procedure P1^{-1}(y)
    PrivateP1^{-1}(y)
    Cleanup()
    return P_1^{-1}(y)
```

```
Game G1 G2 (continued)

private procedure PrivateP1^{-1}(y)
    if (P_1^{-1}(y) ≠ ⊥) return P_1^{-1}(y)
    x ← ReadTape(P_1^{-1}, y, p_1(←, ·))
    AddQuery(1, x, y, ←)
    FreezeLeftValues(x, ⊥)
    LeftQueue ← LeftQueue ∪ (1^+, y)
    return P_1^{-1}(y)

public procedure P2(x)
    if (P_2(x) ≠ ⊥) return P_2(x)
    y ← ReadTape(P_2, x, p_2(→, ·))
    AddQuery(2, x, y, →)
    return P_2(x)

public procedure P2^{-1}(y)
    if (P_2^{-1}(y) ≠ ⊥) return P_2^{-1}(y)
    x ← ReadTape(P_2^{-1}, y, p_2(←, ·))
    AddQuery(2, x, y, ←)
    return P_2^{-1}(y)

public procedure P3(x)
    PrivateP3(x)
    Cleanup()
    return P_3(x)

private procedure PrivateP3(x)
    if (P_3(x) ≠ ⊥) return P_3(x)
    y ← ForcedP3(3^-, x)
    if (y ≠ ⊥) then
        if (y ∈ range(P_3)) then abort
        P_3(x) ← y
        P_3^{-1}(y) ← x
        AddQuery(3, x, y, ⊥)
        RightQueue ← RightQueue ∪ (3^+, y)
    else
        y ← ReadTape(P_3, x, p_3(→, ·))
        AddQuery(3, x, y, →)
    end if
    LeftQueue ← LeftQueue ∪ (3^-, x)
    return P_3(x)

public procedure P3^{-1}(y)
    PrivateP3^{-1}(y)
    Cleanup()
    return P_3^{-1}(y)
```

Fig. 5: Games G_1 and G_2 (first of four sets of procedures). Red lines (commented with '// G_2') are in game G_2 only. All other lines belong both to G_1 and G_2.

A value x_3 is called *1-2-joined* to a value x_1 if there exists a $k \in \mathcal{Z}$ and 1- and 2-queries $(1, x_1, y_1)$, $(2, x_2, y_2)$ such that $y_1 \oplus x_2 = y_2 \oplus x_3 = k$. (We also say that x_1 and x_3 are *k-1-2-joined*, in this case.) Symmetrically, values y_3 and y_5 are *k-4-5-joined* if there exists 4- and 5-queries $(4, x_4, y_4)$, $(5, x_5, y_5)$ such that $y_3 \oplus x_4 = y_4 \oplus x_5 = k$, for some $k \in \mathcal{Z}$.

A *completed path* consists of a 5-tuple of queries $(1, x_1, y_1), \ldots, (5, x_5, y_5)$ such that there exists a value $k \in \mathcal{Z}$ such that (i, x_i, y_i) is k-adjacent to $(i + 1, x_{i+1}, y_{i+1})$ for $i = \{1, 2, 3, 4\}$, and such that $ETable[f^{-1}(k)](x_1 + k) = y_5 + k$. Such a path is also called a *k-completed path*.

Game G_1 G_2 (continued)

private procedure PrivateP3$^{-1}(y)$
  if $(P_3^{-1}(y) \neq \perp)$ **return** $P_3^{-1}(y)$
  $x \leftarrow$ ForcedP3$(3^+, y)$
  if $(x \neq \perp)$ **then**
    if $(x \in \mathrm{domain}(P_3))$ **then abort**
    $P_3(x) \leftarrow y$
    $P_3^{-1}(y) \leftarrow x$
    AddQuery$(3, x, y, \perp)$
    $\mathit{LeftQueue} \leftarrow \mathit{LeftQueue} \cup (3^-, x)$
  else
    ReadTape$(P_3^{-1}, y, p_3(\leftarrow, \cdot))$
    AddQuery$(3, x, y, \leftarrow)$
  end if
  $\mathit{RightQueue} \leftarrow \mathit{RightQueue} \cup (3^+, y)$
  return $P_3^{-1}(y)$

public procedure P4(x)
  if $(P_4(x) \neq \perp)$ **return** $P_4(x)$
  $y \leftarrow$ ReadTape$(P_4, x, p_4(\rightarrow, \cdot))$
  AddQuery$(4, x, y, \rightarrow)$
  return $P_4(x)$

public procedure P4$^{-1}(y)$
  if $(P_4^{-1}(y) \neq \perp)$ **return** $P_4^{-1}(y)$
  $x \leftarrow$ ReadTape$(P_4^{-1}, y, p_4(\leftarrow, \cdot))$
  AddQuery$(4, x, y, \leftarrow)$
  return $P_4^{-1}(y)$

public procedure P5(x)
  PrivateP5(x)
  Cleanup()
  return $P_5(x)$

private procedure PrivateP5(x)
  if $(P_5(x) \neq \perp)$ **return** $P_5(x)$
  $y \leftarrow$ ReadTape$(P_5, x, p_5(\rightarrow, \cdot))$
  AddQuery$(5, x, y, \rightarrow)$
  FreezeRightValues$(y, \perp)$
  $\mathit{LeftQueue} \leftarrow \mathit{LeftQueue} \cup (5^-, x)$
  return $P_5(x)$

public procedure P5$^{-1}(y)$
  if $(P_5^{-1}(y) \neq \perp)$ **return** $P_5^{-1}(y)$
  $x \leftarrow$ ReadTape$(P_5^{-1}, y, p_5(\leftarrow, \cdot))$
  AddQuery$(5, x, y, \leftarrow)$
  return $P_5^{-1}(y)$

Game G_1 G_2 (continued)

private procedure FreezeLeftValues$(x_1, k^\star)$
  forall $k, k' \in \mathcal{Z}$ **do**
    if $(k \neq k' \wedge x_1 \oplus k \oplus k' \in \mathrm{domain}(P_1))$ **then**
      abort
  end forall // G_2
  forall $k \in \mathcal{Z} \backslash \{k^\star\}$ **do**
    if $(x_1 \oplus k \in \mathit{LeftFreezer})$ **then abort**
    forall $K \in \{0,1\}^\kappa$ **do**
      if $(\mathit{ETable}[K](x_1 \oplus k) \neq \perp)$ **then abort**
    end forall // G_2
    $\mathit{LeftFreezer} \leftarrow \mathit{LeftFreezer} \cup \{x_1 \oplus k\}$
  end forall

private procedure FreezeRightValues$(y_5, k^\star)$
  ... // (symmetric to FreezeLeftValues)

private procedure ForcedP3(i, z)
  if $(i = 3^-)$ **then**
    $x_3 \leftarrow z$
    $\mathit{candidate} \leftarrow \emptyset$
    forall $k \in \mathcal{Z}$ **do**
      if $(x_3 \oplus k \notin \mathrm{range}(P_2))$ **continue**
      $y_1 \leftarrow P_2^{-1}(x_3 \oplus k) \oplus k$
      if $(y_1 \notin \mathrm{range}(P_1))$ **continue**
      $x_1 \leftarrow P_1^{-1}(y_1)$
      if $(x_1 \oplus k \in \mathit{LeftFreezer})$ **continue**
      if $(\mathit{candidate} \neq \emptyset)$ **then abort**
      $\mathit{candidate} \leftarrow (k, x_1 \oplus k)$
    end forall // (k)
    if $(\mathit{candidate} = \emptyset)$ **return** $\perp$
    $(k, x) \leftarrow \mathit{candidate}$
    $y_5 \leftarrow \mathrm{E}(f^{-1}(k), x) \oplus k$
    TallyEQuery$(f^{-1}(k), x, \rightarrow)$
    if $(y_5 \notin \mathrm{range}(P_5))$ **return** $\perp$
    $y_4 \leftarrow P_5^{-1}(y_5) \oplus k$
    return P4$^{-1}(y_4) \oplus k$
  end if
  if $(i = 3^+)$ **then**
    ... // (symmetric to case $(i = 3^-)$)
  end if
  return $\perp$

Fig. 6: Games G_1 and G_2 (second of four sets of procedures).

An *execution* is the start-to-finish interaction of the distinguisher with G_2, including the transcript of all internal G_2 actions. An execution is *good* or *non-aborting* if G_2 does not abort. We assume, like in Section 4.3, a fixed (deterministic) distinguisher D that completes all chains.

A set of values of the form

$$\{x_1 \oplus k : k \in \mathcal{Z}, k \neq k^*\}$$

added to *LeftFreezer* by FreezeLeftValues(x_1, k^*) (where, possibly, $k^* = \perp$) is a *left ice tray*. We define a *right ice tray* similarly. We view ice trays as being "erased" or "removed" at the end of each simulator cycle (similarly to the erasure of *LeftFreezer* and *RightFreezer* at the end of each simulator cycle).

Game G_1 G_2 (continued)

```
private procedure ExistsPath(i, z, k)
   if (i = 1⁺) then
      y₁ ← z
      if (y₁ ∉ range(P₁)) return false
      x₁ ← P1⁻¹(y₁)
      (ℓ, x) ← ProbeForward(2, 5, y₁ ⊕ k, k)
      if (ℓ ≠ 5 ∨ x ∉ domain(P₅)) return false
      if (E(f⁻¹(k), x₁ ⊕ k) ≠ P₅(x) ⊕ k) then abort
      TallyEQuery(f⁻¹(k), x₁ ⊕ k, →)
      return true
   end if
   if (i = 3⁻) then
      x₃ ← z
      (ℓ₁, y) ← ProbeBackward(2, 1, x₃ ⊕ k, k)
      (ℓ₂, x) ← ProbeForward(3, 5, x₃, k)
      if (ℓ₁ ≠ 1 ∨ y ∉ range(P₁)) return false
      if (ℓ₂ ≠ 5 ∨ x ∉ domain(P₅)) return false
      if (E(f⁻¹(k), P₁⁻¹(y) ⊕ k) ≠ P₅(x) ⊕ k) then abort
      TallyEQuery(f⁻¹(k), P₁⁻¹(y) ⊕ k, →)
      return true
   end if
   if (i = 3⁺) then
      ... // (symmetric to case (i = 3⁻))
   end if
   if (i = 5⁻) then
      ... // (symmetric to case (i = 1⁺))
   end if

private procedure ProbeForward(i, j, x_i, k)
   // (i, j ∈ {1, 2, 3, 4, 5}, i < j)
   while i < j do
      if (P_i(x_i) = ⊥) break
      x_i ← P_i(x_i) ⊕ k
      i ← i + 1
   end
   return (i, x_i)

private procedure ProbeBackward(i, j, y_i, k)
   // (i, j ∈ {1, 2, 3, 4, 5}, i > j)
   while i > j do
      if (P_i⁻¹(y_i) = ⊥) break
      y_i ← P_i⁻¹(y_i) ⊕ k
      i ← i − 1
   end
   return (i, y_i)
```

Game G_1 G_2 (continued)

```
private procedure EmptyQueue()
   do
      while ¬LeftQueue.empty()
         (i, z) ← LeftQueue.pop()
         if (i = 1⁺) then ProcessNew1Edge(z)
         if (i = 3⁻) then ProcessNew3⁻Edge(z)
      end while
      while ¬RightQueue.empty()
         (i, z) ← RightQueue.pop()
         if (i = 3⁺) then ProcessNew3⁺Edge(z)
         if (i = 5⁻) then ProcessNew5Edge(z)
      end while
   while (¬LeftQueue.empty())

private procedure ProcessNew1Edge(y₁)
   forall k ∈ Z
      if (ExistsPath(1⁺, y₁, k)) then continue
      if (y₁ ⊕ k ∉ domain(P₂)) then continue
      CompletePath1⁺(y₁, k)
   end forall

private procedure ProcessNew3⁻Edge(x₃)
   forall k ∈ Z
      if (ExistsPath(3⁻, x₃, k)) then continue
      if (x₃ ⊕ k ∉ range(P₂)) then continue
      CompletePath3⁻(x₃, k)
   end forall

private procedure ProcessNew3⁺Edge(y₃)
   ... // (symmetric to ProcessNew3⁻Edge)

private procedure ProcessNew5Edge(x₅)
   ... // (symmetric to ProcessNew1Edge)

private procedure Cleanup()
   EmptyQueue()
   LeftFreezer ← ∅
   RightFreezer ← ∅

