Indifferentiability of 8-Round Feistel Networks

Yuanxi Dai and John Steinberger

Institute for Interdisciplinary Information Sciences, Tsinghua University, Beijing.
shusdtc@gmail.com, jpsteinb@gmail.com

Abstract. We prove that a balanced 8-round Feistel network is indifferentiable from a random permutation. This result comes on the heels of (and is part of the same body of work as) a 10-round indifferentiability result for Feistel network recently announced by the same team of authors [10]. The current 8-round simulator achieves similar security, query complexity and runtime as the 10-round simulator and is not significantly more involved. The security of our simulator is also slightly better than the security of the 14-round simulator of Holenstein et al. [18] for a comparable runtime and query complexity.

Table of Contents

1 Introduction

For many cryptographic protocols the only known analyses are in a so-called *ideal primitive model*. In such a model, a cryptographic component is replaced by an idealized information-theoretic counterpart (e.g., a random oracle takes the part of a hash function, or an ideal cipher substitutes for a concrete blockcipher such as AES) and security bounds are given as functions of the query complexity of an information-theoretic adversary with oracle access to the idealized primitive. Early uses of such ideal models include Winternitz [34], Fiat and Shamir [17] (see proof in [27]) and Bellare and Rogaway [2], with such analyses rapidly proliferating after the latter paper.

Given the popularity of such analyses a natural question that arises is to determine the relative "power" of different classes of primitives and, more precisely, whether one class of primitives can be used to "implement" another. E.g., is a random function always sufficient to implement an ideal cipher, in security games where oracle access to the ideal cipher/random function is granted to all parties? The challenge of such a question is partly definitional, since the different primitives have syntactically distinct interfaces. (Indeed, it seems that it was not immediately obvious to researchers that such a question made sense at all [7].)

A sensible definitional framework, however, was proposed by Maurer et al. [22], who introduce a simulation-based notion of *indifferentiability*. This framework allows to meaningfully discuss the instantiation of one ideal primitive by a syntactically different primitive, and to compose such results. (Similar simulation-based definitions appear in [4, 5, 25, 26].) Coron et al. [7] are early adopters of the framework, and give additional insights.

Informally, given ideal primitives Z and Q, a construction C^Q (where C is some stateless algorithm making queries to Q) is *indifferentiable* from Z if there exists a simulator S (a stateful, randomized algorithm) with oracle access to Z such that the pair (C^Q, Q) is statistically indistinguishable from the pair (Z, S^Z). Fig. 1 (which is adapted from a similar figure in [7]) briefly illustrates the rationale for this definition. The more efficient the simulator, the lower its query complexity, and the better the statistical indistinguishability, the more practically meaningful the

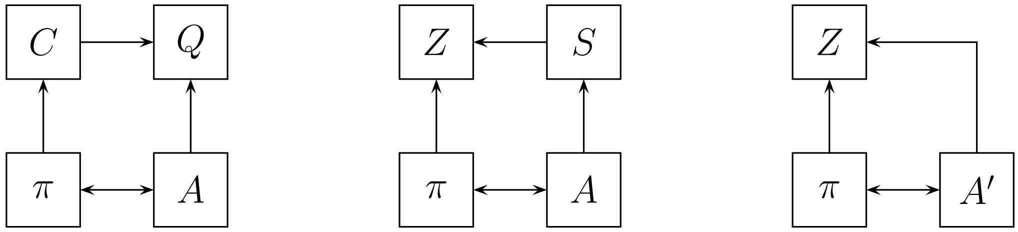

Fig. 1. The cliff notes of indifferentiability, after [7]. (left) Adversary A interacts in a game with protocol π in which π calls a construction C that calls an ideal primitive Q and in which A calls Q directly. (middle) By indifferentiability, the pair (C^Q, Q) can be replaced with the pair (Z, S^Z), where Z is an ideal primitive matching C's syntax, without significantly affecting A's probability of success. (right) Folding S into A gives a new adversary A' for a modified security game in which the "real world" construction C^Q has been replaced by the "ideal world" functionality Z. Hence, a lack of attacks in the ideal world implies a lack of attacks in the real world.

result.

The present paper focuses on the natural question of implementing a permutation from one or more random functions (a small number of distinct random functions can be emulated by a single random function with a slightly larger domain) such that the resulting construction is indifferentiable from a random permutation. This means building a permutation $C : \{0,1\}^{m(n)} \to \{0,1\}^{m(n)}$ where

$$C = C[F_1, \ldots, F_r]$$

depends on a small collection of random functions $F_1, \ldots, F_r : \{0,1\}^n \to \{0,1\}^n$ such that the vector of $r+1$ oracles

$$(C[F_1, \ldots, F_r], F_1, \ldots, F_r)$$

is statistically indistinguishable from a pair

$$(Z, S^Z)$$

where $Z : \{0,1\}^{m(n)} \to \{0,1\}^{m(n)}$ is a random permutation from $m(n)$ bits to $m(n)$ bits, for some efficient simulator S. Thus, in this case, the simulator emulates the random functions $F_1, \ldots, F_r$, and it must use its oracle access to Z to invent answers that make the (fake) random functions $F_1, \ldots, F_r$ look "compatible" with Z, as if Z where really $C[F_1, \ldots, F_r]$. (On the other hand, the simulator does not know what queries the distinguisher might be making to Z.) Here $m(n)$ is polynomially related to n: concretely, the current paper discusses a construction with $m = 2n$.

The construction $C[F_1, \ldots, F_r]$ that we consider in this paper, and as considered in previous papers with the same goal as ours (see discussion below), is an r-round (balanced, unkeyed) *Feistel network*. To wit, given arbitrary functions $F_1, \ldots, F_r : \{0,1\}^n \to \{0,1\}^n$, we define a permutation

$$C[F_1, \ldots, F_r] : \{0,1\}^{2n} \to \{0,1\}^{2n}$$

by the following application: for an input $(x_0, x_1) \in \{0,1\}^{2n}$, values $x_2, \ldots, x_{r+1}$ are defined by setting

$$x_{i+1} = x_{i-1} \oplus F_i(x_i) \tag{1}$$

for $i = 1, \ldots, r$; then $(x_r, x_{r+1}) \in \{0,1\}^{2n}$ is the output of C on input (x_0, x_1). One can observe that C is a permutation since x_{i-1} can be computed from x_i and x_{i+1}, by (1). The value r is the number of *rounds* of the Feistel network. (See, e.g., Fig. 2.)

The question of showing that a Feistel network with a sufficient number of rounds is indifferentiable from a random permutation already has a growing history. Coron, Patarin and Seurin [9] show that an r-round Feistel network cannot be indifferentiable from a random permutation for $r \leq 5$, due to explicit attacks. They also give a proof that indifferentiability is achieved at $r = 6$, but this latter result was found to have a serious flaw by Holenstein et al. [18], who could only prove, as a replacement, that indifferentiability is achieved at $r = 14$ rounds. At the same time, Holenstein et al. found a flaw in the proof of indifferentiability of a 10-round simulator of Seurin's [30] (a simplified alternative to the 6-round simulator of [9]), after which Seurin himself found an explicit attack against his own simulator, showing that the proof could not be patched [31].

In our recent preprint [10] we show that a slight modification of Seurin's 10-round simulator (approximately obtained by switching from a "first-in-first-out" to "last-in-first-out" path completion process) gives a simulator that *is* provably secure, thus bringing the number of rounds that are known to be sufficient for indifferentiability back down to 10. It should be noted that the 14-round simulator of [18] is based on Seurin's 10-round simulator as well, being essentially obtained by adding four "buffer rounds" to flank the two "adapt zones" in the 10-round simulator.

While [10] focuses on the application of a paradigm shift to a pre-existing (but flawed) simulator, the current preprint explores further tweaks and optimizations that can be cleanly and safely carried out in the context of the new paradigm. Specifically, we succeed in building an 8-round version of Seurin's 10-round simulator, so to speak. The security and query complexity of our 8-round simulator is similar to the security and query complexity of the previous 10- and 14-round designs [10, 18], also underscoring the common lineage of these constructions. It remains open whether 6 or 7 rounds might suffice for indifferentiability.

Concerning our optimizations, more specifically, in [10, 18] the "outer detect zone" requires four-out-of-four queries in order to trigger a path completion (the outer detect zone consists of four rounds, these being rounds 1, 2 and $r-1$, r). In the current paper, we optimize by always making the outer detect zone trigger a path completion as soon as possible, i.e., by completing a path whenever three-out-of-four matching queries occur in the outer detect zone. By detecting a little earlier in this fashion, we can move the "adapt zones" on either side by one position towards the left and right edges of the network, effectively removing one round at either end, but things are not quite so simple as two of the four different types of paths detected by the outer detect zone can not make use of the new translated adapt zones, because the translated adapt zones overlap with the query that triggers the path. For these two types of paths (which are triggered by queries at round 2 or at round $r-1$), we use a brand new adapt zone instead, consisting of the middle two rounds of the network. (Rounds 4 and 5, in our 8-round design.) This itself creates a fresh complication, since an adapted query should not trigger a path completion, lest the proof blow up, and since the "middle detect zone" is traditionally made up of rounds 4 and 5 precisely. We circumvent this problem with a fresh trick: We split the middle detect zone into two separate overlapping zones, each of which has *three* rounds: rounds 3, 4, 5 for one zone, rounds 4, 5, 6 for the other; after this change, adapted queries at rounds 4, 5 (and as argued within the proof) do not trigger either of the middle detect zones. The simulator's "termination argument" is slightly affected by the presence of two separate middle detect zones, but not much: one can observe that neither type of middle path detection adds queries at rounds 4 and 5, even though paths triggered by one middle detect zone can trigger a path in the other middle detect zone. Hence, Seurin's original termination argument [30] (used in [10, 18] and in many other contexts since) can go through practically unchanged.

While the entire process of modifications just described might seem (and to a certain extent, is) haphazard, the final 8-round simulator ends up having a highly symmetric structure: It can be abstracted as having four detect zones of three consecutive rounds each, with two "inner zones" (rounds 3, 4, 5 and 4, 5, 6) and two "outer zones" (rounds 1, 2, 8 and 1, 7, 8); each detect zone of three consecutive rounds detects "at either end" (e.g., the detect zone with rounds 3, 4, 5 detects at rounds 3 and 5, etc); the upshot is that each of rounds $1, \ldots, 8$ ends up being a detection point for exactly one of the four three-round detect zones. We refer to Fig. 3 in Section 3. A more leisurely description of our simulator can also be found in Section 3.

Related Work. Recently, and in fact exactly concurrently with [10], Dachman-Soled et al. [19] have published a 10-round indifferentiability result for Feistel networks as well. In order to preserve independence of ideas the current authors have yet not looked at [19].

Before [9], Dodis and Puniya [11] investigated the indifferentiability of Feistel networks in the so-called *honest-but-curious* model, which is incomparable to the standard notion of indifferentiability. They found that in this case, a super-logarithmic number of rounds is sufficient to achieve indifferentiability. Moreover, [9] later showed that super-logarithmically many rounds are also necessary.

Besides Feistel networks, the indifferentiability of many other types of constructions (and particularly hash functions and compression functions) have been investigated. More specifically on the blockcipher side, [1] and [20] investigate the indifferentiability of key-alternating ciphers (with and without an idealized key scheduler, respectively). In a recent eprint note, Dodis et al. [12] investigate the indifferentiability of substitution-permutation networks, treating the S-boxes as independent idealized permutations. As explained in [10], our "LIFO" design philosophy is partly inspired by the simulator in that work.

It should be recalled that indifferentiability does not apply to a cryptographic game for which the adversary is stipulated to come from a special class that does not contain the computational class to which the simulator belongs (the latter class being typically "probabilistic polynomial-time"). See [28].

Finally, Feistel networks have been the subject of a very large body of work in the secret-key (or "indistinguishability") setting, such as in [21, 23, 24, 29] and the references therein.

Paper Organization. In Section 2 we give the few definitions necessary concerning Feistel networks and indifferentiability, and we also state our main result.

In Section 3 we give an hand-wavy overview of our simulator, focusing on high-level behavior. A more technical description of the simulator (starting from scratch, and also establishing some of the terminology used in the proof) is given in Section 4. Section 5 contains the proof itself, starting with a short overview.

2 Definitions and Main Result

Feistel Networks. Let $r \geq 0$ and let $F_1, \ldots, F_r : \{0,1\}^n \to \{0,1\}^n$. Given values $x_0, x_1 \in \{0,1\}^n$ we define values $x_2, \ldots, x_{r+1}$ by

$$x_{i+1} = F_i(x_i) \oplus x_{i-1}$$

for $1 \leq i \leq r$. As noted in the introduction, the application

$$(x_0, x_1) \to (x_r, x_{r+1})$$

defines a permutation of $\{0,1\}^{2n}$. We let

$$\Psi[F_1, \ldots, F_r]$$

denote this permutation. We say that Ψ is an *r-round Feistel network* and that F_i is the *i-th round function* of Ψ.

In this paper, whenever a permutation is given as an oracle, our meaning is that both forward and inverse queries can be made to the permutation. This applies in particular to Feistel networks.

Indifferentiability. A *construction* is a stateless deterministic algorithm that evaluates by making calls to an external set of *primitives*. The latter are functions that conform to a syntax that is specified by the construction. Thus $\Psi[F_1, \ldots, F_r]$ can be seen as a construction with primitives $F_1, \ldots, F_r$. In the general case we notate a construction C with oracle access to a set of primitives Q as C^Q.

A primitive is *ideal* if it is drawn uniformly at random from the set of all functions meeting the specified syntax. A *random function* $F : \{0,1\}^n \to \{0,1\}^n$ is a particular case of an ideal primitive. Such a function is drawn uniformly at random from the set of all functions of domain $\{0,1\}^n$ and of range $\{0,1\}^n$.

A *simulator* is a stateful randomized algorithm that receives and answer queries, possibly being given oracles of its own. We assume that a simulator is initialized to some default state (which constitutes part of the simulator's description) at the start of each experiment. A simulator S with oracle access to an ideal primitive Z is notated as S^Z.

A *distinguisher* is an algorithm that initiates a query-response session with a set of oracles, that has a limited total number of queries, and that outputs 0 or 1 when the query-response session is over. In our case distinguishers are information-theoretic; this implies, in particular, that the distinguisher can "know by heart" the (adaptive) sequence of questions that will maximize its distinguishing advantage. In particular, one may assume without loss of generality that a distinguisher is deterministic.

Indifferentability seeks to determine when a construction C^Q, where Q is a set of ideal primitives, is "as good as" an ideal primitive Z that has the same syntax (interface) as C^Q. In brief, there must exist a simulator S such that having oracle access to the pair (C^Q, Q) (often referred to as the "real world") is indistinguishable from the pair (Z, S^Z) (often referred to as the "simulated world").

In more detail we refer to the following definition, which is due to Maurer et al. [22].

Definition 1. *A construction C with access to a set of ideal primitives Q is (t_S, q_S, ε)-*indifferentiable *from an ideal primitive Z if there exists a simulator $S = S(q)$ such that*

$$\Pr\left[D^{C^Q,Q} = 1\right] - \Pr\left[D^{Z,S^Z} = 1\right] \leq \varepsilon$$

for every distinguisher D making at most q queries in total, and such that S runs in total time t_S and makes at most q_S queries to Z. Here t_S, q_S and ε are functions of q, and the probabilities are taken over the randomness in Q, Z, S and (if any) in D.

As indicated, we allow S to depend on q.[1] The notation

$$D^{C^Q,Q}$$

indicates that D has oracle access to C^Q as well as to each of the primitives in the set Q. We also note that the oracle

$$S^Z$$

offers one interface for D to query for each of the primitives in Q; however the simulator S is "monolithic" and treats each of these queries with knowledge of the others.

Thus, S's job is to make Z look like C^Q by inventing appropriate answers for D's queries to the primitives in Q. In order to do this, S requires oracle access to Z. On the other hand, S doesn't know which queries D is making to Z.

Informally, C^Q is *indifferentiable* from Z if it is (t_S, q_S, ε)-indifferentiable for "reasonable" values of t_S, q_S and for ε negligibly small in the security parameter n. The value q_S in Definition 1 is called the *query complexity* of the simulator.

In our setting C will be the 8-round Feistel network Ψ and Q will be the set $\{F_1, \ldots, F_8\}$ of round functions, with each round function being an independent random function. Consequently, Z (matching C^Q's syntax) will be a random permutation from $\{0,1\}^{2n}$ to $\{0,1\}^{2n}$, queriable (like C^Q) in both directions; this random permutation is notated P in the body of the proof.

Main Result. The following theorem is our main result. In this theorem, Ψ plays the role of the construction C, while $\{F_1, \ldots, F_8\}$ (where each F_i is an independent random function) plays the role of Q, the set of ideal primitives called by C.

Theorem 1. *The Feistel network $\Psi[F_1, \ldots, F_8]$ is (t_S, q_S, ε)-indifferentiable from a random $2n$-bit to $2n$-bit permutation with $t_S = O(q^4)$, $q_S = 32q^4 + 8q^3$ and $\varepsilon = 7400448q^8/2^n$. Moreover, these bounds hold even if the distinguisher is allowed to make q queries to* each *of its 9 (= 8 + 1) oracles.*

The simulator that we use to establish Theorem 1 is described in the two next sections. The three seperate bounds that make up Theorem 1 (for t_S, q_S and ε) are found in Theorems 33, 30 and 97 of sections 5.1, 5.1 and 5.7 respectively.

Miscellaneous Notations. Our pseudocode uses standard conventions from object-oriented programming, including constructors and dot notation '.' for field accesses. (Our objects, however, have no methods save constructors.)

We write $[k]$ for the set $\{1, \ldots, k\}$, $k \in \mathbb{N}$.

The symbol $\perp$ denotes an unitialized or null value (and can be taken to be synonymous with a programming language's **null** value, though we reserve the latter for uninitialized object fields). If T is a table, moreover, we write $x \in T$ to mean that $T(x) \neq \perp$. Correspondingly, $x \notin T$ means $T(x) = \perp$.

[1] This introduces a small amount of non-uniformity into the simulator, but which seems not to matter in practice. While in our case the dependence of S on q is made mainly for the sake of simplicity and could as well be avoided (with a more convoluted proof and a simulator that runs efficiently only with high probability), we note, interestingly, that there is one indifferentiabiliy result that we are aware of—namely that of [14]—for which the simulator crucially needs to know the number of distinguisher queries in advance.

3 High-Level Simulator Overview

In this section we give a somewhat non-technical overview of our 8-round simulator which, like [18] and [10], is a modification of a 10-round simulator by Seurin [30].

Round Function Tables. We recall that the simulator is responsible for 8 interfaces, i.e., one for each of the rounds functions. These interfaces are available to the adversary through a single function, named

$$\mathrm{F}$$

in our pseudocode (see Fig. 4 and onwards), and which takes two inputs: an integer $i \in [8]$ and an input $x \in \{0,1\}^n$.

Correspondingly to these 8 interfaces, the simulator maintains 8 tables, notated $F_1, \dots, F_8$, whose fields are initialized to $\perp$: initially, $F_i(x) = \perp$ for all $x \in \{0,1\}^n$, all $i \in [8]$. (Hence we note that F_i is no longer the name of a round function, but the name of a table. The i-th round function is now $\mathrm{F}(i, \cdot)$. Hopefully this should not cause confusion.) The table F_i encodes "what the simulator has decided so far" about the i-th round function. For instance, if $F_i(x) = y \neq \perp$, then any subsequent distinguisher query of the form $\mathrm{F}(i, x)$ will simply return $y = F_i(x)$. Entries in the tables $F_1, \dots, F_8$ are not overwritten once they have been set to non-$\perp$ values.

The $2n$-bit Random Permutation. Additionally, the distinguisher and the simulator both have oracle access to a random permutation on $2n$ bits, notated

$$\mathrm{P}$$

in our pseudocode (see Fig. 7), and which plays the role of the ideal primitive Z in Definition 1. Thus P accepts an input of the form $(x_0, x_1) \in \{0,1\}^n \times \{0,1\}^n$ and produces an output $(x_8, x_9) \in \{0,1\}^n \times \{0,1\}^n$. P's inverse P^{-1} is also available as an oracle to both the distinguisher and the simulator.

Distinguisher Intuition and Completed Paths. One can think of the distinguisher as checking the consistency of the oracles $\mathrm{F}(1, \cdot)$, …, $\mathrm{F}(8, \cdot)$ with $\mathrm{P}/\mathrm{P}^{-1}$. For instance, the distinguisher could choose random values $x_0, x_1 \in \{0,1\}^n$, construct the values $x_2, \dots, x_9$ by setting

$$x_{i+1} \leftarrow \mathrm{F}(i, x_i) \oplus x_{i-1}$$

for $i = 2, \dots, 9$, and finally check if $(x_8, x_9) = \mathrm{P}(x_0, x_1)$. (In the real world, this will always be the case; if the simulator is doing its job, it should also be the case in the simulated world.) In this case we also say that the values

$$x_1, \dots, x_8$$

queried by the distinguisher form a *completed path*. (The definition of a "completed path" will be made more precise in the next section. The terminology that we use in this section should not be taken too seriously.)

It should be observed that the distinguisher has multiple options for completing paths; e.g., "left-to-right" (as above), "right-to-left" (starting from values x_8, x_9 and evaluating the Feistel network backwards), "middle-out" (starting with some values x_i, x_{i+1} in the middle of the network, and growing a path outwards to the left and to the right), "outward-in" (starting from the endpoints x_0, x_1, x_8, x_9 and going right from x_0, x_1 and left from x_8, x_9), etc, etc. Moreover, the distinguisher can try to reuse the same query for several different paths, can interleave the completion of several paths in a complex manner, and so on.

To summarize, and for the purpose of intuition, one can picture the distinguisher as trying to complete all sorts of paths in a convoluted fashion in order to confuse and/or "trap" the simulator in a contradiction.

THE SIMULATOR'S DILEMMA. Clearly a simulator must to some extent detect which paths a distinguisher is trying to complete, and "adapt" the values along these paths such as to make the (simulated) Feistel network compatible with P. Concerning the latter, one can observe that a pair of missing consecutive queries is sufficient to adapt the two ends of a path to one another; thus if, say,

$$x_0, x_1, x_4, x_5, x_6, x_7, x_8, x_9$$

are values such that

$$F_i(x_i) \neq \bot$$

for $i \in \{1, 4, 5, 6, 7, 8\}$, and such that

$$x_{i+1} = x_{i-1} \oplus F(x_i)$$

for $i \in \{5, 6, 7, 8\}$, and such that

$$\mathrm{P}(x_0, x_1) = (x_8, x_9)$$

and such that

$$F(x_2) = F(x_3) = \bot$$

where $x_2 := x_0 \oplus F_1(x_1)$, $x_3 := F_4(x_4) \oplus x_5$, then by making the assignments

$$F_2(x_2) \leftarrow x_1 \oplus x_3 \tag{2}$$

$$F_3(x_3) \leftarrow x_2 \oplus x_4 \tag{3}$$

the simulator turns $x_1, \ldots, x_8$ into a completed path that is compatible with P. In such a case, we say that the simulator *adapts a path.* The values $F_2(x_2)$ and $F_3(x_3)$ are also said to be *adapted.*

In general, however, if the simulator always waits until the last minute (e.g., until only two adjacent undefined queries are left) before adapting a path, it can become caught in an over-constrained situation whereby several different paths request different adapted values for the same table entry. Hence, it is usual for simulators to give themselves a "safety margin" and to pre-emptively complete paths some time in advance. When pre-emptively completing a path, typical simulators sample all but two values along the path randomly, while "adapting" the last two values as above.

It should be emphasized that our simulator, like previous simulators [9,18,30], makes no distinction between a non-null value $F_i(x_i)$ that is non-null because the distinguisher has made the query $\mathrm{F}(i, x_i)$ or that is non-null because the simulator has set the value $F_i(x_i)$ during a pre-emptive path completion. (Such a distinction seems tricky to leverage, particularly since the distinguisher can know a value $F_i(x_i)$ without making the query $\mathrm{F}(i, x_i)$, simply by knowing adjacent values and by knowing how the simulator operates.) Moreover, the simulator routinely calls its own interface

$$\mathrm{F}(\cdot, \cdot)$$

during the process of path completion, and it should be noted that our simulator, again like previous simulators, makes no difference between distinguisher calls to F and its own calls to F.

One of the basic dilemmas, then, is to decide at which point it is worth it to complete a path; if the simulator waits too long, it is prone to finding itself in an over-constrained situation; if it is too trigger-happy, on the other hand, it runs the danger of creating out-of-control chain reactions of path completions, whereby the process of completing a path sets off another path, and so on. We refer to the latter problem (that is, avoiding out-of-control chain reactions) as the problem of *simulator termination*.