private procedure AddQuery(i, x, y, dir)
   QueryChecks(i, x, y, dir) // G₂
   Queries ← Queries ∪ {(i, x, y, dir, ++qnum)}
   if (qnum > 6q² + q) then abort
```

Fig. 7: Games G_1 and G_2 (third of four sets of procedures).

However, for the time of their existence, ice trays are viewed as "static", in the sense that removal of values from *LeftFreezer* or *RightFreezer* does not affect the ice trays containing those values. (The ice trays are defined as sets, and the sets remain unchanged.) We note that by design, all left ice trays created during a simulator cycle are disjoint, and likewise all right ice trays are disjoint, as long as the game does not abort.

Intuitively, an ice tray holds "certified freshly created random values" that should not have been queried before to E (for left ice trays) or E^{-1} (for right ice trays), under any key. Before querying these values to E or E^{-1} (via EmptyQueue()), such values are carefully removed from *LeftFreezer* or

Game G_1 G_2 (continued)

private procedure CompletePath$1^+(y_1, k)$
  $x_1 \leftarrow P_1^{-1}(y_1)$
  $x_3 \leftarrow P_2(y_1 \oplus k) \oplus k$
  $x_4 \leftarrow \mathrm{PrivateP3}(x_3) \oplus k$
  $x_5 \leftarrow \mathrm{P4}(x_4) \oplus k$
  FinishPath$1^+3^-(x_1, x_5, k)$

private procedure CompletePath$3^-(x_3, k)$
  $x_2 \leftarrow P_2^{-1}(x_3 \oplus k)$
  $x_1 \leftarrow \mathrm{PrivateP1}^{-1}(x_2 \oplus k)$
  $x_4 \leftarrow P_3(x_3) \oplus k$
  $x_5 \leftarrow \mathrm{P4}(x_4) \oplus k$
  FinishPath$1^+3^-(x_1, x_5, k)$

private procedure FinishPath$1^+3^-(x_1, x_5, k)$
  if $(x_1 \oplus k \in \mathit{LeftFreezer})$ **then**
    $\mathit{fresh} \leftarrow$ **true**
    $\mathit{LeftFreezer} \leftarrow \mathit{LeftFreezer} \backslash \{x_1 \oplus k\}$
  else
    $\mathit{fresh} \leftarrow$ **false**
  end if
  $y_5 \leftarrow k \oplus \mathrm{E}(f^{-1}(k), x_1 \oplus k)$
  TallyEQuery$(f^{-1}(k), x_1 \oplus k, \rightarrow)$
  if $(x_5 \in \mathrm{domain}(P_5))$ **then abort**
  if $(y_5 \in \mathrm{range}(P_5))$ **then abort**
  $P_5(x_5) \leftarrow y_5$
  $P_5^{-1}(y_5) \leftarrow x_5$
  AddQuery$(5, x_5, y_5, \perp)$
  $\mathit{RightQueue} \leftarrow \mathit{RightQueue} \cup (5^-, x_5)$
  if (fresh) **then**
    FreezeRightValues(y_5, k)
  end if

private procedure CompletePath$3^+(y_3, k)$
  ... // (symmetric to CompletePath3^-)

private procedure CompletePath$5^-(x_5, k)$
  ... // (symmetric to CompletePath1^+)

private procedure FinishPath$5^-3^+(y_5, y_1, k)$
  ... // (symmetric to FinishPath1^+3^-)

private procedure TallyEQuery(K, z, dir)
  if $(\mathit{dir} = \rightarrow)$ **then**
    if $(\mathit{TallyETable}[K](z) = \perp)$ **then** $++\mathit{Eqnum}$
    $\mathit{TallyETable}[K](z) \leftarrow t \leftarrow \mathrm{E}(K, z)$
    $\mathit{TallyETable}[K]^{-1}(t) \leftarrow z$
  end if
  if $(\mathit{dir} = \leftarrow)$ **then**
    ... // (symmetric to case $\mathit{dir} = \rightarrow$)
  end if
  if $(\mathit{Eqnum} > 2q^2)$ **then abort**

Game G_1 G_2 (continued)

private procedure AddEQuery(K, x, y, dir) // G_2
  if $(x \in \mathit{LeftFreezer})$ **then abort**
  if $(y \in \mathit{RightFreezer})$ **then abort**
  if $(\mathit{dir} = \rightarrow)$ **then**
    forall $k, k' \in \mathcal{Z}$, $k \neq k'$
      if $(y \oplus k \in \mathrm{range}(P_5))$ **then abort**
      if $(y \oplus k \oplus k' \in \mathit{RightFreezer})$ **then abort**
      forall $K \in \{0,1\}^\kappa$
        if $\mathit{ETable}[K](y \oplus k \oplus k') \neq \perp$ **then abort**
      end forall
    end forall
  end if
  if $(\mathit{dir} = \leftarrow)$ **then**
    ... // (symmetric to case $\mathit{dir} = \rightarrow$)
  end forall
  $\mathit{EQueries} \leftarrow \mathit{EQueries} \cup \{(K, x, y, \mathit{dir})\}$

private procedure KeyQueryChecks(k) // G_2
  if $(k \in \mathcal{Z})$ **then abort**
  if $\exists\ k_1, k_2, k_3, k_4 \in \mathcal{Z} \cup \{k\}$ **s.t:**
    (i) $k_1 \oplus k_2 \oplus k_3 \oplus k_4 = 0$
    (ii) $(i \neq j) \implies k_i \neq k_j$
    ...then abort
  for $i = 1$ **to** 4 **do**
    if $\exists (x_i, y_i, x_{i+1}, y_{i+1})$ **s.t:**
      (i) $y_i \oplus x_{i+1} \in (k \oplus 4\mathcal{Z}) \cup \{k\}$
      (ii) $(i, x_i, y_i), (i+1, x_{i+1}, y_{i+1}) \in \mathit{Queries}$
      ...then abort
  end for
  for $i = 1$ **to** 5 **do**
    if $\exists (x_i, y_i, x_i', y_i')$ **s.t:**
      (i) $y_i \oplus y_i' \in k \oplus 5\mathcal{Z} \vee x_i \oplus x_i' \in k \oplus 5\mathcal{Z}$
      (ii) $(i, x_i, y_i), (i, x_i', y_i') \in \mathit{Queries}$
      (iii) $(x_i, y_i) \neq (x_i', y_i')$
      ...then abort
  end for

private procedure QueryChecks(i, x, y, dir) // G_2
  if $(\mathit{dir} = \rightarrow \wedge \exists (x', y')$ **s.t:**
    (i) $(i+1, x', y') \in \mathit{Queries}$
    (ii) $y \oplus x' \in 5\mathcal{Z}$
    ...) **then abort**
  if $(\mathit{dir} = \leftarrow \wedge \exists (x', y')$ **s.t:**
    (i) $(i-1, x', y') \in \mathit{Queries}$
    (ii) $x \oplus y' \in 5\mathcal{Z}$
    ...) **then abort**
  if $(\mathit{dir} = \rightarrow \wedge \exists (x', y')$ **s.t:**
    (i) $(i, x', y') \in \mathit{Queries}$
    (ii) $y \oplus y' \in 6\mathcal{Z}$
    ...) **then abort**
  if $(\mathit{dir} = \leftarrow \wedge \exists (x', y')$ **s.t:**
    (i) $(i, x', y') \in \mathit{Queries}$
    (ii) $x \oplus x' \in 6\mathcal{Z}$
    ...) **then abort**

Fig. 8: Games G_1 and G_2 (fourth of four sets of procedures). Here, $m\mathcal{Z}$ denotes the m-fold direct $\oplus$-sum $\mathcal{Z} \oplus \cdots \oplus \mathcal{Z}$ of $\mathcal{Z}$.

Game G_3

Random tapes: $q_1, \ldots, q_5, r_f$

public procedure $\mathrm{E}(K, x)$
  $k \leftarrow r_f[K]$
  $x_1 \leftarrow x \oplus k$
  for $i = 1$ **to** $i = 4$
    $y_i \leftarrow q_i(\rightarrow, x_i)$
    $x_{i+1} \leftarrow y_i \oplus k$
  end for
  return $q_5(\rightarrow, x_5) \oplus k$

public procedure $\mathrm{E}^{-1}(K, y)$
  $k \leftarrow r_f[K]$
  $y_5 \leftarrow y \oplus k$
  for $i = 5$ **to** $i = 2$
    $x_i \leftarrow q_i(\leftarrow, y_i)$
    $y_{i-1} \leftarrow x_i \oplus k$
  end for
  return $q_5(\leftarrow, y_1) \oplus k$

Game G_4

Random tapes: $q_1, \ldots, q_5, r_f$

public procedure $\mathrm{E}(K, x)$
  ... // (see G_3)

public procedure $\mathrm{E}^{-1}(K, y)$
  ... // (see G_3)

public procedure $\mathrm{f}(K)$
  return $r_f[K]$

public procedure $\mathrm{P1}(x)$
  return $q_1(\rightarrow, x)$

public procedure $\mathrm{P1}^{-1}(y)$
  return $q_1(\leftarrow, y)$

public procedure $\mathrm{P2}(x)$
  return $q_2(\rightarrow, x)$

public procedure $\mathrm{P2}^{-1}(y)$
  return $q_2(\leftarrow, y)$

public procedure $\mathrm{P3}(x)$
  return $q_3(\rightarrow, x)$

public procedure $\mathrm{P3}^{-1}(y)$
  return $q_3(\leftarrow, y)$

public procedure $\mathrm{P4}(x)$
  return $q_4(\rightarrow, x)$

public procedure $\mathrm{P4}^{-1}(y)$
  return $q_4(\leftarrow, y)$

public procedure $\mathrm{P5}(x)$
  return $q_5(\rightarrow, x)$

public procedure $\mathrm{P5}^{-1}(y)$
  return $q_5(\leftarrow, y)$

Fig. 9: Left: the functions E and E^{-1} in game G_3 (and renamed randomness for G_3); the simulator-related procedures of G_3 are the as in G_2 (Figures 5-8). Right: game G_4.

RightFreezer, such as not to cause abortion. On the other hand we will eventually have to check that calls to E and E^{-1} that occur in ExistsPath (the only procedure in which "unguarded" calls to E, E^{-1} occur) don't call E or E^{-1} on a value in *LeftFreezer* or *RightFreezer* respectively. (See Proposition 4 far below.)

Invariants. As should be obvious from the functions QueryChecks() and KeyQueryChecks(), among others, the simulator of game G_2 is a rather "paranoid" simulator that aborts at the first sign of trouble. This propensity for abortion means good executions are endowed with a certain structure. We list here five basic invariants (denoted as Inv0 through Inv4) that hold at any point of *any* execution (good or not) as well as an extra invariant Inv5 that holds at the end of each simulator cycle as long as the simulator has not aborted (Inv5 does not, in general, hold within a simulator cycle); the proof these invariants hold under the conditions stated is given right below.

Inv0. 2-queries and 4-queries (i.e., queries of the form (j, x, y, dir, num) with $j \in \{2, 4\}$) have $dir \in \{\leftarrow, \rightarrow\}$.

Inv1. There does not exist a pair of distinct key queries (K, k, num), (K', k', num') such that $k = k'$, nor a 4-tuple of distinct key queries (K_1, k_1, num_1), (K_2, k_2, num_2), (K_3, k_3, num_3), (K_4, k_4, num_4) such that $k_1 \oplus k_2 \oplus k_3 \oplus k_4 = 0$.

Inv2. There does not exist a pair of queries (i, x, y, dir, num), $(i+1, x', y', dir', num')$ such that either (i) $y \oplus x' \in 5\mathcal{Z}$, $dir = \rightarrow$ and $num > num'$, or (ii) $y \oplus x' \in 5\mathcal{Z}$, $dir' = \leftarrow$ and $num' > num$.

Inv3. There does not exist a pair of queries (i, x, y, dir, num), (i, x', y', dir', num') such that either: (i) $y \oplus y' \in 6\mathcal{Z}$, $dir = \rightarrow$ and $num > num'$, or (ii) $x \oplus x' \in 6\mathcal{Z}$, $dir = \leftarrow$ and $num > num'$. [Note this precludes the case $y \oplus y' \in 6\mathcal{Z}$, $dir = dir' = \rightarrow$ as well as the case $x \oplus x' \in 6\mathcal{Z}$, $dir = dir' = \leftarrow$.]

Inv4. If $(K, x, y, dir) \in \mathit{EQueries}$ for some $K \in \{0,1\}^\kappa$ then $x \notin \mathit{LeftFreezer}$ and $y \notin \mathit{RightFreezer}$.

Inv5. The following holds for $(i, j) \in \{(2,1), (2,3), (4,3), (4,5)\}$: if (i, x_i, y_i, dir, num) and $(j, x_j, y_j, dir', num')$ are two k-adjacent queries such that $num' > num$, then these two queries are part of the same k-completed path.

The invariants that we will most frequently appeal to are invariants Inv2, Inv3 and Inv5. In order to help the reader memorize "which is which" we note that Inv2 concerns the interaction of i-and $(i+1)$-queries whereas Inv3 concerns the interaction of i-queries amongst themselves. Inv5 concerns itself with path completion, and states that the tripwires are really "doing their job".

Lemma 11. *Invariants Inv0–Inv4 hold throughout any execution. Invariant Inv5 holds at the end of each simulator cycle as long as abort has not occurred.*

Proof. We consider each invariant separately:

Inv0: Each time a 2-query or 4-query is created (which occurs only in the procedures P2, P2^{-1}, P4, P4^{-1}), a call to ReadTape occurs and the query is created with direction $\leftarrow$ or $\rightarrow$. Moreover, query directions are never modified after creation.

Inv1: This follows from the first two "ifs" in the procedure KeyQueryChecks (the latter procedure is called before each new key scheduling).

Inv2: This follows from the first two "if" blocks in the procedure QueryChecks (this procedure being called each time a new query is created) and by the first "for" block of KeyQueryChecks. (We note, concerning this for-block, that $k \notin k \oplus 4\mathcal{Z} = \emptyset$ if $\mathcal{Z} = \emptyset$. The crucial point, here, is that $(k \oplus 4\mathcal{Z}) \cup \{k\} \supseteq (5(\mathcal{Z} \cup \{k\})) \backslash 5\mathcal{Z}$. The latter inclusion is easy to check by considering separately the cases $\mathcal{Z} = \emptyset$ and $\mathcal{Z} \neq \emptyset$.)

Inv3: This follows from the second two "if" blocks in QueryChecks and by the second "for" block in KeyQueryChecks. Here we note that $k \oplus 5\mathcal{Z}$ contains all the elements of $6(\mathcal{Z} \cup \{k\}) \backslash 6\mathcal{Z}$ *except* potentially for the element 0, but 0 has no effect since the fact that P_i is a permutation implies $x_i \neq x'_i$ and $y_i \neq y'_i$ for all distinct i-queries (i, x_i, y_i), (i, x'_i, y'_i).

Inv4: We note that values are added to *LeftFreezer*, *RightFreezer* only in procedures FreezeLeftValues, FreezeRightValues, whereas new tuples are added to *EQueries* only from within AddEQuery. Thus, Inv4 follows because of the abort condition in the innermost "forall" loop in FreezeLeftValues, FreezeRightValues, and by the two first abort conditions in AddEQuery.

Inv5: We firstly note that when a new key is scheduled (i.e., when a query to f occurs) the newly scheduled subkey never causes two previously non-adjacent i-, $(i+1)$-queries to become adjacent, by the first "for" block in KeyQueryChecks. Thus, by inspection of the procedure EmptyQueue, it is sufficient to show, by symmetry, that (i) whenever a new 1-query $(1, x_1, y_1, dir, num)$ is created such that there exists an earlier 2-query $(2, x_2, y_2, dir', num') \in \mathit{Queries}$ with $y_1 \oplus x_2 \in \mathcal{Z}$ ($\mathcal{Z}$, here, referring to the set of subkeys already scheduled when the 1-query is created) then $(1^+, y_1)$ is added to *LeftQueue*, and (ii) whenever a new 3-query $(3, x_3, y_3, num, dir)$ is created such that there exists an earlier 2-query $(2, x_2, y_2, dir', num') \in Queries$ with $y_2 \oplus x_3 \in \mathcal{Z}$ (likewise, $\mathcal{Z}$ at the moment the 3-query is created) then $(3^-, x_3)$ is added to *LeftQueue*. To prove (i), we note that 1-queries are only created within the procedures P1, PrivateP1^{-1} and FinishPath5$^-$3$^+$ (for the latter, see the corresponding 5-query creation in FinishPath1$^+$3$^-$). For PrivateP1^{-1} or FinishPath5$^-$3$^+$ the claim is obvious, since the 1-query is automatically added to *LeftQueue* regardless of anything else. For P1, the new 1-query is not added to *LeftQueue*, but this 1-query has direction $\rightarrow$ and causes abortion if it is k-adjacent to a previous 2-query. The analysis for (ii) is similar: if a new 3-query has direction $\rightarrow$ or $\perp$ it's automatically added to *LeftQueue*, whereas if it has direction $\leftarrow$ and it is adjacent to a previous 2-query it causes abort. □

The Bipartite Graphs B_2, B_4 and the Graph B

Our next goal is to establish further structural properties of good executions. In particular, we wish to show that certain sets never grow beyond certain bounds in good executions of G_2. The relevant metrics, however, rely on more definitions to be stated.

In particular, we will define an edge-labeled graph B based on the sets $Queries$ and $KeyQueries$ (or more particularly on $Queries$ and on $\mathcal{Z}$) which conveniently encodes the (most relevant) information from these two sets. Two smaller disjoint graphs B_2 and B_4 lie at the heart of B, and we describe these first. We emphasize that B as well as B_2 and B_4 are time-dependent graphs whose contents depend on the state of $Queries$ and $\mathcal{Z}$.

The graphs B_2 and B_4 are bipartite graphs each of whose two shores is $\{0,1\}^n$. B_2 depends on $\mathcal{Z}$ and on the set of 2-queries, while B_4 depends on $\mathcal{Z}$ and on the set of 4-queries. For concreteness we describe B_2. The definition for B_4 is entirely analogous.

Edges of B_2 are directed and labeled, and constructed as follows: for every 2-query $(2, x, y, dir)$ and every $k \in \mathcal{Z}$ we construct an edge $(x \oplus k, y \oplus k)$ of label k and of direction dir (i.e., from $x \oplus k$ to $y \oplus k$ if $dir = \rightarrow$, the reverse otherwise; by Inv0 this covers all cases). This constitutes all edges of B_2. Thus each edge of B_2 is associated to a pair comprised of one 2-query and of one element of $\mathcal{Z}$. We note that two distinct such pairs cannot give rise to two edges of B_2 with the same endpoints by Inv3. (More precisely, if the key value is the same for each pair this is obvious from the fact that P_2 is a permutation. If the key values are different are the 2-queries are the same, the endpoints are obviously different. Finally if the key values are different and the 2-queries are also different, both endpoints cannot be the same by Inv0 and Inv3.) Hence, B_2 contains no multiple edges. In particular we can associate a unique 2-query to every pair of nodes joined by an edge in B_2, in addition to a unique key value. We also note that edges are added to B_2 not only when new queries are created but also when a value is added to $\mathcal{Z}$.

For every 2-query $(2, x, y)$, the set of B_2-edges

$$\{(x \oplus k, y \oplus k) : k \in \mathcal{Z}\}$$

is called a *parallel set* of edges, and two edges in this set are called *parallel*[17]. Since $\mathcal{Z}$ is a set (as opposed to a multiset[18]) parallel edges have distinct endpoints at either end. We also note that if (x^*, y^*) is an edge of B_2 of label k^*, then

$$\{(x^* \oplus k^* \oplus k, y^* \oplus k^* \oplus k) : k \in \mathcal{Z}\} \tag{6}$$

[17] As a technical comment, we note that if (x, y), (x', y') are parallel edges of B_2 then $x \oplus x' = y \oplus y'$, but the latter condition is not equivalent to (i.e., does not imply) parallelness. Indeed, two distinct 2-queries $(2, x_2, y_2)$, $(2, x_2', y_2')$ might have $x_2 \oplus x_2' = y_2 \oplus y_2'$ without contradicting any of the invariants (even if the existence of such queries is unlikely). Edges are parallel, therefore, if and only if they arise from the same 2-query.

[18] Anyway, the same element is never added twice to $\mathcal{Z}$; see Inv1.

is the set of parallel edges to (x^*, y^*).

Invariants Inv0–Inv3 imply a number of nice structural properties of B_2 and B_4, some of which are collected in the next four lemmas. We emphasize these lemmas state properties that hold at *any* point in *any* execution.

Lemma 12. *Connected components of B_j, $j \in \{2, 4\}$ are directed trees with edges directed away from the root. Moreover, the num*[19] *values on the edges of any (necessarily directed) path in B_j are strictly increasing.*

Proof. By the bracketed remark in Inv3 every vertex of B_j has at most one incoming edge. Next, because queries are totally ordered and edges corresponding to the same query (i.e., parallel edges) don't share endpoints, two adjacent edges in B_j have different *num* values and, by Inv3, these *num* go from smaller to larger according to the edge directions. This in particular implies that B_j has no cycles and, together with the fact that every vertex has indegree at most 1, that connected components of B_j are directed trees. □

Lemma 13. *Two parallel edges of B_j, $j \in \{2, 4\}$, cannot lie on the same directed path. In fact, two parallel edges cannot lie in the same connected component of B_j.*

Proof. The first claim is immediate by Lemma 12. For the second claim, let T be a connected component of B_j, so that T is a tree with edges directed away from its root. Say that two nodes u, v of T are *related* if $u \oplus v \in \mathcal{Z} \oplus \mathcal{Z}$, $u \neq v$, and u and v are either both in an odd level or both in an even level of T (so that u and v are in the same shore of B_j). Then: (a) T cannot contain a pair of related nodes u, v such that u and v are the heads of edges with different values of *num*, as this would contradict Inv3, and: (b) T cannot contain a pair of related nodes u, v such that u is the root of T, because then v is at the head of some edge whose *num* value is greater than the *num* value of the edge leaving u on the path towards v, also contradicting Inv3, given that u and v are in the same shore. But if T contains two parallel edges, then the sources u, v of these edges are related vertices, being in the same shore; in order not to contradict (a) and (b), u and v must be the *heads* of another pair of parallel edges in T (i.e., edges with equal values of *num*), which allows to find a new pair of related vertices higher up in the tree; since this regression cannot go on forever (one reaches the root eventually), one concludes that T cannot contain two parallel edges. □

Lemma 14. *Connected components of B_j containing at least one edge never subsequently merge (i.e., become part of the same connected component of B_2 or B_4; for $j = 2, 4$). More precisely, the root of a connected component of B_j remains forever the root of that component.*

Proof. Fix a point in the execution, and let (x, y) be an edge of B_j of label k such that (say) x has no incoming edges (in particular, the edge (x, y) has direction $\rightarrow$) at that point. Assume by contradiction that, subsequently, a new edge (x, y') of direction $\leftarrow$ appears in B_j. The appearance of the new edge (x, y') can be caused either by the addition of an element to $\mathcal{Z}$ or by the creation of a new 2-query. In the latter case the new 2-query has *num* value greater than the 2-query associated to (x, y), a contradiction to Inv3. In the former case, let k' be the newly scheduled subkey and let $(j, x_j, y_j, dir_j, num_j)$, $(j, x'_j, y'_j, dir'_j, num'_j)$ be the j-queries respectively associated to the edges (x, y), (x, y') of B_j. Then $x \oplus k = x_j$, $x \oplus k' = x'_j$, so $x_j \oplus x'_j = k \oplus k' \subseteq k' \oplus 5(\mathcal{Z} \backslash \{k'\})$, which is a contradiction because it implies the second "for" block of KeyQueryChecks would have aborted before k' was added to $\mathcal{Z}$. □

[19] Since each edge of B_j is associated to a unique j-query we can unambiguously speak of the "*num*" value of edges in B_j as well as of "earlier" and "later" edges, etc.

Here we pause to share some intuition regarding the next lemma, which is maybe a bit more mysterious than the previous three. For a change, we give the intuition in terms of the graph B_4 (this also fits better with later parts in the proof). We have already noted (Lemma 13) that connected components of B_2, B_4 only contain at most one edge each from every set of parallel edges. On the other hand, note that if (x^1, y^1), (x^2, y^2) are two distinct edges in B_4 with $x^1 = x^2$ and with associated 4-queries $(4, x_4^1, y_4^1)$, $(4, x_4^2, y_4^2)$ and labels k_1, k_2 then there exists a different component (different component from the one containing the vertex $x^1 = x^2 = x_4^1 \oplus k_1 = x_4^2 \oplus k_2$, that is) of B_4 that contains a parallel copy of both of these edges; namely the B_4 component containing the two edges

$$(x_4^1 \oplus k_2, y_4^1 \oplus k_2), (x_4^1 \oplus k_1, y_4^2 \oplus k_1)$$

which indeed lie the same component because $x_4^1 \oplus k_2 = x_4^2 \oplus k_1$. Hence, every adjacency between two edges in B_4 (or B_2) is "replicated somewhere else" inside B_4 (resp. B_2), in the sense that there exist parallel copies of these two edges that are also adjacent. The content of the following lemma is essentially that a 3-wise adjacency (three edges meeting at a vertex) is, by contrast, never "replicated somewhere else" and even, more strongly, that no connected component of B_4 contains a parallel copy of all three edges involved in a 3-wise adjacency.

Lemma 15. *Let $(4, x_4^1, y_4^1)$, $(4, x_4^2, y_4^2)$, $(4, x_4^3, y_4^3)$ be three distinct 4-queries such that there exist values $k_1, k_2, k_3 \in \mathcal{Z}$ with $x_4^1 \oplus k_1 = x_4^2 \oplus k_2 = x_4^3 \oplus k_3$. Then no component of B_4 contains edges parallel to each of these three 4-queries, except for the component containing the vertex $u := x_4^1 \oplus k_1$. The symmetric statement also applies to the graph B_2.*

Proof. Let T be a connected component of B_4 purportedly containing parallel copies of the 4-queries $(4, x_4^1, y_4^1)$, $(4, x_4^2, y_4^2)$, $(4, x_4^3, y_4^3)$, but not containing u. Let ℓ_1, ℓ_2, $\ell_3 \in \mathcal{Z}$ be the edge labels on these parallel copies—i.e., $(x_4^i \oplus \ell_i, y_4^i \oplus \ell_i)$ is an edge of label ℓ_i in T for $i = 1, 2, 3$. Then $\ell_i \neq k_i$ for all i since otherwise T would contain u, and $\ell_i \neq \ell_j$ for $i \neq j$ since $x_4^i \neq x_4^j$. (Indeed, $x_4^i = x_4^j$ would also imply $y_4^i = y_4^j$, as the 4-queries represent a partial permutation.) One can similarly observe that $k_i \neq k_j$ for $i \neq j$.

If $x_4^1 \oplus \ell_1 = x_4^2 \oplus \ell_2 = x_4^3 \oplus \ell_3$ then

$$\ell_i \oplus \ell_j = k_i \oplus k_j$$

for all $i, j \in \{1, 2, 3\}$, $i \neq j$. The second half of Inv1 and the fact that $\ell_i \neq k_i$, $\ell_i \neq \ell_j$ then implies $\ell_i = k_j$; but since $\ell_i = k_j$ for two different j's, this contradicts the distinctness of k_1, k_2, k_3. We can thus assume without loss of generality that $v_1 \neq v_2$, where $v_1 := x_4^1 \oplus \ell_1$, $v_2 = x_4^2 \oplus \ell_2$.

If neither v_1 nor v_2 is the root of T then since $v_1 \oplus v_2 = \ell_1 \oplus \ell_2 \oplus k_1 \oplus k_2 \in 4\mathcal{Z}$, we obtain a contradiction to Inv3 by considering the two 4-queries associated to the 4-edges whose heads are v_1 and v_2. (These two 4-queries have different values of *num* by Lemma 13.) On the other hand, a similar contradiction to Inv3 is obtained if v_1 is the root by considering the 4-edge leaving v_1 in direction of v_2, and the 4-edge arriving at v_2. (These latter two 4-edges are distinct, since v_1 and v_2 must either both be in even layers, or both be in odd layers of T.) $\square$

We now discuss the graph B, which is a common extension of the graphs B_2 and B_4 of vertex set $(\{0,1\}^n)^6$. Basically, B is obtained by "gluing" together B_2 and B_4 with queries of the form $(3, x, y)$, and by gluing extra pendant edges to the left shore of B_2 (one for every query of the form $(1, x, y)$) and to the right shore of B_4 (one for every query of the form $(5, x, y)$)[20]. The additional glued edges are *unlabeled*, by contrast to the edges of B_2 and B_4.

[20] At this juncture we warn about a potential source of confusion: we often refer to a generic edge of B_2 as "(x, y)" when the left shore of B_2 actually corresponds to "y_1 values" (outputs of P_1) and the right shore corresponds to "x_3 values"

More precisely, B has six "shores" equal to $\{0,1\}^n$; a copy of B_2 is placed between shores 2 and 3, whereas a copy of B_4 is placed between shores 4 and 5. For $i = 1,3,5$, a query (i,x,y) becomes a (possibly directed) unlabeled edge from node x in shore i to node y in shore $i+1$. Edge directions are ascribed the natural way; for example, query $(1,x,y,dir,num)$ becomes an edge directed from shore 1 to shore 2 if $dir = \rightarrow$, and so on; queries with $dir = \perp$ become undirected edges. Edges also inherit the "timeline" of their associated queries, so that we can speak of "earlier" and "later" edges of B. (As we know, not all edges of B are comparable this way, since some edges of B_2 and B_4 correspond to the same query.)

We note that every query of the form $(1,x,y)$, $(3,x,y)$ or $(5,x,y)$ corresponds to exactly one edge in B, by contrast to queries of the form $(2,x,y)$ and $(4,x,y)$ that correspond to $|\mathcal{Z}|$ edges in B. We also note that a *k-completed path* of queries corresponds to a path of 5 edges in B, starting in shore 1 and ending on shore 6, such that both labeled edges are labeled by k, and such that the endpoints of the path are compatible with $E_{f^{-1}(k)}$. (The labeled edges being the residents of B_2 and B_4; for convenience, we will identify B_2 and B_4 with their copies in B.) Moreover, every edge of B_2 and B_4 is in at most one k-completed path, because of its label.

For shorthand, an *i-edge* of the graph B is an edge between shores i and $i+1$; thus i-edges correspond to i-queries. However, this correspondence is only 1-to-1 for $i = 1,3,5$. We will occasionally, for the sake of convenience, confuse the notion of "i-edge" and "i-query" for $i = 1,3,5$, but we will carefully keep the two notions separate for $i = 2,4$. We note that the set of all i-edges forms a partial matching between shores i and $i+1$ for $i = 1,3,5$ (because these edges encode the partially defined permutations P_1, P_3 and P_5). In particular, every node in shore 2 is adjacent to at most one 1-edge, etc.

The notion of "1-2-join" translates as follows into the graph B: a value x_3 in shore 3 is "k-1-2-joined" to a value x_1 in shore 1 if and only if there is a path of length two (edge directions ignored) from x_1 to x_3 where the 2-edge of the path has label k. Symmetrically for "4-5-join".

Invariant Inv5 above translates as follows into the graph language: for $(i,j) \in \{(2,1),(2,3),(4,3),(4,5)\}$: if an i-edge and j-edge are adjacent, but not in a completed path, the j-edge must be earlier than the i-edge. Moreover, the i-edge must therefore be directed "away" from the j-edge, in order not to contradict Inv3.

Invariants Inv2 and Inv3 above imply, in particular, the following: if an edge e of B is adjacent to a $\{\leftarrow,\rightarrow\}$-directed edge e' of B, such that e and e' are adjacent at the "arrow end" of e' (i.e., the "head" of e'), then e must be a later edge than e'.

We note that each tuple added to *LeftQueue* and *RightQueue* is associated to a unique 1-edge, 3-edge or 5-edge in the obvious way. We will therefore sometimes speak of a 1-edge (or 3-edge, etc) being "added to *LeftQueue*" or "popped from *LeftQueue*" with the obvious intent.

To re-emphasize, and for future reference, the left and right shores of B_2 are shores 2 and 3 of B while the left and right shores of B_4 are shores 4 and 5 of B.

Pebbling. A node in shore 2, 3, 4 or 5 of B that is adjacent to a 1-edge, 3-edge or 5-edge is said to be *pebbled*. In this case we also say the node is *pebbled by* the relevant 1-, 3- or 5-edge. Note the edge pebbling a node necessarily unique. One can also note that nodes in shore 2 can only be pebbled by 1-edges, that nodes in shores 3 and 4 can only be pebbled 3-edges, and that nodes in shore 5 can only be pebbled by 5-edges.

The following property of pebbling plays a crucial role in our proof:

(inputs of P_3). (Thus x refers to a y_1-value while y refers to an x_3-value.) Similarly the left and right shores of B_4 correspond to y_3- and x_5-values respectively, even though an edge of B_4 is generically called "(x,y)". Along the same lines, inputs and outputs to the cipher E are generically labeled x, y when more logical names might be y_0 and x_6, instead.

Lemma 16. *At any moment of execution such that Inv5 holds the connected components of B_2 and B_4 are "pebbled upwards": if a node in a connected component of B_j is pebbled, then so is the parent of that node (recall the connected component is a directed tree by Lemma 12). In particular, if any node of a component is pebbled, then so is the root.*

Proof. Let x_3 be, say, a pebbled node in shore 3 such that there exists a 2-query of the form $(2, y_1 \oplus k, x_3 \oplus k, \rightarrow, num)$ for some k, num; in other words, the vertex x_3 has parent y_1 in B_2 (via an edge of label k). Because of the direction $y_1 \rightarrow x_3$ of the edge (y_1, x_3) in B_2, the 3-edge adjacent to x_3 must be a later edge than the edge $(y_1, x_3) \in B_2$ in order not to contradict Inv3; which means by Inv5 that there exists a completed path containing that edge and the edge $(y_1, x_3) \in B_2$. But the presence of a path means y_1 is pebbled by a 1-edge. Other cases are symmetric. □

Live trees. We write $B_j(v)$ for the connected component containing a node v in B_j. We will be especially interested in the size of the largest connected component of B_j formed by *non-pebbled* nodes. We clarify this with a definition.

Fix a moment in the execution when Inv5 holds. Let v be a non-pebbled vertex of B_j. Then $B_j(v)$ is a directed tree with edges directed away from the root. Let u be the highest non-pebbled node in $B_j(v)$ above v (possibly $u = v$). Then no nodes beneath u are pebbled (or else u would already be pebbled). If u is not the root, let $\overline{u}$ be the parent of u in $B_j(v)$. Let $B_j(v)^-$ be obtained from $B_j(v)$ by removing all nodes whose (undirected) path to v passes through $\overline{u}$, if $\overline{u}$ exists, but not removing $\overline{u}$ itself. We define the *live tree anchored at* v $\mathrm{Li}(v)$ to be the tree obtained by "dangling" $B_j(v)^-$ by v, i.e., making v the new root of the tree. One can note that $\mathrm{Li}(v)$ is no longer a directed tree in the sense of having all edges directed towards/away from the root (in fact, edge directions will no longer be important to us in $\mathrm{Li}(v)$), and that at most one node of $\mathrm{Li}(v)$ is pebbled, namely $\overline{u}$, which, if present, is a leaf in $\mathrm{Li}(v)$.

We note that $\mathrm{Li}(v) = \{v\}$ if v is not adjacent to any edges. For convenience, we also define $\mathrm{Li}(v) = \{v\}$ if v is a pebbled node.

The above definition of $\mathrm{Li}(v)$ presupposes that Inv5 holds. It will be convenient, however, if $\mathrm{Li}(v)$ is defined for all nodes v at all moments of execution. For this we use the following generalized definition: $\mathrm{Li}(v)$ is the tree obtained by "dangling" the connected component of B_j containing v by v, such that v is the root, and then pruning all portions of the "dangled" tree that lie beneath a pebbled node (thus a non-leaf node of the tree is never pebbled, and $\mathrm{Li}(v)$ reduces to v if v is pebbled). It is easy to see this definition is indeed a generalization of (i.e., compatible with) the original definition of $\mathrm{Li}(v)$.

The *size* of $\mathrm{Li}(v)$ is the number of *non-pebbled* nodes in $\mathrm{Li}(v)$. We define

$$\gamma(B_j) = \max_v \ \mathrm{size}(\mathrm{Li}(v))$$

where the max is taken over all vertices of B_j. We also define

$$\gamma(B) = \max\{\gamma(B_2), \gamma(B_4)\}.$$

As we will see, $\gamma(B)$ is closely related to maximum running time of a simulator cycle.

Fans and brooms. We make a last few special-purpose definitions that will be convenient for the proofs in the next subsection.

A (maximal) connected component of B_4 consisting of a single vertex in shore 4 connected to t different vertices in shore 5, with all t edges having direction $\rightarrow$, is called a *rightward B_4-fan* of degree t. (We note $t \leq |\mathcal{Z}|$, for obvious reasons.) The vertex in shore 4 is called the *apex* of the fan. The union of all edges that are parallel to edges in a rightward B_4-fan F is called the *parallel completion* of F. We note that because each edge in a B_4-fan has a different label k, different edges of a B_4-fan do not undergo the exact same "parallel duplication" process, cf. (6). Thus the parallel completion of a B_4-fan of degree t is not simply the disjoint union of $|\mathcal{Z}|$ different B_4-fans of same degree t. Indeed, it is easy to see from Inv3[21], from Lemma 15 and from the remarks preceding that lemma that the parallel completion of a B_4-fan of degree t consists of $\binom{t}{2}$ (disjoint) B_4-fans of degree 2, of $(|\mathcal{Z}|-1)t - 2\binom{t}{2}$ (disjoint) B_4-fans of degree 1, and of the original B_4-fan of degree t.

The parallel completion of is called *self-contained* if no edges of B_4 outside the parallel completion are adjacent to any edges inside the self-completion. By the above remarks we have

$$\text{size}(\text{Li}(v)) \leq 3$$

for every vertex v in the parallel completion of a self-contained fan such that v is not in the same component as the fan's apex.

A natural process for constructing a rightward B_4-fan, that will occur many times below, is as follows: (i) a 3-query $(3, x_3, y_3, \to)$ of direction $\to$ is created; (ii) a number of 4-edges of direction $\to$ adjacent at the left to y_3 are subsequently created; more precisely, the i-th such edge comes about as the result of a new 4-query $(4, y_3 \oplus k_i, \ldots, \to)$ being created, for some sequence of values $k_1, \ldots, k_t \in \mathcal{Z}$. If the process just described doesn't cause abort to occur, and if no other queries are created than the ones just described, it is easy to see that the resulting B_4-fan of degree t that is created is self-contained regardless of the previous state of B_4. (This uses, in particular, Inv2; for when we select a random vertex y_3 in shore 4 as the head of the 3-edge and as the fan's apex, we need y_3 to be selected such that there do not exist values $k, k', k'' \in \mathcal{Z}$ and $x_4 \in \text{domain}(P_4)$ such that

$$y_3 \oplus k \oplus k' = x_4 \oplus k''.$$

Indeed, note that the set

$$\{y_3 \oplus k \oplus k' : k, k' \in \mathcal{Z}\}$$

is a superset of the shore 4 vertices in the parallel completion of any fan of apex y_3, whereas the set

$$\{x_4 \oplus k'' : x_4 \in \text{domain}(P_4), k'' \in \mathcal{Z}\}$$

is the set of all shore 4 vertices already adjacent to an edge in B_4. As for adjacencies in shore 5, these cannot be created because of Inv3 and the fact that the newly created 4-queries all have direction $\to$.)

It is also easy to see (the arguments are the same) that if the above two-step process is interleaved for several fans at once the resulting set of fans created each have a self-contained parallel completion, presuming abort doesn't occur. Furthermore, if other non-4-queries are created at the same time this obviously doesn't change anything either (as the final condition concerns B_4). In fact, new 4-queries can also be created during the fan creation process without affecting the self-containment of fans as long as each such newly created 4-query is "anchored in B_4"; i.e., as long as $x_4 \oplus \mathcal{Z}$ is a shore 4 vertex adjacent to a previous edge of B_4 for each new 4-query of the form $(4, x_4, y_4, \to)$ and as long as $y_4 \oplus \mathcal{Z}$ is a shore 5 vertex adjacent to a previous edge of B_4 for each new 4-query of the form $(4, x_4, y_4, \leftarrow)$. (The details all involve Inv2 and Inv3, and the arguments are entirely similar to above.)

A 3-edge of direction $\to$ attached to a rightward B_4-fan, where each edge of the B_4-fan is additionally attached to a 5-edge of direction $\perp$, is called a *B_4-broom*; the *parallel completion* of the broom is obtained by adding the parallel completion of the B_4-fan to the broom. We say the broom is *self-contained* if the broom's fan has a self-contained parallel completion. Importantly, one can note that

$$\text{size}(\text{Li}(v)) \leq 3$$

[21] Inv3 precludes edges in the parallel completion from having adjacencies in shore 5.

for *every* vertex v of B_4 adjacent to an edge in the parallel completion of a self-contained broom, since the vertices inside the fan are pebbled (and hence have $\mathrm{size}(\mathrm{Li}(v)) = 0$).

Symmetric definitions (leftward fan, broom, etc) hold for B_2. For example, the apex of a leftward B_2-fan is a shore 3 vertex.

Inside the Simulator Cycle

In the previous subsection we mainly introduced the graphs B_2, B_4 and B, the metrics $\gamma(B_2)$, $\gamma(B_4)$ and $\gamma(B)$ and notions of fans and brooms. In this section our main concern is to bound the maximum per-simulator-cycle growth of $\gamma(B)$ as well as of the sets *Queries*, *KeyQueries* and *EQueries*. Recall that a "simulator cycle" is the portion of execution allotted to answering a single adversarial query. The upper bounds we obtain depend on the type of adversarial query, and a schematic summary of the results is given in Table 1. (The two $\gamma(B)$'s appearing in the last row of this table refer, more precisely, to the value of $\gamma(B)$ at the *start* of the simulator cycle.)

Unsurprisingly, simulator cycles that answer an adversarial query to f, P1, $\mathrm{P2}^{\pm 1}$, $\mathrm{P4}^{\pm 1}$ or $\mathrm{P5}^{-1}$ are quite easy to analyze whereas all other simulator cycles (namely those answering a query to $\mathrm{P1}^{-1}$, $\mathrm{P3}^{\pm 1}$ and P5) are quite involved to analyze. Hence the upper bounds are proved in two separate lemmas, for the "easy" and "hard" cases. The following lemma covers the "easy" cases of Table 1:

Table 1: Bounds on the growth of key quantities per adversarial query.

adds at most ... to:	$\gamma(B)$	$\lvert KeyQueries\rvert$	$\lvert Queries\rvert$	$\lvert EQueries\rvert$	Lemma
query to E or E^{-1}	0	0	0	1	17
query to f	0	1	0	0	17
query to P1 or $\mathrm{P5}^{-1}$	0	0	1	0	17
query to P2, $\mathrm{P2}^{-1}$, P4 or $\mathrm{P4}^{-1}$	2	0	1	0	17
query to $\mathrm{P1}^{-1}$, P3, $\mathrm{P3}^{-1}$ or P5	2	0	$6\gamma(B)$	$2\gamma(B)$	18

Lemma 17. *An adversarial query to E, E^{-1}, f, P1, P2, $P2^{-1}$, P4, $P4^{-1}$ or $P5^{-1}$ influences the values $\gamma(B)$, $\lvert KeyQueries\rvert$, $\lvert Queries\rvert$ and $\lvert EQueries\rvert$ as follows:*

- *$\gamma(B)$ is increased by at most 2 in queries to P2, $P2^{-1}$, P4 or $P4^{-1}$, and by* 0 *in other queries.*
- *$\lvert KeyQueries\rvert$ is increased by at most 1 in queries to f, and by* 0 *in other queries.*
- *$\lvert Queries\rvert$ is increased by at most 1 in queries to P1, P2, $P2^{-1}$, P4, $P4^{-1}$ or $P5^{-1}$, and by* 0 *in other queries.*
- *$\lvert EQueries\rvert$ is increased by at most 1 in queries to E or E^{-1}, and by* 0 *in other queries.*

Proof. We examine each quantity separately:

- $\gamma(B)$. Note that $\gamma(B_2)$ is defined by P_2 only. An adversarial query to E, E^{-1}, P1, P4, $\mathrm{P4}^{-1}$ or $\mathrm{P5}^{-1}$ clearly does not add new elements to P_2 (particularly as procedure EmptyQueue() is never evaluated). A query to P2 or $\mathrm{P2}^{-1}$ increases $\gamma(B_2)$ by at most one by Lemmas 13 and 14. (This also holds if B_2 is empty to start with, since $\gamma(B_2) = 1$ if B_2 is empty.) Each query to f adds as many edges to B_2 as there are 2-queries (unless abort occurs). However these new edges all go into their own isolated component because of the second "for" loop in KeyQueryChecks and, hence,

$\gamma(B_2)$ can only augment by 1 as a result. As similar bounds hold for $\gamma(B_4)$ the claims on $\gamma(B)$ follow.

- *KeyQueries.* This is obvious: an adversarial query to f results in at most one new element added to *KeyQueries* whereas queries to other public functions don't increase the number of scheduled subkeys as the simulator never calls f. (Indeed, outside f the simulator only uses the set $\mathcal{Z}$ and the table f^{-1}.)
- *Queries.* In a query to P1, P2, P2^{-1}, P4, P4^{-1} or P5^{-1}, procedure EmptyQueue() is never evaluated, and at most one element is added to *Queries*. An adversarial query to E, E^{-1} or f does not result in new elements.
- *EQueries.* An adversarial query to E or E^{-1} results in at most one new element added to *EQueries*. Queries to f, P1, P2, P2^{-1}, P4, P4^{-1} or P5^{-1} do not result in calls to E or E^{-1}. ☐

The following lemma proves the "hard part" of Table 1. In many ways, this lemma's proof constitutes the heart of our whole analysis (and is therefore worth spending some time on despite its depressing length).

Lemma 18. *An adversarial query to P1^{-1}, P3, P3^{-1} or P5 affects $\gamma(B)$, $|KeyQueries|$, $|Queries|$ and $|EQueries|$ as follows: $\gamma(B)$ is increased by at most 2, $|Queries|$ increases by at most $6\gamma(B)$ (referring to the value of $\gamma(B)$ at the start of the simulator cycle), and $|EQueries|$ increases by at most $2\gamma(B)$ (ditto). Any of these queries leaves KeyQueries invariant.*

Proof. The fact that *KeyQueries* is left unchanged is obvious, since the simulator doesn't query f internally. For the remaining statements we only consider queries to P1^{-1} and P3, which is sufficient by symmetry. We start by considering a query to P1^{-1}.

Assume the adversary makes a query P1$^{-1}(y_1^*)$. If y_1^* is adjacent to a 1-edge then the simulator returns immediately, so we can assume that y_1^* is not adjacent to a 1-edge when the query P1$^{-1}(y_1^*)$ is made, i.e. that y_1^* is unpebbled at that moment. We let $T_{y_1^*}$ be the live tree $\mathrm{Li}(y_1^*)$ as it stands at the start of the simulator cycle. (The purpose of "renaming" $\mathrm{Li}(y_1^*)$ as $T_{y_1^*}$ is to avoid ambiguity, since $\mathrm{Li}(y_1^*)$, technically, changes during the simulator cycle; by our definition, $T_{y_1^*}$ is a specific "snapshot" of $\mathrm{Li}(y_1^*)$.) We note that $T_{y_1^*}$ has at least one non-pebbled node, namely y_1^* itself, and that y_1^* is the root of $T_{y_1^*}$. If y_1^* is the unique vertex of $T_{y_1^*}$ then "nothing happens" when the query P1$^{-1}(y_1^*)$ is made (more precisely, a 1-edge of direction $\leftarrow$ is lazy-sampled, added to *LeftQueue*, then *LeftQueue* is emptied with nothing happening, and the simulator returns). We will therefore assume that $T_{y_1^*}$ has at least two nodes in what follows.