Seurin's 10-Round Simulator. Our 8-round simulator is based on "tweaking" our previous 10-round simulator [10] which is itself based on modifying Seurin's (flawed) 10-round simulator [30]. Unfotunately (and after some failed efforts of ours to find shortcuts) it still seems that the shortest and safest way to understand the 8-round simulator is to start back with Seurin's 10-round simulator, followed by the modifications of [10] and by the "tweaks" that bring the network down to 8 rounds.

In a nutshell, Seurin's simulator completes a path for *every* pair of values (x_5, x_6) such that $F_5(x_5)$ and $F_6(x_6)$ are defined, as well as for every 4-tuple of values

$$x_1,\ x_2,\ x_9,\ x_{10}$$

such that

$$F_1(x_1),\ F_2(x_2),\ F_9(x_9),\ F_{10}(x_{10})$$

are all defined, and such that

$$\mathrm{P}(x_0, x_1) = (x_{10}, x_{11})$$

where $x_0 := F_1(x_1) \oplus x_2$, $x_{11} := x_9 \oplus F_{10}(x_{10})$. By virtue of this, rounds 5 and 6 are called the *middle detect zone* of the simulator, while rounds 1, 2, 9, 10 are called the *outer detect zone*. (Thus whenever a detect zone "fills up" with matching queries, a path is completed.) Paths are adapted either at positions 3, 4 or else at positions 7, 8, as depicted in Fig. 2.

In a little more detail, a function call $\mathrm{F}(5, x_5)$ for which $F_5(x_5) = \perp$ triggers a path completion for every value x_6 such that $F_6(x_6) \neq \perp$; such paths are adapted at positions 3 and 4. Symmetrically, a function call $\mathrm{F}(6, x_6)$ for which $F_6(x_6) = \perp$ triggers a path completion for every value x_5 such that $F_5(x_5) \neq \perp$; such paths are adapted at positions 7 and 8. For the outer detect zone, a call $\mathrm{F}(2, x_2)$ such that $F_2(x_2) = \perp$ triggers a path completion for every tuple of values x_1, x_9, x_{10} such that $F_1(x_1)$, $F_9(x_9)$ and $F_{10}(x_{10})$ are defined, and such that the constraints listed above are satisfied (verifying these constraints thus requires a call to P or P^{-1}); such paths are adapted at positions 3, 4. Paths that are symmetrically triggered by a query $\mathrm{F}(9, x_9)$ are adapted at positions 7, 8. Function calls to $\mathrm{F}(2, \cdot)$, $\mathrm{F}(5, \cdot)$, $\mathrm{F}(6, \cdot)$ and $\mathrm{F}(9, \cdot)$ are the only ones to trigger path completions. (Indeed, one can easily convince oneself that sampling a new value $F_1(x_1)$ or $F_{10}(x_{10})$ can only trigger the outer detect zone with negligible probability; hence, this possibility is entirely ignored by the simulator.) To summarize, in all cases the completed path is adapted at positions that are immediately *next to* the query that triggers the path completion.

To more precisely visualize the process of path completion, imagine that a query

$$\mathrm{F}(2, x_2)$$

has just triggered the second type of path completion, for some corresponding values x_1, x_9 and x_{10}; then Seurin's simulator (which would immediately lazy sample the value $F_2(x_2)$ even before checking if this query triggers any path completions) would (a) make the queries

$$\mathrm{F}(8, x_8), \ldots, \mathrm{F}(6, x_6), \mathrm{F}(5, x_5)$$

to itself in that order, where $x_{i-1} := F_i(x_i) \oplus x_{i+1} = \mathrm{F}(i, x_i) \oplus x_{i+1}$ for $i = 9, \ldots, 6$, and (b) adapt the values $F_3(x_3)$, $F_4(x_4)$ as in (2), (3) where $x_3 := x_1 \oplus F_2(x_2)$, $x_4 := F_5(x_5) \oplus x_6$. In general, some subset of the table entries

$$F_8(x_8), \ldots, F_5(x_5)$$

(and more exactly, a prefix of this sequence) may be defined even before the queries $\mathrm{F}(8, x_8)$, $\ldots, \mathrm{F}(5, x_5)$ are made. The crucial fact to argue, however, is that $F_3(x_3) = F_4(x_4) = \perp$ right before these table entries are adapted.

Extending this example a little, say moreover that $F_6(x_6) = \perp$ at the moment when the above-mentioned query

$$\mathrm{F}(6, x_6)$$

is made. This will trigger another path completion for every value x_5^* such that $F_5(x_5^*) \neq \perp$ at the moment when the query $\mathrm{F}(6, x_6)$ occurs. Analogously, such a path completion would proceed by making (possibly redundant) queries

$$\mathrm{F}(4, x_4^*), \ldots, \mathrm{F}(1, x_1^*), \mathrm{F}(10, x_{10}^*), \mathrm{F}(9, x_9^*)$$

for values $x_4^*, \ldots, x_1^*, x_0^*, x_{11}^*, x_{10}^*, x_9^*$ that are computed in the obvious way (with a query to P to go from (x_0^*, x_1^*) to (x_{10}^*, x_{11}^*), where $x_0^* := F_1(x_1^*) \oplus x_2^*$), before adapting the path at positions 7, 8. The crucial fact to argue would again be that $F_7(x_7^*) = F_8(x_8^*) = \perp$ when the time comes to adapt these table values, where $x_8^* := F_{10}(x_{10}^*) \oplus x_{11}^*$, $x_7^* := x_5^* \oplus F_6(x_6)$.

In Seurin's simulator, moreover, paths are completed on a first-come-first-serve (or FIFO[2]) basis: while paths are "detected" immediately when the query that triggers the path completion is made, this information is shelved for later, and the actual path completion only occurs after all previously detected paths have been completed. In our example, for instance, the path triggered by the query $\mathrm{F}(2, x_2)$ would be adapted before the path triggered by the query $\mathrm{F}(6, x_6)$. The imbroglio of semi-completed paths is rather difficult to keep track of, however, and indeed Seurin's simulator was later found to suffer from a real "bug" related to the simultaneous completion of multiple paths [18, 31].

CHANGES TO SEURIN'S SIMULATOR. For the following discussion, we will say that x_2, x_5 constitute the *endpoints* of a path that is adapted at positions 3, 4; likewise, x_6, x_9 constitute the *endpoints* of a path that is adapted at positions 7, 8. Hence, the endpoints of a path are the two values that flank the adapt zone. We say that an endpoint x_i is *unsampled* if $F_i(x_i) = \perp$ and *sampled* otherwise. Succinctly, our simulator's philosophy is to not sample the endpoints of a path until right before the path is about to be adapted or (even more succinctly!) *to sample randomness at the moment it is needed*. This essentially results in two main differences for our simulator, which are (i) changing the order in which paths are completed and (ii) doing "batch adaptations" of paths, i.e., adapting several paths at once, for paths that happen to share endpoints.

To illustrate the first point, return to the above example of a query

$$\mathrm{F}(2, x_2)$$

that triggers a path completion of the second type with respect to some values x_1, x_9, x_{10}. Then by definition

$$F_2(x_2) = \perp$$

[2] FIFO: First-In-First-Out. LIFO: Last-In-First-Out.

at the moment when the call $\mathrm{F}(2, x_2)$ is made. Instead of immediately sampling $F_2(x_2)$, as in the original simulator, we will keep this value "pending" (the technical term that we use in the proof is "pending query") until it comes time to adapt the path. Moreover, and keeping the notations from the previous example, note that the query

$$\mathrm{F}(6, x_6)$$

will not result in $F_6(x_6)$ being immediately lazy sampled either (assuming, that is, $F_6(x_6) = \perp$) as long as there is at least one value x_5^* such that $F_5(x_5^*) \neq \perp$, since in such a case x_6 is the endpoint of a path-to-be-completed (namely, the path which we notated as $x_1^*, \ldots, x_5^*, x_6, x_7^*, \ldots, x_{10}^*$ above), and, according to our policy, this endpoint must be kept unsampled until that path is adapted. In particular, the value $x_5 = F_6(x_6) \oplus x_7$ from the "original" path *cannot be computed* until the "secondary" path containing x_5^* and x_6 has been completed (or even more: until *all* secondary paths triggered by the query $\mathrm{F}(6, x_6)$ have been completed). In other words, the query $\mathrm{F}(6, x_6)$ "holds up" the completion of the first path. In practical terms, paths that are detected during the completion of another path take precedence over the original path, so that path completion becomes a LIFO process.

Implicitly, we note that our requirement that both endpoints of a path *remain* unsampled until further notice means that both endpoints are *initially* unsampled. For the "starting" endpoint of the path (i.e., where the path is detected) this is obvious, since the path cannot be triggered otherwise, while for the "far" endpoint of the path one can argue, as we do in the proof, that it holds with high probability.

As for "batch adaptations" the intuitive idea is that paths that share unsampled endpoints must be adapted (and in particular have their endpoints lazy sampled) simultaneously. In this event, the group of paths that are collectively sampled[3] and adapted will be an equivalence class in the transitive closure of the relation "shares an endpoint with". Note that paths adapted at 3, 4 can only share their endpoints[4] with other paths adapted at 3, 4, while paths adapted at 7, 8 can only share their endpoints with other paths adapted at 7, 8. Hence the paths in such an equivalence class will, in particular, all have the same adapt zone. Moreover, the batch adaptation of such a group of paths cannot happen at any point in time, but must happen when the group of paths is "stable": none of the endpoints of the paths in the group should currently be a trigger for a path completion that has not yet been detected, or that has started to complete but that has not yet reached its far endpoint. It so turns out, moreover, that the topological structure of such an equivalence class (with endpoints as nodes and paths as edges) will be a tree with all but negligible probability, simplifying many aspects of the simulator and of the proof.

While this describes the (simple) high-level idea of batch adaptations, the implementation details are more tedious. In fact, we follow with a separate and slightly more technical discussion of these details.

Further Details: Pending Queries, Trees, Etc. Whenever a query $\mathrm{F}(i, x_i)$ occurs with $F_i(x_i) = \perp$ and $i \in \{2, 5, 6, 9\}$, the simulator creates a so-called *pending query* at that position, and for that value of x_i. (Strictly speaking, the pending query is the pair (i, x_i).) One can think of a pending query as a kind of "beacon" that periodically[5] checks for new paths to trigger, as per the

[3] In this context we use the verb "sampled" as a euphemism for "have their endpoints sampled".

[4] Recall that the endpoints of a path with adapt zone 3, 4 are x_2 and x_5, and that the endpoints of a path with adapt zone 7, 8 are x_6 and x_9.

[5] Our simulator is not multi-threaded! But this metaphor is still helpful.

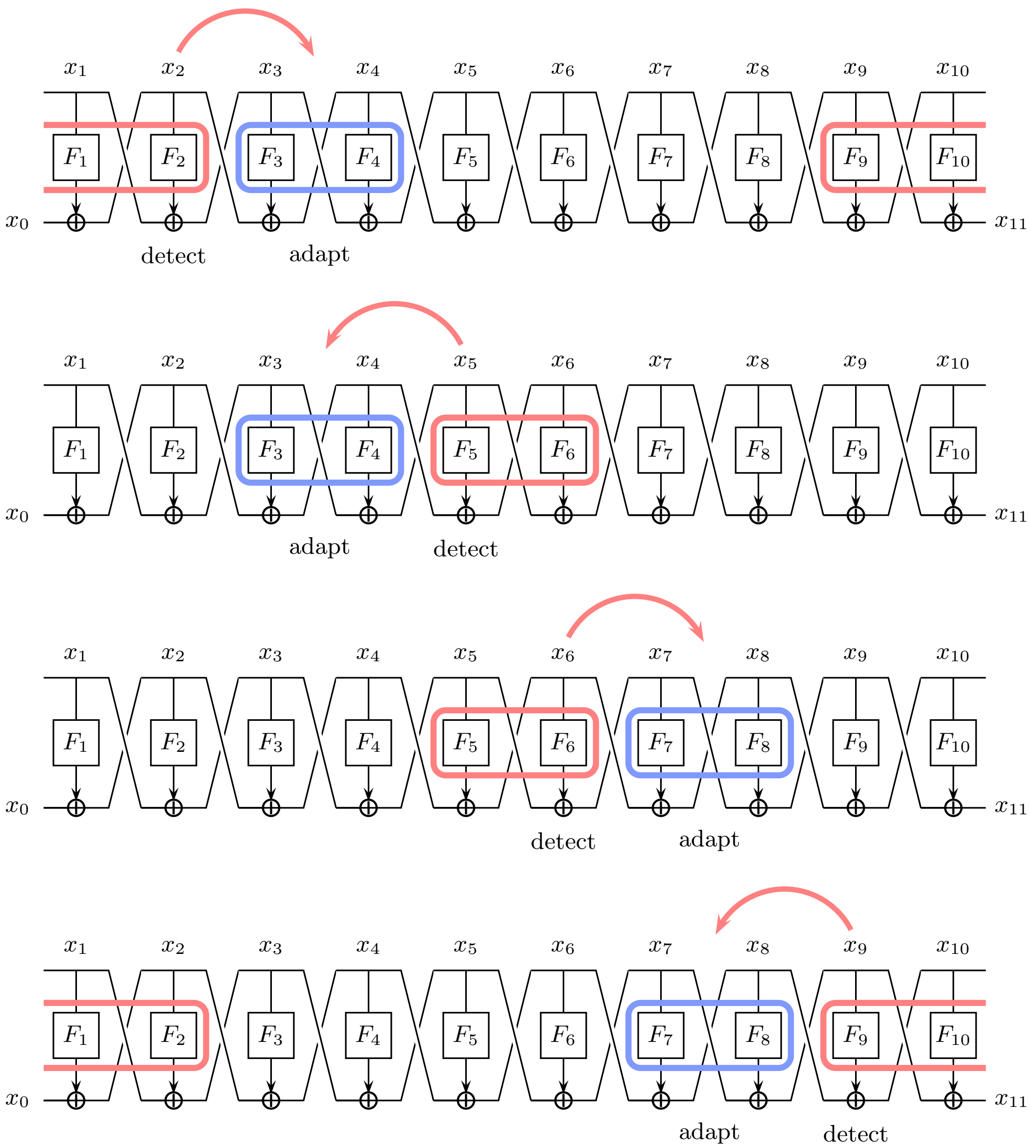

Fig. 2. A sketch of the 10-round simulator from [10] (and also Seurin's 10-round simulator). Rounds 5 and 6 form one detect zone; rounds 1, 2, 9 and 10 form another detect zone; rounds 3 and 4 constitute the left adapt zone, 7 and 8 constitute the right adapt zone; red arrows point from the position where a path is detected (a.k.a., "pending query") to the adapt zone for that path.

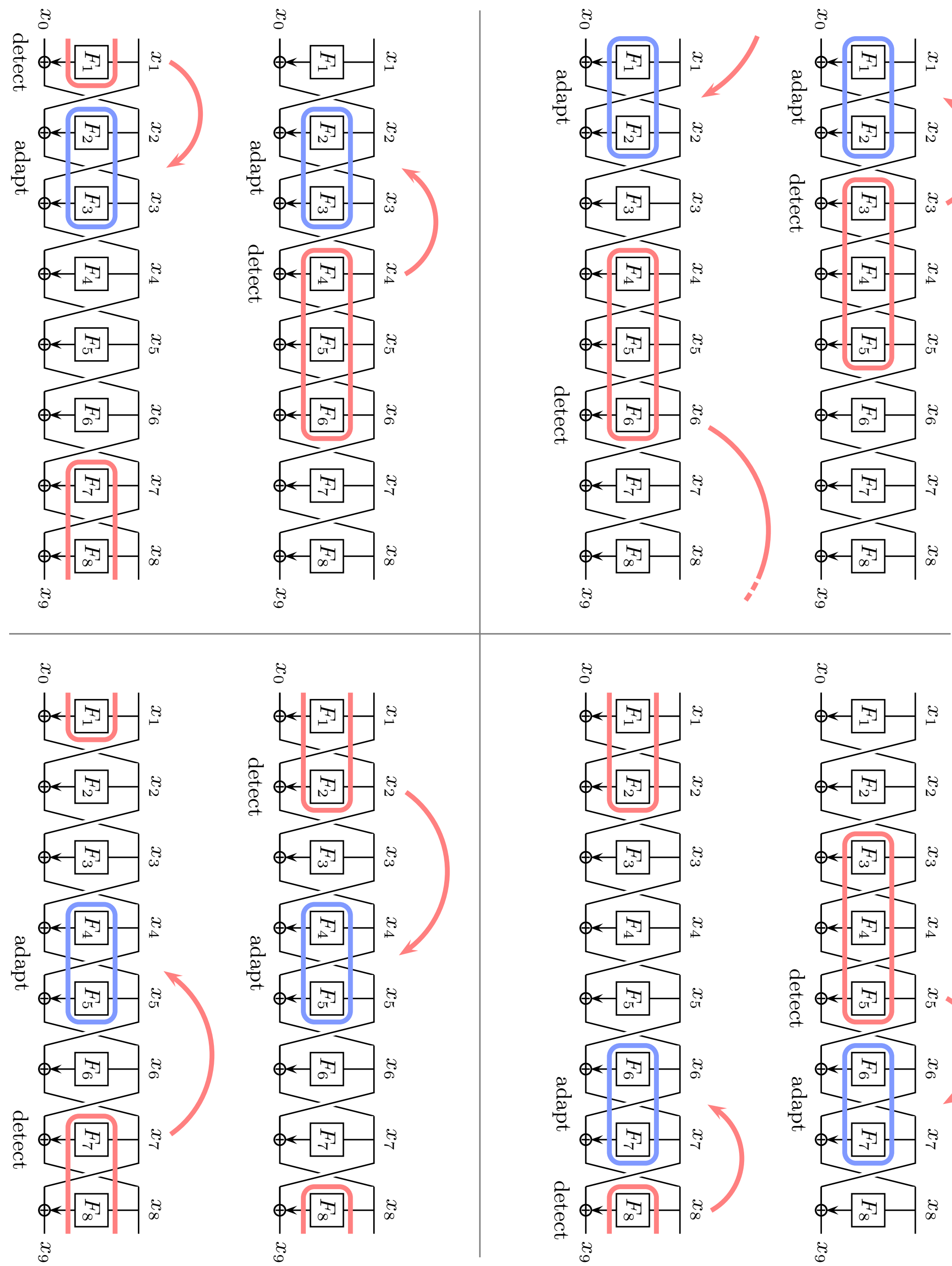

Fig. 3. A sketch of our 8-round simulator drawn in the same style as Fig. 2. Red groups of three queries are detect zones; when a query completing a detect zone (a.k.a., “pending query”) occurs at one of the endpoints of the zone, a path completion is triggered; the adapt zone for that path completion is shown in blue; the four quadrants correspond to the four possible adapt zones. (The adapt zone at positions F_1, F_2 in the upper right quadrant could equivalently be moved to F_7, F_8.)

rules of Fig. 2. E.g., a pending query

$$(2, x_2)$$

will trigger a new path to complete for any tuple of values x_1, x_9, x_{10} such that (same old!)

$$F_1(x_1) \neq \perp, F_9(x_9) \neq \perp, F_{10}(x_{10}) \neq \perp$$

and such that

$$\mathrm{P}(x_0, x_1) = (x_{10}, x_{11})$$

where $x_0 := F_1(x_1) \oplus x_2$, $x_{11} := x_9 \oplus F_{10}(x_{10})$. The tuple of queries x_1, x_9, x_{10} is also called a *trigger* for the pending query $(2, x_2)$. For a pending query $(9, x_9)$, a trigger is a tuple x_1, x_2, x_{10} subject to the symmetric constraints. For a pending query $(5, x_5)$, a trigger is any value x_6 such that $F_6(x_6) \neq \perp$, and likewise any value x_5 such that $F_5(x_5) \neq \perp$ is a trigger for any pending query $(6, x_6)$. We note that a pending query *triggers* a path when there exists a *trigger* for the pending query. Hence there the word "trigger" has two slightly different uses (as a noun and as a verb).

We differentiate the endpoints of a path according to which one triggered the path: the pending query that triggered the path is called the *origin* of the path, while the other endpoint (if and when present) is the *terminal* of the path.

While pending queries are automatically created each time a function call $\mathrm{F}(i, x_i)$ occurs with $F_i(x_i) = \perp$ and with $i \in \{2, 5, 6, 9\}$, the simulator also has a separate mechanism[6] at its disposal for directly creating pending queries without calling $\mathrm{F}(\cdot, \cdot)$. In particular, whenever the simulator reaches the terminal of a path, the simulator turns the terminal into a pending query.

In short: (i) all path endpoints are pending queries, so long as the path has not been sampled and adapted; (ii) pending queries keep triggering paths as long as there are paths to trigger.

For the following, we introduce some extra terminology:

- A path is *ready* when it has been extended to the terminal, and the terminal has been made pending.
- A ready path with endpoints 2, 5 is called a "$(2, 5)$-path", and a ready path with endpoints 6, 9 is called a "$(6, 9)$-path".
- Two ready paths are *neighbors* if they share an endpoint; let a *neighborhood* be an equivalence class of ready paths under the transitive closure of the neighbor relation. We note that a neighborhood consists either of all $(2, 5)$-paths or consists all of $(6, 9)$-paths.
- A pending query is *stable* if it has no "new" triggers (that is, no triggers for which the simulator hasn't already started to complete a path), and if paths already triggered by the pending query are ready.
- A neighborhood is *stable* if all the endpoints of all the paths that it contains are stable.

A neighborhood can be visualized as a graph with a node for each endpoint and an edge for each ready path. As mentioned above, these neighborhoods actually turn out to be trees with high probability. (The simulator aborts otherwise.) We will thus speak of a $(2, 5)$-*tree* for a neighborhood consisting of $(2, 5)$-paths and of a $(6, 9)$-*tree* for a neighborhood consisting of $(6, 9)$-paths.

To summarize, when a query $\mathrm{F}(i, x_i)$ triggers a path completion, the simulator starts growing a tree that is "rooted" at the pending query (i, x_i); for other endpoints of paths in this tree (i.e.,

[6] This might sound a bit ad-hoc right now, but it actually corresponds to the most natural way of programming the simulator, as will become clearer in the technical simulator overview.

besides (i, x_i)), the simulator "plants" a pending query at that endpoint without making a call to $\mathrm{F}(\cdot,\cdot)$, which pending query tests for further paths to complete, and which may thus cause the tree to grow even larger, etc. If and when the tree becomes stable, the simulator samples all endpoints of all paths in the tree, and adapts all these paths.[7]

The growth of a $(2,5)$-tree may at any moment be interrupted by the apparition of a new $(6,9)$-tree (specifically, when a query to $\mathrm{F}(6,\cdot)$ or $\mathrm{F}(9,\cdot)$ triggers a new path completion), in which case the $(2,5)$-tree is put "on hold" while the $(6,9)$-tree is grown, sampled and adapted; vice-versa, a $(6,9)$-tree may be interrupted by the apparition of a new $(2,5)$-tree. In this fashion, a "stack of trees" that alternates between $(2,5)$- and $(6,9)$-trees is created. Any tree that is not the last tree on the stack contains a non-ready path (the one, that is, that was interrupted by the next tree on the stack), and so is not stable. For this reason, in fact, the only tree that can become stable at a given moment is the last tree on the stack.

We also note that in certain cases (and more specifically for pending queries at positions 5 and 6), trees higher up in the stack can affect the stability of nodes of trees lower down in the stack: a node that used to be stable loses its stability after a higher-up tree has been created, sampled and adapted. Hence, our simulator always re-checks all nodes of a tree "one last time" before deeming a tree stable, after a tree stops growing—and such a check will typically, indeed, uncover new paths to complete that weren't there before. Moreover, because the factor that determines when these new paths will be adapted is the timestamp of the *pending query* to which they are attached, rather than the timestamp of the *actual last query* that completed a trigger for this pending query, it is a matter of semantic debate whether our simulator is really "LIFO" or not. (But at least, we personally tend to think of our simulator as a LIFO simulator.)