Recall that $T_{y_1^*}$ has either one or zero pebbled leaves, since Inv5 holds at the beginning of the simulator cycle (see the original definition of a live tree). We divide the analysis into three cases according to whether $T_{y_1^*}$ has a pebbled leaf or not, and if so in which shore.

Case 1: $T_{y_1^*}$ has no pebbled leaf. G_2 starts by adding an edge $(1, x_1^*, y_1^*, \leftarrow, num)$ where x_1^* is chosen randomly. At this point the values $\{x_1^* \oplus k : k \in \mathcal{Z}\}$ are added to *LeftFreezer* (which in fact consists only of those values, since *LeftFreezer* starts out empty). (If abort does not occur[22] then we are guaranteed, in particular, that $ETable[K](x_1^* \oplus k) = \perp$ for every $K \in \{0,1\}^\kappa$ and every $k \in \mathcal{Z}$, by the checks made in FreezeLeftValues.) The pair $(1^+, y_1^*)$ is finally added to *LeftQueue*. Before this *LeftQueue*, like *RightQueue*, is empty. Then EmptyQueue is called.

[*Some handy vocabulary:* In a nutshell, now, the new 1-edge $(1, x_1*, y_1^*, \leftarrow)$ will be popped from *LeftQueue*, and the simulator will complete a path for every child of y_1^* in $T_{y_1^*}$; these children of y_1^*

[22] While the lemma still holds for simulator cycles during which abort occurs we will tacitly carry out the analysis assuming abort doesn't occur. Checking that nothing extraordinary happens if abort occurs (which isn't important anyway for the rest of the proof) is left to the reader.

will therefore become pebbled by 3-edges, and these 3-edges are added to *LeftQueue*; when each 3-edge is popped from *LeftQueue*, third-level nodes of the tree become pebbled by 1-edges that are also added to *LeftQueue*, and so on. As a convention, we say that a node of the tree is *processed* when the 1- or 3-edge containing it is popped from *LeftQueue* and the relevant call to ProcessNew1Edge or ProcessNew3$^-$Edge is made (i.e., the call containing the node's name); during that same call, we say that the node's children are *pre-processed*. For example, when the first element $(1^+, y_1^*)$ is popped from *LeftQueue* and ProcessNew1Edge(y_1^*) is called, we say that y_1^* is *processed* and that the children of y_1^* in $T_{y_1^*}$ are *pre-processed* by the call.]

Let $x_3^1, \ldots, x_3^\ell$ be the children of y_1^* in $T_{y_1^*}$ (these are "second-level" nodes of the tree) and let k^j be the label on the 2-edge (y_1^*, x_3^j). We assume the children $x_3^1, \ldots, x_3^\ell$ are ordered such that $k^1 < \ldots < k^\ell$, and that a loop of the type "forall $k \in \mathcal{Z}$" iterates from the smallest value of $k \in \mathcal{Z}$ to the largest. (We will make similar ordering assumptions without mention in the future.)

As per our convention above, y_1^* is "processed" when the pair $(1^+, y_1^*)$ is popped off *LeftQueue* and ProcessNew1Edge(y_1^*) is called. Within that call, the first child x_3^1 of y_1^* is "pre-processed" when the "forall" loop in ProcessNew1Edge iterates with $k = k^1$. At this point, CompletePath1$^+(y_1^*, k^1)$ is called, and PrivateP3(x_3^1) is called from within the latter function. Because $T_{y_1^*}$ has no pebbled leaves to start with, $P_3(x_3^1) = \perp$ at this point. Moreover, the only value x_1 that x_3^1 is k-1-2-joined to for some $k \in \mathcal{Z}$ is $x_1 = x_1^*$, with $k = k^1$, and we have $x_1^* \oplus k^1 \in$ *LeftFreezer*. Thus ForcedP3(x_3^1) will return $\perp$ when it is called by PrivateP3(x_3^1), and PrivateP3(x_3^1) creates a new 3-edge $(3, x_3^1, y_3^1, \rightarrow)$ by calling ReadTape. This new 3-edge is added to *LeftQueue*. Then CompletePath1$^+$ creates a new 4-query $(4, y_3^1 \oplus k^1, y_4^1, \rightarrow)$ (indeed, this query will be *new* because otherwise its adjacency to the newer query $(3, x_3^1, y_3^1)$ would contradict Inv2), removes $x_1^* \oplus k^1$ from *LeftFreezer*, and finally creates a new 5-query $(5, y_4^1 \oplus k^1, y_5^1, \perp)$ where $y_5^1 = E_{f^{-1}(k^1)}(x_1^* \oplus k^1) \oplus k^1$. (We note in passing that since $E_{f^{-1}(k^1)}(x_1^*)$ is randomly sampled at this point, $y_5^1 \notin \text{range}(P_5)$ or else E would abort from within AddEQuery. Moreover $y_4^1 \oplus k^1$ cannot be in domain(P_5), either, without contradicting Inv2, the 4-query $(4, y_3^1 \oplus k^1, y_4^1, \rightarrow)$ being new.)

The new 3-query $(3, x_3^1, \ldots)$ and the new 5-query $(5, x_5^1, \ldots)$ each correspond to a new 3-edge and 5-edge of directions $\rightarrow$ and $\perp$ respectively that are added to B, while the new 4-query $(4, x_4^1, \ldots)$ becomes a collection of $|\mathcal{Z}|$ different parallel 4-edges added to $B_4 \subseteq B$, where only one of these 4-edges (i.e., the one with label k^1) is adjacent to the newly created 3-edge and 5-edge; the other newly created $|\mathcal{Z}| - 1$ 4-edges are, by Inv2 and Inv3, not adjacent to any previous edges of B. Thus, the parallel completion of a self-contained "degree 1 B_4-broom" (i.e., broom whose fan has degree 1) is added to B.

The pre-processing of subsequent second-level nodes of $T_{y_1^*}$ (via the same call ProcessNew1Edge) is not affected by the degree 1 B_4-broom added by the pre-processing of x_3^1, or by the removal of $x_1^* \oplus k^1$ from *LeftFreezer*, since each such second-level node is connected to y_1^* by a 2-edge of different label. Thus, since $T_{y_1^*}$ has ℓ second-level nodes, ℓ parallel completions of degree 1 brooms are sequentially added to B as the game pre-processes the second-level nodes of $T_{y_1^*}$ and all these parallel completions are self-contained.

Let (x_3^j, y_3^j) be the 3-edge (broom "handle") containing x_3^j. We also point out (since this will shortly play a role), that the sets

$$\{y_3^j \oplus k : k \in \mathcal{Z}\} \text{ and } \{y_3^i \oplus k : k \in \mathcal{Z}\} \tag{7}$$

are disjoint from one another for $i \neq j$, by Inv3, and also that

$$\{y_3^j \oplus k : k \in \mathcal{Z}, k \neq k^j\} \cap \text{domain}(P_4) = \emptyset \tag{8}$$

after the pre-processing of second-level nodes, by Inv2.

We also note that while the newly created 5-edges are placed on *RightQueue*, they will later be "innocuously popped" from the queue, being adjacent to a single 4-edge with whom they are part of a completed path. (The fact that these newly created 5-edges indeed remain adjacent to a single 4-edge—i.e., that new adjacent 4-edges aren't added—will follow by Inv2 and by the analysis below.)

To describe the processing of second-level nodes of $T_{y_1^*}$, let x_3 be the first second-level node with at least one child and let $y_1^1, \ldots, y_1^t$ be the t third-level children of x_3 in $T_{y_1^*}$ (these are nodes in shore 2 of B). Let k_j be the label on the 2-edge from x_3 to y_1^j (note that above, we used k^j, not k_j—thus we are not (yet) overwriting our definitions), and let k^* be the label on the 2-edge from x_3 to y_1^*. We note $k^* \neq k_1, \ldots, k_t$ since all these are labels of distinct 2-edges adjacent at the same node x_3 of B_2. (Confusingly, we also note that $k^* = k^j$ for some j, because $k^1, \ldots, k^\ell$ are the labels of the 2-edges leaving y_1^*.) Also, let (x_3, y_3) be the 3-edge containing x_3.

When $(3^-, x_3)$ is popped from *LeftQueue*, ProcessNew3$^-$Edge(x_3) is called and iterates its "forall" loop. When the "forall" loop iterates with $k = k_j$ an adjacency with a 2-edge is detected and, since y_1^j is unpebbled (in particular, not in any completed path), CompletePath3$^-$(x_3) is called. CompletePath3$^-$ starts by creating a new query $(1, x_1^j, y_1^j, \leftarrow)$, and the values $\{x_1^j \oplus k : k \in \mathcal{Z}\}$ become a new ice tray. Then CompletePath3$^-$ makes the call P4($P_3(x_3) \oplus k_j$) (equivalent to P4($y_3 \oplus k_j$)), which results in the creation of a new 4-query $(4, x_{4,j}, y_{4,j}, \rightarrow)$ with $x_{4,j} = y_3 \oplus k_j$. The query $(4, x_{4,j}, y_{4,j}, \rightarrow)$ is *new*, indeed, because $y_3 \oplus k_j \notin \text{domain}(P_4)$, as per the remarks above (cf. (7) and (8)), given that $k_j \neq k^*$. Next $x_1^j \oplus k_j$ is removed from *LeftFreezer* (indeed, no modification of *LeftFreezer* has occurred since the (recent) creation of the edge $(1, x_1^j, y_1^j, \leftarrow)$) and the value $y_{5,j} = E_{f^{-1}(k_j)}(x_1^j \oplus k_j)$ is computed. Again, since $x_1^j \oplus k_j$ is a "fresh" call to $E_{f^{-1}(k_j)}$ that involves calling AddEQuery, $y_{5,j} \notin \text{range}(P_5)$ or else AddEQuery would abort, and since the 4-query $(4, x_{4,j}, y_{4,j}, \rightarrow)$ is new we also have $y_{4,j} \oplus k_j \notin \text{domain}(P_5)$ by Inv2. Thus, with high probability[23], the game adds a new 5-query $(5, y_{4,j} \oplus k_j, y_{5,j}, \perp)$ without aborting. We note that $(y_3, y_{4,j} \oplus k_j)$ becomes a new 4-edge in B_4, along with its attendant parallel edges.

Carrying out the above process for $j = 1, \ldots, t$ has the effect of creating a B_4-broom of degree $t+1$, where the, "handle" of the broom is the query $(3, x_3, y_3, \rightarrow)$ and where edge labels on the broom's fan are $k^*, k_1, \ldots, k_t$. The broom is self-contained, as it follows the canonical "two-step creation process" described in our introduction to brooms. We can also note that the broom's degree is exactly equal to the degree of the second-level node x_3 in $T_{y_1^*}$ (edge to parent included). Hence, the net effect of processing the new 3-edge $(3, x_3, y_3, \rightarrow)$ in ProcessNew3$^-$Edge is the creation of a self-contained B_4-broom whose degree is the same as that of x_3 in B_2. We note again that the new 5-edges of direction $\perp$ created (the "broom hairs") are adjacent to only one 4-edge, with whom they lie on a completed path.

While the above assumed for simplicity that x_3 was the first second-level child of y_1^* with third-level children, nothing substantive changes when processing subsequent second-level nodes: each second-level node gives rise to a self-contained broom of degree equal to its own degree in $T_{y_1^*}$.

The processing of level 3 nodes is similar to the processing of the level 1 node y_1^*. More precisely, let y_1^j be a level 3 node (as in the third-to-last paragraph), with associated 1-edge $(1, x_1^j, y_1^j, \leftarrow)$ added (as just described) during the processing of x_3 and during the pre-processing of y_1^j. Then level 4 nodes beneath y_1^j are pre-processed the same way with respect to the (newly created) 1-edge (x_1^j, y_1^j) as level 2 nodes were pre-processed with respect to the (then newly created) 1-edge (x_1^*, y_1^*). In particular, the presence of the 2-edge (y_1^j, x_3) of label k_1^j has no effect because this edge is in a completed k_1^j-path, and is therefore ignored by ProcessNew1Edge(y_1^j). Subsequently, level 4 nodes are processed similarly to level 2 nodes, and so on. The interleaved, "breadth-first search" aspect of the $T_{y_1^*}$ traversal obviously makes no difference either.

[23] But we emphasize that the probability of abortion is not this lemma's topic. Phrases such as "with high probability" are only included for readability (to remind that abort might have occurred, and in which case we basically don't care) and have no mathematical intent.

In summary, as a result of the game answering the query $P_1^{-1}(y_1^*)$, no new edges are added to B_2, while some new edges are added to B_4. The edges added to B_4 are in (the parallel completions of) self-contained B_4-brooms; each B_4-broom is associated to an even-level node in $T_{y_1^*}$, and the degree of the broom equals, exactly, the degree of that node in $T_{y_1^*}$. Because

$$\text{size}(\text{Li}(v)) \leq 3$$

for every shore 4 and shore 5 vertex v in the completion of a self-contained broom, we therefore see $\gamma(B_j)$, $j = 2, 4$, increase by at most 2 (i.e., from 1 to 3) as a result of the game answering the query $P_1^{-1}(y_1^*)$.

Each odd-level node in $T_{y_1^*}$ causes the game to create one new 1-query (including y_1^* itself), whereas each even-level node of degree t (edge to parent included) causes the game to create one new 3-query, t new 4-queries, and t new 5-queries (all part of the same broom). Thus, some simple accounting shows that the total number of queries created by the game is

$$|V(T_{y_1^*})| + 2|E(T_{y_1^*})|$$

where $V(T_{y_1^*})$ is the vertex set of $T_{y_1^*}$ and $E(T_{y_1^*})$ is the edge set of $T_{y_1^*}$. Since a tree with m vertices has (at most) $m - 1$ edges, the number of queries created by the game is therefore at most

$$|V(T_{y_1^*})| + 2(|V(T_{y_1^*})| - 1) \leq 3 \cdot \text{size}(T_{y_1^*}) = 3 \cdot \text{sizeLi}(y_1^*) \leq 3\gamma(B)$$

since $\text{size}(T_{y_1^*}) = |V(T_{y_1^*})|$, where the last two quantities refer to the start of the simulator cycle. Finally, the number of calls to E is one for each edge of $T_{y_1^*}$, so the simulator makes at most $\text{size}(T_{y_1^*})$ calls to E. This concludes our analysis of case 1.

Case 2. $T_{y_1^*}$ has a pebbled leaf at an even-level node (i.e., a node in shore 3). Let $x_3^\circ = \overline{u}$ be the pebbled leaf of $T_{y_1^*}$ and let (x_3°, y_3°) be the 3-edge adjacent to that leaf. Let y_1° be the odd-level parent of x_3° in $T_{y_3^\circ}$ (possibly $y_1^\circ = y_1^*$), and let k° be the label of the edge (y_1°, x_3°) in B_2. We consider two different subcases, according to whether y_3° is or is not adjacent to a 4-edge of label k° at the start of the simulator cycle.

Subcase 2.1. y_3° is adjacent to a 4-edge of label k°. Let (y_3°, x_5°) be the 4-edge of label k° and let $T_{x_5^\circ} = \text{Li}(x_5^\circ)$ as this tree stands at the start of the simulator cycle. We first argue that x_5° cannot be in a 5-edge (x_5°, y_5°) when the adversary makes the query $P_1^{-1}(y_1^*)$. Indeed, if this were the case, then because the 4-edge (y_3°, x_5°) has a direction ($\leftarrow$ or $\rightarrow$) either the 5-edge (x_5°, y_5°) or the 3-edge (x_3°, y_3°) would have to be later than the 4-edge (y_3°, x_5°), placing all three edges (x_3°, y_3°), (y_3°, x_5°) and (x_5°, y_5°) on a common k°-completed path, together also with the 2-edge (y_1°, x_3°) of label k°; but then y_1° would have to be adjacent to a 1-edge, a contradiction to the fact that x_3° is the only vertex of $T_{y_1^*}$ adjacent to a 3-edge or 1-edge. Thus, x_5° is not adjacent to a 5-edge, and $T_{x_5^\circ} \neq \{x_5^\circ\}$; in fact we know the tree $T_{x_5^\circ}$ has a pebbled leaf, this being y_3°, as y_3° is adjacent to a 3-edge at the start of the simulator cycle, and as (y_3°, x_5°) is a 4-edge.

When G_2 processes the tree $T_{y_1^*}$, at some point an edge $(1, x_1^\circ, y_1^\circ, \leftarrow)$ is created and put onto the queue (maybe right away, if $y_1^\circ = y_1^*$). When the edge $(1, x_1^\circ, y_1^\circ, \leftarrow)$ is popped from the queue and y_1° is processed, the game eventually executes the body of the "forall" loop in ProcessNew1Edge(y_1°) with $k = k^\circ$. ProcessNew1Edge() will call PrivateP3(x_3°) to learn y_3° (and to no other effect, since the 3-query $(3, x_3^\circ, y_3^\circ)$ already exists) and will then call P4($y_3^\circ \oplus k^\circ$) to compute the value x_5° (and to no other effect). As already argued, $x_5^\circ \notin \text{domain}(P_5)$ (the technical reason this hasn't changed since the game started processing the tree $T_{y_1^*}$ is that all 5-edges that have been added have been adjacent to a new 4-edge of direction $\to$, and Inv3). We also claim that $x_1^\circ \oplus k^\circ \in$ *LeftFreezer*, at this point. Indeed, since the creation of the 1-query $(1, x_1^\circ, y_1^\circ, \leftarrow)$, the only values in the left ice tray

$$\{x_1^\circ \oplus k : k \in \mathcal{Z}\}$$

that have been removed from *LeftFreezer* are values of the form $x_1^\circ \oplus k$ for $k \neq k^\circ$ (k the label of some other 2-edge incident to y_1°—either the edge to the parent of y_1° in $T_{y_1^*}$, or to an "earlier" child of y_1°). Thus $E_{f^{-1}(k^\circ)}(x_1^\circ \oplus k^\circ)$ is a "fresh" (i.e., *fresh* = **true**) query to $E_{f^{-1}(k^\circ)}$ and so the 5-edge (x_5°, y_5°) where $y_5^\circ := \mathrm{E}_{f^{-1}(k^\circ)}(x_1^\circ \oplus k^\circ) \oplus k^\circ$ is created without causing an abortion, with high probability. (Namely, AddEQuery doesn't abort w.h.p., and the latter also implies $y_5^\circ \notin \mathrm{range}(P_5)$.) At this point, the 5-edge (x_5°, y_5°) is added to *RightQueue*. This concludes the processing of key value k° in the "forall" loop associated to *LeftQueue* element y_1^-. We note that (x_3°, y_3°) is neither added to *LeftQueue* or to *RightQueue*, since this is not a new edge. In particular, we note that x_3° might be a pebbled leaf on another live tree $\mathrm{Li}(v)$, where the intersection of $\mathrm{Li}(v)$ and $T_{y_1^*}$ is just x_3°; vertices in such a tree T_v will not be affected (i.e., not become pebbled) by the processing of $T_{y_1^*}$. (Likewise, y_3° might be the "pebbled leaf common intersection" of $T_{x_5^\circ}$ with some other live tree $\mathrm{Li}(u)$, for u in shore 4 or 5.)

The rest of the processing of $T_{y_1^*}$ is not affected by the presence of the edge (x_3°, y_3°). When *LeftQueue* is emptied, the nodes of $T_{y_1^*}$ have been pebbled. A self-contained B_4-broom of degree t is added, like in Case 1, for every shore 3 vertex of $T_{y_1^*}$ of degree t, except for vertex x_3°. Here x_3° is only "responsible" for the addition of a single 5-edge, namely the 5-edge (x_5°, y_5°), and for no new 4-edges.

Later, when the 5-edges on the "broom hairs" are popped from *RightQueue*, they cause no effect, being only adjacent to a single 4-edge, with which they form a completed path (as already discussed in Case 1). When the 5-edge (x_5°, y_5°) is popped from *RightQueue*, however, the tree $T_{x_5^\circ}$ is processed and pebbled like the tree $T_{y_1^*}$ in Case 1 (just as if the adversary had just made the query $\mathrm{P5}^{-1}(x_5^\circ)$, resulting in a new query (x_5°, y_5°)). In particular, when the element $(5^-, x_5^\circ)$ is popped from *RightQueue*, and when the "forall" loop executes with $k = k^\circ$, k° is ignored because (x_5°, y_5°) is already on a k°-completed path. Thus, here, the presence of the "pebbled leaf" y_3° on the tree $T_{x_5^\circ}$ is, effectively, ignored, and $T_{x_5^\circ}$ is processed like a Case 1 tree.

After the processing of $T_{x_5^\circ}$, B_2-brooms of degree t have been added for every shore 4 vertex in $T_{x_5^\circ}$ of degree t. The 1-edges on these brooms have been added to *LeftQueue*, but they are innocuously popped off *LeftQueue*, being adjacent to a single 2-edge with which they form completed path. Subsequently, both *LeftQueue* and *RightQueue* are empty, and EmptyQueue() terminates.

In summary, because the only edges added to B_2 and B_4 are in brooms, $\gamma(B_2)$, $\gamma(B_4)$ increase by at most 2 each during the simulator cycle (i.e., from 1 to 3). Similar accounting as in Case 1 shows the total number of new queries created during this simulator cycle is at most $3\mathrm{size}(T_{y_1^*}) + 3\mathrm{size}(T_{x_5^\circ}) \leq 6\gamma(B)$. (In particular, we note that the pebbled leaves x_3° and y_3° of $T_{y_1^*}$ and $T_{x_5^\circ}$ don't contribute any brooms, and indeed contribute no queries at all (since we "bill" the 5-query (x_5°, y_5°) to x_5° itself).) As before, the number of calls made to E and E^{-1} is at most the total number of edges in $T_{y_1^*}$ and $T_{x_5^\circ}$, which is at most $\mathrm{size}(T_{y_1^*}) + \mathrm{size}(T_{x_5^\circ}) \leq 2\gamma(B)$ since $\mathrm{size}(T_{y_1^*})$, $\mathrm{size}(T_{x_5^\circ})$ are exactly equal to the number of edges in these trees (as the two pebbled vertices don't count in the size). This concludes the analysis of subcase 2.1.

Subcase 2.2: y_3° is not adjacent to a 4-edge of label k°. (The definitions of x_3°, y_3°, y_1°, k° remain as outlined in Case 2.) Again, when the game processes $T_{y_1^*}$ at some point an edge $(1, x_1^\circ, y_1^\circ, \leftarrow)$ is created and put onto the *LeftQueue*. When this edge is popped from the queue and $k = k^\circ$ is processed in the "forall" loop of ProcessNew1Edge() then, again, $\mathrm{PrivateP3}(x_3^\circ)$ is queried to no effect, since the edge (x_3°, y_3°) already exists. At this point $\mathrm{P4}(y_3^\circ \oplus k^\circ)$ will still be undefined since the only 4-queries that have been created since the start of the processing of $T_{y_1^*}$ are left-adjacent to new 3-edges of direction $\to$. After the new 4-query $(4, x_4^\circ, y_4^\circ, \to)$ has been created by P4, where $x_4^\circ = y_3^\circ \oplus k^\circ$ and y_4°

was selected by ReadTape, the situation is indistinguishable from subcase 2.1, except that the 4-query $(4, x_4^\circ, y_4^\circ, \rightarrow, num)$ now has a larger value of num; but since this value of num played no role in our analysis of subcase 2.1, the remainder of simulator cycle proceeds as in subcase 2.1, with $x_5^\circ := y_4^\circ \oplus k^\circ$ and $T_{x_5^\circ} := \mathrm{Li}(x_5^\circ)$ as the tree $\mathrm{Li}(x_5^\circ)$ stands after the creation of the 4-edge (y_3°, x_5°). In this case, in particular, since $T_{x_5^\circ}$ reduces to the single edge (y_3°, x_5°), nothing happens when the edge (x_5°, y_5°) is popped from *RightQueue*, and the "right-hand side" of EmptyQueue immediately terminates.

The main difference with subcase 2.1 is only in the final accounting: in the current subcase, a new edge (and its parallel attendants) is added to B_4 that is not part of a broom, namely the edge (y_3°, x_5°). The edge (y_3°, x_5°) does not increase the size(Li(v)) for any v in shore 4 or 5 at the end of the simulator cycle because nodes y_3° and x_5° are pebbled, but the parallel edges to (y_3°, x_5°) might cause size(Li(v)) to increase for certain v's. Since parallel edges go into distinct connected components of B_4 by Lemma 13, size(Li(v)) can only increase by 1 for every v in shore 4 or 5 as a result of the parallel edges to (y_3°, x_5°), and, moreover, connected components in the parallel completion of one of the new B_4 brooms aren't affected by these parallel edges, as is easy to see by Inv2, Inv3. Thus, like in the previous case, $\gamma(B_4)$ increases by at most 2 during the simulator cycle, and likewise for $\gamma(B_2)$. Despite the new "non-broom" 4-edge added to B_4, basic accounting (distinguishing between the cases when size$(T_{y_1^*}) = 1$ and size$(T_{y_1^*}) > 1$) will again show that the game creates at most 3size$(T_{y_1^*}) \leq 3\gamma(B)$ new queries. Finally, the simulator makes at most size$(T_{y_1^*}) \leq \gamma(B)$ calls to E because the number of these calls is (again) the number of edges in $T_{y_1^*}$.

Case 3: $T_{y_1^*}$ has a pebbled leaf at an odd-level node. Let $y_1^\circ = \overline{u}$ (in shore 2) be the pebbled leaf of $T_{y_1^*}$, and let (x_1°, y_1°) be the 1-edge containing y_1°. Let x_3° be the parent of y_1° in $T_{y_1^*}$, and let y_1' be the parent of x_3° in $T_{y_1^*}$ (possibly, $y_1' = y_1^*$). Let k° be the label on the 2-edge (y_1°, x_3°) and let k' be the label on the 2-edge (y_1', x_3°).

As the game processes the tree $T_{y_1^*}$, at some point a 1-query of the form $(1, x_1', y_1', \leftarrow)$ is added, pebbling y_1', and the pair $(1^+, y_1')$ is added to *LeftQueue*. When this pair is popped from *LeftQueue*, the "forall" loop in ProcessNew1Edge() is eventually executed with $k = k'$; at this point PrivateP3(x_3°) is called. ForcedP3(x_3°) will create a *candidate* consisting of the pair $(k^\circ, x_1^\circ \oplus k^\circ)$; this will be the only candidate, since $x_1' \oplus k'$ is still in *LeftFreezer* at this point (using disjointness of left ice trays), and since y_1° is the only pebbled node in $T_{y_1^*}$ (meaning, in particular, that x_3° has no other pebbled children aside from y_1° in the original tree $T_{y_1^*}$). After the "forall" loop, ForcedP3(x_3°) evaluates $y_5^\circ := E_{f^{-1}(k^\circ)}(x_1^\circ \oplus k^\circ) \oplus k^\circ$. We now distinguish several subcases:

Subcase 3.1. $y_5^\circ \in range(P_5)$, and $P_5^{-1}(y_5^\circ) \oplus k^\circ \in range(P_4)$. Let $x_5^\circ = P_5^{-1}(y_5^\circ)$, $y_4^\circ = x_5^\circ \oplus k^\circ$, $x_4^\circ = P_4^{-1}(y_4^\circ)$, $y_3^\circ = x_4^\circ \oplus k^\circ$.

We start by claiming that the query $E_{f^{-1}(k^\circ)}(x_1^\circ \oplus k^\circ)$ is "new" when ForcedP3() makes this query if and only if the same query is "new" at the beginning of the simulator cycle; i.e., that $ETable[f^{-1}(k^\circ)](x_1^\circ \oplus k^\circ) = \perp$ at the beginning of the simulator cycle if and only if it holds in ForcedP3(x_3°). Indeed, the only values z for which $ETable[K](z)$ has become defined during this simulator cycle for some key K are values in the left ice trays that have been created since the start of the simulator cycle. By design, these left ice trays cannot contain a value of the form $x_1^\circ \oplus k$ for any $k \in \mathcal{Z}$, since $x_1^\circ \in \mathrm{domain}(P_1)$ at the start of the cycle (see the first "forall" loop in FreezeLeftValues). Hence the claim.

We next claim that the 4- and 5-queries $(4, x_4^\circ, y_4^\circ)$, $(5, x_5^\circ, y_5^\circ)$ already existed at the beginning of the current simulator cycle, before the processing of $T_{y_1^*}$ began. Indeed, the previous claim directly implies that the query $E_{f^{-1}(k^\circ)}(x_1^\circ \oplus k^\circ)$ cannot be a new query when ForcedP3(x_3°) makes this query (otherwise E would abort, given that $y_5^\circ \in \mathrm{range}(P_5)$), i.e., $ETable[f^{-1}(k^\circ)](x_1^\circ \oplus k^\circ)$ is already defined at the start of the simulator cycle. Secondly, all 5-queries added since the beginning of the processing of $T_{y_1^*}$ have been created with *fresh* = **true** (the first time that *fresh* = **false** will occur is when $(3^-, x_3^\circ)$ will be popped from *LeftQueue* and processed; however this has not occurred yet; indeed, $(3^-, x_3^\circ)$ has not even been added to *LeftQueue* yet). Let $(5, x_5, y_5)$ be such a new 5-query, created inside FinishPath1^+3^-. Then, because if *fresh* = **true** when this query is created, either: (i) $k = k^\circ$ when this query is created in FinishPath1^+3^-, and having $y_5 = y_5^\circ$ is impossible because then $y_5 \oplus k = y_5^\circ \oplus k^\circ$ would already have been in the range of $ETable[f^{-1}(k)] = ETable[f^{-1}(k^\circ)]$ at the start of the simulator cycle, contradicting the freshness of the query "E$(f^{-1}(k), x_1 \oplus k)$" in FinishPath$1^+3^-$ (more particularly a contradiction to Inv4), or else (ii) $k \neq k^\circ$ in FinishPath1^+3^-, and having $y_5 = y_5^\circ$ would cause FreezeRightValues(y_5, k) to abort, given that $y_5^\circ \oplus k^\circ \in$ range$(ETable[f^{-1}(k^\circ)])$ (we have $y_5^\circ \oplus k^\circ = ETable[f^{-1}(k^\circ)](x_1^\circ \oplus k^\circ)$). Thus, either way, $y_5 \neq y_5^\circ$, and so the 5-query $(5, x_5^\circ, y_5^\circ)$ already existed at the start of the simulator cycle. Since all 4-queries added since the start of the simulator cycle have add direction $\to$, this also implies the 4-query $(4, x_4^\circ, y_4^\circ)$ existed at the start of the simulator cycle.

We thirdly claim that $y_3^\circ \notin$ range(P_3). Firstly, it is clear that $y_3^\circ \notin$ range(P_3) before the processing of $T_{y_1^*}$ starts (i.e., at the beginning of the current simulator cycle), because otherwise, given that the 4- and 5-queries $(4, x_4^\circ, y_4^\circ)$, $(5, x_5^\circ, y_5^\circ)$ existed at the start of the simulator cycle, these queries would have been part of a k°-completed path by invariant Inv5. Moreover, because all 3-queries created since the start of this simulator cycle have direction $\to$, having one of these newly created queries end at y_3° would contradict Inv2, given that $y_3^\circ \oplus k^\circ = x_4^\circ$, where $(4, x_4^\circ, y_4^\circ)$ is an older 4-query. Thus $y_3^\circ \notin$ range(P_3) when ForcedP3(x_3°) returns y_3° to PrivateP3(x_3°), and the latter procedure creates a new 3-query $(3, x_3^\circ, y_3^\circ, \perp)$ without aborting. Since this query is adapted it is put both into *LeftQueue* and into *RightQueue* by PrivateP3().

After PrivateP3(x_3°) returns, $x_5' := x_5 :=$ P4$(y_3^\circ \oplus k') \oplus k'$ is computed by ProcessNew1Edge (still during the processing of y_1', and during the pre-processing of x_3°) where the call P4$(y_3^\circ \oplus k')$ may or may not create a new 4-query (i.e., the 4-query $(4, y_3^\circ \oplus k', x_5' \oplus k')$ might be pre-existing, which is fine). FinishPath1^+3^- sets *fresh* to **true** , because $x_1' \oplus k' \in$ *LeftFreezer*, given that $(1, x_1', y_1', \leftarrow)$ is a 1-edge newly created this simulator cycle. Then $y_5' := y_5 := k' \oplus E_{f^{-1}(k')}(x_1' \oplus k')$ is computed (a "fresh" query to E, as we had $x_1' \oplus k' \in$ *LeftFreezer*). Clearly $y_5' \notin$ range(P_5), or else E would abort. We also claim $x_5' \notin$ domain(P_5). If the P4-call P4$(y_3^\circ \oplus k')$ results in the creation of a new 4-query, this is obvious, so assume the 4-query $(4, y_3^\circ \oplus k', x_5' \oplus k')$ pre-existed. It is easy to see, then, that this 4-query already existed at the start of the simulator cycle. Now consider the connected component $T_{y_3^\circ}$ of B_4, where $T_{y_3^\circ}$ is a snapshot of the live tree Li(y_3°) *at the start of the simulator cycle* (notice, among others, that the value y_3° is "already defined" at the start of the simulator cycle, in the current subcase). Note Li(y_3°) contained a pebbled leaf at the start of the simulator cycle, this being x_5°, so the node x_5', also in $T_{y_3^\circ}$ (connected to y_3° by a 4-edge of label k', more precisely) could *not* also be pebbled at the start of the simulator cycle. Moreover, given that 5-queries created so far have all been attached to newly created 4-edges of direction $\to$ (in newly created B_4-brooms), x_5' remains unpebbled until now; thus, $x_5' \notin$ domain(P_5), and the new 5-query $(5, x_5', y_5')$ is added by FinishPath1^+3^- without causing abort, completing a k'-path $(x_1', y_1', x_3^\circ, y_3^\circ, x_5', y_5')$ in B. As usual, the new 5-edge $(5, x_5^\circ, y_5^\circ)$ is added to *RightQueue*. (We note this edge is preceded on *RightQueue* by the edge $(3, x_3^\circ, y_3^\circ)$, added to *RightQueue* by PrivateP3(x_3°).)

The "next interesting thing" to happen in the processing of $T_{y_1^*}$ is the popping of the 3-edge $(3, x_3^\circ, y_3^\circ)$ from *LeftQueue*. As keys k are iterated through such that x_3° is adjacent to a 2-edge of label k (the values $k = k'$ and $k = k^\circ$ are skipped because already part of a completed path), potentially new 4-edges are added at y_3°, each connected to a new 5-edge (indeed, it is easy to see the relevant value $x_1 \oplus k$ will be in *LeftFreezer*, thus resulting in a "fresh" value y_5; arguing that the corresponding value x_5 is not already in domain(P_5) when the new 5-edge is ready to be created can be done as above, when we argued $x_5' \notin$ domain(P_5)). Each newly added 4-edge adjacent to y_3° adds size at most

1 to size(Li(v)) for every v in shores 4 and 5, because of that edge's parallel copies by Lemma 13. Moreover, by Lemma 15, size(Li(v)) can only increase by at most 2 as the result of *all* newly created 4-edges (including, if it was new, the 4-edge $(4, y_3^\circ \oplus k', x_5' \oplus k')$ from above), for any v in shores 4 and 5. We also have that the parallel completions of the new B_4-brooms created by the processing of $T_{y_1^*}$ can make size(Li(v)) increase by at most 2 for each v. As a necessary technicality, we note that the two increases (from parallel copies of the newly created 4-edges and parallel completions of brooms) do not add up: the shore 4 v's adjacent to an edge in the parallel completion of a B_4-broom cannot also receive parallel copies of the new 4-edges created adjacent to y_3°, by Inv2 and by the fact that the 4-edge $(4, x_4^\circ, y_4^\circ)$ exists at the start of the simulator cycle; the same statement is true, and easier to see, for shore 5 v's by Inv3. (The former use of Inv2 applies, for once, the "full force" of Inv2, i.e., uses the fact that Inv2 is stated for $5\mathcal{Z}$ and not for $3\mathcal{Z}$: one needs, for every new broom handle $(3, x_3, y_3, \rightarrow)$, that the equation $y_3 \oplus k_1 \oplus k_2 = x_4^\circ \oplus k^\circ \oplus k_3 \oplus k_4$ have no solution $k_1, k_2, k_3, k_4 \in \mathcal{Z}$.) Thus $\gamma(B_4)$ increases by at most 2 as the result of the creation of these new 4-edges and of the new B_4-brooms.

After $(3^-, x_3^\circ)$ has been popped from *LeftQueue* and processed, the subsequent processing of $T_{y_1^*}$ occurs like in Case 1. In particular: (i) no new edges are added to B_2; (ii) the 1-edge (x_1°, y_1°), being an old edge that already existed at the start of the simulator cycle, is never put onto *LeftQueue*; (iii) the 2-edge (y_1°, x_3°) doesn't reappear in the processing of $T_{y_1^*}$.

Subsequently, at some point, the 3-edge $(3^+, y_3^\circ)$ is popped from *RightQueue*. At this point the tree $T_{y_3^\circ}$ is processed. In the meanwhile since the start of the simulator cycle, extra 4-edges have potentially been tacked on to y_3° (these are the "new 4-edges" discussed above) with attendant 5-edges; however, since these new 4-edges are already in completed paths, they have no effect when y_3° is processed. (Also note the 5-edges on these paths are already in *RightQueue*, ready to be processed later. For newly added 4-edges adjacent to y_3° these 5-edges are adjacent to nothing else, and they are popped innocuously from *RightQueue*, whereas for non-new 4-edges adjacent to y_3° already at the start of the simulator cycle, this processing can be viewed as part of the processing[24] of $T_{y_3^\circ}$.) Moreover, the 4-edge (y_3°, x_5°) to the "original pebbled leaf" x_5° of $T_{y_3^\circ}$ is also in a completed path, now, so has no effect either. Altogether, therefore, the processing of $T_{y_3^\circ}$ unfolds like a Case 1 processing (albeit "starting" on shore 4), contributing nothing to $\gamma(B_4)$ and contributing at most 1 to $\gamma(B_2)$ (which might increase from 2 to 3). After *RightQueue* has been emptied, *LeftQueue* is "innocuously" emptied one last time (potentially containing the 1-edges in leftward B_2-brooms created by the processing of $T_{y_3^\circ}$), and EmptyQueue() returns.

Basic accounting (that we skip) shows that the total number of queries created by the game during this cycle is at most $3 \cdot \text{size}(T_{y_1^*}) + 3 \cdot \text{size}(T_{y_3^\circ}) \leq 6\gamma(B)$. Finally, the simulator makes at most $\text{size}(T_{y_1^*}) + \text{size}(T_{y_3^\circ})$ calls to E, where we note the extra call to E made during the call ForcedP3(x_3°) can be "billed" to the edge (y_1°, x_3°) of $T_{y_1^*}$, so that at most one call to E occurs per edge of $T_{y_1^*}/T_{x_3^\circ}$.

Subcase 3.2. $y_5^\circ \in range(P_5)$, and $P_5^{-1}(y_5^\circ) \oplus k^\circ \notin range(P_4)$. In this case, when ForcedP3(x_3°) calls P4$^{-1}(y_4^\circ)$ with $y_4^\circ := P_5^{-1}(y_5^\circ) \oplus k^\circ$, a new 4-query $(4, x_4^\circ, y_4^\circ, \leftarrow)$ is created, and ForcedP3() returns $y_3^\circ := x_4^\circ \oplus k^\circ$. We note that, by the "newness" of $(4, x_4^\circ, y_4^\circ, \leftarrow)$, the node y_3° is connected to $x_5^\circ = y_4^\circ \oplus k^\circ$ in B_4, and to nothing else, at this point.

Subsequently, PrivateP3(x_3°) creates the 3-query $(3, x_3^\circ, y_3^\circ)$. The subsequent processing of $T_{y_1^*}$ resembles subcase 3.1. Attentive readers might have noted that the analysis of subcase 3.1 does use in one place (when it comes to upper bounding the total increase so size(Li(v)) for B_4 vertices v) the fact that the 4-query $(4, x_4^\circ, y_4^\circ)$ exists at the start of the simulator cycle, but this argument is easily

[24] The "original pebbled leaf" on $T_{y_3^\circ}$ is x_5°. When these 5-edges are popped the edge to y_3° is ignored because on a completed path, and so the subtree attached to the 5-edge is "cut off" from its original pebbled leaf, and now simply processed as a Case 1 subtree.

replaced by an application of Inv2; readers who have made it this far should be able to fill in the details, which we skip.

We also note that while the node x_5° was potentially in a large component of B_4 to start with, the pebbling of this node prevents the game from ever processing that component (because the edge $(5, x_5^\circ, y_5^\circ)$ is never placed on *RightQueue*). Thus, no pre-existing B_4 component is processed in this case (i.e., *RightQueue* empties innocuously, with nothing happening, like in Case 1), and the game creates at most $3 \cdot \text{size}(T_{y_1^*}) \leq 3\gamma(B)$ new queries before returning. Exactly $\text{size}(T_{y_1^*}) \leq \gamma(B)$ calls to E are made, or one for every edge of $T_{y_1^*}$, and $\gamma(B_2)$ increases by at most 1 while $\gamma(B_4)$ increases by at most 2 like in subcase 3.1.

Subcase 3.3. $y_5^\circ \notin range(P_5)$. In this case ForcedP3($x_3^\circ$) returns $\perp$, despite the pre-existing edge (x_1°, y_1°). Then PrivateP3() creates a new 3-edge $(3, x_3^\circ, y_3^\circ, \rightarrow)$, ProcessNew1Edge() calls P4($y_3^\circ \oplus k'$) and thus creates a new 4-edge $(4, x_4', y_4', \rightarrow)$ with $x_4' := y_3^\circ \oplus k'$, and then creates a new 5-edge (with *fresh* = **true**) $(5, x_5', y_5')$ with $x_5' := y_4' \oplus k'$ and $y_5' = E_{f^{-1}(k')}(x_1' \oplus k')$.

Later, when $(3^-, x_3^\circ)$ is popped from *LeftQueue*, and with $k = k^\circ$ in the "forall" loop of procedure ProcessNew3$^-$Edge(), a particularity occurs in that, since $x_1^\circ \oplus k^\circ \notin$ *LeftFreezer*, the value $y_5^\circ = E_{f^{-1}(k^\circ)}(x_1^\circ \oplus k^\circ)$ is computed with *fresh* = **false** (this value is indeed non-random), and FreezeRightValues(y_5°, k°) is not called. On the other hand, the value $y_4^\circ = \text{P4}(y_3^\circ \oplus k^\circ)$ is randomly chosen at this point (by Inv2 and by the random choice of y_3°), so the new 5-edge $(5, x_5^\circ, y_5^\circ)$ with $x_5^\circ = y_4^\circ \oplus k^\circ$ is only adjacent to one 4-edge, namely the 4-edge (y_3°, x_5°) of label k°. Hence, even while the set of values $\{y_5^\circ \oplus k : k \in \mathcal{Z}\}$ is non-random (and not in *RightFreezer*), this makes no difference when *RightQueue* is emptied because vertex x_5° is only adjacent to one 4-edge, that is in a completed path with the 5-edge $(5, x_5^\circ, y_5^\circ)$ (obviously, no 4-edges with endpoint x_5° have been created in the meanwhile). Thus, in this case, nothing happens when *RightQueue* is emptied of its 5-edges, as in subcase 3.2.

Altogether, $\gamma(B_2)$, $\gamma(B_4)$ increase by at most 1 and 2 respectively like in subcase 3.2. While the query bean-counting is slightly different than in subcase 3.2 (e.g., the 5-query $(5, x_5^\circ, y_5^\circ)$ is now billed to the current query cycle), the game still creates at most $3 \cdot \text{size}(T_{y_1^*}) \leq 3\gamma(B)$ new queries during the query cycle, as can be checked using accounting similar to Case 1. Again the simulator makes exactly $\text{size}(T_{y_1^*}) \leq \gamma(B)$ distinct calls to E (even while it makes one "wasted" call to E in ForcedP3(x_3°): the same call is made again later, anyway), as in subcase 3.2.

Case 3 concludes our analysis of a query to P1^{-1} (or to P5, by symmetry).

Assume now the adversary makes a query $P_3(x_3^*)$, where x_3^* is not adjacent to a 3-edge (or the query returns immediately and there is nothing to discuss). We define $T_{x_3^*} = \text{Li}(x_3^*)$ at the start of the simulator cycle. We divide our analysis into the similar cases and subcases according to whether $T_{x_3^*}$ has a pebbled leaf, etc. (The analysis is altogether very similar to the case of P1^{-1}-queries, and, essentially, there is nothing new, but we review the cases and subcases for completeness.)

Case 4: $T_{x_3^*}$ has no pebbled leaf. At the first-level node x_3^* of the tree (the root), the game adds an edge $(3, x_3^*, y_3^*, \rightarrow, num)$ where y_3^* is chosen randomly. Indeed, because $T_{x_3^*}$ has no pebbled leaves to start with, $P_3(x_3^*) = \perp$ at this point, and ForcedP3(x_3^*) will return $\perp$ when it is called by PrivateP3(x_3^*). The pair $(3^-, x_3^*)$ is added to *LeftQueue* (before this *LeftQueue*, like *RightQueue*, is empty). We point out (since this will shortly play a role), that

$$\{y_3^* \oplus k : k \in \mathcal{Z}\} \cap \text{domain}(P_4) = \emptyset$$

after the processing of this first-level node, by Inv2.

The second-level nodes of the tree $T_{x_3^*}$ are pre-processed when the pair $(3^-, x_3^*)$ is popped off of *LeftQueue* in EmptyQueue(). Let $y_1^1, \ldots, y_1^\ell$ be the children of x_3^* in $T_{x_3^*}$ (these are the "second-level" nodes) and let k^j be the label on the 2-edge (y_1^j, x_3^*). We assume the children $y_1^1, \ldots, y_1^\ell$ are ordered such that $k^1 < \ldots < k^\ell$, and that a loop of the type "forall $k \in \mathcal{Z}$" iterates from the smallest value of $k \in \mathcal{Z}$ to the largest.

When the "forall" loop iterates with $k = k^1$, ExistsPath$(3^-, x_3^*, k^1)$ returns false since y_1^1 is still unpebbled, and CompletePath3$^-(x_3^*)$ is called, which starts by calling PrivateP1$^{-1}(y_1^1)$. Because $T_{x_3^*}$ has no pebbled leaves to start with, $P_1^{-1}(y_1^1) = \perp$ at this point. Thus PrivateP1$^{-1}(y_1^1)$ creates a new 1-edge $(1, x_1^1, y_1^1, \leftarrow)$ by calling ReadTape. At this point the ice tray $\{x_1^1 \oplus k : k \in \mathcal{Z}\}$ is added to *LeftFreezer*. Then ProcessNew3$^-$Edge() creates a new 4-query $(4, y_3^* \oplus k^1, y_4^1, \rightarrow)$ (indeed, this query will be *new* because otherwise its adjacency to the newer query $(3, x_3^*, y_3^*)$ would contradict Inv2), removes $x_1^1 \oplus k^1$ from *LeftFreezer*, and finally creates a new 5-query $(5, y_4^1 \oplus k^1, y_5^1, \perp)$ where $y_5^1 = E_{f^{-1}(k)}(x_1^1 \oplus k^1) \oplus k^1$. We note that since $E_{f^{-1}}(x_1^*, k^1)$ is randomly sampled at this point, y_5^1 has negligible chance of being in range(P_5), and $y_4^1 \oplus k^1$ cannot be in domain(P_5), either, without contradicting Inv2.

As in case 1, the new 3-query $(3, x_3^*, \ldots)$, the $|\mathcal{Z}|$ different parallel 4-edges corresponding to $(4, x_4^1, \ldots)$, and the new 5-query $(5, x_5^1, \ldots)$ form the parallel completion of a self-contained "degree 1 B_4-broom".

ProcessNew3$^-$Edge(x_3^*) pre-processes the other nodes $y_1^2, \ldots, y_1^\ell$ in its "forall" loop similarly. When the forall loop iterates with $k = k^j$, it starts by creating a new query $(1, x_1^j, y_1^j, \leftarrow)$. At this point the ice tray $\{x_1^j \oplus k : k \in \mathcal{Z}\}$ is added to *LeftFreezer* (which does not overlap with previously added ice trays by Inv3). Then it makes the call P4$(P_3(x_3^*) \oplus k^j)$, which results in the creation of a new 4-query $(4, x_4^j, y_4^j, \rightarrow)$ with $x_4^j = y_3^* \oplus k^j$. Next $x_1^j \oplus k^j$ is removed from *LeftFreezer* (indeed, no modification of *LeftFreezer* has occurred since the (recent) creation of the edge $(1, x_1^j, y_1^j, \leftarrow)$) and the value $y_5^j = E_{f^{-1}(k^j)}(x_1^j \oplus k^j)$ is computed. Since $x_1^j \oplus k^j$ is a "fresh" lookup in *ETable*, $y_5^j \notin$ range(P_5) probability 1 (or else AddEQuery would have aborted), and since the 4-query $(4, x_4^j, y_4^j, \rightarrow)$ is new, we have $y_4^j \oplus k^j \notin$ domain(P_5) by Inv2. Thus the game adds a new 5-query $(5, y_4^j \oplus k^j, y_5^j, \perp)$ without aborting, w.h.p..

Carrying out the above process for $j = 1, \ldots, \ell$ has the effect of creating a B_4-broom of degree ℓ, where the, "handle" of the broom is the query $(3, x_3^*, y_3^*, \rightarrow)$ and where edge labels on the broom's fan are $k^1, \ldots, k^\ell$; it is quite easy to check that the broom is self-contained with high probability from the fact that all 3-queries and 4-queries created while processing level 2 and level 3 nodes have direction $\rightarrow$. We can also note that the broom's degree is exactly equal to the degree of the first-level node x_3^* in $T_{x_3^*}$.