Afterthoughts: Structural vs. Conceptual Changes. Of the main changes that we introduce to Seurin's simulator, we note that "batch adaptations" are in some sense a conceptual convenience. Indeed, one way or another every non-null value

$$F_j(x_j)$$

for $j \notin \{3,4,7,8\}$ ends up being randomly and independently sampled in our simulator, as well as in Seurin's; so one might as well load a random value into $F_j(x_j)$ as soon as the query $\mathrm{F}(j, x_j)$ is

[7] In more detail, when a tree becomes stable the simulator lazy samples

$$F_i(x_i)$$

for every endpoint (a.k.a., pending query) in the tree. Then if the tree is, say, a $(2,5)$-tree, the simulator can compute the values

$$x_3 := x_1 \oplus F_2(x_2)$$
$$x_4 := F_5(x_5) \oplus x_6$$

and set

$$F_3(x_3) := x_2 \oplus x_4$$
$$F_4(x_4) := x_3 \oplus x_5$$

for each path in the tree. If two paths "collide" by having the same value of x_3 or x_4 the simulator aborts. Likewise the simulator aborts if either $F_3(x_3) \neq \perp$ or $F_4(x_4) \neq \perp$ for some path, before adapting those values. We call this two-step process "sampling and adapting" the $(2,5)$-tree. The process of sampling and adapting a $(6,9)$-tree is analogous.

made, as in Seurin's original simulator, as long as we take care to keep on completing paths in the correct order. While correct, this approach is conceptually less convenient, because the "freshness" of the random value $F_j(x_j)$ is harder to argue when that randomness is needed (e.g., to argue that adapted queries do not collide, etc). In fact, our simulator is an interesting case where the search for a syntactically convenient usage of randomness naturally leads to structural changes that turn out to be critical for correctness.

We also point out that the idea of batch adaptations already appears explicitly in the simulator of [12], and which indeed formed part of the inspiration for our own work. In [12], however, batch adaptations are purely made for conceptual convenience, and not for structural reasons.

Readers seeking more concrete insights can consult Seurin's attack against his own 10-round simulator [31] and check this attack fails when the simulator is switched to ours.

8-ROUND SIMULATOR. In the 10-round simulator, the outer detect zone is in some sense unnecessarily large: for any set of four matching queries that complete the outer detect zone, the simulator can "see" the presence of matching queries already by the third query.

To wit, say the distinguisher chooses random values x_0, x_1, makes the query

$$(x_{10}, x_{11}) \leftarrow \mathrm{P}(x_0, x_1)$$

to P, then queries $\mathrm{F}(1, x_1)$ and $\mathrm{F}(10, x_{10})$. At this point, even if the simulator knows that the values x_1 and x_{10} are related by some query to P, the simulator has no hope of finding *which* query to P, because there are exponentially many possibilities to try for x_0 and/or x_{11}. However, as soon as the distinguisher makes either of the queries

$$\mathrm{F}(2, x_2) \qquad \text{or} \qquad \mathrm{F}(9, x_9)$$

where $x_2 := x_0 \oplus \mathrm{F}(1, x_1)$, $x_9 := \mathrm{F}(10, x_{10}) \oplus x_{11}$, then the simulator has enough information to draw a connection between the queries being made at the left- and right-hand sides of the network. (E.g., if the query $\mathrm{F}(2, x_2)$ is made, the simulator can compute x_0 from $F_1(x_1)$ and x_2, can call $\mathrm{P}(x_0, x_1)$, and recognize, in P's output, the value x_{10} for which it has already answered a query.) More generally, anytime the distinguisher makes three-out-of-four matching queries in the 10-round outer detect zone, the simulator has enough information to reverse-engineer the relevant query to $\mathrm{P}/\mathrm{P}^{-1}$ and, thus, to see a connection between the queries being made at either side of the network.

This observation motivates the division of the 4-round outer detect zone into two separate outer detect zones of three (consecutive) rounds each. In the eight-round simulator, then, these two three-round outer detect zones are made up of rounds 1, 2, 8 and rounds 1, 7, 8, respectively. Both of these detect zones detect "at the edges" of the detect zone. I.e., the 1, 7, 8 detect zone might trigger a path completion through queries to $\mathrm{F}(7, \cdot)$ and $\mathrm{F}(1, \cdot)$, whereas the 1, 2, 8 detect zone might trigger a path completion through queries to $\mathrm{F}(2, \cdot)$ or to $\mathrm{F}(8, \cdot)$. (Once again the possibility of "completing" a detect zone by a query at the middle of the detect zone is ignored because this event has negligible chance of occuring.)

E.g., a query

$$\mathrm{F}(7, x_7)$$

such that $F_7(x_7) = \bot$ and for which there exists values x_0, x_1, x_8 such that $F_8(x_8) \neq \bot$, $F_1(x_1) \neq \bot$, and such that $\mathrm{P}^{-1}(x_8, x_9) = (x_0, x_1)$ where $x_9 = x_7 \oplus F_8(x_8)$ would trigger the 1, 7, 8 detect zone, and produce a path completion. Similarly, a query

$$\mathrm{F}(1, x_1)$$

such that $F_1(x_1) = \perp$ and for which there exists values x_0, x_7, x_8 such that $F_7(x_7) \neq \perp$, $F_8(x_8) \neq \perp$, and such that $\mathrm{P}^{-1}(x_8, x_9) = (x_0, x_1)$ where $x_9 = x_7 \oplus F_8(x_8)$ would trigger the 1, 7, 8 detect zone as well.

When a path is detected at position 1 or at position 8, we can respectively adapt the path at positions 2, 3 or at positions 6, 7—i.e., we adapt the path in an adapt zone that is immediately adjacent to the position that triggered the path completion, as in the 10-round simulator. However, for paths detected at positions 2 and 7, the same adapt zones cannot be used, and we find it more convenient to adapt the path at rounds 4, 5, as depicted in the bottom left quadrant of Fig. 2.

To keep the proof manageable, however, one of the imperatives is that an "adapted" query should not trigger a new path completion. If we kept the middle detect zone as rounds 4, 5 only (by analogy with the 10-round simulator, where the middle detect zone consists of rounds 5 and 6), then the queries that we adapt at rounds 4, 5 would trigger new paths completions of themselves—a mess! However, this problem can be avoided by splitting the middle detect zone into two *enlarged* middle detect zones of three queries each: one middle detect zone consisting of rounds 3, 4, 5 and one consisting of rounds 4, 5, 6. As before, each of these zones detects "at the edges". After this change, bad dreams go away, and the 8-round simulator recovers essentially the same functioning as the 10-round simulator. The sum total of detect and adapt zones, including which adapt zone is used for paths detected at which point, is shown in Fig. 3.

The 8-round simulator utilizes the same "pending query" mechanism as the 10-round simulator. In particular, now, each query

$$\mathrm{F}(j, x_j)$$

with $F_j(x_j) = \perp$ creates a new pending query (j, x_j), because paths are now detected at all positions, and each pending query will detect for paths as depicted[8] in Fig. 3, with there being exactly one type of "trigger" for each position j. A path triggered by a pending query is first extended to a designated terminal (the "other" endpoint of the path), the position of which is a function of the pending query that triggered the path (this position is shortly to be discussed), which becomes a new pending query of its own, etc. As in the 10-round simulator, the simulator turns the terminal into a pending query without making a call to $\mathrm{F}(\cdot, \cdot)$.

For the 10-round simulator, we recall that the possible endpoint positions of a path are 2, 5 and 6, 9. The 8-round simulator has more variety, as the endpoints of a path do not always directly flank the adapt zone for that path. Specifically:

- paths detected at positions 1 and 4, as in the top left quadrant of Fig. 3, have endpoints 1, 4; before such paths are adapted, they include only the values x_1, x_4, x_5, x_6, x_7, x_8
- paths detected at positions 3 and 6, as in the top right quadrant of Fig. 3, have endpoints 3, 6; before such paths are adapted, they include only the values x_3, x_4, x_5, x_6
- paths detected at positions 2 and 7, as in the bottom left quadrant of Fig. 3, have endpoints 2, 7; before such paths are adapted, they include only the values x_1, x_2, x_7, x_8
- paths detected at positions 5 and 8, as in the bottom right quadrant of Fig. 3, have endpoints 5, 8; before such paths are adapted, they include only the values x_1, x_2, x_3, x_4, x_5, x_8

Hence, paths with endpoints 1, 4 or 5, 8 are familiar from the 10-round simulator. (Being the

[8] To solidify things with some examples, a "trigger" for a pending query $(5, x_5)$ is a pair values of x_3, x_4 such that $F_3(x_3) \neq \perp$, $F_4(x_4) \neq \perp$ and such that $x_3 \oplus F_4(x_4) = x_5$, corresponding to the rightmost, bottommost diagram of Fig. 3; a "trigger" for a pending query $(1, x_1)$ is pair of values x_7, x_8 such that $F_7(x_7) \neq \perp$, $F_8(x_8) \neq \perp$, and such that $\mathrm{P}^{-1}(x_8, x_9) = (*, x_1)$ where $x_9 := x_7 \oplus F_8(x_8)$, corresponding to the leftmost, topmost diagram of Fig. 3. Etc.

analogues, respectively, of paths with endpoints 2, 5 or 6, 9.) On the other hand, paths with endpoints 3, 6 or 2, 7 are shorter, containing only four values before the adaptation takes place. As in the 10-round simulator, we speak of an "(i, j)-path" for paths with endpoints i, j. We also say that a path is *ready* once it has reached both its endpoints and these have been turned into pending queries, and that two ready paths are *neighbors* if they share an endpoint.

Since, by virtue of the endpoint positions, a $(1, 4)$-path can only share an endpoint with a $(1, 4)$-path, a $(2, 7)$-path can only share an endpoint with a $(2, 7)$-path, a $(3, 6)$-path can only share an endpoint with $(3, 6)$-path, and a $(5, 8)$-path can only share an endpoint with a $(5, 8)$-path, neighborhoods (which are the transitive closure of the neighbor relation) are always comprised of the same kind of (i, j)-path. As in the 10-round simulator, these neighborhoods are actually topological trees, giving rise, thus, to "$(1, 4)$-trees", "$(2, 7)$-trees", "$(3, 6)$-trees" and "$(5, 8)$-trees". Given this, the 8-round simulator functions entirely analogously to the 10-round simulator, only with more different types of paths and of trees (which does not make an important difference) and with a slightly modified mechanism for adapting $(2, 7)$- and $(3, 6)$-trees, which are the trees for which the path endpoints are not directly adjacent to the adapt zone (which does not make an important difference either).

To wit, concerning the latter point, when a $(2, 7)$- or $(3, 6)$-tree is adapted, some additional queries have to be lazy sampled for each path before reaching the adapt zone. (In the case of a $(3, 6)$-tree, each path even requires a query to P^{-1}.) But because the endpoints of each path are lazy sampled as the first step of the batch adaptation process, there is negligible chance that these extra queries will trigger a new path completion. So for those queries the 8-round simulator directly lazy samples the tables F_i without even calling its own $\mathrm{F}(\cdot, \cdot)$ interface.

As a small piece of trivia (since it doesn't really matter to the simulator), one can check, for instance, that a $(1, 4)$-tree may be followed either by a $(2, 7)$-, $(3, 6)$-, or a $(5, 8)$-tree on the stack—i.e., while making a $(1, 4)$-path ready, we may trigger any of the other three types of paths—and symmetrically the growth of a $(5, 8)$-tree may be interrupted by any of the three other types of trees. On the other hand, $(2, 7)$-trees and $(3, 6)$-trees have shorter paths, and in fact when such trees are grown *no* queries to $\mathrm{F}(\cdot, \cdot)$ are made, which means that such trees never see their growth interrupted by anything. In other words, a $(3, 6)$- or $(2, 7)$-tree will only appear as the last tree in the tree stack, if at all.

Overall, it is imperative that pending queries be kept *unsampled* until the relevant tree becomes stable, and is adapted. In particular, the simulator must not overwrite the pending queries of trees lower down in the tree stack while working on the current tree.

In fact, our simulator *cannot* overwrite pending queries because it keeps a list of all pending queries, and aborts rather than overwrite a pending query. Nonetheless, one must show that the chance of such an event is negligible. The analysis of this bad event is lengthy but also straightforward. Briefly, this bad event can only occur if ready and non-ready paths arrange to form a certain type of cycle, and the occurence of such cycles can be reduced to the occurence of a few different "local" bad events whose (negligible) probabilities are easily bounded.

The Termination Argument. The basic idea of Seurin's termination argument (which only needs to be lightly adapted for our use) is that each path detected in one of the outer detect zones is associated with high probability to a P-query previously made by the distinguisher. Since the distinguisher only has q queries total, this already implies that the number of path completions triggered by the outer detect zones is at most q with high probability.

Secondly, whenever a path is triggered by one of the middle detect zones, this path completion does not add any new entries to the tables F_4, F_5, that would not have been added had the path completion not occured. Hence, only two mechanisms add entries to the tables F_4 and F_5: queries directly made by the distinguisher and path completions triggered by the outer detect zones. Each of these accounts for at most q table entries in each of F_4, F_5, so that the tables F_4, F_5 do not exceed size $2q$. But *every* completed path corresponds to a *unique* pair of entries in F_4, F_5. (I.e., no two completed paths have the same x_4 *and* the same x_5.) So the total number of paths ever completed is at most $(2q)^2 = 4q^2$.

4 Technical Simulator Description and Pseudocode Overview

In this section we "reboot" the simulator description, with a view to the proof of Theorem 1. A number of terms introduced informally in Section 3 are given precise definitions here. As already admonished, indeed, the provisory definitions and terminology of Section 3 should not be taken seriously as far as the main proof is concerned.

The pseudocode describing our simulator is given in Figs. 4–6, and more specifically by the pseudocode for game G_1, which is the simulated world. In Fig. 4 one finds the function F (to be called with an argument $(i, x) \in [8] \times \{0,1\}^n$), which is the simulator's only interface to the distinguisher. The random permutation P and its inverse P^{-1}—which are the other interfaces available to the distinguisher—can be found on the left-hand side of Fig. 7, which is also part of game G_1.

Our pseudocode uses *explicit random tapes*, similarly to [18]. On the one hand there are tapes $f_1, ..., f_8$ where f_i is a table of 2^n random n-bit values for each $1 \le i \le 8$, i.e., $f_i(x)$ is a uniform independent random n-bit value for each $1 \le i \le 8$ and each $x \in \{0,1\}^n$. Moreover there is a tape $p : \{0,1\}^{2n} \to \{0,1\}^{2n}$ that implements a random permutation from $2n$ bits to $2n$ bits. The inverse of p is accessible via p^{-1}. The only procedures to access p and p^{-1} are P and P^{-1}.

As described in the previous section, the simulator maintains a table $F_i : \{0,1\}^n \to \{0,1\}^n$ for the i-th round function, $1 \le i \le 8$. Initially, $F_i(x) = \perp$ for all $1 \le i \le 8$ and all $x \in \{0,1\}^n$. The simulator fills the tables F_i progressively, and never overwrites a value $F_i(x)$ such that $F_i(x) \neq \perp$. If a call to $F(i, x)$ occurs and $F_i(x) \neq \perp$, the call simply returns $F_i(x)$.

The permutation oracle P/P^{-1} also maintains a pair of private tables T/T^{-1} that encode a subset of the random tapes p/p^{-1}. We refer to Fig. 7 for details (briefly, however, the tables T/T^{-1} remember the values on which P/P^{-1} have already been called). These tables serve no tangible purpose in G_1, where P/P^{-1} implement black-box two-way access to a random permutation, but they serve a role subsequent games, and they appear in some of the definitions below.

In certain situations, and following [1], our simulator explicitly aborts ('**abort**'). In such cases the distinguisher is notified of the abort and the game ends.

In order to describe the operation of the simulator in further detail we introduce some more terminology.

A *query cycle* is the portion of simulator execution from the moment the distinguisher makes a query to $F(\cdot, \cdot)$ until the moment the simulator either returns a value to the distinguisher or aborts. A query cycle is *non-aborted* if the simulator does not abort during that query cycle.

A *query* is a pair $(i, x) \in [8] \times \{0,1\}^n$. The value i is the *position* of the query.

A query (i, x) is *defined* if $F_i(x) \neq \perp$. Like many other predicates defined below, this is a time-dependent property.

Our simulator's central data type is a **Node**. (See Fig. 4.) Nodes are arranged into *trees*. A node n is the *root* of its tree if and only if $n.parent = \textbf{null}$. Node b is the *child* of node a if and only if $b \in a.children$ and if and only if $b.parent = a$. Each tree has a root.

Typically, several disjoint trees will coexist during a given query cycle. Distinct trees are never brought to merge. Moreover, new tree nodes are only added beneath existing nodes, as opposed to above the root. (Thus the first node of a tree to be created is the root, and this node remains the root as long as the tree exists.) Nodes are never deleted from trees, either. However, a tree is "lost" once the last reference to the root pops off the execution stack, at which point we say that the tree and its nodes have been *discarded*. Instead of garbage collecting discarded nodes, however, we assume that such nodes remain in memory somewhere, for convenience of description within the proof. Thus, once a node is created it is not destroyed, and we may refer to the node and its fields even while the node has no more purpose for the simulator.

Besides the parent/child fields, a node contains a *beginning* and an *end*, that are both queries, possibly **null**, i.e., $\mathit{beginning}, \mathit{end} \in \{[8] \times \{0,1\}^n, \textbf{null}\}$. (The *beginning* and *end* fields correspond to the "endpoints" of a path, mentioned in Section 3.)

The *beginning* and *end* fields are never overwritten after they are set to non-**null** values. A node n such that $n.end \neq \textbf{null}$ is said to be *ready*, and a node cannot have children unless it is ready. The root n of a tree has $n.\mathit{beginning} = \textbf{null}$, while a non-root node n has $n.\mathit{beginning} = n.parent.end$ (which is non-**null** since the parent is ready). Hence n is the root of its tree if and only if $n.\mathit{beginning} = \textbf{null}$.

A query (i, x) is *pending* if and only if $F_i(x) = \perp$ and there exists a node n such that $n.end = (i, x)$. Intuitively, a query (i, x) is pending if $F_i(x) = \perp$ but the simulator has already decided to assign a value to $F_i(x)$ during that query cycle. In particular, one can observe from the pseudocode that when a call $\mathrm{F}(i, x)$ occurs such that $F_i(x) = \perp$, a call $\mathrm{NewTree}(i, x)$ occurs that results a new tree being created, with a root n such that $n.end = (i, x)$, so that (i, x) becomes a pending query.

The following additional useful facts about trees will be seen in the proof:

1. We have
$$a.end \neq b.end$$
for all nodes $a \neq b$, presuming $a.end, b.end \neq \textbf{null}$, and regardless of whether a and b are in the same tree or not. (Thus all query fields in all trees are distinct, modulo the fact that a child's *beginning* is the same as its parent's *end*.)
2. If $n.\mathit{beginning} = (i, x_i) \neq \textbf{null}$, $n.end = (j, x_j) \neq \textbf{null}$ then
$$\{i, j\} \in \{\{1, 4\}, \{5, 8\}, \{2, 7\}, \{3, 6\}\}.$$
3. Each tree has at most one non-ready node, i.e., at most one node n with $n.end = \textbf{null}$. This node is necessarily a leaf, and, if it exists, is called the *non-ready leaf* of the tree.
4. GrowTree(*root*) is only called once per root *root*, as syntactically obvious from the code. While this call has not yet returned, moreover, we have $F_i(x) = \perp$ for all (i, x) such that $n.end = (i, x)$ for some node n of the tree. (In other words, a pending query remains pending as long as the node to which it is associated belongs to a tree which has not finished growing.)

The *origin* of a node n is the position of $n.\mathit{beginning}$, if $n.\mathit{beginning} \neq \textbf{null}$. The *terminal* of a node n is the position of $n.end$, if $n.end \neq \textbf{null}$. (Thus, as per the second bullet above, for each ready non-root node the origin and the terminal uniquely determines each other.)

A *2chain* is a triple of the form $(i, x_i, x_{i+1}) \in \{0, 1, \ldots, 8\} \times \{0,1\}^n \times \{0,1\}^n$. The *position* of the 2chain is i.

Each node has a 2chain field called id, which is non-**null** if and only if the node isn't the root of its tree. Intuitively, each node is associated to a path which "needs to be completed", and the id contains two queries that are on the path at the moment when the node is created; indeed the two queries (in adjacent positions) are enough to uniquely determine the path.

Detect zone	Origin	Position of id
8, 1, 2	2 or 8	1
7, 8, 1	1 or 7	7
3, 4, 5	3 or 5	3
4, 5, 6	4 or 6	4

Table 1. For nodes triggered by each detect zone, position of the id and possible values of the origin.

The value of id is assigned to a non-root node when the node is created by Trigger, such that id lies in the relevant detect zone. Specifically, for a node of origin $i \in [8]$, the position of id is 7 if $i \in \{1, 7\}$, is 1 if $i \in \{2, 8\}$, is 3 if $i \in \{3, 5\}$, and is 4 if $i \in \{4, 6\}$ (see Table 1). We note that the position of id is always the (cyclically) first position of the detect zone except for the outer detect zone 8, 1, 2, because the first two positions of the latter detect zone are not adjacent.

The simulator also maintains a global list N of nodes that are ready. This list is maintained for the convenience of the procedure IsPending, which would otherwise require searching through all trees that have not yet been discarded (and, in particular, maintaining a set of pointers to the roots of such trees).

Recursive call structure. Trees are grown according to a somewhat complex recursive mechanism. Here is the overall recursive structure of the stack:

- F calls NewTree (at most one call to NewTree per call to F)
- NewTree calls GrowTree (one call to GrowTree per call to NewTree)
- GrowTree calls GrowTreeOnce (one or more times)
- GrowTreeOnce calls FindNewChildren (one or more times) and also calls GrowTreeOnce (zero or more times)
- FindNewChildren calls Trigger (zero or more times)
- Trigger calls MakeNodeReady (at most one call to MakeNodeReady per call to Trigger)
- MakeNodeReady calls Prev or Next (zero or more times)
- Prev and Next call F (zero or once)

We observe that new trees are only created by calls to F. Moreover, a node n is not ready (i.e., $n.end = $ **null**) when MakeNodeReady(n) is called, and n is ready (i.e., $n.end \neq$ **null**) when MakeNodeReady(n) returns, whence the name of the procedure. Since MakeNodeReady calls Prev and Next (which themselves call F), entire trees might be created and discarded while making a node ready.

Tree Growth Mechanism and Path Detection. Recall that every pending query (i, x) is uniquely associated to some node n (in some tree) such that $n.end = (i, x)$. Every pending query is susceptible of triggering zero or more *path completions*, each of which incurs the creation of a new node that will be a child of n. The trigger mechanism (implemented by the procedures

FindNewChildren and Trigger) is now discussed in more detail.

Firstly we must define *equivalence* of 2chains. This definition relies on the functions Val^+, Val^- and is implemented by the function Equivalent, which we invite the reader to consult at this point. (See Figs. 5–6.) Briefly, a 2chain (i, x_i, x_{i+1}) is *equivalent* to a 2chain (j, x'_j, x'_{j+1}) if and only if:

(1) (i, j) equals $(7, 4)$, $(1, 7)$, $(3, 1)$ or $(4, 3)$ and $\mathrm{Val}^-(i, x_i, x_{i+1}, h) = x'_h$ for $h = j, j + 1$ (or, equivalently, $\mathrm{Val}^+(j, x'_j, x'_{j+1}, h) = x_h$ for $h = i, i + 1$), or
(2) the 2chain (j, x'_j, x'_{j+1}) is equivalent to (i, x_i, x_{i+1}) in the sense of case (i), or
(3) the two 2chains are identical, i.e., $i = j$, $x_i = x'_j$ and $x_{i+1} = x'_{j+1}$.

Equivalence is defined in these specific cases only, and it is symmetric but not transitive. It can be noted that equivalence is time-dependent (like most of our definitions), in the sense that entries keep being added to the tables F_i.

Let (i, x_i) be a pending query and let n be the node such that $n.end = (i, x_i)$. (We remind that such a node n exists and is unique; existence follows by definition of *pending*, uniqueness is argued within the proof.)