We also note that while the newly created 5-edges are placed on *RightQueue*, they will later be "innocuously popped" from the queue, being adjacent to a single 4-edge with whom they are part of a completed path.

Subsequently, even-level nodes (in shore 2) are processed like odd-level nodes in Case 1, and odd-level nodes (in shore 3) are processed like even-level nodes in Case 1. As in Case 1, the net result is the creation of a degree t self-contained B_4-broom for every node in shore 3 of degree t. In addition, a new 1-query is created for every shore 2 node. Nothing happens when *RightQueue* is emptied, given that every 5-edge in *RightQueue* is attached to a single 4-edge on an already-completed path. The same edge- and vertex-based accounting as in Case 1 shows that the game creates at most $3 \cdot \text{size}(T_{x_3^*}) \leq \gamma(B)$ new queries. As in Case 1, also, $\gamma(B_2)$ doesn't increase while $\gamma(B_4)$ increases by at most 2, while at most size$(T_{x_3^*}) \leq \gamma(B)$ calls are made to E.

We next analyze the cases in which the adversary makes a query $P_3(x_3^*)$ for x_3^* in shore 3 not adjacent to a 3-edge, but where the tree $T_{x_3^*} = \text{Li}(x_3^*)$ contains a pebbled leaf. There are two main cases to consider, since the pebbled leaf might be either in an odd or even level of $T_{x_3^*}$. If the pebbled leaf is in an even level, then it is a shore 2 node, and in if it is in an odd level, then it is a shore 3 node.

Case 5. $T_{x_3^*}$ has a pebbled leaf at an odd-level node (i.e., a node in shore 3). Let $x_3^\circ = \overline{u}$ be the pebbled leaf of $T_{x_3^*}$ and let (x_3°, y_3°) be the 3-edge adjacent to that leaf. Note that $x_3^\circ \neq x_3^*$ as x_3^* is not adjacent to a 3-edge when it is queried. Let y_1° be the even-level parent of x_3° in $T_{y_3^\circ}$, and let k' be the label of the edge (y_1°, x_3°) in B_2. Here the analysis is exactly the same as case 2. One can show that when a 4-edge (y_3°, x_5°) of label k' is assumed to exist ("subcase 5.1" for future reference), the game creates at most $3 \cdot \text{size}(T_{x_3^*}) + 3 \cdot \text{size}(T_{x_5^\circ}) \leq 6\gamma(B)$ new queries, where $T_{x_5^\circ} = \text{Li}(x_5^\circ)$ at the start of the simulator cycle, and that the simulator makes at most $\text{size}(T_{x_3^*}) + \text{size}(T_{x_5^\circ}) \leq 2\gamma(B)$ calls to E. As before, $\gamma(B_j)$, $j = 2, 4$ increase by at most 2 (in fact, $\gamma(B_2)$ increases by at most 1 in the current case, since now $\gamma(B_2) \geq 2$ to start with). In subcase 5.2, where such an edge is assumed not to exist, at most $3\text{size}(T_{x_3^*}) \leq 3\gamma(B)$ new queries are made and the simulator makes at most $\text{size}(T_{x_3^*}) \leq \gamma(B)$ calls to E. As in subcase 5.1, $\gamma(B_4)$ increases by at most 2 while $\gamma(B_2)$ increases by at most 1 during the simulator cycle.

Case 6: $T_{x_3^*}$ has a pebbled leaf at an even-level node. Let $y_1^\circ = \overline{u}$ (in shore 2) be the pebbled leaf of $T_{x_3^*}$, and let (x_1°, y_1°) be the 1-edge containing y_1°. Let x_3° be the parent of y_1° in $T_{x_3^*}$ (possibly, $x_3^\circ = x_3^*$). Let k° be the label on the 2-edge (y_1°, x_3°).

On the one hand, suppose that $x_3^\circ = x_3^*$. Then, at this first-level node of the tree (the root), PrivateP3(x_3°) is called. ForcedP3(x_3°) will create a *candidate* consisting of the pair $(k^\circ, x_1^\circ \oplus k^\circ)$; this will be the only candidate, since y_1° is the only pebbled node in $T_{x_3^*}$. On the other hand, suppose that $x_3^\circ \neq x_3^*$. Let y_1' be the parent of x_3° in $T_{x_3^*}$, and let k' be the label on the 2-edge (y_1', x_3°). As the game processes the tree $T_{x_3^*}$, at some point a 1-edge of the form $(1, x_1', y_1')$ is added, pebbling y_1', and the pair $(1^+, y_1')$ is added to *LeftQueue*. When this pair is popped from *LeftQueue*, the "forall" loop in ProcessNew1Edge is eventually executed with $k = k'$; at this point PrivateP3(x_3°) is called. ForcedP3(x_3°) will create a *candidate* consisting of the pair $(k^\circ, x_1^\circ \oplus k^\circ)$; this will be the only candidate, since $x_1' \oplus k'$ is still in *LeftFreezer* at this point (using disjointness of left ice trays), and since y_1° is the only pebbled node in $T_{x_3^*}$. In both cases, after the "forall" loop, ForcedP3(x_3°) evaluates $y_5^\circ := E_{f^{-1}(k^\circ)}(x_1^\circ \oplus k^\circ) \oplus k^\circ$.

From this point on the analysis proceeds exactly the same as in Case 3, and we can in particular subdivide the analysis into subcases 6.1, 6.2 and 6.3 analogous to subcases 3.1, 3.2 and 3.3. In subcase 6.1, where a 5-edge (x_5°, y_5°) and a 4-edge (y_3°, x_5°) of label k° are assumed to exist, the game creates at most $3 \cdot \text{size}(T_{x_3^*}) + 3 \cdot \text{size}(T_{y_3^\circ}) \leq 6\gamma(B)$ new queries and makes at most $\text{size}(T_{x_3^*}) + \text{size}(T_{y_3^\circ}) \leq 2\gamma(B)$ calls to E, while $\gamma(B_4)$ and $\gamma(B_2)$ increase by at most 2 during the simulator cycle (the latter a slight departure from subcase 3.1, since now $\gamma(B_2)$ might initially be 1, i.e., is no longer at least 2 at the start of the simulator cycle). In subcases 6.2 and 6.3, where at least the 4-edge (y_3°, x_5°) of label k° is assumed not to exist at the start of the simulator cycle, at most $3\text{size}(T_{x_3^*}) \leq 3\gamma(B)$ new queries are created, at most $\text{size}(T_{x_3^*})\gamma(B)$ calls to E are made and, as ever, $\gamma(B_2)$, $\gamma(B_4)$ increase by at most 2. □

Corollary 3. *In any execution, we have, for $j \geq 0$,*

- $\gamma(B) \leq 2j + 1$;
- $|KeyQueries| \leq j$;
- $|Queries| \leq 6j^2$;
- $|EQueries| \leq 2j^2$.

after the j-th query is answered.

Proof. The bound on $|KeyQueries|$ is trivial (see Lemma 17). The bound on $\gamma(B)$ follows from Lemma 18, and from the fact that $\gamma(B) = 1$ after 0 queries. The bounds on $|Queries|$ and $|EQueries|$ follow from the same lemmas and the facts that $\sum_{i=1}^{j} 6(2(i-1)+1) = 12\sum_{i=1}^{j} i - 6j = 12(j+1)j/2 - 6j = 6j^2$, $\sum_{i=1}^{j} 2(2(i-1)+1) = 2j^2$. □

Crucially, we note that the proof of Corollary 3 does not depend on the presence of the abort conditions in TallyEQuery and AddQuery related respectively to the growth of the query counters *Eqnum* and *qnum*, as we have completely ignored the existence of those abort conditions until now. The bounds on $|Queries|$ and $|EQueries|$ are trivial to prove when these abort conditions are taken into account but we have not, so far, made any use of these abort conditions. In fact, it is precisely because the proof of Corollary 3 doesn't rely on these abort conditions that we know these abort conditions occur with probability zero in game G_2. (They may occur, however, with nonzero probability in game G_1, which lacks some of the structure[25] of game G_2.)

Bounding The Abort Probability of G_2

To upper bound the probability of abortion we prove individual upper bounds on the probability of various types of abortions, and then apply a union bound. All abort probabilities assume G_2 is interacting with a q-query distinguisher D.

Proposition 2. *The probability that G_2 aborts inside the procedure ReadTape is at most* $36q^4/(2^n - 6q^2)$.

Proof. By design (and by direct inspection of the simulator's code) ReadTape($Table, x, p$) is never called with arguments x, $Table$ such that $Table(x) \neq \perp$, so the procedure's first abort never occurs.

The probability that $Table^{-1}(y) \neq \perp$ where $y = p(x)$ is zero when $Table = ETable[K]$, since $ETable[K]$ contains no "adapted" entries (all entries of $ETable[K]$ come straight from the random tape $p_E[K]$, which encodes a permutation).

When $Table = P_i$ for some i, the probability that $y \in \text{range}(P_i)$ is upper bounded by $|Queries|/(2^n - |Queries|)$, which is at most $6q^2/(2^n - 6q^2)$ by Corollary 3. In turn, the total number of times that ReadTape is called with $Table = P_i$ for some i is also upper bounded by the final size of *Queries*. Taking a union bound over all these calls, we therefore find that the probability of aborting in ReadTape is upper bounded by

$$6q^2 \frac{6q^2}{2^n - 6q^2}$$

as claimed. □

Proposition 3. *The probability that G_2 aborts inside a call to E or E^{-1} not made from within the procedure ExistsPath is at most* $(52q^8 + 26q^6 + 12q^5 + 26q^4)/(2^n - 2q^2)$.

Proof. E and E^{-1} can only abort from within the subcall to AddEQuery. We note that AddEQuery is called each time after a "fresh" value is downloaded by ReadTape from $p_E[K]$ or $p_E[K]^{-1}$, and that this value comes uniformly at random from a set of size at least $2^n - |EQueries|$, which is at least $2^n - 2q^2$ by Corollary 3.

[25] The reader may be confused about why these abort conditions are introduced in the first place, if they play no role in game G_2. To wit, these abort conditions unambiguously bound the simulator's running time and query complexity in game G_1. A query sequence and set of random tapes that leads to the triggering of one of these abort conditions in G_1 will also lead to abortion in G_2 (since G_2 executes like G_1, only with possibly more abortions) albeit the abortion will occur at a different, earlier point in G_2.

If the query to $\mathrm{E}/\mathrm{E}^{-1}$ is made by the adversary then *LeftFreezer*, *RightFreezer* are empty and the probability of abortion is zero. Since we are not considering calls to $\mathrm{E}/\mathrm{E}^{-1}$ made from within ExistsPath (see this lemma's statement) this leaves calls to $\mathrm{E}/\mathrm{E}^{-1}$ made within FinishPath1^+3^-, FinishPath5^-3^+.

Recall that *LeftFreezer*, *RightFreezer* are emptied after each simulator cycle. The total number of times the procedures FreezeLeftValues, FreezeRightValues are called during a simulator cycle is at most the number of 1- and 5-queries created during that simulator cycle, which is upper bounded by $6\gamma(B) \leq 6(2q+1) = 12q + 6 \leq 13q$ by Corollary 3. Each such call adds at most $|\mathcal{Z}| \leq q$ elements to *LeftFreezer*, *RightFreezer* so the maximum size that *LeftFreezer* and *RightFreezer* can grow during a simulator cycle is upper bounded by $13q^2$.

If E is called from within FinishPath1^+3^-, therefore, the probability that $y \in \mathit{RightFreezer}$ is at most $13q^2/(2^n - 2q^2)$ (and the probability that $x \in \mathit{LeftFreezer}$ is zero, since FinishPath1^+3^- explicitly removes this value from the freezer before the call), and similarly the probability that $x \in \mathit{LeftFreezer}$ for a call to E^{-1} made from within FinishPath5^-3^+ is at most $13q^2(2^n - 2q^2)$ (and the probability that $y \in \mathit{RightFreezer}$ is zero). Either way, therefore, the probability that one of the first two abort conditions is triggered is at most $13q^2/(2^n - 2q^2)$ per call to AddEQuery.

By similar considerations, the second abort condition in the "forall" loop has probability at most $|\mathcal{Z}|^2 13q^2/(2^n - 2q^2) \leq 13q^4/(2^n - q^2)$ of occurring, while the innermost abort condition has probability at most $|\mathcal{Z}|^2|\mathit{EQueries}|13q^2/(2^n - 2q^2) \leq 26q^6/(2^n - 2q^2)$ since there are at most $|\mathit{EQueries}|$ pairs (K, z) such that $\mathit{ETable}[K](z) \neq \perp$ and since $|\mathit{EQueries}| \leq 2q^2$ by Corollary 3.

Lastly, the first abort condition in the forall loop has probability at most $|\mathcal{Z}||\mathit{Queries}|/(2^n - 2q^2) \leq 6q^3/(2^n - 2q^2)$ of causing abort.

Since at most $2q^2$ calls to AddEQuery occur altogether during an execution, a union bounds gives that the probability of ever aborting while calling E or E^{-1} from within one of the FinishPath procedures is at most

$$2q^2\Big(\frac{26q^6}{2^n - 2q^2} + \frac{13q^4}{2^n - 2q^2} + \frac{6q^3}{2^n - 2q^2} + \frac{13q^2}{2^n - 2q^2}\Big)$$

as claimed. □

Proposition 4. *The probability that G_2 aborts inside a call to E or E^{-1} made from within the procedure ExistsPath is zero.*

Proof. We note that ExistsPath is called from within ProcessNew1Edge, ProcessNew3^{-1}Edge, etc; more specifically, ExistsPath is called each time an edge is popped from the queue, and for every $k \in \mathcal{Z}$ for that edge (so $|\mathcal{Z}|$ calls to ExistsPath occur per edge popped from the queues).

In a nutshell, to summarize the argument's general idea before going into details, when (say) a 1-edge $(1, x_1, y_1, \leftarrow)$ is popped from *LeftQueue* this occurs as part of the breadth-first-search processing of a live tree of B_2 in which y_1 is a node, for each value $k \in \mathcal{Z}$ one of three things will happen: (i) $y_1 \oplus k \notin \mathrm{domain}(P_2)$, and ExistsPath$(1^+, y_1, k)$ will return without making any calls to $\mathrm{E}/\mathrm{E}^{-1}$, (ii) $x_2 := y_1 \oplus k \in \mathrm{domain}(P_2)$ and $x_3 := P_2(x_2) \oplus k$ is the (necessarily already pebbled) parent of y_1 in the live tree being processed by EmptyQueue, and by design the 2-edge $(2, y_1, x_3)$ is already on a k-completed path[26] created during the processing of x_3 (and, more exactly, during the pre-processing of y_1) so that ExistsPath returns true without aborting (more specifically, the call to E made from within ExistsPath is already in *ETable*, so AddEQuery is never called), (iii) $x_3 := P_2(y_1 \oplus k) \oplus k$ is a child of y_1 in the tree being processed and therefore not yet not pebbled, i.e. not adjacent to a

[26] We emphasize that the definition of "k-completed path" includes the existence of (and consistence of) the relevant calls to $\mathrm{E}/\mathrm{E}^{-1}$. See the definition at the beginning of this appendix.

3-edge, and ExistsPath returns false without ever calling E because a 3-edge is missing. With respect to (ii) it can be useful to note that the FinishPath procedures really do complete well-formed paths: if a call FinishPath$1^+3^-(x_1, x_5, k)$ returns without aborting, then after the call returns there exists a k-completed path containing the vertices x_1 in shore 1 and x_5 in shore 5.

We proceed to check that whenever an edge is popped from *LeftQueue* or *RightQueue* then (the analogue of) (i), (ii) or (iii) occurs for each $k \in Z$. Basically (since (i), (ii), (iii) were stated above for the case of a 1-edge): (i) the value of k leads nowhere (no k-adjacency to a 2-edge for the case of the left queue, no k-adjacency to a 4-edge in a right queue); (ii) the k-adjacent 2-edge or 4-edge is already on a k-completed path, or (iii) the k-adjacent 2-edge or 4-edge leads to an unpebbled node. We rely on the case analysis of Lemma 18 as a backdrop and assume familiarity with the vocabulary therein.

In the case when a "broom hair" 1-edge is popped off *LeftQueue* (as will be created when a live tree of B_4 is processed, cf. Lemma 18) the broom hair is only adjacent to one 2-edge and this 2-edge is on a completed path, so (i) or (ii) occur. If a 1-edge or 3-edge is popped as part of the processing of a "Case 1" tree (referring, here and later, to the numbering of Lemma 18) it is also obvious that (i), (ii) or (iii) occur.

In Case 1 (cf. Lemma 18) it is obvious that (i), (ii) or (iii) occur each time a 1-edge or a 3-edge is popped off of *LeftQueue*, for each $k \in \mathcal{Z}$. Moreover, when a "broom hair" 5-edge is later popped off of *RightQueue* the broom hair is only adjacent to one 4-edge and this 4-edge is on a completed path, so (i) or (ii) occur in this case. The same holds for broom hairs in other cases of Lemma 18.

In subcase 2.1 the only edge of additional interest to be popped is the 5-edge (x_5°, y_5°), which is popped off of *RightQueue* when *RightQueue* is emptied. But since (y_3°, x_5°) is on a k°-completed path and since y_3° is the "original pebbled node" of $\mathrm{Li}(x_5^\circ)$ and since $\mathrm{Li}(x_5^\circ)$ (in particular the pebbling thereof) hasn't been affected since the start of the simulator cycle (except for the pebbling of x_5°), here too it is easy to see that one of (i), (ii) or (iii) occur for every $k \in \mathcal{Z}$. Subcase 2.2 offers nothing new over subcase 2.1.

In subcase 3.1, when the 3-edge (x_3°, y_3°) is popped from *LeftQueue* it is adjacent to two pebbled nodes in shore 2 of B, but both of these 2-edges are on completed paths. When the same edge is later popped from *RightQueue* y_3° is, potentially, adjacent to many different pebbled nodes in shore 5 but the relevant 4-edges are all on completed paths, since the nodes in shore 5 were either x_5° (obviously on a completed path with y_3°) or else were pebbled during this simulator cycle as part of the completion of a path using the edge (x_3°, y_3°). Other edges popped from *LeftQueue* or *RightQueue* are subsumed by previous cases.

Subsequent cases and subcases behave similarly to either Case 1, subcase 2.1 or subcase 3.1. $\square$

Proposition 5. *The probability that G_2 aborts inside a call to f (including the subcall to procedure KeyQueryChecks) is at most* $(72q^{10} + 36q^9 + 36q^5 + q^4 + q^2)/2^n$.

Proof. By Corollary 3 *qnum* never grows larger than $6q^2 + q$, so the last "if" in procedure f actually never causes an abort. (See the comments after Corollary 3.)

It remains to examine the probability that KeyQueryChecks aborts.

The probability that $k \in \mathcal{Z}$ is obviously at most $|\mathcal{Z}|/2^n \leq q/2^n$, so the first line of KeyQueryChecks causes abort with probability at most $q/2^n$ for each query to f.

Since a nontrivial solution to $k_1 \oplus k_2 \oplus k_3 \oplus k_4 = 0$, $k_i \neq k_j$, must involve the newly scheduled subkey k, it is easy to see the second abort condition of KeyQueryChecks has chance at most $|\mathcal{Z}|^3/2^n \leq q^3/2^n$ of causing abort (for each query to f).

The second abort condition has chance at most $|Queries|^2(q^4+1)/2^n \leq 36q^4(q^4+1)/2^n$ of causing abort, as is easy to see. Finally, it is also easy to see that the third abort condition of KeyQueryChecks has chance at most $2|Queries|^2 q^5/2^n \leq 72q^9/2^n$ per query to f.

Adding the above upper bounds and multiplying by q to account for the q total possible queries to f, we find that the overall probability that an abort occurs within f is at most

$$q(72q^9 + 36q^8 + 36q^4 + q^3 + q)/2^n$$

as claimed. □

Proposition 6. *The probability that G_2 aborts inside a call to AddQuery is at most* $(36q^{10}+36q^9)/(2^n - 6q^2)$.

Proof. By Corollary 3 $|Queries| + |KeyQueries|$ never grows larger than $6q^2 + q$ in G_2, so the last line of AddQuery never causes abort (see the comments after Corollary 3). It remains to upper bound the probability that QueryChecks aborts each time it is called.

Assuming a query of direction $\rightarrow$, the probability that the first abort condition of QueryChecks is triggered is at most $|Queries||\mathcal{Z}|^5/(2^n - |Queries|) \leq 6q^7/(2^n - 6q^2)$ (since y is chosen uniformly at random from a set of size at least $2^n - |\text{range}(P_i)| \geq 2^n - |Queries|$) while the probability that the third abort condition is triggered is at most $|Queries||\mathcal{Z}|^6/(2^n - |Queries|) \geq 6q^8/(2^n - 6q^2)$. The case of a query of direction $\leftarrow$ is symmetric, whereas a query of direction $\perp$ cannot cause abort. Therefore, since AddQuery is called at most $6q^2$ times per execution, the total probability of abort occurring in AddQuery is at most

$$6q^2(6q^8 + 6q^7)/(2^n - 6q^2)$$

as claimed. □

Proposition 7. *The probability that G_2 aborts inside a call to FreezeLeftValues or FreezeRightValues is at most* $90q^5/(2^n - 6q^2)$.

Proof. We note that PrivateP1^{-1} (resp. PrivateP5) selects the argument x (resp. y) for FreezeLeftValues (resp. FreezeRightValues) uniformly from a set of size at least $2^n - |Queries| \geq 2^n - 6q^2$, whereas FinishPath1$^+3^-$ (resp. FinishPath5$^-$3$^+$) only calls FreezeRightValues (resp. FreezeLeftValues) when *fresh* = **true** in which case E (resp. E^{-1}) returns a value sampled uniformly at random from a set of size at least $2^n - |EQueries| \geq 2^n - 2q^2$. Hence, in all cases, the argument x or y to FreezeLeftValues and FreezeRightValues comes uniformly at random from a set of size at least $2^n - 6q^2$. We now look in more detail at the probability of abortion of FreezeLeftValues.

The probability that FreezeLeftValues aborts because $x_1 \in (\mathcal{Z}\backslash\{k^\star\}) \oplus \mathit{LeftFreezer}$ is at most $|\mathcal{Z}|13q^2/(2^n - 6q^2) \leq 13q^3/(2^n - 6q^2)$.

Finally, since there are at most $|EQueries| \leq 2q^2$ pairs (K, x) such that $ETable[K](x) \neq \perp$ (and in particular at most $2q^2$ values x such that $ETable[K](x) \neq \perp$ for some K), the probability of aborting in the innermost forall loop is at most $|\mathcal{Z}|2q^2/(2^n - 6q^2) \leq 2q^3/(2^n - 6q^2)$.

Taking into account that FreezeLeftValues, FreezeRightValues are only called after the creation of a new i-query, and since $|Queries| \leq 6q^2$, the overall probability of aborting inside one of these two functions is therefore at most

$$6q^2(13q^3 + 2q^3)/(2^n - 6q^2)$$

per execution. □

Proposition 8. *The probability that G_2 aborts inside a call to ForcedP3 (not counting within subroutine calls to E/E^{-1} and TallyEQuery) called by ForcedP3) is zero.*

Proof. We note, in passing, that the probability of aborting in the call to TallyEQuery is zero by Corollary 3 and that the probability of aborting within calls to E/E^{-1} is accounted for by propositions 3 and 4, but these calls are not the topic of this proposition.

We can assume without loss of generality that $i = 3^-$ in the call to ForcedP3. We note the only abort condition internal to ForcedP3 is the "**if** (*candidate* $\neq \emptyset$) ..." abort condition. In a nutshell, the reason two distinct candidates are never created is that (#1) x_3 will only be k-1-2-adjacent to at most two values x_1 in shore 1 because, more particularly, it is adjacent to at most two pebbled shore 2 vertices of B, these being, if present, (a) its pebbled parent, and (b) the "original pebbled leaf" y_1° of the live tree being processed (notation of Lemma 18), and (#2) the relevant value of $x_1 \oplus k$ is in *LeftFreezer* for the 1-2-adjacency through the pebbled parent, so there only remains, at most, the 1-2-adjacency through the original pebbled leaf. To check all this occurs as just described one can revisit the case analysis of Lemma 18 and note, in particular, that ForcedP3 only ever returns a non-$\perp$ value in Case 3 and Case 6 and that in these cases, moreover, exactly *one* call to ForcedP3 per simulator cycle returns a non-$\perp$ value. (Even more: exactly one call to ForcedP3 per Case 3 or Case 6 simulator cycle results in the creation of a *candidate*.) The details of checking this are left to the reader. □

Proposition 9. *The probability that G_2 aborts inside a call to PrivateP3 or PrivateP3^{-1} is zero.*

Proof. We consider a call to PrivateP3. We note that abort can only occur at one place in PrivateP3, namely if ForcedP3 returns a non-$\perp$ value already in range(P_3). However, as noted in the previous proposition, there is only at most one call to PrivateP3 per simulator cycle for which ForcedP3 returns a non-$\perp$ value, this being in subcases 3.1, 3.2, 6.1 and 6.2 of Lemma 18 (and for the case of an adversarial query to P1^{-1} or P3, *no* calls to PrivateP3^{-1} result in a non-$\perp$ value being returned by ForcedP3^{-1}), and the fact that the values returned by ForcedP3 do not cause abort in these specific calls is actually argued in detail within the proof of Lemma 18. Hence the proposition follows from the proof of Lemma 18. □

Proposition 10. *The probability that G_2 aborts inside of FinishPath1^+3^- or FinishPath5^-3^+ (subroutine calls excluded) is zero.*

Proof. We consider FinishPath1^+3^-. The issue is to show that as long as the call to E does not cause abort we have $x_5 \notin$ domain(P_5) and $y_5 \notin$ range(P_5). In fact, this has already been argued in detail within the case analysis of Lemma 18. Specifically, if the query to E is fresh then $y_5 \notin$ range(P_5) by the abort condition in AddEQuery, and if the call to P4 in CompletePath1^+ or CompletePath3^- defining x_5 is new then $x_5 \notin$ domain(P_5) by Inv2. Cases when the call to E isn't fresh occur in subcase 3.3 and subcase 6.3 of Lemma 18 and are analyzed there (one can note that at most one such "non-fresh" path completion occurs per simulator cycle); cases when the call to P4 doesn't result in a new 4-query occur in subcases 2.1, 3.1, 5.1 and 6.1 of Lemma 18 and are analyzed there. In all events, we refer the reader back to the case analysis of Lemma 18. □

A quick inspection of G_2 shows that propositions 2–10 cover all possible types of abortion in that game. Hence, by a simple union bound, we arrive at a final upper bound for the abort probability of G_2:

Lemma 19. *The probability that G_2 aborts while interacting with a q-query distinguisher D is at most*

$$160q^{10}/2^n$$

for all $q \geq 1$.

Proof. Taking the sum of the upper bounds in propositions 2–10 while noting that $(2^n - 6q^2)^{-1} \geq (2^n - 2q^2)^{-1} \geq (2^n)^{-1}$, we find that the abortion probability is upper bounded by

$$(108q^{10} + 72q^9 + 52q^8 + 26q^6 + 138q^5 + 63q^4 + q^2)/(2^n - 6q^2).$$

For $q \geq 3$ it's easy to check this quantity is at most $150q^{10}/(2^n - 6q^2)$. Moreover for $q = 1$ it is easy to check by inspection that G_2 has, in fact, probability 0 of aborting, and similarly G_2 aborts with probability $1/2^n$ only for $q = 2$, so that in all cases the probability of abortion is at most $150q^{10}/(2^n - 6q^2)$.

Finally, one can note that

$$150q^{10}/(2^n - 6q^2) \leq 160q^{10}/2^n$$

as long as $160q^{10} \leq 2^n$ (using $6q^2 \leq 6q^{10} \leq 6 \cdot 2^n/160$, $2^n - 6q^2 \geq 2^n(1 - 6/160) \geq 2^n(15/16)$), which implies the stated bound. $\square$

D Proof of Lemma 4

Proposition 11. *While $LeftQueue$ is being emptied, every connected component in the graph B_4 remains "pebbled upward"*[27]*. Similarly, every connected component of B_2 remains "pebbled upward" while $RightQueue$ is being emptied. In particular, an unpebbled node in the left (resp. right) shore of B_4 is never adjacent to more than one pebbled node in the right (resp. left) shore of B_4 while $LeftQueue$ is being emptied, and symmetrically for B_2 while $RightQueue$ is being emptied.*

Proof. This follows by inspecting the case analysis in the proof of Lemma 18 (Appendix C), that describes in detail how $LeftQueue$ is emptied. However, in order not to simply "dump" the task of revisiting this case analysis on the reader's shoulders, we re-summarize here the main observations that relate to the current proposition. Case numbers below correspond to the case numbers in Lemma 18 (while we briefly re-summarize the premise each case, the reader is referred to the proof of Lemma 18 for a more complete description):

Case 1: A query $P1^{-1}(y_1^*)$ where $T_{y_1^*}$ has no pebbled leaves. The only changes that occur to B_4 as the result of the emptying of $LeftQueue$ are the addition of "B_4-brooms" and their attendant parallel edges. The connected components in the parallel completions of these brooms are either completely pebbled or not pebbled at all (for the degree 2 or degree 1 brooms created by parallel edges), in accordance with the proposition's conclusion. Moreover, since the broom's apex is pebbled from the start and since the broom's edges are rightward, it is easy to check that each broom remains pebbled upward while edges are added to it.

Subcase 2.1: $T_{y_1^*}$ has an even-level pebbled leaf x_3°, and the 3-edge (x_3°, y_3°) is adjacent to a 4-edge (y_3°, x_5°) of label k', where k' is the label on the 2-edge from x_3° to its parent in $T_{y_1^*}$. We can first observe that the 4-edge (y_3°, x_5°) has direction $\rightarrow$; indeed, if it had direction $\leftarrow$, then the 3-edge (x_3°, y_3°) would have to be a later edge by invariant Inv2, and, by invariant Inv5, these two edges would already be on a common k'-completed path at the start of the simulator cycle. Thus when the 5-edge (x_5°, y_5°) is added, the B_4 component containing x_5° remains upward pebbled because x_5° is right beneath the already-pebbled node y_3° in this component. Nothing else of "of note" happens to B_4 while $LeftQueue$ finishes emptying (in particular, no further changes occur to the connected component containing x_5°). And as explained in Lemma 18, when $(5^-, x_5^\circ)$ is later popped off of $RightQueue$, $T_{x_5^\circ}$ is processed in case 1 fashion, so that the subsequent emptying of $RightQueue$ falls under case 1.

Subcase 2.2: As subcase 2.1, but (x_3°, y_3°) is not adjacent to any 4-edge of label k'. The only difference with subcase 2.1 is that the 4-edge (y_3°, x_5°) of direction $\rightarrow$ and label k' is created on-the-fly by the

[27] Spelled out in full: if $k \in \mathcal{S}$ and $(4, x_4, y_4, dir)$ is a 4-query, then $(dir = \rightarrow \wedge\ y_4 \oplus k \in \text{domain}(P_5))$ implies $x_4 \oplus k \in \text{range}(P_3)$, and $(dir = \leftarrow \wedge\ x_4 \oplus k \in \text{range}(P_3))$ implies $y_4 \oplus k \in \text{domain}(P_5)$.

simulator; the connected component then containing x_5° and y_3° is then still obviously pebbled upward, since the newly added unpebbled node x_5° is a leaf of this component, and parallel copies of this edge also leave their respective components upward pebbled, since each of these only adds a new leaf to the component. Apart from this, subcase 2.2 is identical to subcase 2.1.

Subcase 3.1: (See Lemma 18 for details.) We observe that the 4-edges (y_3°, x_5°) has direction $\leftarrow$. Indeed, if (y_3°, x_5°) had direction $\rightarrow$ this would imply the 5-edge $(5, x_5^\circ, y_5^\circ)$ is later than the 4-edge (y_3°, x_5°) by invariant Inv2, implying that these two edges are already in a k°-completed path at the start of the simulator cycle, a contradiction. Hence, when node y_3° becomes pebbled by the addition of the 3-edge (x_3°, y_3°), the B_4 connected component containing y_3° and x_5° remains pebbled upward, sine y_3° is a child of the already-pebbled node x_5° in this component. The $\leftarrow$-direction of the 4-edge (y_3°, x_5°) also implies that the (potentially newly created) 4-edge (y_3°, x_5') has direction $\rightarrow$. Thus when x_5' subsequently becomes pebbled, the same connected component of B_4 (now potentially augmented with x_5') is still upward pebbled because x_5' is below the already-pebbled node y_3° in this component. When $(3^-, x_3^\circ)$ is popped from *LeftQueue* more 4-edges adjacent to y_3° of direction $\rightarrow$ are potentially created (if not already present), and a path is completed for each; since the nodes in shore 5 that become pebbled as a result are, like x_5', children of the already-pebbled y_3°, the connected component of B_4 that is involved (which is always the component containing y_3°) remains pebbled upward during this process as well. The subsequent emptying of *LeftQueue* unfolds as in case 1. The subsequent emptying of *RightQueue* also occurs as in case 1, as sketched in Lemma 18.

Subcase 3.2: (See Lemma 18 for details.) This case is subsumed by subcase 3.1 in the sense that after the creation of the 4-edge $(4, y_3^\circ, x_5^\circ)$ (of direction $\leftarrow$) we can fall back on the analysis of subcase 3.1.

Subcase 3.3: (See Lemma 18 for details.) In this case all 4-edges created during the emptying of *LeftQueue* are, again, rightward, and in B_4-brooms. While the B_4-broom attached to y_3° has the particularity of having a "non-random" vertex in shore 6, this changes nothing as far as B_4 is concerned, and this case is similar to case 1 (see Lemma 18 for more details).

Cases 4, 5 and 6 present no significant changes from, respectively, cases 1, 2 and 3 above, and we leave their verification to the curious reader. □

Proposition 12. *Consider a good execution D^{G_2}. Then a 1-query of direction $\leftarrow$ never completes a k-chain for $k \in \mathcal{S}$ (referring to the set $\mathcal{S}$ at the time of the 1-query's creation), and likewise a 5-query of direction $\rightarrow$ never completes a k-chain for $k \in \mathcal{S}$.*

Proof. We prove the proposition for 1-queries, the proof for 5-queries being similar. Consider the only times that 1-queries of direction $\leftarrow$ are created: when the adversary makes a query $\mathrm{P1}^{-1}(y_1)$, and when $\mathrm{PrivateP1}^{-1}(y_1)$ is called from within ProcessNew1Edge() or ProcessNew3$^-$Edge(). Obviously, the former type of query cannot complete a k-chain for $k \in \mathcal{S}$, since the 4 other queries in the chain would already be there at the start of the simulator cycle, contradicting Inv5.

For the second type of 1-query of direction $\leftarrow$, say $(1, x_1, y_1, \leftarrow)$ is a 1-query created by a call $\mathrm{PrivateP1}^{-1}(y_1)$ issued when *LeftQueue* is being emptied. For simplicity, assume first that *RightQueue* has never been emptied, i.e., that *LeftQueue* is nonempty when EmptyQueue() is called, and that the 1-query $(1, x_1, y_1)$ is created while *LeftQueue* is emptied for the first time. Then there exists a vertex v in shore 2 or 3 of the graph B such that the adversary's query, at this simulator cycle, was either $\mathrm{P1}^{-1}(v)$ (if v is in shore 2) of $\mathrm{P3}(v)$ (if v is in shore 3), and such that the tree T_v of root v (as it stands at the beginning of the simulator cycle) contains y_1. (We note that $v \neq y_1$, since otherwise the adversary queried $\mathrm{P1}^{-1}(y_1)$, and we are in the first case above.) It is sufficient to show that when the 1-edge $(1, x_1, y_1, \leftarrow)$ is created, there does not exist a path consisting of a 2-edge, a 3-edge, a 4-edge and a 5-edge in B, where the 2-edge and 4-edge have the same label $k \in \mathcal{S}$, and where the 2-edge is incident to y_1. For shortness, we call such a path a "4-path from y_1".

When EmptyQueue() starts emptying $LeftQueue$, the only element in $LeftQueue$ is the newly created edge adjacent at v, and the only pebbled nodes in the tree T_v are v and, potentially, a leaf of T_v. We let x_3 be the parent of y_1 in T_v (possibly, $v = x_3$) and let k' be the label on the 2-edge (y_1, x_3). Let $y_1^1, \ldots, y_1^\ell$ be the other nodes besides y_1 incident to x_3 in T_v, if any (if $x_3 \neq v$, then one of these nodes is actually the parent of x_3 in T_v), and let k^j be label on the 2-edge (y_1^j, x_3). Let u be the "original pebbled leaf" of T_v, if such existed, and, *if* y_1 is adjacent to u, let k_u be the label on the 2-edge (y_1, u).

When the 1-edge $(1, x_1, y_1, \leftarrow)$ is created, the only pebbled nodes in shore 3 that y_1 is adjacent to are x_3 and, potentially, u. Thus any 4-path from y_1 must either pass through u or x_3. We first argue that such a 4-path cannot pass through u, if u exists and is adjacent to y_1. Indeed, let (u, y_3^u) be the 3-edge pebbling u (i.e., $(3, u, y_3^u) \in Queries$), since such an edge exists by assumption. Then at the start of EmptyQueue(), y_3^u might be the endpoint of a 4-edge (y_3^u, x_5^u) of label k_u, but x_5^u cannot be in domain(P_5), as is easy to argue from invariant Inv5, since otherwise there would already be a completed k_u-path going through y_1, a contradiction to the fact that y_1 is initially not pebbled. Moreover, the connected component of B_4 containing y_3^u does not[28] change from the start of the simulator cycle until the parent of u (which is y_1) has an edge created, and this edge is popped from the stack and processed; hence when the edge $(1, x_1, y_1, \leftarrow)$ is created, this component is still intact and there is no 4-path from y_1 using the 2-edge (y_1, u).

Next, it remains to argue that a 4-path going through x_3 and starting at y_1 cannot exist when the edge $(1, x_1, y_1, \leftarrow)$ is created. We note that at some point during this simulator cycle, the call PrivateP3(x_3) was made to create a 3-edge (x_3, y_3). We distinguish two additional cases, according to whether ForcedP3(x_3) returned $\perp$ or not inside this call to PrivateP3(x_3).

If ForcedP3(x_3) returned $\perp$ then the 3-edge (x_3, y_3) has direction $\rightarrow$, and it is easy to see that the only 4-edges adjacent to y_3, if any, at the moment the query $(1, x_1, y_1, \leftarrow)$ is created, have edge labels in the set $\{k^1, \ldots, k^\ell\}$. Since this set does not contain k', a 4-path starting at y_1 and going through x_3 cannot exist.

For the second case, assume ForcedP3(x_3) returned $y_3 \neq \perp$ when PrivateP3(x_3) was called. Then one of the x_3-neighbors $y_1^1, \ldots, y_1^\ell$ had to be initially pebbled, i.e., u had to be one of the nodes $y_1^1, \ldots, y_1^\ell$. (To be more precise, if y_1^j was not initially pebbled, and if a 1-edge (x_1^j, y_1^j) exists when ForcedP3(x_3) is called, then $x_1^j \oplus k^j \in \mathit{LeftFreezer}$ when ForcedP3(x_3) is called, since all 1-queries created during the emptying of $LeftQueue$ have direction $\leftarrow$, and since "left ice trays" are disjoint; hence, the edge (x_1^j, y_1^j) could not cause ForcedP3(x_3) to return a non-$\perp$ value.) For simplicity, assume without loss of generality that $u = y_1^1$. Let $y_5^1 = E_K(y_1^1 \oplus k^1) \oplus k^1$; then y_5^1 was in range(P_5) when ForcedP3(x_3) was called. Let (x_5^1, y_5^1) be the corresponding 5-edge. Then either ForcedP3(x_3) created a 4-query $(4, x_4^1, y_4^1, \leftarrow)$ with $y_4^1 := x_5^1 \oplus k^1$, $x_4^1 = y_3 \oplus k^1$ (the latter being how y_3 becomes defined), or this 4-query already exists. (If it already exists its direction will also be $\leftarrow$, since otherwise it would be in a k^1-completed path, given the (necessarily later) 5-query (x_5^1, y_5^1).) If the 4-query $(4, x_4^1, y_4^1, \leftarrow)$ is created on-the-fly in ForcedP3(x_3) then no other 4-edge adjacent at y_3 will exist except for the 4-edge $(y_3, x_5^1) = (x_4^1 \oplus k^1, y_4^1 \oplus k^1)$ of label k^1, and in particular no 4-edge of label k' adjacent at y_3 will exist; it is also easy to verify that no such 4-edge will be added until the moment the query $(1, x_1, y_1, \leftarrow)$ is created. (All 4-edges adjacent to y_3 that are created in the meanwhile will have labels in the set $\{k^1, \ldots, k^\ell\}$.) Thus, no 4-path starting at y_1 and going through x_3 will exist in this case, and we can

[28] See subcases 2.1 and 2.2 in Lemma 18.

assume the 4-query $(4, x_4^1, y_4^1, \leftarrow)$ was pre-existing in ForcedP3(x_3) and, moreover, that there already existed a 4-edge (y_3, x_5') of label k' adjacent at y_3 (since no 4-edge of label k' adjacent at y_3 will otherwise be added before the creation of $(1, x_1, y_1, \leftarrow)$). While such a 4-edge may exist, it must have direction $\rightarrow$, being the opposite direction of the 4-edge (y_3, x_5^1), and so x_5' cannot be already adjacent to a 5-edge without the edge (y_3, x_5^1) being in a k'-completed path at the start of the simulator cycle (this would be a contradiction, among others, to the fact that PrivateP3(x_3) requires $y_3 \notin \text{range}(P_3)$ in order not to abort). Once again, it is easy to argue that x_5' remains non-adjacent to any 5-edges until $(1, x_1, y_1, \leftarrow)$ is created (the only changes that occur to the B_4 component containing y_3 are the addition of 4-edges whose labels, again, are in the set $\{k^1, \ldots, k^\ell\}$—while the endpoints of these edges in shore 5 become pebbled, these endpoints do not include x_5'). Hence, in this case as well a 4-path starting at y_1 and passing through x_3 cannot exist when the edge $(1, x_1, y_1, \leftarrow)$ is created.

This completes our analysis of the case in which $RightQueue$ has never been emptied prior to the emptying of $LeftQueue$ in which the 1-edge $(1, x_1, y_1, \leftarrow)$ is created. However, when $LeftQueue$ is being emptied after $RightQueue$, the components of B_2 being processed by EmptyQueue() are several disjoint trees whose processing also unfolds disjointly, and if an edge $(1, x_1, y_1, \leftarrow)$ is added at y_1 then y_1 is (obviously) a vertex in one of these disjoint trees. (For more details, see subcases 2.1 and 3.2 in Lemma 18, as well as subcases 5.1 and 6.2. These are all cases when the emptying of $RightQueue$ after $LeftQueue$ will potentially cause new queries to be created; here we are considering the symmetric version of these cases, with $LeftQueue$ and $RightQueue$ reversed.) From this, it is easy to see that this case offers no novelties, and can be analyzed similarly to the previous one. □

Proof (Proof of Lemma 4). By Lemma 2, for every pair (K, x) such that $ETable[K](x) \neq \perp$ at the end of the (non-aborting) execution, there exists a 5-tuple of queries

$$(1, x_1, y_1, dir_1, num_1), (2, x_2, y_2, dir_2, num_2), \ldots, (5, x_5, y_5, dir_5, num_5) \tag{9}$$

that consists of a completed path for $\mathrm{E}(K, x)$. We note that K must have been scheduled (i.e., that the query $f(K)$ must have been made) before the path is completed, since otherwise the simulator would abort.

We first argue that the query with the largest value of num_i is adapted. Note that a query with $dir \in \{\leftarrow, \rightarrow\}$ in the above chain cannot be flanked on either side with queries with lower values on num without contradicting invariant Inv2. Therefore, since a non-adapted query has $dir \in \{\leftarrow, \rightarrow\}$, the queries $(2, x_2, y_2)$, $(3, x_3, y_3)$ and $(4, x_4, y_4)$ cannot be simultaneously the last query added to the chain and also be non-adapted. But if $(1, x_1, y_1)$ or $(5, x_5, y_5)$ are the last query added to the chain they are adapted by Proposition 12, so this proves the claim (i.e., that the query with the largest value of num_i is adapted).

Now let $(K', x') \neq (K, x)$ be a second pair such that $ETable[K](x) \neq \perp$, and let

$$(1, x_1', y_1', dir_1', num_1'), (2, x_2', y_2', dir_2', num_2'), \ldots, (5, x_5', y_5', dir_5', num_5') \tag{10}$$

be the corresponding completed path. We argue that the last query in (10) cannot equal the last query in (9). This is equivalent to showing that every adapted query completes at most one path.

First, say that a 5-query is adapted; then, this occurs when $LeftQueue$ is being emptied. By Proposition 11, if the adapted 5-query is $(5, x_5, y_5)$, then x_5 is adjacent to at most one pebbled node in shore 4 of the graph B right before the query is created. Since the relevant 4-edge has a unique label k, this directly implies the new 5-query can only be completing one path. Adapted 1-queries are treated symmetrically.

The other case to consider is when a 3-query is adapted. Say $(3, x_3, y_3)$ is the new adapted 3-query. If this 3-query is adapted while $LeftQueue$ is being emptied, then Proposition 11 implies that right before the query is created, y_3 is adjacent via a 4-edge to at most one pebbled node in shore 5, and hence, the edge $(3, x_3, y_3)$ can be on at most one new completed path. A similar observation can be made if the query $(3, x_3, y_3)$ is created as a direct result of an adversarial query $\mathrm{P3}(x_3)$ (though, in this case, the same observation could be made as well for x_3). Obviously, a symmetric argument can be made if the 3-query is adapted while *RightQueue* is being emptied, so this completes the proof that an adapted query cannot be the last query of two distinct completed paths.

Since every adapted query is obviously the last query on some completed path (this is easy to directly see from the simulator), this establishes a bijection between adapted queries and completed paths (whereby an adapted query corresponds to the unique path that it completes). Finally, there is a bijection between completed paths and pairs (K, x) such that $ETable[K](x) \neq \perp$ by Lemma 2, which completes the lemma. $\square$

E Indifferentiability of $\mathbf{KA_5}$ with a Permutation-Based Key Derivation

Game G_1' G_2'

Random tapes: $p_0, p_1, \ldots, p_5, \{p_E[K] : K \in \{0,1\}^\kappa\}$