We define *triggers* for (i, x_i) as follows, considering different values of i (note that the cases with $i = 1, 2, 3, 4$ are symmetric to cases with $i = 8, 7, 6, 5$ respectively):

– If $i = 1$, a trigger for $(i, x_i) = (1, x_1)$ is a pair $(x_7, x_8) \in F_7 \times F_8$ such that $\mathrm{P}^{-1}(x_8, x_9) = (*, x_1)$ where $x_9 = x_7 \oplus F_8(x_8)$ and such that the 2chain $(7, x_7, x_8)$ is not equivalent to $n.id$ and not equivalent to $c.id$ for any existing child c of n.
– If $i = 2$, a trigger for $(i, x_i) = (2, x_2)$ is a pair $(x_8, x_1) \in F_8 \times F_1$ such that $\mathrm{P}(x_0, x_1) = (x_8, *)$ where $x_0 = F_1(x_1) \oplus x_2$ and such that the 2chain $(1, x_1, x_2)$ is not equivalent to $n.id$ and not equivalent to $c.id$ for any existing child c of n.
– If $i = 3$, a trigger for $(i, x_i) = (3, x_3)$ is a pair $(x_4, x_5) \in F_4 \times F_5$ such that $x_5 = x_3 \oplus F_4(x_4)$ and such that the 2chain $(3, x_3, x_4)$ is not equivalent to $n.id$ and not equivalent to $c.id$ for any existing child c of n.
– If $i = 4$, a trigger for $(i, x_i) = (4, x_4)$ is a pair $(x_5, x_6) \in F_5 \times F_6$ such that $x_6 = x_4 \oplus F_5(x_5)$ and such that the 2chain $(4, x_4, x_5)$ is not equivalent to $n.id$ and not equivalent to $c.id$ for any existing child c of n.
– If $i = 5$, a trigger for $(i, x_i) = (5, x_5)$ is a pair $(x_3, x_4) \in F_3 \times F_4$ such that $x_3 = F_4(x_4) \oplus x_5$ and such that the 2chain $(3, x_3, x_4)$ is not equivalent to $n.id$ and not equivalent to $c.id$ for any existing child c of n.
– If $i = 6$, a trigger for $(i, x_i) = (6, x_6)$ is a pair $(x_4, x_5) \in F_4 \times F_5$ such that $x_4 = F_5(x_5) \oplus x_6$ and such that the 2chain $(4, x_4, x_5)$ is not equivalent to $n.id$ and not equivalent to $c.id$ for any existing child c of n.
– If $i = 7$, a trigger for $(i, x_i) = (7, x_7)$ is a pair $(x_8, x_1) \in F_8 \times F_1$ such that $\mathrm{P}^{-1}(x_8, x_9) = (*, x_1)$ where $x_9 = x_7 \oplus F_8(x_8)$ and such that the 2chain $(7, x_7, x_8)$ is not equivalent to $n.id$ and not equivalent to $c.id$ for any existing child c of n.
– If $i = 8$, a trigger for $(i, x_i) = (8, x_8)$ is a pair $(x_1, x_2) \in F_1 \times F_2$ such that $\mathrm{P}(x_0, x_1) = (x_8, *)$ where $x_0 = F_1(x_1) \oplus x_2$ and such that the 2chain $(1, x_1, x_2)$ is not equivalent to $n.id$ and not equivalent to $c.id$ for any existing child c of n.

We note that the trigger and (i, x_i) form three queries in a detect zone, where i is not in the middle of the detect zone. The non-equivalence conditions ensures that the triggered path is not the same as the path corresponding to the parent node n or another path that has already been triggered.

The procedure that checks for triggers is FindNewChildren. Specifically, FindNewChildren takes as argument a node n, and checks if there exist triggers for the pending query[9] $n.end$. FindNewChildren(n) enumerates all "potential triggers" (i.e., all pairs of defined queries in the specific positions) and calls Trigger to check whether it is a valid trigger.

The arguments of Trigger consist of a position i, three values x_i, x_{i+1} and u, and a node *node*. The position i specifies one of the four detect zones, and the three values constitute queries in the three positions of the detect zone; in particular, the first two values corresponds to queries in positions i and $i+1$ respectively. If Trigger identifies a trigger, it creates a new child c for n; the *id* of c is set to (i, x_i, x_{i+1}). Therefore the relations between i and detect zones follow Table 1. After creating c, Trigger calls MakeNodeReady(c).

As a subtlety, one should observe that, in FindNewChildren(n), certain value pairs that are *not* triggers before a call to Trigger might be triggers *after* the call, because Trigger has called MakeNodeReady, which has created fresh table entries. However one can also observe that FindNewChildren will in any case be called again on node n by virtue of having returned $child\_added = \mathbf{true}$. (Indeed, GrowTree(*root*) only returns after doing a complete traversal of the tree such that no calls to FindNewChildren($\cdot$) during the traversal result in a new child.)

Partial Paths and Completed Paths. We define an (i, j)-*partial path*[10] to be a sequence of values $x_i, x_{i+1}, \ldots, x_j$ if $i < j$, or a sequence x_i, x_{i+1}, $\ldots, x_9, x_0, x_1, \ldots, x_j$ if $i > j$ satisfying the following properties: $x_h \in F_h$ and $x_{h-1} \oplus F_h(x_h) = x_{h+1}$ for subscripts h such that $h \notin \{i, j, 0, 9\}$; if $i > j$, then $i \leq 8$, $j \geq 1$, and $T(x_0, x_1) = (x_8, x_9)$; if $i < j$, then $0 \leq i < j \leq 9$.

We notate the partial path as $\{x_h\}_{h=i}^{j}$ regardless of whether $i < j$ or $i > j$, with the understanding that x_9 is followed by x_0 if $i > j$.

The values i and j are called the *endpoints* of the path. One can observe that two adjacent values x_h, x_{h+1} on a partial path ($h \neq 9$) along with two endpoints (i, j) uniquely determine the partial path, if it exists.

An (i, j)-partial path $\{x_h\}_{h=i}^{j}$ *contains* a 2chain $(\ell, y_\ell, y_{\ell+1})$ if $x_\ell = y_\ell$ and $x_{\ell+1} = y_{\ell+1}$; moreover if $i = j + 1$, the case $\ell = j$ is excluded.

We say an (i, j)-partial path $\{x_h\}_{h=i}^{j}$ is *full* if $1 \leq i, j \leq 8$ and if $x_i \notin F_i$, $x_j \notin F_j$.

A *completed path* is a $(0, 9)$-partial path $\{x_h\}_{h=0}^{9}$ such that $T(x_0, x_1) = (x_8, x_9)$.

The MakeNodeReady Procedure. Next we discuss the procedure MakeNodeReady. One can firstly observe that MakeNodeReady(*node*) is not called if *node* is the root of its tree, as clear from the pseudocode. In particular $node.beginning \neq null$ when MakeNodeReady(*node*) is called.

MakeNodeReady(*node*) behaves differently depending on the origin i of *node*. If $i = 1$ then $node.id = (7, u_7, u_8)$ for some values u_1, u_2, where $(1, u_1) = node.beginning$. Starting with $j = 7$, MakeNodeReady executes the instructions

$$(u_1, u_2) \leftarrow \text{Prev}(j, u_1, u_2)$$
$$j \leftarrow j - 1 \mod 11$$

until $j = 4$. One can note (from the pseudocode of Prev) that after each call of the form Prev(j, u_1, u_2)

[9] Let $n.end = (i, x_i)$. By definition, then, (i, x_i) is "pending" only if $F_i(x_i) = \perp$. This is indeed always the case when FindNewChildren(n) is called—and throughout the execution of that call—as argued within the proof.

[10] This is a slightly simplified definition. The "real" definition of a partial path is given by Definition 7, Section 5.1. However, the change is very minor, and does not affect any statement or secondary definition made between here and Definition 7.

with $j \neq 0$, $F_j(u_1) \neq \bot$. (When $j = 0$ the call Prev(j, u_1, u_2) entails a call to P^{-1} instead of to F.) Thus, after this sequence of calls, there exists a partial path $x_4, x_5, \ldots, x_1$ with endpoints $(i, j) = (1, 4)$ and with $(7, x_7, x_8) = node.id$.

We also have $F_1(x_1) = \bot$ by item 4 above and, if MakeNodeReady doesn't abort, $F_4(x_4) = \bot$ as well when MakeNodeReady returns. In particular, $x_4, x_5, \ldots, x_1$ is a full $(4, 1)$-partial path when MakeNodeReady returns, containing *node.id*.

For other origins i, MakeNodeReady similarly creates a partial path whose endpoints are the origin and terminal of the node by repeated calls to Prev (if $i = 2, 5, 6$) or Next (if $i = 3, 4, 7, 8$). The partial path is also full when MakeNodeReady returns, and likewise contains *node.id*. In Table 2, the positions of queries issued by MakeNodeReady are listed in the column "MakeNodeReady". The non-colored positions are those that are defined when the node is created. In particular, we observe that when $i = 2, 3, 6, 7$, MakeNodeReady doesn't issue new queries to F.

In summary, when MakeNodeReady(*node*) returns one has $node.beginning \neq$ **null**, $node.end \neq$ **null**, and there exists a full (i, j)-partial path containing *node.id* such that

$$\{(i, x_i), (j, x_j)\} = \{node.beginning, node.end\}.$$

Origin	Terminal	Existing	MakeNodeReady	PrepareTree	AdaptNode
1	4	7, 8	7, 6, 5		2, 3
2	7	1, 8	8	3, 6	4, 5
3	6	4, 5	5	7, 8	1, 2
4	1	5, 6	5, 6, 7, 8		2, 3
5	8	3, 4	3, 2, 1		6, 7
6	3	4, 5	4	7, 8	1, 2
7	2	8, 1	8, 1	3, 6	4, 5
8	5	1, 2	2, 3, 4		6, 7

Table 2. This table shows the positions of queries issued by MakeNodeReady and of table values sampled by PrepareTree and adapted by AdaptNode as a function of the origin (and terminal) of a path, as well as the positions of queries that are already defined when MakeNodeReady is called (the 'Existing' column). In the MakeNodeReady column, queries in black are already defined when MakeNodeReady issues the query to $\mathrm{F}(\cdot, \cdot)$, so that F returns immediately for those queries. By contrast, blue positions may spawn a $(2, 7)$- or $(3, 6)$-tree while red positions may spawn a $(1, 4)$- or $(5, 8)$-tree.

Path Completion Process. We say that node n is *stable* if no triggers exist for the query $n.end$.

When GrowTree(*root*) returns in NewTree, each node in the tree rooted at *root* is both ready and stable. (This is rather easy to see syntactically from the pseudocode.) Moreover each non-root node of the tree is associated to a partial path, which is the unique partial path containing that node's *id* and whose endpoints are the node's origin and terminal.

After GrowTree(*root*) returns, SampleTree(*root*) is called, which calls ReadTape(i, x) for each (i, x) such that $(i, x) = n.end$ for some node n in the tree rooted at *root*. This effectively assigns a uniform independent random value to $F_i(x)$ for each such pair (i, x).

One can observe that the only nodes whose stability is potentially affected by a change to the table F_i are nodes with terminal $i \pm 1, 2$ (taken modulo 8) since each detect zone has length 3. Given that all nodes in the tree have terminals $i \in \{1, 4\}$, $i \in \{5, 8\}$, $i \in \{2, 7\}$ or $i \in \{3, 6\}$, the calls to ReadTape that occur in SampleTree(*root*) do not affect the stability of the nodes the current tree, i.e., the tree rooted at *root*. (On the other hand the stability of nodes of trees lower down in the stack is potentially affected.)

After SampleTree(*root*) returns, PrepareTree(*root*) is called to further extend the partial paths associated[11] to each non-root node of the tree until only the queries about to be adapted are undefined. If the origin of a node is 1, 4, 5 or 8, PrepareTree does nothing since the associated partial path only contains two undefined queries after SampleTree returns; on the other hand, PrepareTree samples queries in positions 3 and 6 if the origin is 2 or 7, and samples queries in positions 7 and 8 if the origin is 3 or 6. We note that queries sampled by PrepareTree relies on the randomness sampled in SampleTree, thus it is unlikely that they trigger a new path completion together with pre-existing queries or amongst themselves (this is also true for queries adapted by AdaptTree).

Finally, AdaptTree(*root*) is called, which "adapts" each associated partial path into a completed path. In more detail, the two undefined queries in the path are adapted (by a call to the procedure Adapt) as in equations (2) and (3); the positions of the adapted queries are shown in Table 2.

Further Pseudocode Details: The Tables $T_{\text{sim}}/T_{\text{sim}}^{-1}$. In order to reduce its query complexity, and following an idea of [12], our simulator keeps track of which queries it has already made to P or P^{-1} via a pair of tables T_{sim} and T_{sim}^{-1}. These tables are maintained by the procedures SimP and SimP^{-1} (Fig. 4), which are "wrapper functions" that the simulator uses to access P and P^{-1}. If the simulator did not use the tables T_{sim} and T_{sim}^{-1} to remember its queries to $\mathrm{P}/\mathrm{P}^{-1}$, the query complexity would be quadratically higher: $O(q^8)$ instead of $O(q^4)$. (This is the route taken by [18], and their query complexity could be indeed be lowered from $O(q^8)$ to $O(q^4)$ by using the trick of remembering past queries to $\mathrm{P}/\mathrm{P}^{-1}$.)

We also note that the tables T_{sim}, T_{sim}^{-1} are accessed by the procedures Val^+ and Val^- of game G_1 (see Fig. 6), while in games G_2–G_4 Val^+ and Val^- access the tables T and T^{-1} directly, which are not accessible to the simulator in game G_1. As it turns out, games G_1–G_4 would be unaffected if the procedures Val^+, Val^- called $\mathrm{SimP}/\mathrm{SimP}^{-1}$ (or even $\mathrm{P}/\mathrm{P}^{-1}$) instead of doing table look-ups "by hand", because it turns out that Val^+, Val^- never return $\perp$ in any of games G_1–G_4 (see Lemma 21); but we choose the latter presentation (i.e., accessing the tables $T_{\text{sim}}/T_{\text{sim}}^{-1}$ or T/T^{-1}, depending) in order to emphasize—and to more easily argue within the proof—that calls to Val^+, Val^- do not cause "new" queries to $\mathrm{P}/\mathrm{P}^{-1}$.

5 Proof of Indifferentiability

In this section we give a proof for Theorem 1, using the simulator described in Section 4 as the indifferentiability simulator.

In order to prove that our simulator successfully achieves indifferentiability as defined by Definition 1, we need to upper bound the time and query complexity of the simulator, as well as the advantage of any distinguisher. These three bounds are the objects of Theorems 33, 30 and 97 respectively.

Game Sequence. Our proof uses a sequence of five games, $\mathrm{G}_1, \ldots, \mathrm{G}_5$, with G_1 being the simulated world and G_5 being the real world. Games G_1–G_4 are described by the pseudocode of Figs. 4–7 while game G_5 is given by the pseudocode of Fig. 8. Every game offers the same interface to the distinguisher, consisting of functions F, P and P^{-1}.

[11] The partial path is namely uniquely determined by the node's *id*.

A brief synopsis of the changes that occur in the games is as follows:

In G_2: The simulator's procedures CheckP^+ and CheckP^-(Fig. 5) used by the simulator in FindNewChildren (Fig. 4) "peeks" at the table T: CheckP^+ returns $\perp$ if $(x_8, x_9) \notin T^{-1}$, and CheckP^- returns $\perp$ if $(x_0, x_1) \notin T$; this modification ensures that a call to $\text{CheckP}^+/\text{CheckP}^-$ does not result in a "fresh" call to P. Also, the procedures Val^+, Val^- use the tables T, T^{-1} instead of T_{sim}, T_{sim}^{-1}. (As mentioned at the end of the last section, the second change does not actually alter the behavior of Val^+, Val^-, despite the fact that the tables T_{sim}, T_{sim}^{-1} may be proper subsets of the tables T, T^{-1} (see Lemma 21). On the other hand, the change to $\text{CheckP}^+/\text{CheckP}^-$ may result in "false negatives" being returned.)

In G_3: The simulator adds a number of checks that may cause it to abort in places when it did not abort in G_2. Some of these involve peeking at the random permutation table T, which means they cannot be included in G_1. Otherwise, G_3 is identical to G_2, so the only difference between G_2 and G_3 is that G_3 may abort when G_2 does not. The pseudocode for the new checking procedures called by G_3 are in Figs. 9–10.

In G_4: The only difference occurs in the implementation of the oracles P, P^{-1} (see Fig. 7). In G_4, these oracles no longer rely on the random permutation table $p : \{0,1\}^{2n} \to \{0,1\}^{2n}$, but instead evaluate an 8-round Feistel network using the random tapes $f_1, \ldots, f_8$ as round functions.

In G_5: This is the real world, meaning that $\text{F}(i, x)$ directly returns the value $f_i(x)$. As will be shown in the proof, the only "visible" difference between G_4 and G_5 is that G_4 may abort, while G_5 does not.

The *advantage* of a distinguisher D at distinguishing games G_i and G_j is defined as

$$\Delta_D(G_i, G_j) = \Pr_{G_i}[D^{F,P,P^{-1}} = 1] - \Pr_{G_j}[D^{F,P,P^{-1}} = 1] \tag{4}$$

where the probabilities are taken over the coins of the relevant game as well as over D's coins, if any. Most of the proof is concerned with upper bounding $\Delta_D(G_1, G_5)$ for a distinguisher D that is limited to q queries (in a nonstandard sense defined below); the simulator's efficiency, as well its query complexity (Theorems 32 and 30 respectively) will be established as byproducts along the way.

Normalizing the Distinguisher. In the following proof we fix an information-theoretic distinguisher D with access to oracles F, P, and P^{-1}. The distinguisher can issue at most q queries to $\text{F}(i, \cdot)$ for each $i \in [8]$ and at most q queries to P and P^{-1} in total. In particular, the distinguisher is allowed to make q queries to *each* round of the Feistel network, which is a relaxed condition. The same relaxation is implicitly made in most if not all previous work in the area, but explicitly acknowledging the extra power of the distinguisher actually helps to improve the final bound, as we shortly explain.

Since D is information-theoretic, we can assume without loss of generality that D is deterministic by fixing the best possible sequence of coin tosses for D. (See, e.g., the appendix in the proceedings version of [6].)

We can also assume without loss of generality that D outputs 1 if an oracle abort. Indeed, since the real world G_5 does not abort, this can only increase the distinguishing advantage $\Delta_D(G_1, G_5)$.

Some of our lemmas, moreover, only hold if D is a distinguisher that *completes all paths*, as per the following definition:

Definition 1. A distinguisher D *completes all paths* if at the end of every non-aborted execution, D has made the queries $\text{F}(i, x_i)$ for $i = 1, 2, \ldots, 8$ where $x_i = \text{F}(i-1, x_{i-1}) \oplus x_{i-2}$ for $i = 2, 3, \ldots, 8$, for every pair (x_0, x_1) such that D has either queried P at (x_0, x_1) at some point during the execution or such that P^{-1} returned (x_0, x_1) to D at some point during the execution.

Lemmas that only hold if D completes all paths (and which are confined to sections 5.5, 5.7) are marked with a (*).

It is not difficult to see that for every distinguisher D that makes at most q queries to each of its oracles, there is a distinguisher D^* that completes all paths, that achieves the same distinguishing advantage as D, and that makes at most $2q$ queries to each of its oracles. Hence, the cost of assuming a distinguisher that completes all paths is a factor of two in the number of queries. (Previous papers [1, 18, 20] pay for the same assumption by giving r times as many queries to the distinguisher, where r is the number of rounds. Our trick of explicitly giving the distinguisher the power to query each of its oracles q times reduces this factor to 2 without harming the final bound; indeed, current proof techniques *effectively* give the distinguisher q queries to each of its oracles anyway. Our trick also partially answers a question posed in [1].)

Miscellaneous. Unless otherwise specified, an *execution* refers to a run of one of the games G_1, G_2, G_3, G_4 (excluding, thus, G_5) with the fixed distinguisher D mentioned above.

5.1 Efficiency of the Simulator

We start the proof by proving that the simulator is efficient in games G_1 through G_4. This part is similar to previous efficiency proofs such as [12, 18], and ultimately relies on Seurin's termination argument, outlined at the end of Section 3.

Unless otherwise specified, lemmas in this section apply to games G_1 through G_4. As the proof proceeds, and for ease of reference, we will restate some (but not all) of the definitions made in Section 4.

Definition 2. A query (i, x_i) is *defined* if $F_i(x_i) \neq \bot$. It is *pending* if it is not defined and there exists a node n such that $n.end = (i, x_i)$.

Definition 3. A *completed path* is a sequence $x_0, \ldots, x_9$ such that $x_{i+1} = x_{i-1} \oplus F_i(x_i)$ for $1 \leq i \leq 8$ and such that $T(x_0, x_1) = (x_8, x_9)$.

Definition 4. A node n is *created* if its constructor has returned. It is *ready* if $n.end = (i, x_i) \neq$ **null**, and it is *sampled* if $F_i(x_i) \neq \bot$. A node n is *completed* if there exists a completed path $x_0, x_1, \ldots, x_9$ containing the 2chain $n.id$.

We emphasize that a completed node is also a sampled node, that a sampled node is also a ready node, etc. We thus have the following chain of containments:

$$\text{created nodes} \supseteq \text{ready nodes} \supseteq \text{sampled nodes} \supseteq \text{completed nodes}$$

We also note that a root node r cannot become completed because $r.id =$ **null** (and remains **null**) for root nodes. Moreover, we remind that nodes are never deleted (even after the last reference to a node is lost).

Lemma 1. *The parent, id, beginning, and end fields of a node are never overwritten after they are assigned a non-***null** *value.*

Proof. This is easy to see from the pseudocode. The *parent*, *id* and *beginning* of a node are only assigned in the constructor. The only two functions to edit the *end* field of a node are NewTree and MakeNodeReady. NewTree creates a root with a **null** *end* field and immediately assigns the *end* field to a non-**null** value, while MakeNodeReady(n) is only called for nodes n that are not roots, and is called at most once for each node. □

Lemma 2. *A node is a root node if and only if it is a root node after its constructor returns, and if and only if it is created in the procedure* NewTree*.*

Proof. Recall that by definition a node n is a root node if and only if $n.beginning = \textbf{null}$. The first "if and only if" therefore follows from the fact that the *beginning* field of a node is not modified outside the node's constructor.

The second "if and only if" follows by inspection of the procedures NewTree and Trigger (Fig. 4), which are the only two procedures to create nodes. □

The above lemmas show that all fields of a node are invariant after the node's definition, except for the set of children, which grows as new paths are discovered. Therefore when we refer to these variables in the following discussions, we don't need to specify exactly what time we are talking about (as long as they are defined).

Lemma 3. *The entries of the tables F_i are not overwritten after they are defined.*

Proof. The only two procedures that modify tables F_i are ReadTape and Adapt. In both procedures the simulator checks that $x_i \notin F_i$ (and aborts if otherwise) before assigning a value to $F_i(x_i)$. □

Lemma 4. *Entries in tables T and T^{-1} are never overwritten and T_{sim} (T_{sim}^{-1}) is a subset of T (T^{-1}). In G_1, G_2 and G_3, the tables T and T^{-1} are compatible with the permutation encoded by tape p and its inverse.*

Proof. The tables T and T^{-1} are only modified in P or P^{-1}. Entries are added according to a permutation, which is the permutation encoded by the random tape p in G_1, G_2 and G_3, and is the 8-round Feistel network built from the round functions (random tapes) $f_1, \ldots, f_8$ in G_4. By inspection of the pseudocode, the entries are never overwritten.

The table T_{sim} is only modified in SimP and SimP^{-1}. The entry added to T_{sim} is obtained via a call to P or P^{-1}, where the corresponding entry in T is returned, and hence the same entry also exists in T. □

Lemma 5. *A node is immediately added to the set N after becoming ready.*

Proof. A node becomes ready when its *end* is assigned a query. This only occurs in NewTree and MakeNodeReady, and in both cases the node is added into N immediately after the assignment. □

Lemma 6. *Let n be a ready node with $n.end = (i, x_i)$. Then* IsPending$(i, x_i) = \textbf{true}$ *or $x_i \in F_i$ from the moment when n is added to N until the end of the execution.*

Proof. The procedure IsPending(i, x_i) returns **true** while n is in N. Note that n is removed from N only in SampleTree, right after ReadTape($n.end$). Therefore, at the moment when n is removed from N we already have $x_i \in F_i$. Since entries in F_i are not overwritten, this remains true for the rest of the execution. □

Lemma 7. *We have $n_1.end \neq n_2.end$ for distinct nodes n_1 and n_2 with $n_1.end \neq$ **null**.*

Proof. Assume by contradiction that there exist two nodes n_1, n_2 such that $n_1.end = n_2.end = (i, x_i)$. Without loss of generality, suppose n_1 becomes ready before n_2.

If n_2 is the root of a tree, it becomes ready after it is created in NewTree, called by $\mathrm{F}(i, x_i)$. Between the time when $\mathrm{F}(i, x_i)$ is called and the time NewTree executes its second line, no modification is made to the other nodes, so n_1 is already ready when the call $\mathrm{F}(i, x_i)$ occurs. By Lemmas 5 and 6, when $\mathrm{F}(i, x_i)$ is called, we have $\mathrm{IsPending}(i, x_i) = \textbf{true}$ or $x_i \in F_i$. But $\mathrm{F}(i, x_i)$ aborts if $\mathrm{IsPending}(i, x_i) = \textbf{true}$, and it returns $F_i(x_i)$ directly if $x_i \in F_i$. NewTree is not called in either case, leading to a contradiction.