```
public procedure P0(K)
    if P_0(K) ≠ ⊥ return P_0(K)
    k ← ReadTape(P_0, K, p_0(→, ·)) ⊕ K
    KeyQueryChecks(k) // G'_2
    Z ← Z ∪ k
    f^{-1}(k) ← K
    KeyQueries ← KeyQueries ∪ {(K, k, ++qnum)}
    if (qnum > 6q^2 + q) then abort
    return P_0(K)

public procedure P0^{-1}(Q)
    if P_0^{-1}(Q) ≠ ⊥ return P_0^{-1}(Q)
    k ← ReadTape(P_0^{-1}, Q, p_0(←, ·)) ⊕ Q
    KeyQueryChecks(k) // G'_2
    Z ← Z ∪ k
    f^{-1}(k) ← Q ⊕ k
    KeyQueries ← KeyQueries ∪ {(K, k, ++qnum)}
    if (qnum > 6q^2 + q) then abort
    return P_0^{-1}(Q)
```

Game G_4'

Random tapes: $p_0, q_1, \ldots, q_5$

```
public procedure P0(K)
    return p_0(→, K)

public procedure P0^{-1}(Q)
    return p_0(←, Q)
```

Fig. 10: Defining the games G_1', G_2', G_3', G_4'. At left the procedures P0 and $\mathrm{P0}^{-1}$ in games G_1' and G_2', which replace the procedure f of G_1 and G_2, and the random tapes for G_1' and G_2'. At right, the procedures P0, $\mathrm{P0}^{-1}$ in G_4' and the random tapes for G_3', G_4'. Game G_3' is identical to G_3 except the procedure f of G_3 is replaced in G_3' by the procedures P0, $\mathrm{P0}^{-1}$ of G_2' above.

In this appendix we outline how the results of Section 4 can be modified to cover the case when the key scheduling function f is instantiated via a Davies-Meyer transform, i.e., $f(K) = P_0(K) \oplus K$ where P_0 is an independent random permutation and where, in this case, we assume the key space is $\{0,1\}^n$. We note that in this case the simulator does not provide an interface to f; it provides, instead, an interface to P_0 and P_0^{-1} (these procedures are denoted P0 and $\mathrm{P0}^{-1}$ in our code).

Theorem 4. *Let $P_0, \ldots, P_5$ be independent random n-bit permutations. Let D be an arbitrary information-theoretic distinguisher that makes at most q queries. Then there exists a simulator $\mathcal{S}$*

such that

$$\mathrm{Adv}^{\mathrm{indif}}_{KA_5,\mathcal{IC},\mathcal{S}}(D) \le 320 \cdot 6^{10}\left(\frac{q^{10}}{2^n} + \frac{q^4}{2^n}\right) = O\left(\frac{q^{10}}{2^n}\right),$$

where $\mathcal{S}$ makes at most $2q^2$ queries to the ideal cipher $\mathcal{IC}$ and runs in time $O(q^3)$.

We note the bounds of Theorem 4 are exactly the same as those of Theorem 3 (see the reason for this below).

The modified simulator is given by game G_1' in Figure 10. More precisely, G_1' is identical to G_1 except that procedure f is replaced by the two new procedures P0, $\mathrm{P0}^{-1}$; moreover the random oracle tape r_f of G_1 is replaced by a random permutation tape p_0 in G_1', and G_1' has a new pair of tables P_0, P_0^{-1} (that hold a partially defined permutation, similarly to P_1, P_1^{-1}, etc). While the table f does not appear in G_1' we note that f^{-1} still appears in G_1' as a means of mapping subkeys back to master keys. The subsequent games G_2', G_3' and G_4' follow straightforward modifications of G_2, G_3 and G_4, as outlined in Figure 10.

With these changes in place it is easy to adapt the proof of indifferentiability outlined in Section 4.3. While upper bounding the abort probability of G_2', the only affected portion of Appendix C is Proposition 5, which must be replaced by the following statement:

Proposition 13. *The probability that G_2' aborts inside a call to P0 or $P0^{-1}$ (including the subcalls to KeyQueryChecks) is at most $(72q^{10} + 36q^9 + 36q^5 + q^4 + q^2)/(2^n - q)$.*

The proof of Proposition 13 is entirely parallel to that of Proposition 5, and simply relies on the fact that when a fresh lookup is made in the tables $p_0(\rightarrow, \cdot)$ or $p_0(\leftarrow, \cdot)$ the answer comes uniformly at random from a set of size at least $2^n - q$.

Since anyway $1/2^n$ is upper bounded by $1/(2^n - 6q^2)$ in the proof of Lemma 19, and since $1/(2^n - q)$ is likewise upper bounded by $1/(2^n - 6q^2)$, the slight difference between the bounds of propositions 5 and 13 makes no difference for the final abort bound. Namely, the probability of abortion in G_2' is still upper bounded by $160q^{10}/2^n$, like G_2.

Since the explicit randomness has changed form (going from r_f to p_0) the randomness mapping τ of Section 4.3 must be modified, but the modification is trivial: while τ was defined as the identity on r_f here we use a map τ' that is the same as τ except that τ' is the identity on p_0. Other necessary changes to the lemmas and propositions of Section 4.3 are similarly superficial, and a quick revisitation of the proof will show that everything essentially carries through "as-is", with no change in the final indistinguishability bounds, simulator efficiency or query-complexity.