If n_2 is not a root node, its *end* is assigned in MakeNodeReady. Before $n_2.end$ is assigned, two assertions are checked. Since no modification is made to the other nodes during the assertions, n_1 is ready before the assertions. By Lemmas 5 and 6, we must have $\mathrm{IsPending}(i, x_i) = \textbf{true}$ (violating the second assertion) or $x_i \in F_i$ (violating the first assertion). In both cases the simulator aborts before the assignment, which is also a contradiction. □

Lemma 8. FindNewChildren(n) *is only called if n is a ready node.*

Proof. Recall that ready nodes never revert to being non-ready (cf. Lemma 1).

If n is created by NewTree then $n.end$ is assigned by NewTree immediately after creation, and hence n is ready.

If n is created by AddChild, on the other hand, then AddChild calls MakeNodeReady(n) immediately, which does not return until n is ready. Moreover, while MakeNodeReady(n) calls further procedures, it does not pass on a reference to n to any of the procedures that it calls, so it is impossible for a call FindNewChildren(n) to occur while MakeNodeReady(n) has not yet returned. □

Lemma 9. *A node n is a child of n' if and only if $n.beginning = n'.end \neq$ **null**.*

Proof. If $n' = n.parent$, then in the constructor of n, its *beginning* is assigned the same value as $n'.end$. Since FindNewChildren is only called on ready nodes, $n'.end \neq$ **null**. By Lemma 1, neither $n.beginning$ nor $n'.end$ can be overwritten, thus $n.beginning = n'.end \neq$ **null** until the end of the execution.

On the other hand, if $n.beginning = n'.end \neq$ **null**, then n is a non-root node. As proved in the "if" direction, we must have $n.parent.end = n.beginning = n'.end$. By Lemma 7, the *end* of ready nodes are distinct, thus $n.parent = n'$. □

Lemma 10. *For every node n, the query $n.end$ only becomes defined when* SampleTree(n) *is called.*

Proof. Consider an arbitrary node n with $n.end = (i, x_i)$. $n.end$ can only be assigned in a call to NewTree or MakeNodeReady. NewTree(i, x_i) must be called in a call to $\mathrm{F}(i, x_i)$, when (i, x_i) is not defined or pending. The MakeNodeReady procedure aborts if the query being assigned to $n.end$ is defined or pending. Therefore, $n.end$ is not defined or pending when n becomes ready.

After n becomes ready, it is added to the set N immediately and will not be removed from N until SampleTree(n) is called. Before n is removed, $\mathrm{IsPending}(i, x_i)$ always returns **true**. Thus calls to $\mathrm{ReadTape}(i, x_i)$ and $\mathrm{Adapt}(i, x_i, \cdot)$ will abort without defining the queries. Therefore, the query $n.end = (i, x_i)$ remains undefined before SampleTree(n) is called. □

Lemma 11. *When* FindNewChildren(n) *is called, as well as during the call, $n.end$ is pending.*

Proof. By definition, we only need to prove that the query $n.end$ has not been defined. By Lemma 10, $n.end$ is not defined before SampleTree(n) is called. Let r be the root of the tree containing n. Observe that FindNewChildren(n) is only called before GrowTree(r) returns, while the call to SampleTree(r) (and to SampleTree(n)) occurs after GrowTree(r) returns. □

Lemma 12. *The set N consists of all nodes that are ready but not sampled, except for the moments right before a node is added to N or right before a node is deleted from N.*

Proof. By Lemma 5, a node is added to N right after it becomes ready. On the other hand, a node is added to N only in procedures NewTree and MakeNodeReady, and in both procedures the *end* of the node is assigned a non-**null** value before it is added.

Then we prove that a node is removed from N if and only if it becomes sampled, which immediately follows from an observation of the pseudocode: A node n must be removed from N during the call to SampleTree(n), and right after ReadTape($n.end$) is called.

Therefore, the set N always equals the set of nodes that are ready but not sampled, except for the gaps right before the sets are changed. □

Lemma 13. *At all points when calls to* IsPending *occur in the pseudocode, the call* IsPending(i, x_i) *returns* **true** *if and only if the query (i, x_i) is pending.*

Proof. IsPending(i, x_i) returns **true** if and only if there exists a node n in N such that $n.end = (i, x_i)$. Since IsPending is not called immediately before a modification to N, Lemma 12 implies that this occurs if and only if there exists a node n such that $n.end = (i, x_i)$ and such that $F_i(x_i) = \perp$. □

Definition 5. Let $\tilde{F}_i$ denote the set of queries in position i that are pending or defined, for $i \in [8]$.

For any $i \in [8]$, since F_i is the set of defined queries in position i, we have $F_i \subseteq \tilde{F}_i$. The sets $\tilde{F}_i$ are time-dependent, like the sets F_i.

Lemma 14. *The sets $\tilde{F}_i$ are monotone increasing, i.e., once a query becomes pending or defined, it remains pending or defined for the rest of the execution.*

Proof. By Lemma 3, we know that after an entry is added to a table, it will not be overwritten. Therefore any defined query will remain defined through the rest of the execution.

For each pending query (i, x_i), there exists a node n such that $(i, x_i) = n.end$. By Lemma 1, $n.end$ will not change and thus (i, x_i) must be pending if it is not defined. □

Lemma 15. *At the end of a non-aborted query cycle, there exist no pending queries (i.e., all pending queries have been defined).*

Proof. Observe that in each call to NewTree, SampleTree is called on every node in the tree before NewTree returns, unless the simulator aborts. Therefore, all pending queries in the tree become defined before NewTree successfully returns. A non-aborted query cycle ends only after all calls to NewTree have returned, so all pending queries are defined by then. □

Next we upper bound the number of nodes created by the simulator and the sizes of the tables. We will separate the nodes into two types as in the following definition, and upper bound the number of each type. Recall that in the simulator overview we defined the *origin* and *terminal* of a non-root node n to be the positions of $n.beginning$ and $n.end$ respectively.

Definition 6. A non-root node is an *outer node* if its origin is 1, 2, 7 or 8, and is an *inner node* if its origin is 3, 4, 5 or 6.

The names imply by which detect zone a path is triggered: an inner node is associated with a path triggered by an inner detect zone; an outer node is associated with a path triggered by an outer detect zone.

Lemma 16. *The number of outer nodes created in an execution is at most q.*

Proof. It is easy to see from the pseudocode that before an outer node is added in FindNewChildren, the counter *NumOuter* is incremented by 1. The simulator aborts when the counter exceeds q, so the number of outer nodes is at most q. □

Now we give a formal definition of *partial path*, superseding (or rather augmenting) the definition given in Section 4.

Definition 7. An (i, j)*-partial path* is a sequence of values $x_i, x_{i+1}, \ldots, x_j$ if $i < j$, or a sequence x_i, x_{i+1}, $\ldots, x_9, x_0, x_1, \ldots, x_j$ if $i > j$, satisfying the following properties: $i \neq j$ and $0 \leq i, j \leq 9$; $x_h \in F_h$ and $x_{h-1} \oplus F_h(x_h) = x_{h+1}$ for subscripts h such that $h \notin \{i, j, 0, 9\}$; if $i > j$, we also require $(i, j) \neq (9, 0)$, $T(x_0, x_1) = (x_8, x_9)$ if $1 \leq j < i \leq 8$, $T(x_0, x_1) = (*, x_9)$ if $i = 9$, and $T^{-1}(x_8, x_9) = (x_0, *)$ if $j = 0$.

As can be noted, the only difference with the definition given in Section 4 is that the cases $i = 9$ and $j = 0$ (though not both simultaneously) are now allowed.

Let $\{x_h\}_{h=i}^{j}$ be an (i, j)-partial path. Each pair (h, x_h) with

$$h \in \{i, i+1, \ldots, j\}$$

if $i < j$, or with

$$h \in \{i, i+1, \ldots, 9\} \cup \{0, 1, \ldots, j\}$$

if $i > j$ is said to be *in* the partial path. We also say the partial path *contains* (h, x_h). We may also say that x_h *is in* the partial path (or that the partial path *contains* x_h) without mentioning the index h, if h is clear from the context.

Note that a partial path may contain pairs of the form $(9, x_9)$ and $(0, x_0)$ even though such pairs aren't queries, technically speaking.

As previously, a partial path $\{x_h\}_{h=i}^{j}$ *contains* a 2chain $(\ell, x_\ell, x_{\ell+1})$ (with $0 \leq \ell \leq 8$) if (ℓ, x_ℓ) and $(\ell + 1, x_{\ell+1})$ are both in $\{x_h\}_{h=i}^{j}$ and if $\ell \neq j$.

There are two different versions of Val^+ and Val^- in the pseudocode: one is used in G_1 (the G_1*-version*) and the other is used in $\text{G}_2, \text{G}_3, \text{G}_4$ (the G_2*-version*). In the following definition, as well as for the rest of the proof, Val^+ and Val^- refer to the G_2-version of these procedures.

Lemma 17. *Given a 2chain $(\ell, x_\ell, x_{\ell+1})$ and two* endpoints *i and j, there exists at most one (i, j)-partial path $\{x_h\}_{h=i}^{j}$ that contains the 2chain. Moreover, the values in the partial path can be obtained by $x_h = \text{Val}^+(\ell, x_\ell, x_{\ell+1}, h)$ if x_h is to the right of $x_{\ell+1}$ in the sequence $x_i, \ldots, x_j$*[12]*, and by $x_h = \text{Val}^-(\ell, x_\ell, x_{\ell+1}, h)$ if x_h is to the left of x_ℓ in the sequence $x_i, \ldots, x_j$.*

[12] The sequence $x_i, \ldots, x_j$ has the form $x_i, \ldots, x_{11}, x_0, \ldots, x_j$ if $j < i$ and $x_i, x_{i+1}, \ldots, x_j$ if $j > i$.

Proof. By Definition 7, we can see that each pair of values x_i, x_{i+1} uniquely determines the previous and the next value in the sequence (if they exist), and x_8, x_9 uniquely determines x_0, x_1 and vice versa. Thus, starting from x_ℓ and $x_{\ell+1}$, we can evaluate the path in each direction step by step according to the definition.

Moreover, we can see from the pseudocode that the procedures Val^+ and Val^- implements the above iterations and thus return the corresponding value in the partial path. □

Definition 8. Define the *length* of a partial path $\{x_h\}_{h=i}^{j}$ as $j-i+1$ if $i < j$ and equals $j-i+11$ if $i > j$.

Thus the length of a partial path $\{x_h\}_{h=i}^{j}$ is the number of distinct values of h for which there exists a pair (h, x_h) in the path, including possibly the values $h = 0$ and $h = 9$.

We note that a partial path cannot have length more than 10, because Definition 7 doesn't allow "self-overlapping" paths.

Definition 9. Let n be a non-root node with origin $h \in [8]$. If $h \in \{3, 4, 7, 8\}$ the *maximal path* of n is the longest (i, j)-partial path with $i = h$ containing $n.id$. If $h \in \{1, 2, 5, 6\}$ the *maximal path* of n is the longest (i, j)-partial path with $j = h$ containing $n.id$.

We note that a node's maximal path can have length at most 10, being defined as a partial path, even if the path could be further extended past its endpoint (in the standard Feistel sense) in some pathological cases.

Lemma 18. *A non-root node has a unique maximal path.*

Proof. This directly follows from the fact that a partial path's length is upper bounded by 10 by definition, and that an (i, j)-partial path is uniquely determined by the values i, j and by any 2chain contained in the path. □

The following lemma gives the observation that if a query is added to the sets $\tilde{F}_i$ in a procedure related to n, it must be in the maximal path of n.

Lemma 19. *The following statements hold for every non-root node n:*

1. *Let $n.id = (i, x_i, x_{i+1})$, then (i, x_i) and $(i+1, x_{i+1})$ are in the maximal path of n.*
2. *After $\mathrm{F}(i, x_i)$ is called in $\mathrm{MakeNodeReady}(n)$, the query (i, x_i) is in the maximal path of n.*
3. *After $\mathrm{SimP}(x_0, x_1)$ is called in $\mathrm{MakeNodeReady}(n)$, both $(0, x_0)$ and $(1, x_1)$ are in the maximal path of n; after the call returns with value (x_8, x_9), $(8, x_8)$ and $(9, x_9)$ are in the maximal path of n. Symmetrically for a call to SimP^{-1}.*
4. *The query that is assigned to $n.end$ is in the maximal path of n (even if the assignment doesn't occur because the assertions fail).*

Proof. In the following we assume that the origin of n is 1, 2, 5 or 6. The other four cases are symmetric.

We note that since the table entries and $n.id$ are not overwritten, if (i, x_i) is in the maximal path of n at some point in the execution, it remains so until the end of the execution.

The first statement directly follows from the definition of a maximal path, which is a partial path containing the 2chain $n.id$.

In a call to MakeNodeReady, F and SimP are called in $\mathrm{Prev}(i, x_i, x_{i+1})$. We prove by induction on the number of times Prev has been called in MakeNodeReady that both x_i and x_{i+1} are in the maximal path of n, and as well as the two output values of Prev(i, x_i, x_{i+1}) (whose positions may be $i-1$ and i or 8 and 9) are in the maximal path of n. In fact the latter statement follows from the former, since if $i > 0$ the output values of Prev are x_i and $x_{i-1} = \mathrm{F}(i, x_i) \oplus x_{i+1}$, which are in the same partial path as x_i and x_{i+1}, whereas if $i = 0$ the output values are $(x_8, x_9) = T(x_0, x_1)$, which are in the same partial path as x_0 and x_1, given that we are not overextending the partial path past length 10.

Since the next input to Prev is its former output (except for the first call) all that remains is to show the base case, i.e., that the first argument (i, x_i, x_{i+1}) given to Prev in MakeNodeReady is in the maximal path of n. However $(i, x_i, x_{i+1}) = n.id$ for the first call, so this is the case.

The query (j, x_j) is also in the output of Prev, so it is also in the maximal path by the above argument. □

In the following discussion, we will use x_i to denote queries in the maximal path of n unless no node n is involved or otherwise specified.

Lemma 20. *If a non-root node n is not ready, it has been created in a call to* AddChild *and the call hasn't returned. Specifically, each tree contains at most one non-ready node at any point of the execution.*

Proof. The first part is a simple observation: the call to AddChild returns only after MakeNodeReady(n) returns, at which point n has become ready.

Now consider any tree with root r. The node r becomes ready right after it is created. Non-root nodes are created in FindNewChildren via AddChild; before AddChild returns, no new node is added to the tree (the nodes created in F called by MakeNodeReady are in a new tree). Therefore, other nodes can be added to the tree only after AddChild returns, when the previous new node has become ready. □

Lemma 21. *The calls to* Val$^+$ *and* Val$^-$ *in procedures* Equivalent *and* AdaptNode *don't return* $\perp$.

Proof. The procedure Equivalent(C_1, C_2) is called inside FindNewChildren(n), either directly or via InChildren, where C_1 and C_2 are 2chains. The first 2chain C_1 is either $n.id$ or the *id* of a child of n. In the latter case, C_1 has the same position as C_2, therefore the values are directly compared without calling Val$^+$ or Val$^-$.

Now consider the first case, when $C_1 = n.id$. If n is the root of a tree, Equivalent returns **false** without calling Val$^+$ or Val$^-$. Otherwise, since AddChild(n) must have returned before FindNewChildren(n) can be called, n is ready by Lemma 20. By Lemma 19, the maximal path of n contains $n.end$. We can check that in every case, the calls to Val$^+$ or Val$^-$ won't "extend" over the terminal of the node. We show the cases where the origin of n is 1 for example: if the origin of n is 1, then the position of $n.id$ is 7 and the terminal of n is 4. A call to Equivalent$(n.id, (4, x_4, x_5))$ is made in FindNewChildren(n), in which Val$^-(n.id, 4)$ and Val$^-(n.id, 5)$ are called. Since n is ready, by Lemma 19 we know $n.end$ is in the maximal path of n, so Val$^-(n.id, 4) = n.end \neq \perp$. This also implies Val$^-(n.id, 5) \neq \perp$. The other cases are similar.

The call to AdaptNode(n) occurs after SampleTree(n) and PrepareTree(n), therefore the path containing $n.id$ has defined queries in all positions except possibly the two positions to be adapted, and Val$^+$ and Val$^-$ called in AdaptNode(n) will return a non-$\perp$ value. □

Lemma 22. *After* AdaptNode(n) *returns, the node n is completed. In particular, the queries in n's maximal path forms a completed path.*

Proof. Recall that n is completed if $n.id$ is contained in a completed path. Consider the execution in AdaptNode(n). Since the calls to Val^+ and Val^- don't return $\perp$ by Lemma 21, there exists a partial path $\{x_h\}_{h=m+1}^{m}$ containing $n.id$. Moreover, in AdaptNode(n) the queries (m, x_m) and $(m+1, x_{m+1})$ are adapted such that $F_m(x_m) = x_{m-1} \oplus x_{m+1}$ and $F_{m+1}(x_{m+1}) = x_m \oplus x_{m+2}$. Along with the properties of a partial path, it is easy to check that $\{x_h\}_{h=0}^{9}$ is a completed path, which contains $n.id$. ☐

Lemma 23. *The children of a node n must be created in* AddChild *called by* FindNewChildren(n). *The following properties hold for any node n: (i) n doesn't have two children with the same id; (ii) If n is a non-root node, the maximal path of n doesn't contain both queries in $c.id$ for any $c \in n.children$.*

Proof. It is easy to see from the pseudocode that a non-root node is only created in AddChild, which is only called in FindNewChildren(n) and the node becomes a child of n.

Before AddChild is called in FindNewChildren(n), a call to InChildren is made to check that the *id* of the new node doesn't equal the *id* of any existing child of n. All children of n have the same position of *id* and in this case, Equivalent returns **true** when the input 2chains are identical.

Property (ii) is ensured by the Equivalent call in FindNewChildren. By Lemma 21, the calls to Val^+ and Val^- in Equivalent return non-$\perp$ values. Therefore, when c is created and $c.id = (i, x_i, x_{i+1})$, the maximal path of n already contains queries in positions i and $i+1$, and at least one of them is different to the corresponding query in $c.id$. ☐

Lemma 24. *The maximal path of an inner node contains pending or defined queries in positions 4 and 5. Moreover, for any two distinct inner nodes n_1 and n_2, their maximal paths contain different pairs of queries in positions 4 and 5.*

Proof. If n is an inner node, it must be created in a call to Trigger($i, j, x, u, v, node$) where $i \in \{3, 4, 5, 6\}$. We can observe from the pseudocode that $j \in \{3, 4\}$ in this case, and the three queries (j, x), $(j+1, u)$ and $(j+2, v)$ must consist of one pending query and two defined queries, all of which should be in the maximal path of *node*. Therefore the maximal path of every newly-created inner node already contains pending or defined queries in positions 4 and 5, and at least one of the two queries is defined.

Now we prove the second part of the lemma. Assume by contradiction that there exist distinct inner nodes n_1 and n_2 such that their maximal paths both contain queries $(4, x_4)$ and $(5, x_5)$. When they are created, at least one of the $(4, x_4)$ and $(5, x_5)$ is defined as discussed before. Without loss of generality, assume $(4, x_4)$ becomes defined after $(5, x_5)$; then $(5, x_5)$ is defined when n_1 or n_2 is created.

The origins of n_1 and n_2 cannot be the same: Otherwise, by assumption their maximal paths both contain queries $(4, x_4)$ and $(5, x_5)$, which uniquely determines their *id* and *beginning* with the origin. Thus $n_1.id = n_2.id$ and $n_1.beginning = n_2.beginning$. By Lemmas 7 and 9, the parents of n_1 and n_2 should be the unique node whose *end* equals $n_1.beginning$. This implies that n_1 and n_2 are siblings with the same *id*, contradicting Lemma 23.

Next we show that the origins of n_1 and n_2 cannot be 4 or 5: We prove by contradiction, and without loss of generality, we assume that the origin of n_1 is 4. By Lemma 10, the query $(4, x_4)$

is not defined until SampleTree($n_1.parent$) is called. The origin of n_2 is not 4, so it must be 3, 5 or 6. In all these cases, $(4, x_4)$ should be defined when n_2 is created, so n_2 must be created after n_1 is completed. However, this is not possible since after n_1 is completed the queries $(3, x_3)$, $(5, x_5)$ and $(6, x_6)$ are all defined and hence cannot be $n_2.beginning$ (we let $x_3 = F_4(x_4) \oplus x_5$ and $x_6 = x_4 \oplus F_5(x_5)$ as usual).

The only possibility left is that the origins of n_1 and n_2 are 3 and 6 (or 6 and 3, which is symmetric) respectively. Without loss of generality, assume n_1 is created before n_2. After n_1 is created, MakeNodeReady(n_1) is called and immediately assigns $n_1.end = (6, x_6)$. Since $n_2.beginning = (6, x_6)$, by Lemma 7 we have $n_2.parent = n_1$. However, both queries of $n_2.id$ are contained by the maximal path of n_1, contradicting the second part of Lemma 23. □

Lemma 25. *The simulator creates at most $4q^2 - q$ inner nodes in an execution.*

Proof. By Lemma 24, each inner node corresponds to a unique pair of defined or pending queries in positions 4 and 5.

Other than the queries issued by the distinguisher, a query can only be added to $\tilde{F}_i$ in calls to MakeNodeReady(n) or AdaptNode(n). Specifically, for $i \in \{4, 5\}$, a query is added to $\tilde{F}_i$ only when n is an outer node. Indeed, the issued queries are in the maximal path of n by Lemma 19. If n is an inner node, the queries in positions 4 and 5 of its maximal path are already pending or defined when n is created.

For each outer node n, at most one query in each of positions 4 and 5 becomes pending or defined during MakeNodeReady(n) and AdaptNode(n). By Lemma 16, there exist at most q outer nodes during the execution.

On the other hand, the distinguisher issues at most q queries in each position. Therefore we have $|\tilde{F}_4| \leq 2q$, $|\tilde{F}_5| \leq 2q$. This is enough for upper bounding the number of inner nodes by $4q^2$; the following discussion will improve the bound to $4q^2 - q$.

If the number of outer nodes is less than q, or if at least one outer node n contains a query in position 4 or 5 in its maximal path such that the query is defined or pending before the query is issued in MakeNodeReady(n) or AdaptNode(n), the size of one of $\tilde{F}_4$ and $\tilde{F}_5$ is at most $2q - 1$. In this case we have $|\tilde{F}_4| \cdot |\tilde{F}_5| \leq 2q(2q - 1) < 4q^2 - q$, implying that there are less than $4q^2 - q$ inner nodes.

On the other hand, assume the above event does not occur, i.e., there are q outer nodes and each of them issues or adapts new queries in positions 4 and 5. Consider an arbitrary outer node n whose maximal path contains $(4, x_4)$ and $(5, x_5)$. We will prove that there exists no inner node n' whose maximal path contains $(4, x_4)$ and $(5, x_5)$.

If the origin of n is 2 or 7, then $(4, x_4)$ and $(5, x_5)$ are defined in AdaptNode(n) and $(3, x_3)$ and $(6, x_6)$ are defined in PrepareTree(n)[13]. None of the queries has been pending and therefore no inner path can be triggered by any of them.

Otherwise the origin of n is 1 or 8, and we showcase the former case. The query $(4, x_4)$ only becomes pending at the end of the call to MakeNodeReady(n), before which $(6, x_6)$ and $(5, x_5)$ are queried. Since $(4, x_4)$ has not been defined when $(6, x_6)$ and $(5, x_5)$ become defined, the origin of n' cannot be 5 or 6. If the origin is 4, then $n'.beginning = (4, x_4)$. Since $n.end = (4, x_4)$, by Lemma 9 n' must be a child of n, which contradicts Lemma 23. If the origin of n' is 3, n' must be created after

[13] By assumption $(4, x_4)$ and $(5, x_5)$ are adapted successfully in AdaptNode(n). When PrepareTree(n) is called neither $(3, x_3)$ nor $(6, x_6)$ is defined or pending, otherwise the simulator aborts before adapting the two queries.

$(4, x_4)$ is defined and before the query $(3, x_3)$ is adapted in AdaptNode(n).[14] However, after $(4, x_4)$ is defined and before $(3, x_3)$ is adapted, the simulator is in procedures SampleTree, PrepareTree and AdaptTree where no node is created.

By the above discussion, we know each inner node corresponds to a distinct pair of queries in $\tilde{F}_4 \times \tilde{F}_5$, such that the two queries are not contained in the maximal path of an outer node. Since by assumption q outer nodes exist, whose maximal paths contain different pairs of queries in positions 4 and 5, the number of inner nodes is upper bounded by

$$|\tilde{F}_4| \cdot |\tilde{F}_5| - q \leq (2q)^2 - q = 4q^2 - q.$$

□

Lemma 26. *At most $4q^2$ non-root nodes are created in an execution.*

Proof. A non-root node is either an inner node or an outer node. By Lemmas 25 and 16, The total number of non-root nodes is upper bounded by $(4q^2 - q) + q = 4q^2$. □

Lemma 27. *At any point of an execution, the number of pending or defined queries satisfies $|\tilde{F}_i| \leq 2q$ for $i \in \{4, 5\}$, $|\tilde{F}_i| \leq 4q^2$ for $i \in \{1, 8\}$, and $|\tilde{F}_i| \leq 4q^2 + q$ for $i \in \{2, 3, 6, 7\}$.*

Proof. In the proof for Lemma 25, we proved that $|\tilde{F}_4| \leq 2q$, $|\tilde{F}_5| \leq 2q$.

The queries in $\tilde{F}_i$ for $i \in \{1, 8\}$ are added by distinguisher queries or if the query is in the maximal path of an inner node (similarly to the analysis in the proof of Lemma 25, the queries in positions 1 and 8 in the maximal path of an outer node are defined or pending when the node is created). There are at most q distinguisher queries and at most $4q^2 - q$ inner nodes by Lemma 25, therefore each of $\tilde{F}_1$ and $\tilde{F}_8$ contains at most $4q^2$ queries.

The queries positions 2, 3, 6 and 7 can become pending or defined for both inner nodes and outer nodes. There are at most $4q^2$ non-root nodes and at most q distinguisher queries in each position, thus the size of $\tilde{F}_i$ is upper bounded by $4q^2 + q$ for $i \in \{2, 3, 6, 7\}$. □

Lemma 28. *We have $|F_i| \leq 2q$ for $i \in \{4, 5\}$, $|F_i| \leq 4q^2$ for $i \in \{1, 8\}$, and $|F_i| \leq 4q^2 + q$ for $i \in \{2, 3, 6, 7\}$. In games $\mathrm{G}_2, \mathrm{G}_3$ and G_4, we have $|T| \leq 4q^2$.*

Proof. Since F_i are subsets of $\tilde{F}_i$, the upper bounds on $|F_i|$ follow by Lemma 27.

In $\mathrm{G}_2, \mathrm{G}_3$ and G_4, procedures CheckP$^+$ and CheckP$^-$ do not add entries to T. Therefore, new queries are added to T only by distinguisher queries or by simulator queries in MakeNodeReady. Moreover, if n is an outer node, the permutation query made in MakeNodeReady(n) (if exists) is the one queried in the call to CheckP$^+$ or CheckP$^-$ before n is added (which preexists in T even before the call occurs). Thus only when n is an inner node does the simulator make new permutation queries in MakeNodeReady(n). By Lemma 25, the number of inner nodes is at most $4q^2 - q$. The distinguisher queries the permutation oracle at most q times, so the size of T is upper bounded by $(4q^2 - q) + q = 4q^2$. □

Lemma 29. *Consider an execution of G_1. If the simulator calls $\mathrm{SimP}(x_0, x_1)$, we have $x_1 \in F_1$ and $x_2 := F_1(x_1) \oplus x_0 \in \tilde{F}_2$. Symmetrically, if the simulator calls $\mathrm{SimP}^{-1}(x_8, x_9)$, we have $x_8 \in F_8$ and $x_7 := F_8(x_8) \oplus x_9 \in \tilde{F}_7$.*

[14] The call to Adapt$(3, x_3, y_3)$ may define $(3, x_3)$ or may abort; in both cases n' cannot be created after the call.

Proof. First consider the first statement. The simulator queries SimP only in procedures CheckP^- and MakeNodeReady.

If the query to SimP is made in a call to CheckP^-, we can see from the pseudocode that x_1 and x_2 equals the first two arguments of the call. CheckP^- is only called by FindNewChildren(n) (via Trigger) when the origin of n is 8 or 2. If the origin is 2, $x_1 \in F_1$ and $(2, x_2) = n.end$ (so $x_2 \in \tilde{F}_2$ by Lemma 6); if the origin is 8, we must have $x_1 \in F_1$ and $x_2 \in F_2$ by observing the FindNewChildren procedure.

If the query to SimP is made in a call to MakeNodeReady(n), then the origin of n must be 1, 2, 5 or 6 so that Prev is used. If the origin of n is 1, $\text{Prev}(i, x_i, x_{i+1})$ is never called with $i = 0$ and SimP is never called. If the origin is 2, then the call to SimP is exactly the same as the call in CheckP^- right before the node is created (and the result has been proved for this case). If the origin is 5 or 6, MakeNodeReady(n) calls Prev and make queries in positions 2 and 1 before the permutation query is made in the next Prev call. By the time the permutation query is called, both $\text{F}(2, x_2)$ and $\text{F}(1, x_1)$ have been called and thus both queries are defined.

For the second statement, SimP^{-1} is called by CheckP^+, MakeNodeReady and PrepareTree. The first two cases are symmetric to the first statement. For the call in PrepareTree, we can observe that right before $\text{SimP}^{-1}(x_8, x_9)$ is called, $(7, x_7)$ and $(8, x_8)$ are defined in the two calls to ReadTape made by PrepareTree. □

The next lemma upper bounds the query complexity of the simulator (in G_1).

Theorem 30. *In the simulated world* G_1*, the simulator calls each of P and* P^{-1} *for at most* $16q^4 + 4q^3$ *times.*

Proof. The simulator calls P and P^{-1} via the wrapper functions SimP and SimP^{-1} respectively. The functions maintain tables T_{sim} and T_{sim}^{-1}, consisting of previously made permutation queries. When SimP (resp. SimP^{-1}) is called, they first check whether the query already exists in the tables; if so, the table entry is directly returned without actually calling P (P^{-1}). Therefore, P (P^{-1}) is queried only when SimP (SimP^{-1}) receives a new query for the first time, and we only need to upper bound the number of distinct queries to SimP and SimP^{-1}.

Again we only give a proof for SimP, and the proof for SimP^{-1} is symmetric. By Lemma 29, if the simulator calls $\text{SimP}(x_0, x_1)$, we have $x_1 \in F_1$ and $x_2 := F_1(x_1) \oplus x_0 \in \tilde{F}_2$ by the end of the execution. Note that each pair of x_1 and x_2 determines a unique query $(F_1(x_1) \oplus x_2, x_1)$. Thus, the number of distinct queries to SimP is at most

$$|F_1 \times \tilde{F}_2| = |F_1| \cdot |\tilde{F}_2| \leq (4q^2) \cdot (4q^2 + q) = 16q^4 + 4q^3$$

where the inequality is due to Lemmas 27 and 28. □

Lemma 31. *The number of root nodes created in an execution is upper bounded by* $24q^2 + 8q$.

Proof. Root nodes immediately become ready after being created. By Lemma 7, each root node has a distinct *end* (i, x_i) which is in the set $\tilde{F}_i$. Therefore, the number of root nodes is upper bounded by the sum of sizes of $\tilde{F}_i$, which is at most $24q^2 + 8q$ by Lemma 27. □

Finally we upper bound the time complexity of the simulator.

Theorem 32. *The running time of the simulator in* G_1 *is* $O(q^{10})$.

Proof. Note that most procedures in the pseudocode runs in constant time without making calls to other procedures, thus can be treated as a single command. We only need to upper bound the running time of the procedures with loops and those that call other procedures. Unless otherwise specified, the following discussion considers the running time inside a procedure, i.e., the running time of the called procedures (that are not constant-time) is not included in the running time of the caller.

First consider the procedures that are called at most once per node, including procedures AddChild, MakeNodeReady, SampleTree, PrepareTree and AdaptNode. AdaptTree is called once for each tree (i.e., each root node). Next and Prev are called in MakeNodeReady for a constant number of times. F is called once in each call to Next or Prev, and at most $8q$ times by the distinguisher. NewTree is only called in F, and IsPending is called in F and MakeNodeReady. By Lemmas 26 and 31, there are at most $4q^2$ non-root nodes and at most $24q^2 + 8q$ root nodes, thus each of these procedures is called $O(q^2)$ times.

The running time of the above procedures are also related to the number of nodes. The loops in IsPending, SampleTree, AdaptTree, and AdaptNode iterates over a subset of all nodes, whose size is at most $O(q^2)$. The other procedures run in constant time. Therefore, the total running time of the aforementioned procedures is $O(q^2) \cdot O(q^2) = O(q^4)$.

Now consider a call to GrowTree(*root*) where *root* is the root of a tree that has τ nodes after GrowTree returns. GrowTree repeatedly calls GrowTreeOnce to add newly triggered nodes into the current tree, until no change is made in an iteration. At most $\tau - 1$ calls can add new nodes to the tree, therefore GrowTreeOnce(*root*) is called at most τ times. GrowTreeOnce is called recursively on every node in the tree, and calls FindNewChildren on each node. Therefore, FindNewChildren is called at most τ times in each iteration, and thus a total of τ^2 times in GrowTree(*root*).

The procedure FindNewChildren iterates through two tables. By Lemma 28, the number of pairs in the tables is at most $(4q^2 + q)^2 < 25q^4$, which is the number of times Trigger is called. Equivalent runs in constant time and InChildren runs in $O(q^2)$ time (the node has at most $O(q^2)$ children by Lemma 26). Thus the total running time of the life cycle of FindNewChildren is at most $O(q^6)$, including the called procedures.

By Lemmas 26 and 31, the total number of nodes in the trees is at most $4q^2 + 24q^2 + 8q = O(q^2)$, i.e., $\sum \tau = O(q^2)$. The time complexity the GrowTree cycle is dominated by the the running time of FindNewChildren, which is

$$\left(\sum \tau^2\right) \cdot O(q^6) \le \left(\sum \tau\right)^2 \cdot O(q^6) = \left(O\left(q^2\right)\right)^2 \cdot O(q^6) = O(q^{10}).$$

In conclusion, the time complexity of the simulator in G_1 is $O(q^{10})$. □

The last theorem upper bounds the running time of our simulator as programmed in our pseudocode. However, it is possible to significantly speed up the tree-growing procedures by using hash tables instead of tree traversals. In particular, we can improve the running time of the simulator to $O(q^4)$ at the cost of spending $O(q^4)$ extra space to store the hash tables.

For the following modified analysis, we assume that hash tables can be accessed with constant time, just like the tables F_i and T. In a real world implementation, the overhead of the tables is at most logarithmic.

Theorem 33. *The simulator of game* G_1 *can be implemented to run in time* $O(q^4)$*.*

Proof Sketch. Recall that every position $i \in [8]$ is at the endpoint of a unique detect zone. For this proof, the two other positions in the detect zone will be called the *trigger positions* of i. For example, the trigger positions of 1 are 7 and 8, and so on.

We can optimize the simulator as follows, while keeping an equivalent functionality:

- For each position $i \in [8]$, we maintain a set $\mathsf{Pending}_i$ containing the values x_i such that (i, x_i) is a pending query.
- For each position $i \in [8]$, we maintain a hash table $\mathsf{Trigger}_i$ that maps a n-bit string x_i to a stack,[15] such that if (i, x_i) is a pending query then $\mathsf{Trigger}_i(x_i)$ contains the set of triggers for (i, x_i).
- The procedure FindNewChildren(*node*) with $node.end = (i, x_i)$ doesn't iterate through all pairs of defined queries in the trigger positions of i; instead, it empties $\mathsf{Trigger}_i(x_i)$ and calls Trigger with these values only.
- The procedure Trigger does not call CheckP$^+$, CheckP$^-$, Equivalent or InChildren, but creates the child without checking anything.

The correctness follows by the assumption that $\mathsf{Trigger}_i(x_i)$ contains exactly the triggers for (i, x_i) at all points in time such that (i, x_i) is a pending query. The latter assumption will be ensured by the following properties, maintained throughout an execution: (1) before a query (i, x_i) becomes pending, $\mathsf{Trigger}_i(x_i)$ is empty; (2) when a query (i, x_i) becomes pending, all of (i, x_i)'s triggers are pushed onto $\mathsf{Trigger}_i(x_i)$; (3) when a query becomes defined, any new triggers involving this query are pushed onto the relevant stacks; (4) when a non-root node is created, the trigger responsible for creating the node is no longer in the stack.

We give more details about how the hash tables and stacks are updated. The tables $\mathsf{Pending}_i$ are modified whenever the set of ready nodes N is modified, which costs $O(q^2)$ time since there are at most $O(q^2)$ nodes by Lemmas 26, 31.

The stack $\mathsf{Trigger}_i(x_i)$ is initialized when the query (i, x_i) becomes pending. This can be done in time $O(q^2)$ if (i, x_i) is adjacent to one of i's trigger positions. For example, if $i = 2$, then the simulator checks all $x_1 \in F_1$, queries $\mathrm{P}(x_0, x_1) = (x_8, x_9)$ for $x_0 = F_1(x_1) \oplus x_2$, and pushes (x_8, x_1) onto the stack only if $x_8 \in F_8$. However, if $i = 1$ (or $i = 8$), the simulator has to check all pairs of (x_7, x_8), which takes $O(q^4)$ time.

We can reduce this cost by additionally maintaining two hash tables:

- For $i \in \{1, 8\}$, a hash table Perm_i maps a value x_i to a stack of pairs, where $\mathsf{Perm}_1(x_1)$ contains all pairs (x_7, x_8) such that $x_7 \in F_7$, $x_8 \in F_8$ and such that $\mathrm{P}(x_8, x_9) = (*, x_1)$ for $x_9 = x_7 \oplus F_8(x_8)$, and where $\mathsf{Perm}_8(x_8)$ contains all pairs (x_1, x_2) such that $x_1 \in F_1$, $x_2 \in F_2$ and such that $\mathrm{P}(x_0, x_1) = (x_8, *)$ for $x_0 = F_1(x_1) \oplus x_2$.

The stacks are only updated when a query in position 1, 2, 7 or 8 are defined, and each update requires going through all defined queries in one position, taking $O(q^2)$ time. At most $O(q^2)$ queries are defined, and the total cost of the tables is $O(q^4)$.

With these tables, when $(1, x_1)$ becomes pending, the simulator can empty the stack $\mathsf{Perm}_1(x_1)$ and check for triggers.[16] Since the sum of the sizes of the lists is at most

$$|F_7| \cdot |F_8| \leq O(q^4),$$

[15] Initially, the table maps everything to null-value. A stack $\mathsf{Trigger}_i(x_i)$ is created when $\mathsf{Trigger}_i(x_i)$ is called for the first time. Therefore the space consumption is due to the sizes of the stacks, and unused stacks will not take up space. This is also how we implement the other mappings in the proof.

[16] At most one of the entries is not a trigger, i.e., if $(1, x_1)$ is the *end* of a non-root node n, the pair contained by the maximal path of n is not a trigger.

the total time for initializing $\mathsf{Trigger}_1$ is $O(q^4)$. The same bound can be proved for $\mathsf{Trigger}_8$.

When a query (h, x_h) becomes defined (either in ReadTape or in Adapt), the stack $\mathsf{Trigger}_i$ must be updated for every i such that h is a trigger position of i. If the trigger positions of i are adjacent (i.e., $i \neq 2, 7$), the update is easy: for example, if $(2, x_2)$ becomes defined, the simulator checks all $x_1 \in F_1$ and computes $x_8 = \mathrm{Val}^-(1, x_1, x_2, 8)$; if $(8, x_8)$ is pending and $(1, x_1, x_2)$ is not in the maximal path of n (with $n.end = (8, x_8)$), we push (x_1, x_2) onto $\mathsf{Trigger}_8(x_8)$. (Note that we only need to check that the trigger is not contained in the maximal path of the node; no child can be equivalent since the query has just been defined.)

However, the trigger positions of 2 (and 7) are 1 and 8, which are not adjacent and cannot be updated using this naïve approach. For these positions we need more hash tables $\mathsf{TrigHelper}_i$:

- For $i \in \{1, 8\}$, we maintain a hash table $\mathsf{TrigHelper}_i$ that maps a value x_i to a stack of pairs, where $\mathsf{TrigHelper}_8(x_8)$ contains all pairs (x_1, x_2) such that $x_1 \in F_1$, $x_2 \in \tilde{F}_2$ and $\mathrm{P}(F_1(x_1) \oplus x_2, x_1) = (x_8, *)$, and where $\mathsf{TrigHelper}_1(x_1)$ contains all pairs (x_7, x_8) such that $x_8 \in F_8$, $x_7 \in \tilde{F}_7$ and $\mathrm{P}(x_8, x_7 \oplus F_8(x_8)) = (*, x_1)$.

The stacks $\mathsf{TrigHelper}_8$ are updated each time a query in position 2 becomes pending or a query in position 1 becomes defined, and symmetrically for the stacks $\mathsf{TrigHelper}_1$. The cost for every such update is $O(q^2)$ as before.

Now we describe how triggers for pending queries in position 2 are updated; it is symmetric for pending queries in position 7. When $(2, x_2)$ becomes pending, the simulator scans all pairs of defined queries in positions 1 and 8, adding triggers for $(2, x_2)$ to $\mathsf{Trigger}_2(x_2)$. (It also updates $\mathsf{TrigHelper}_8$ and performs the other jobs described before.) Whenever a query in position 1 becomes defined, the simulator updates $\mathsf{Trigger}_2(x_2)$ for all $x_2 \in \mathsf{Pending}_2$ by checking whether $\mathrm{P}(F_1(x_1) \oplus x_2, x_1) = (x_8, *)$ for some $x_8 \in F_8$; if so, (x_8, x_1) is pushed onto $\mathsf{Trigger}_2(x_2)$. On the other hand, when a query in position 8 becomes defined, the simulator checks whether $\mathsf{TrigHelper}_8(x_8)$ is nonempty; if so, the entries are "translated" to entries in $\mathsf{Trigger}_2(x_2)$. More formally, when $(8, x_8)$ is defined, for each entry $(x_1, x_2) \in \mathsf{TrigHelper}_8(x_8)$, an entry (x_8, x_1) is pushed onto $\mathsf{Trigger}_2(x_2)$.

We note that the stacks $\mathsf{Trigger}_i(x_i)$ are also modified by FindNewChildren: as mentioned before, when FindNewChildren creates a node using a trigger, the trigger is popped from $\mathsf{Trigger}_i(x_i)$.

When a query becomes pending or defined, the cost of updating all of the stacks is $O(q^2)$. Since at most $O(q^2)$ queries become pending or defined throughout the execution, and combining other upper bounds, the total cost of maintaining the stacks is at most $O(q^4)$.

As proved in Theorem 32, the running time of the procedures outside GrowTree is $O(q^4)$. The running time inside GrowTree is dominated by FindNewChildren. For the improved version, the running time of FindNewChildren is constant for each call plus a constant for each triggered path. Recall that FindNewChildren is called $O(q^4)$ (proved in Theorem 32) times and the number of triggered paths is $O(q^2)$. Therefore, the total running time of the optimized simulator is $O(q^4)$.

Although the optimized simulator makes some additional permutation queries, each permutation query still always corresponds to some pair $x_1 \in F_1$, $x_2 \in \tilde{F}_2$ or to some pair $x_7 \in \tilde{F}_7$, $x_8 \in F_8$, so the same query complexity bound (cf. Theorem 30) holds as before. □

5.2 Transition from G_1 to G_2

Modifications in G_2. The game G_2 differs from G_1 in two places: the procedures CheckP^+ and CheckP^- and the procedures Val^+ and Val^-. We will use the previous convention and call the version of a procedure used in G_1 the G_1-*version* of the procedure, while the version used in G_2 is called the G_2-*version* of the procedure. We note that the G_2-version of CheckP^+, CheckP^-, Val^+ and Val^- are also used in games G_3 and G_4.

In the G_2-version of CheckP^+ and CheckP^-, the simulator checks whether the permutation query already exists in the table T; if not, **false** is returned without calling SimP or SimP^{-1}. Therefore, if a permutation query is issued in CheckP^+ or CheckP^-, it must already exist in the table T, i.e., the query has been issued by the distinguisher or by the simulator before.

Note that the CheckP^+ and CheckP^- are called by FindNewChildren and are responsible for determining whether a triple of queries are in the same path. The G_2-version may return false negatives if the permutation query in the path hasn't been made, and the path won't be triggered in G_2. We will prove that such a path is unlikely to be triggered in G_1, either.

We say two executions of G_1 and G_2 are *identical* if every procedure call returns the same value in the two executions. Since the distinguisher is deterministic and it only interacts with procedures F, P, and P^{-1}, it issues the same queries and outputs the same value in identical executions.

Lemma 34. *We have*

$$\Delta_D(G_1, G_2) \leq 500q^6/2^n.$$

Proof. This proof uses the divergence technique of Lemma 40 in [12].

Note that if $q \geq 2^{n-2}$ the bound trivially holds, so we can assume $q \leq 2^{n-2}$.

Consider executions of G_1 and G_2 with the same random tapes $f_1, \ldots, f_8, p$.

We say the two executions *diverge* in a call to $\text{CheckP}^+(x_7, x_8, x_1)$ (resp. $\text{CheckP}^-(x_1, x_2, x_8)$), if in the execution of G_2, we have $p^{-1}(x_8, x_9) = (*, x_1)$ and $(x_8, x_9) \notin T^{-1}$ (resp. $p(x_0, x_1) = (x_8, *)$ and $(x_0, x_1) \notin T$). Note that x_9 and x_0 are defined as in the pseudocode of CheckP^+ and CheckP^-, i.e., according to the Feistel construction. It is easy to check that a call to CheckP^+ or CheckP^- returns the same answer in the two executions, unless the two executions diverge in this call.

Now we argue that two executions are identical if they don't diverge. We do this by induction on the number of function calls. Firstly note that the only procedures to use the tables T, T^{-1} and/or T_{sim}, T_{sim}^{-1} are CheckP^+, CheckP^- P, P^{-1}, PSim, PSim^{-1}, Val^+ and Val^-. CheckP^+ and CheckP^- always return the same answer as long as divergence doesn't occur, as discussed above. The procedures P, P^{-1}, PSim, PSim^{-1} always return the same values as well, because the value returned by these procedures is in any case compatible with p. Lastly the table entries read by Val^+ and Val^- in G_1 and G_2 must exist by Lemma 21, and are in both cases compatible with the tape p by Lemma 4, so Val^+ and Val^- behave identically as well. Moreover CheckP^+, CheckP^-, P, P^{-1}, PSim, PSim^{-1}, Val^+ and Val^- do not make changes to other global variables besides the tables T, T^{-1}, T_{sim} and T_{sim}^{-1}, so the changes to these tables do not propagate via side-effects. Hence, two executions are identical if they do not diverge.

Next we upper bound the probability that the two executions diverge. The probability is taken over the choice of the random tapes. Note that divergence is well defined in G_2 alone: An execution of G_2 diverges if and only if in a call to CheckP^+ we have $(x_8, x_9) \notin T^{-1}$ and $p^{-1}(x_8, x_9) = (*, x_1)$, or in a call to CheckP^- we have $(x_0, x_1) \notin T$ and $p(x_0, x_1) = (x_8, *)$. We compute the probability of the above event in G_2.

Due to symmetry of CheckP^+ and CheckP^-, we only discuss CheckP^- below, and the same bound applies for CheckP^+. We will upper bound the probability that divergence occurs in each call to CheckP^-, and then apply a union bound.

If $(x_0, x_1) \in T$, divergence won't occur. Otherwise if $(x_0, x_1) \notin T$, the tape entry $p(x_0, x_1)$

hasn't been read in the execution, because p is only read in P and P^{-1} and an entry is added to T immediately after it is read. The value of $p(x_0, x_1)$ is uniformly distributed over $\{0,1\}^{2n} \backslash \{T(x'_0, x'_1) : x'_0, x'_1 \in \{0,1\}^n\}$. By Lemma 28, the size of T is at most $4q^2$, so $p(x_0, x_1)$ is distributed over at least $2^{2n} - 4q^2$ values. There are 2^n values of the form $(x_8, *)$, and $p(x_0, x_1)$ equals one of them with probability at most $2^n/(2^{2n} - 4q^2)$. In both cases, the probability that divergence occurs in the CheckP$^-$ call is upper bounded by $2^n/(2^{2n} - 4q^2)$.

CheckP$^-$ is only called in FindNewChildren, and its arguments correspond to three queries that are pending or defined in positions 1, 2, 8. By Lemma 27, the number of pending or defined queries in each of these positions is at most $4q^2 + q$, and CheckP$^-$ is called on at most $(4q^2 + q)^3$ distinct arguments.

If CheckP$^-$ is called multiple times with the same argument (x_1, x_2, x_8), divergence either occurs for the first of these calls or else occurs for none of these calls. Thus we only need to consider the probability of divergence in the first call to CheckP$^-$ with a given argument. Using a union bound over the set of all distinct arguments with which CheckP$^-$ is called, the probability that divergence occurs is at most

$$(4q^2 + q)^3 \cdot \frac{2^n}{2^{2n} - 4q^2} \leq 125q^6 \cdot \frac{2^n}{2^{2n} - 2^{2n}/4} \leq \frac{250q^6}{2^n}$$

where the first inequality is due to the assumption mentioned at the start of the proof that $q \leq 2^{n-2}$.

The same upper bound holds for the probability that divergence occurs in a call to CheckP$^+$. With a union bound, divergence occurs in an execution of G_2 with probability at most $500q^6/2^n$.

The distinguisher D outputs the same value in identical executions, so the probability that D has different outputs in the two executions is upper bounded by $500q^6/2^n$, which also upper bounds the advantage of D in distinguishing G_1 and G_2. □

5.3 Transition from G_2 to G_3

Modifications in G_3. Compared to G_2, the calls to procedures CheckBadP, CheckBadR, SetToPrep and CheckBadA are added in G_3. These procedures make no modification to the tables; they only cause the simulator to abort in certain situations. Thus a non-aborted execution of G_3 is identical to the G_2-execution with the same random tapes.

There is no need to compute $\Delta_D(\mathrm{G}_2, \mathrm{G}_3)$; instead, we prove that the advantage of D in distinguishing between G_3 and G_5 is greater than or equal to that between G_2 and G_5.

Lemma 35. *We have*

$$\Delta_D(\mathrm{G}_2, \mathrm{G}_5) \leq \Delta_D(\mathrm{G}_3, \mathrm{G}_5).$$

Proof. By the definition of advantage in equation (4), we have

$$\Delta_D(\mathrm{G}_i, \mathrm{G}_5) = \Pr_{\mathrm{G}_i}[D^{\mathrm{F},\mathrm{P},\mathrm{P}^{-1}} = 1] - \Pr_{\mathrm{G}_5}[D^{\mathrm{F},\mathrm{P},\mathrm{P}^{-1}} = 1].$$

Thus we only need to prove that D outputs 1 in G_3 with at least as high a probability as in G_2. This trivially follows from the observation that the only difference between G_3 and G_2 is that additional abort conditions are added in G_3, and that the distinguisher outputs 1 when the simulator aborts. □

5.4 Bounding the Abort Probability in G_3

Categorizing the Aborts. The simulator in G_3 aborts in many conditions. We can categorize the aborts into two classes: those that occur in the Assert procedure, and those that occur in procedures CheckBadP, CheckBadR, and CheckBadA. As will be seen in the proof, the Assert procedure *never* aborts in G_3. On the other hand, CheckBadP, CheckBadR and CheckBadA will abort with small probability.

Let BadP, BadR, and BadA denote the events that the simulator aborts in CheckBadP, CheckBadR, and CheckBadA respectively. These three events are collectively referred to as the *bad events*. CheckBadP is called in P and P^{-1}, CheckBadR is called in ReadTape, and CheckBadA is called right before the nodes are adapted in NewTree. A more detailed description of each bad event will be given later.

This section consists of two parts. We first upper bound the probability of bad events. Then we prove that in an execution of G_3, the simulator does not abort inside of calls to Assert.

5.4.1 Bounding Bad Events

We start by making some definitions. In this section, we say a query is *active* if it is pending or defined. A 2chain is *active* if it is both left active and right active as defined below:

Definition 10. A 2chain (i, x_i, x_{i+1}) is *left active* if $i \geq 1$ and the query (i, x_i) is active, or if $i = 0$ and $(x_i, x_{i+1}) \in T$. Symmetrically, the 2chain is *right active* if $i \leq 7$ and the query $(i+1, x_{i+1})$ is active, or if $i = 8$ and $(x_i, x_{i+1}) \in T^{-1}$.

We note that ActiveQueries returns not only the active queries, but also the queries in the set *ToPrep*. The reason will be clear after seeing the definition of the event BadRPrepare. If we treat the queries in BadRPrepare as active, the procedures IsLeftActive, IsRightActive, and IsActive in the pseudocode check whether a 2chain is left active, right active, and active, respectively.

Incidences Between 2Chains and Queries. The following definitions involve the procedures Val^+ and Val^-, which are defined in the pseudocode. Recall that we are using the G_2-version of the procedures.

The answers of Val^+ and Val^- are time dependent: $\perp$ may be returned if certain queries in the path hasn't been defined. Thus the following definitions are also time dependent.

The notion of a query being "incident" with a 2chain is defined below, which will be used in the bad events.

Definition 11. A query (i, x_i) is *incident* with a 2chain (j, x_j, x_{j+1}) if $i \notin \{j, j+1\}$ and if either $\text{Val}^+(j, x_j, x_{j+1}, i) = x_i$ or $\text{Val}^-(j, x_j, x_{j+1}, i) = x_i$.

Lemma 36. *A query (i, x_i) is incident with an active 2chain if and only if at least one of the following is true:*

- *$i \geq 2$ and there exists an active 2chain $(i-2, x_{i-2}, x_{i-1})$ such that* $\text{Val}^+(i-2, x_{i-2}, x_{i-1}, i) = x_i$*;*
- *$i \in \{0, 1\}$ and there exists an active 2chain $(8, x_8, x_9)$ such that* $\text{Val}^+(8, x_8, x_9, i) = x_i$*;*
- *$i \leq 7$ and there exists an active 2chain $(i+1, x_{i+1}, x_{i+2})$ such that* $\text{Val}^-(i-2, x_{i-2}, x_{i-1}, i) = x_i$*;*
- *$i \in \{8, 9\}$ and there exists an active 2chain $(0, x_0, x_1)$ such that* $\text{Val}^-(0, x_0, x_1, i) = x_i$*.*

Proof. The "if" direction is trivial since the query (i, x_i) is incident with the active 2chain in each case.

For the "only if" direction, suppose the query is incident with an active 2chain (k, x'_k, x'_{k+1}) where $i \notin \{k, k+1\}$. We assume $\mathrm{Val}^+(k, x'_k, x'_{k+1}, i) = x_i$, and the other case is symmetric.

From the implementation of Val^+, we observe that there exists a partial path $\{x'_h\}_{h=k}^{i}$ such that $x'_i = x_i$, where x'_h equals the value of the variable x_h in the pseudocode.

If $i \geq 2$, since $i \notin \{k, k+1\}$, x'_{i-2} and x'_{i-1} exist in the partial path. If $k = i - 2$, the 2chain $(i-2, x'_{i-2}, x'_{i-1}) = (k, x'_k, x'_{k+1})$ and is active by assumption. Otherwise, neither $i-1$ nor $i-2$ is an endpoint of the partial path, which implies that $x'_{i-1} \in F_{i-1}$ and that $x'_{i-2} \in F_{i-2}$ if $i > 2$ and $(x'_{i-2}, x'_{i-1}) \in T$ if $i = 2$. Thus the 2chain is active. Moreover, $\mathrm{Val}^+(i-2, x'_{i-2}, x'_{i-1}, i) = x'_i = x_i$.

If $i \in \{0, 1\}$, we have $k > i$. Similarly one can see that the 2chain $(8, x'_8, x'_9)$ is active by looking separately at the cases $k = 8$ and $k < 8$, and that $\mathrm{Val}^+(8, x'_8, x'_9, i) = x_i$. □

Number of Active 2chains. In order to upper bound the probability of bad events, we need to upper bound the number of active 2chains.

Recall $\tilde{F}_i$ is the set of active queries in position i. By Definition 10, if a 2chain (i, x_i, x_{i+1}) is left active and $i \geq 1$, we must have $x_i \in \tilde{F}_i$; if (i, x_i, x_{i+1}) is right active and $i \leq 7$, $x_{i+1} \in \tilde{F}_{i+1}$.

We extend the definition of sets $\tilde{F}_i$ for $i = 0, 9$ as follows: $\tilde{F}_0$ is the set of values x_0 such that $(0, x_0, x_1)$ is left active for some x_1, while $\tilde{F}_9$ is the set values of x_9 such that $(8, x_8, x_9)$ is right active for some x_8. Or, equivalently:

$$\tilde{F}_0 := \{x_0 : \exists x_1 \text{ s.t. } T(x_0, x_1) \neq \perp\}, \text{ and}$$
$$\tilde{F}_9 := \{x_9 : \exists x_8 \text{ s.t. } T^{-1}(x_8, x_9) \neq \perp\}.$$

In particular we have $|\tilde{F}_0| \leq |T|$ and $|\tilde{F}_9| \leq |T|$.

Lemma 37. *If a 2chain (i, x_i, x_{i+1}) is left active, $x_i \in \tilde{F}_i$; if it is right active, $x_{i+1} \in \tilde{F}_{i+1}$.*

Proof. Recall that $\tilde{F}_i$ is the set of active queries (i, x_i) for $1 \leq i \leq 8$. This lemma follows from the definition of left active, right active, and from the definition of the sets $\tilde{F}_i$ for $0 \leq i \leq 9$. □

We note that Lemma 37 is not if-and-only-if; for example, if x_0, x_1 are values such that $x_0 \in \tilde{F}_0$ and $T(x_0, x_1) = \perp$, then $(0, x_0, x_1)$ is not left active. (However, the first part of Lemma 37 is if-and-only-if for $1 \leq i \leq 8$, and symmetrically, the second part is if-and-only-if for $0 \leq i \leq 7$.)

Lemma 38. *We have $|\tilde{F}_i| \leq 4q^2$ for $i \in \{0, 9\}$, and $|\tilde{F}_i| \leq 4q^2 + q$ for all $\tilde{F}_i$.*

Proof. By Lemma 28, we have $|\tilde{F}_0| \leq |T| \leq 4q^2$ and $|\tilde{F}_9| \leq |T| \leq 4q^2$. The second statement then follows by Lemma 27. □

Definition 12. Let $\mathcal{C}_i$ denote the set of x_i such that (i, x_i) is incident with an active 2chain.

Lemma 39. *We have $|\mathcal{C}_i| \leq 2(4q^2 + q)^2$, i.e., the number of queries in position i that are incident with an active 2chain is at most $2(4q^2 + q)^2$.*

Proof. By Lemma 36, a query (i, x_i) is incident with an active 2chain only if there exists an active 2chain (j, x_j, x_{j+1}) for

$$j = \begin{cases} i-2 & \text{if } i \geq 2 \\ 8 & \text{if } i \leq 1 \end{cases}$$

such that $\mathrm{Val}^+(j, x_j, x_{j+1}, i) = x_i$, or if there exists an active 2chain (j, x_j, x_{j+1}) for

$$j = \begin{cases} i+1 & \text{if } i \leq 7 \\ 0 & \text{if } i \geq 8 \end{cases}$$

such that $\mathrm{Val}^-(j, x_j, x_{j+1}, i) = x_i$. Moreover, the total number of active 2chains in each position is at most $(4q^2 + q)^2$ by Lemma 38. □

The explicit definitions of the bad events BadP, BadR and BadA given below in Definitions 13, 15 and 16 are equivalent to the abort conditions that are checked in the procedures CheckBadP, CheckBadR and CheckBadA respectively.

Bad Permutation. The procedure CheckBadP is called in P and P^{-1}. BadP is the event that a new permutation query "hits" a query in position 1 or 8 (depending on the direction of the permutation query) that is active or is incident with an active 2chain:

Definition 13. BadP occurs in $\mathrm{P}(x_0, x_1)$ if at the beginning of the procedure, we have $(x_0, x_1) \notin T$ and for $(x_8, x_9) = p(x_0, x_1)$, either $x_8 \in \tilde{F}_8$ or $x_8 \in \mathcal{C}_8$. Similarly, BadP occurs in $\mathrm{P}^{-1}(x_8, x_9)$ if at the beginning of the procedure, $(x_8, x_9) \notin T^{-1}$ and for $(x_0, x_1) = p^{-1}(x_8, x_9)$, either $x_1 \in \tilde{F}_1$ or $x_1 \in \mathcal{C}_1$.

Lemma 40. *The probability that* BadP *occurs in an execution of* G_3 *is at most* $432q^6/2^n$.

Proof. As in Lemma 34 we can assume that $q \leq 2^{n-2}$ since the statement trivially holds otherwise.

When a query $\mathrm{P}(x_0, x_1)$ is issued with $(x_0, x_1) \notin T$, the tape entry $p(x_0, x_1)$ has not been read. Since p encodes a permutation, and since whenever an entry of p is read it is added to the table T, the value of $p(x_0, x_1)$ is uniformly distributed on the $2n$-bit strings that are not in T. By Lemma 28 we have $|T| \leq 4q^2$, thus $p(x_0, x_1)$ is distributed on at least $2^{2n} - 4q^2$ values, and each value is chosen with probability at most $1/(2^{2n} - 4q^2)$.

BadP occurs in $\mathrm{P}(x_0, x_1)$ only if $x_8 \in \tilde{F}_8 \cup \mathcal{C}_8$ where x_8 is the first half of the tape entry $p(x_0, x_1) = (x_8, x_9)$. By Lemma 28 we have $|F_8| \leq 4q^2$, and by Lemma 39 we have $|\mathcal{C}_8| \leq 2(4q^2+q)^2$. There are at most 2^n possible values for x_9, therefore BadP occurs when (x_8, x_9) equals one of the at most $(4q^2 + 2(4q^2+q)^2) \cdot 2^n$ pairs. The probability of each pair is at most $1/(2^{2n} - 4q^2)$, so BadP occurs in $\mathrm{P}(x_0, x_1)$ with probability at most $2^n \cdot (4q^2 + 2(4q^2+q)^2)/(2^{2n} - 4q^2)$.

The same bound can be proved symmetrically for a call to $\mathrm{P}^{-1}(x_8, x_9)$ with $(x_8, x_9) \notin T^{-1}$.

Each call to $\mathrm{P}(x_0, x_1)$ with $(x_0, x_1) \notin T$ or to $\mathrm{P}^{-1}(x_8, x_9)$ with $(x_8, x_9) \notin T^{-1}$ adds an entry to the table T. By Lemma 28, the size of T is at most $4q^2$, so the total number of such calls is upper bounded by $4q^2$. With a union bound, the probability that BadP occurs in at least one of these calls is at most

$$4q^2 \cdot \frac{2^n \cdot (4q^2 + 2(4q^2+q)^2)}{2^{2n} - 4q^2} \leq \frac{216q^6}{2^n - 4q^2/2^n}.$$

Since $q \leq 2^{n-2}$, $4q^2/2^n < 2^{n-1}$ and $216q^6/(2^n - 4q^2/2^n) < 432q^6/2^n$. □

Type of Trees. At this point it will be useful to establish some terminology for distinguishing trees that have nodes with different origin/terminal. Indeed:

Lemma 41. *A ready node with origin 1 (resp. 2, 3, 4, 5, 6, 7, 8) has terminal 4 (resp. 7, 6, 1, 8, 3, 2, 5).*

Proof. This is obvious from MakeNodeReady. □

Moreover, recall that a non-root node's origin is the terminal of its parent (Lemma 9). In particular, it follows from Lemma 41 that the terminal of r determines the origins and terminals of nodes in a tree rooted at r.

Definition 14. A tree is called a $(1,4)$-*tree* if its root has terminal 1 or 4; a tree is a $(2,7)$-*tree* if its root has terminal 2 or 7; a tree is a $(3,6)$-*tree* if its root has terminal 3 or 6; a tree is a $(5,8)$-*tree* if its root has terminal 5 or 8.

By the above remarks, every ready node of a (i,j)-tree has terminal i or j.

Bad Read. The procedure CheckBadR is called in ReadTape, before the new query is written to the table. We emphasize that the new entry has *not* been added to the tables at this moment. The event BadR defined below occurs if and only if CheckBadR aborts.

Note that the set *ToPrep* is maintained by procedures SetToPrep and CheckBadR during PrepareTree(r) if r is the root of a $(2,7)$-tree. The set contains undefined queries that are about to be defined in PrepareTree(n) for some node n in the tree; its size decreases as ReadTape is called in PrepareTree, and when PrepareTree(r) returns *ToPrep* is empty.

Definition 15. Let ReadTape be called with argument (i, x_i) such that $x_i \notin F_i$. Then we define the following four events:

- BadRHit is the event that there exists x_{i-1} and x_{i+1} such that $x_{i-1} \oplus x_{i+1} = f_i(x_i)$, such that the 2chain $(i-1, x_{i-1}, x_i)$ is left active, and such that the 2chain (i, x_i, x_{i+1}) is right active.
- BadREqual is the event that there exists $x'_i \in F_i$ such that $f_i(x_i) = F_i(x'_i)$.
- BadRCollide is the event that there exists x_{i-1} such that the 2chain $(i-1, x_{i-1}, x_i)$ is left active and the query $(i+1, x_{i-1} \oplus f_i(x_i))$ is incident with an active 2chain, or that there exists x_{i+1} such that the 2chain (i, x_i, x_{i+1}) is right active and the query $(i-1, f_i(x_i) \oplus x_{i+1})$ is incident with an active 2chain.
- Suppose ReadTape(i, x_i) is called in PrepareTree(n) where n is a node in a $(2,7)$-tree. If $i = 3$, BadRPrepare is the event that there exists $(6, x'_6) \in$ *ToPrep* such that $u_2 \oplus f_3(x_3) = F_5(u_5) \oplus x'_6$ for some $u_2 \in \tilde{F}_2$ and $u_5 \in F_5$. Symmetrically if $i = 6$, BadRPrepare is the event that there exists $(3, x'_3) \in$ *ToPrep* such that $f_6(x_6) \oplus u_7 = x'_3 \oplus F_4(u_4)$ for some $u_4 \in F_4$ and $u_7 \in \tilde{F}_7$.

Moreover, we let BadR = BadRHit ∨ BadREqual ∨ BadRCollide ∨ BadRPrepare.

Lemma 42. BadRHit *occurs in a call to* ReadTape(i, x_i) *with probability at most* $(4q^2 + q)^2/2^n$.

Proof. BadRHit only occurs if $x_i \notin F_i$, in which case the value of $f_i(x_i)$ is uniformly distributed over $\{0,1\}^n$.

By Lemma 37, BadRHit occurs only if there exists $x_{i-1} \in \tilde{F}_{i-1}$ and $x_{i+1} \in \tilde{F}_{i+1}$ such that $f_i(x_i) = x_{i-1} \oplus x_{i+1}$, i.e., only if $f_i(x_i) \in \tilde{F}_{i-1} \oplus \tilde{F}_{i+1}$. By Lemma 38, we have

$$|\tilde{F}_{i-1} \oplus \tilde{F}_{i+1}| \leq |\tilde{F}_{i-1}| \cdot |\tilde{F}_{i+1}| \leq (4q^2 + q)^2.$$

Therefore, the probability that BadRHit occurs is at most $(4q^2 + q)^2/2^n$. □

Lemma 43. BadREqual *occurs in a call to* ReadTape(i, x_i) *with probability at most* $(4q^2 + q)/2^n$.

Proof. By Lemma 28, we have $|F_i| \leq 4q^2 + q$. Since $x_i \notin F_i$ by the assertion in ReadTape, the value of $f_i(x_i)$ is uniformly distributed and is independent of existing queries in F_i. The probability that $f_i(x_i)$ equals $F_i(x_i')$ for $x_i' \in F_i$ is at most $|F_i|/2^n \leq (4q^2 + q)/2^n$. □

The event BadRPrepare is similar to BadRCollide; in fact, if we include the queries in *ToPrep* in the set of "active queries", then BadRCollide ∨ BadRPrepare = BadRCollide. (This is exactly how BadRPrepare is detected in our pseudocode: the procedure ActiveQueries returns not only active queries but also queries in *ToPrep*. We can check that this modification does not affect the correctness of other bad events, since *ToPrep* is non-empty only during PrepareTree of $(2, 7)$-trees.)

Lemma 44. *The probability that* BadRCollide *or* BadRPrepare *occurs in a call to* ReadTape(i, x_i) *is at most* $5(4q^2 + q)^3/2^n$*.*

Proof. BadRCollide and BadRPrepare only occur if $x_i \notin F_i$, in which case the value of $f_i(x_i)$ is uniformly distributed over $\{0,1\}^n$.

Consider the first part of BadRCollide. If $(i-1, x_{i-1}, x_i)$ is left active, we must have $x_{i-1} \in \tilde{F}_{i-1}$ by Lemma 37. We also require that $x_{i+1} \coloneqq x_{i-1} \oplus f_i(x_i) \in \mathcal{C}_{i+1}$. Therefore, $f_i(x_i) = x_{i-1} \oplus x_{i+1} \in \tilde{F}_{i-1} \oplus \mathcal{C}_{i+1}$. By Lemmas 38 and 39, we have

$$|\tilde{F}_{i-1} \oplus \mathcal{C}_{i+1}| \leq (4q^2 + q) \cdot 2(4q^2 + q)^2 = 2(4q^2 + q)^3.$$

We can interpret BadRPrepare in a similar way: Let $(3, x_3)$ be defined during PrepareTree(n) and let r be the root of the tree containing n. Recall that *ToPrep* is the set of queries that will be defined during PrepareTree(r) (but hasn't been defined). The event BadRPrepare occurs if there exists $u_2 \in \tilde{F}_2$, $u_5 \in F_5$ and $(6, x_6') \in$ *ToPrep* such that $f_3(x_3) = u_2 \oplus F_5(u_5) \oplus x_6'$. The nodes in the tree are all outer nodes, and by Lemma 16 the tree contains at most q nodes. Each node contributes at most one query in position 6 to the set *ToPrep*, so there are at most q different possible values for x_6'. By Lemma 27 we have $|\tilde{F}_5| \leq 2q$ and $|\tilde{F}_2| \leq 4q^2 + q$. Therefore BadRPrepare occurs if $f_3(x_3)$ equals one of the (at most) $q \cdot 2q \cdot (4q^2 + q) = 2q^2(4q^2 + q)$ values.

Symmetrically the bounds can be proved for the second part of BadRCollide and BadRPrepare. Note that BadRPrepare occurs with probability 0 for ReadTape calls not issued by PrepareTree(n), thus the upper bound applies to all ReadTape calls. The two events occur for at most

$$2 \cdot (2(4q^2 + q)^3 + 2q^2(4q^2 + q)) \leq 5(4q^2 + q)^3$$

values of $f_i(x_i)$, which are chosen with probability at most $5(4q^2 + q)^3/2^n$. □

Lemma 45. *In an execution of* G_3*,* BadR *occurs with probability at most* $26200q^8/2^n$*.*

Proof. Every time ReadTape(i, x_i) is called with $x_i \notin F_i$, an entry is added to the tables. Therefore the number of such calls is at most $\sum_i |F_i| \leq 8q + 32q^2 \leq 40q^2$, where the first inequality is obtained by Lemma 28.

By Lemmas 42, 43 and 44 and by applying a union bound, the probability that BadRHit, BadREqual, BadRCollide or BadRPrepare occurs in one of the calls to ReadTape(i, x_i) with $x_i \notin F_i$ is thus upper bounded by

$$40q^2 \cdot \Big(\frac{(4q^2 + q)^2}{2^n} + \frac{4q^2 + q}{2^n} + \frac{5(4q^2 + q)^3}{2^n}\Big) \leq \frac{26200q^8}{2^n}.$$

□

PROPERTIES OF G_3. Before we give the definition of the last bad event BadA, we show some properties of executions of G_3 that are obtained due to the fact that the simulator aborts when BadP or BadR occurs. These properties will be used when we upper bound the probability of BadA. They are also useful for the equivalence of the event BadA and the abortion in CheckBadA.

Lemma 46. *Consider an execution of* G_3*. When a query is sampled in* PrepareTree(n)*, it is not pending or defined, and it is not incident with an active 2chain unless the 2chain is contained by the maximal path of* n*.*

Proof. The proof relies on the fact that a query sampled in PrepareTree is determined by the adjacent query that is freshly sampled. If it is already defined or pending when the adjacent query is sampled, BadRHit occurs and the simulator should have aborted. Moreover, two calls to PrepareTree will not try to sample the same query, otherwise BadRCollide occurs. Similarly the query is not incident with an active 2chain, or BadRCollide occurs.

We will showcase the proof for the case where n is a non-root node with $n.end = (3, x_3)$. The proof for other positions is similar.

Let r be the root of the tree containing n. The tree is a $(3, 6)$-tree, so the queries sampled during SampleTree(r) (including the sub-calls) are in positions 3 and 6. Recall the convention that (h, x_h) denotes the queries in the maximal path of n, and in particular $(7, x_7)$ and $(8, x_8)$ are the queries sampled in PrepareTree(n).

Note that $x_7 = F_6(x_6) \oplus x_5$, where $F_6(x_6)$ is sampled in SampleTree$(n.parent)$ and the query $(5, x_5)$ has been defined in MakeNodeReady(n). When ReadTape$(6, x_6)$ is called, the query $(7, x_7)$ is not pending or defined, since otherwise BadRHit occurs for the left active 2chain $(5, x_5, x_6)$. Furthermore, $(7, x_7)$ is not incident with an active 2chain or BadRCollide occurs.

After the call to ReadTape$(6, x_6)$, $(7, x_7)$ is incident with an active 2chain $(5, x_5, x_6)$ (and (i, x_i, x_{i+1}) for $i = 3, 4$); however, the 2chain is in the maximal path of n and is excluded in the lemma.

Similarly, we have $x_8 = F_7(x_7) \oplus x_6$, where ReadTape$(7, x_7)$ has just been called and $x_6 \in F_6$. The query $(8, x_8)$ is not pending or defined, and isn't incident with an active 2chain, since otherwise BadRHit or BadRCollide occurs. After ReadTape$(7, x_7)$ returns the query is incident with 2chains contained in the maximal path of n, which is compatible with the lemma.

We note that ReadTape$(8, x_8)$ is called immediately after ReadTape$(7, x_7)$, therefore the proof is done for this case. However, more queries are sampled between the calls to ReadTape$(6, x_6)$ and ReadTape$(7, x_7)$, and we need to prove that the result still holds after the changes.

If $(7, x_7)$ is pending or defined when PrepareTree(n) is called, it must be sampled in PrepareTree(n') which is called before PrepareTree(n). Let (h, x'_h) denote the queries in the maximal path of n'. Since the origin of n' is also 3 or 6, the query $(6, x'_6)$ is sampled in SampleTree(n') (if the origin is 3) or SampleTree$(n'.parent)$ (if the origin is 6). Moreover, $x'_5 \in F_5$ before SampleTree(r) is called. If ReadTape$(6, x_6)$ is called after ReadTape$(6, x'_6)$, we have $f_6(x_6) \oplus x_5 = x_7 = x'_7 = F_6(x'_6) \oplus x'_5$ where $x_5, x'_5 \in F_5$, so BadRCollide occurs in ReadTape$(6, x_6)$. Symmetrically if ReadTape$(6, x'_6)$ is called later, BadRCollide occurs in ReadTape$(6, x'_6)$.

If $(7, x_7)$ is incident with an active 2chain, by Lemma 36 it is incident with an active 2chain $(5, x'_5, x'_6)$ or $(8, x'_8, x'_9)$. Note that such an active 2chain did not exist when ReadTape$(6, x_6)$ is called. Between the calls ReadTape$(6, x_6)$ and PrepareTree(n), no query becomes pending and only queries in positions 3, 6, 7 or 8 get sampled in SampleTree or PrepareTree. Therefore, if the incident active 2chain is $(5, x'_5, x'_6)$, ReadTape$(6, x'_6)$ must be called after ReadTape$(6, x_6)$ while $(5, x'_5)$ has been defined before. However, BadRCollide occurs in $\text{ReadTape}(6, x'_6)$ since $(5, x'_5, x'_6)$ is left active and $(7, x_7)$ is incident with an active 2chain $(5, x_5, x_6)$. Similarly, if the active 2chain is $(8, x'_8, x'_9)$, BadRCollide occurs in the call to $\text{ReadTape}(8, x'_8)$. ☐

Lemma 47. *In an execution of* G_3*, if the origin of a node* n *is 3 or 6, then the call to* SimP^{-1} *in* $\text{PrepareTree}(n)$ *defines a new permutation query (i.e., the parameter of the call* $(x_8, x_9) \notin T^{-1}$*).*

Proof. Before the call to $\text{SimP}^{-1}(x_8, x_9)$, $\text{ReadTape}(8, x_8)$ has just been called. If (x_8, x_9) is already in T, then BadRHit occurs since $(7, x_7, x_8)$ is left active and $(8, x_8, x_9)$ is right active. ☐

Lemma 48. *Consider an execution of* G_3*. When a query* (i, x_i) *is adapted in* $\text{AdaptNode}(n)$ *it is not pending or defined. Moreover, if* $i = 1$*, there don't exist* x'_8 *and* x'_9 *such that* $(8, x'_8, x'_9)$ *is not in the maximal path of* n *and such that* $T^{-1}(x'_8, x'_9) = (*, x_1)$*.*

Proof. If $i \neq 1$, the query (i, x_i) must be adjacent to a freshly sampled query in the maximal path of n (which is sampled in $\text{SampleTree}(n)$ or $\text{PrepareTree}(n)$). Therefore, it can be proved like in Lemma 46 that the adapted query is not pending or defined when the adjacent query is sampled, since otherwise BadRHit occurs.

The query can also become defined in another call to AdaptNode. Now we prove that (i, x_i) is not adapted in $\text{AdaptNode}(n')$ for another node $n' \neq n$. This is also similar to the counterpart for PrepareTree.

Let (h, x_h) and (h, x'_h) denote the queries in the maximal paths of n and n' respectively, and assume by contradiction that the adapted queries $x_i = x'_i$ for $i \neq 1$. Without loss of generality, assume i is in the left of the adapt zone (i.e., the adapt zone used by the tree is $(i, i+1)$). Then $\text{ReadTape}(i-1, x_{i-1})$ is called during $\text{SampleTree}(n)$ or $\text{PrepareTree}(n)$ (and similarly for x'_{i-1} and n'). We assume without loss of generality that $\text{ReadTape}(i-1, x_{i-1})$ is called before $\text{ReadTape}(i-1, x'_{i-1})$. Since both n and n' are about to be adapted, when $\text{ReadTape}(i-1, x'_{i-1})$ is called $(i-2, x'_{i-2}, x'_{i-1})$ is left active and $(i, x'_i) = (i, x_i)$ is incident with an active 2chain $(i-2, x_{i-2}, x_{i-1})$. Thus BadRCollide occurs, leading to a contradiction.

Now consider the case $i = 1$. If $(1, x_1)$ is adapted in $\text{AdaptNode}(n)$, n's origin is 3 or 6 and a permutation query $\text{SimP}^{-1}(x_8, x_9)$ has been made in $\text{PrepareTree}(n)$. By Lemma 47 $(x_8, x_9) \notin T^{-1}$ before the call. Since BadP did not occur, $(1, x_1)$ is not pending or defined, and there does not exist (x'_8, x'_9) such that $(x'_8, x'_9) \neq (x_8, x_9)$ and $T^{-1}(x'_8, x'_9) = (*, x_1)$.

Now prove that $(1, x_1)$ is not adapted in $\text{AdaptNode}(n')$ for another node n' in the same tree. Let (h, x'_h) denote the queries in the maximal path of n' and assume by contradiction that $x_1 = x'_1$. Without loss of generality, let $\text{PrepareTree}(n)$ be called before $\text{PrepareTree}(n')$. Then the entry $T^{-1}(x_8, x_9) = (*, x_1)$ exists when $\text{SimP}^{-1}(x'_8, x'_9)$ is called. Since $(x'_8, x'_9) \notin T^{-1}$ by Lemma 47 and since $p^{-1}(x'_8, x'_9) = (x'_0, x'_1) = (x'_0, x_1)$, BadP occurs in $\text{P}^{-1}(x'_8, x'_9)$.

The above discussion also proves that no entry of the form $(*, x_1)$ is added to T in PrepareTree(n'). Moreover, no permutation query is defined during AdaptTree, thus $T(x_0, x_1) = (x_8, x_9)$ is the only entry in T that has the form $(*, x_1)$ when $\text{AdaptNode}(n)$ is called. ☐

Bad Adapt. The CheckBadA procedure is called once per tree. For a tree with root r, CheckBadA(r) is called after $\text{PrepareTree}(r)$ returns and before $\text{AdaptTree}(r)$ is called. At this point, for every node n in the tree, all queries in the full partial path containing $n.id$, except the two to be adapted, are defined. Such a full partial path is a $(i+1, i)$-partial path where (i, x_i) and $(i+1, x_{i+1})$ are about to be adapted, and we say it is *ready to be adapted*.

Recall that by Lemmas 10 and 19, before SampleTree(r) is called, each node n in the tree is associated to a (unique) full partial path containing $n.id$ and whose endpoints are the origin and terminal of the node. The call to SampleTree assigns the values $f_i(x_i)$ and $f_j(x_j)$ to $F_i(x_i)$, $F_j(x_j)$ where (i, j) are the endpoints of the path, extending the full partial path by one query in each direction. In the case of a $(1, 4)$-tree, each node in the tree is associated to a unique $(3, 2)$-partial path; in the case of a $(5, 8)$-tree, each node in the tree is associated to a unique $(7, 6)$-partial path. In the aforementioned cases, the paths associated to the nodes are ready to be adapted. However, in other cases, more queries are sampled by PrepareTree before adaptations occur: for a $(2, 7)$-tree, PrepareTree assigns $F_3(x_3) = f_3(x_3)$ and $F_6(x_6) = f_6(x_6)$; for a $(3, 6)$-tree, PrepareTree assigns $F_7(x_7) = f_7(x_7)$ and $F_8(x_8) = f_8(x_8)$, and calls the permutation query $\mathrm{SimP}^{-1}(x_8, x_9)$. Then each node in the tree can be associated to a unique $(5, 4)$- or $(2, 1)$-partial path in the two cases respectively. After PrepareTree(r) returns, each node in the tree is associated to a unique $(i + 1, i)$-partial path which is ready to be adapted.

Focusing for concreteness on the case of a $(1, 4)$-tree, AdaptTree assigns

$$F_2(x_2) \leftarrow x_1 \oplus x_3$$
$$F_3(x_3) \leftarrow x_2 \oplus x_4$$

for each non-root node n, where $\{x_h\}_{h=3}^{2}$ is the $(3, 2)$-partial path associated to n. (See the procedures AdaptTree, AdaptNode and Adapt.) We say that the queries $(2, x_2)$ and $(3, x_3)$ are *adapted* in the call to AdaptTree. The assignments to F_2 and F_3 are also called *adaptations*. Thus, two adaptations occur per non-root node in the tree.

As mentioned, the procedure CheckBadA(r) is called before any adaptations take place. To briefly describe this procedure, CheckBadA starts by "gathering information" about all the adaptations to take place for the current tree, i.e., two adaptations per non-root node. For this it uses the **Adapt** class. The **Adapt** class has four fields: *query, value, left* and *right*.

For example, given a non-root node n in a $(1, 4)$-tree with associated $(3, 2)$-partial path $\{x_h\}_{h=3}^{2}$, and letting

$$y_2 = x_1 \oplus x_3$$
$$y_3 = x_2 \oplus x_4$$

be the future values of $F_2(x_2)$ and $F_3(x_3)$ respectively, GetAdapts will create the two instances of **Adapt** with the following settings:

$$(\textit{query, value, left, right}) = ((2, x_2), y_2, x_1, x_4),$$
$$(\textit{query, value, left, right}) = ((3, x_3), y_3, x_1, x_4).$$

These two instances are added to the set $\mathcal{A}$, which contains all the instance of **Adapt** for the current tree ($\mathcal{A}$ is reset to $\emptyset$ at the top of CheckBadA).

In our proof, $\mathcal{A}$ refers to the state of this set after GetAdapts(r) returns. Abusing notation a little, we will write

$$(i, x_i, y_i) \in \mathcal{A}$$

as a shorthand to mean that there exists some $a \in \mathcal{A}$ of the form

$$((i, x_i), y_i, *, *)$$

after GetAdapts returns. We may even omit y_i and simply say a query (i, x_i) is in $\mathcal{A}$.

Lemma 49. *Assume that* GetAdapts(r) *has returned. Then for every* $(i, x_i, y_i) \in \mathcal{A}$ *there exists a unique* $a \in \mathcal{A}$ *of the form* $((i, x_i), *, *, *)$.

Proof. In the proof of Lemma 48 we showed that two calls AdaptNode(n) and AdaptNode(n') for $n \neq n'$ would not adapt the same query. In particular, we note that the proof still holds even if the simulator aborts before one of the calls is made (i.e., the queries supposed to be adapted must be distinct, even in the cases where abortion occurs). Each entry in $\mathcal{A}$ corresponds to one query about to be adapted by a call to AdaptNode. Since each query (i, x_i) is adapted at most once, the set $\mathcal{A}$ contains at most one entry of the form $((i, x_i), *, *, *)$. □

By Lemma 49 each tuple $(i, x_i, y_i) \in \mathcal{A}$ can be uniquely associated to a node in the tree being adapted, specifically the node n whose associated partial path contains (i, x_i). For convenience we will say that (i, x_i, y_i) is *adapted in* n or, equivalently, *adapted in the path* $\{x_h\}$, where $\{x_h\}$ is a shorthand for $\{x_h\}_{h=3}^{2}$ (for $(1,4)$-trees), for $\{x_h\}_{h=7}^{6}$ (for $(5,8)$-trees), for $\{x_h\}_{h=2}^{1}$ (for $(3,6)$-trees) or for $\{x_h\}_{h=5}^{4}$ (for $(2,7)$-trees).

Definition 16. Let r be a root node, and consider the point in the execution after GetAdapts(r) is called. Then we define the following bad events with respect to the state of the tables at this point (in particular, before AdaptTree(r) is called):

- BadAHit is the event that for some $(i, x_i, y_i) \in \mathcal{A}$, $i \neq 1$, there exist $x'_{i-1} \in \tilde{F}_{i-1}$ and $x'_{i+1} \in \tilde{F}_{i+1}$ such that $y_i = x'_{i-1} \oplus x'_{i+1}$.
- If the tree rooted at r is *not* a $(3,6)$-tree, BadAPair is the event that there exists two tuples $(i, x_i, y_i) \in \mathcal{A}$ and $(i+1, u_{i+1}, v_{i+1}) \in \mathcal{A}$ adapted in different paths $\{x_i\}$ and $\{u_i\}$, such that $x_{i-1} \neq u_{i-1}$ and the query $(i+2, x_i \oplus v_{i+1})$ is active or is incident with an active 2chain, or such that $x_{i+2} \neq u_{i+2}$ and the query $(i-1, y_i \oplus u_{i+1})$ is active or is incident with an active 2chain.
- If r is the root of a $(2,7)$-tree, BadAEqual is the event that for some $(i, x_i, y_i) \in \mathcal{A}$, there exists $x'_i \in F_i$ such that $y_i = F_i(x'_i)$ or there exists $(i, x'_i, y'_i) \in \mathcal{A}$ such that $x'_i \neq x_i$ and $y_i = y'_i$.
- If r is the root of a $(2,7)$-tree, BadAMid is the event that there exists $(4, x_4, y_4)$, $(5, x'_5, y'_5)$, $(4, u_4, v_4)$ and $(5, u'_5, v'_5)$ such that $x_4 \oplus y'_5 = u_4 \oplus v'_5 \notin F_6$, where $y_i = F_i(x_i)$ or $(i, x_i, y_i) \in \mathcal{A}$ for $i = 4, 5$, where $v_i = F_i(u_i)$ or $(i, u_i, v_i) \in \mathcal{A}$ for $i = 4, 5$, where $(x_4, x'_5) \neq (u_4, u'_5)$, and where at least one of $(4, x_4, y_4)$, $(5, x'_5, y'_5)$, $(4, u_4, v_4)$ and $(5, u'_5, v'_5)$ is in $\mathcal{A}$.

Moreover, we let BadA = BadAHit $\vee$ BadAPair $\vee$ BadAEqual $\vee$ BadAMid.

In the rest of this section, we will let r denote the root of the tree being adapted as in the above definition.

The probabilities in the following lemmas are over the randomness of tape entries read in SampleTree(r) and PrepareTree(r). If CheckBadA is called, the simulator did not abort in SampleTree(r) and PrepareTree(r). This implies that the sampled queries are not defined before, and thus are sampled uniformly at random. When we use the notations $\{x_h\}_{h=i+1}^{i}$ (often shortened to $\{x_h\}$, as above) for the path associated to a node n in the tree rooted at r, our meaning is that the endpoints x_i and x_{i+1} of the path are random variables defined over the coins read by SampleTree and PrepareTree. More precisely, x_{i+1} is determined by $f_{i+2}(x_{i+2})$, while x_i is determined by $f_{i-1}(x_{i-1})$ if $i > 1$ and by $p^{-1}(x_8, x_9)$ if $i = 1$. By extension, each $(i, x_i, y_i) \in \mathcal{A}$ is a random variable over the same set of coins.

Lemma 50. *Let* n *be a non-root node in the tree rooted at* r, *then the probability that* BadAHit *occurs for a query adapted in* AdaptNode(n) *is at most* $2(4q^2 + q)^2/2^n$.

The probability that BadAHit *occurs in a* G$_3$*-execution is at most* $200q^6/2^n$.

Proof. As in Lemma 34 we can assume that $q \le 2^{n-2}$ since the statement trivially holds otherwise.

If n is a node in a $(1,4)$-tree, let the path associated to n be $\{x_h\}$, and consider the adapted query $(2, x_2, y_2) \in \mathcal{A}$. We have $y_2 = x_1 \oplus x_3 = x_1 \oplus F_4(x_4) \oplus x_5$, where $F_4(x_4) = f_4(x_4)$ is uniformly distributed and is independent of x_1, x_5 and the sets $\tilde{F}_i$. Each of $\tilde{F}_1$ and $\tilde{F}_3$ contains at most $4q^2+q$ queries by Lemma 38. Therefore, the probability that $x_1 \oplus f_4(x_4) \oplus x_5$ equals a value in $\tilde{F}_1 \oplus \tilde{F}_3$ is at most $(4q^2+q)^2/2^n$. Similarly, the same bound can be proved for $(3, x_3, y_3) \in \mathcal{A}$. The lemma follows from a union bound.

The proof when n is a node in a $(5,8)$- or $(2,7)$-tree is the same. (In the latter case, one difference is that the query $F_3(x_3) = f_3(x_3)$ or $F_6(x_6) = f_6(x_6)$ is sampled in PrepareTree instead of SampleTree, and the set of active queries $\tilde{F}_i$ is changed during PrepareTree. However, only queries in *ToPrep* are added to $\tilde{F}_i$, which are fixed before $f_3(x_3)$ or $f_6(x_6)$ is sampled and thus are independent of the sampled values.)

If n is a node in a $(3,6)$-tree, then we only need to consider the query $(2, x_2, y_2) \in \mathcal{A}$ (since the event requires $i \ne 1$). We have $y_2 = x_1 \oplus x_3$, where x_1 equals the second half of $p(x_8, x_9)$. By Lemma 47 the permutation query is newly made, after x_3, $\tilde{F}_1$ and $\tilde{F}_3$ are fixed (indeed, for a $(3,6)$-tree PrepareTree only defines queries in positions 7 and 8). The number of different $y_2 \in \tilde{F}_1 \oplus \tilde{F}_3$ is at most $(4q^2+q)^2$, thus $\mathsf{BadAHit}$ occurs for $(4q^2+q)^2$ different $x_1 = y_2 \oplus x_3$. Moreover, there are at most 2^n possible values for x_0. As discussed in the proof for Lemma 40, $p(x_8, x_9)$ equals any value with probability at most $1/(2^{2n} - 4q^2)$. Thus the probability that $p(x_8, x_9)$ equals one of the pairs (x_0, x_1) that cause $\mathsf{BadAHit}$ is at most

$$(4q^2+q)^2 \cdot 2^n/(2^{2n} - 4q^2) \le (4q^2+q)^2 \cdot 2^n/(2^{2n} - 2^{2n}/2) = 2(4q^2+q)^2/2^n$$

where the inequality is due to the assumption that $q \le 2^{n-2}$.

By Lemma 26, there are at most $4q^2$ non-root nodes in an execution. With a union bound, the probability that $\mathsf{BadAHit}$ occurs in an execution is at most $4q^2 \cdot 2(4q^2+q)^2/2^n \le 200q^6/2^n$. □

Lemma 51. *Let n_1 and n_2 be non-root nodes in the tree rooted at r, where the tree is not a $(3,6)$-tree. The probability that* $\mathsf{BadAPair}$ *occurs for the position-i query adapted in n_1 and the position-$(i+1)$ query adapted in n_2 is at most $(8q^2 + 2q + 4(4q^2+q)^2)/2^n$.*

The probability that $\mathsf{BadAPair}$ *occurs in a* G_3*-execution is at most $(1760q^8 - 440q^6)/2^n$.*

Proof. We have $i = 2$ if r is the root of a $(1,4)$-tree, $i = 6$ if r is the root of a $(5,8)$-tree, and $i = 4$ if r is the root of a $(2,7)$-tree. Let (i, x_i, y_i) and $(i+1, u_{i+1}, v_{i+1})$ be adapted in paths $\{x_h\}$ and $\{u_h\}$ of n_1 and n_2 respectively.

Note that $x_i = x_{i-2} \oplus f_{i-1}(x_{i-1})$ and $v_{i+1} = u_i \oplus u_{i+2} = u_{i-2} \oplus f_{i-1}(u_{i-1}) \oplus u_{i+2}$. If $x_{i-1} \ne u_{i-1}$, the value of $x_i \oplus v_{i+1}$ is uniformly distributed since $f_{i-1}(x_{i-1})$ and $f_{i-1}(u_{i-1})$ are uniform and independent. Moreover, the values are also independent of $\tilde{F}_{i+2}$ and of $\mathcal{C}_{i+2}$ (which is determined by F_{i+1} and $\tilde{F}_i$). Thus the probability that the value is in $\tilde{F}_{i+2} \cup \mathcal{C}_{i+2}$ is

$$|\tilde{F}_{i+2} \cup \mathcal{C}_{i+2}|/2^n \le (|\tilde{F}_{i+2}| + |\mathcal{C}_{i+2}|)/2^n \le (4q^2 + q + 2(4q^2+q)^2)/2^n$$

where the second inequality uses Lemmas 38 and 39.

By a symmetric argument, the same bound can be proved for the event that $x_{i+2} \ne u_{i+2}$ and $(i-1, y_i, u_{i+1})$ is active or is incident with an active 2chain. The first part of the lemma follows by a union bound on the above results.

By Lemma 26, the number of non-root nodes is at most $4q^2$. Moreover, if $n_1 = n_2$ we have

$x_{i-1} = u_{i-1}$ and $x_{i+2} = u_{i+2}$, so BadAPair wouldn't occur. With a union bound over the $4q^2(4q^2-1)$ ways of choosing distinct n_1 and n_2, the probability of BadAPair in an execution is at most

$$(16q^4 - 4q^2) \cdot \frac{8q^2 + 2q + 4(4q^2+q)^2}{2^n} \le \frac{1760q^8 - 440q^6}{2^n}.$$

□

Lemma 52. *The number of queries in each of positions 4 and 5 that are defined or are in $\mathcal{A}$ is upper bounded by $2q$.*[17]

Proof. As discussed in the proof of Lemma 27, the distinguisher makes at most q queries in each position and the simulator adapts or defines queries in positions 4 and 5 only when completing an outer node, where there are at most q outer nodes in an execution (cf. Lemma 16). Note that a query is in $\mathcal{A}$ only if it is about to be adapted, so the number of queries in position 4 (resp. 5) that are defined or are in $\mathcal{A}$ is at most $2q$. □

Lemma 53. *Let n be a non-root node in a $(2,7)$-tree. The probability that* BadAEqual *occurs for a query adapted in* AdaptNode(n) *is at most $4q/2^n$.*

The probability that BadAEqual *occurs in a* G_3*-execution is at most $4q^2/2^n$.*

Proof. Let the maximal path of n be $\{x_h\}$, and consider the adapted query $(4, x_4, y_4) \in \mathcal{A}$. We have $y_4 = x_3 \oplus x_5 = x_3 \oplus F_6(x_6) \oplus x_7$. By Lemma 46, the value $F_6(x_6) = f_6(x_6)$ (sampled in PrepareTree(n)) is uniformly distributed and is not used in other calls to PrepareTree. Thus, the value of $F_6(x_6)$ is independent of queries adapted in other paths as well as x_3, x_7 and F_4. By Lemma 52, there are at most $2q$ x'_4 such that $x'_4 \in F_4$ or $(4, x'_4, y'_4) \in \mathcal{A}$ for some y'_4. The probability that $y_4 = x_3 \oplus F_6(x_6) \oplus x_7$ equals one of $F_4(x'_4)$ or y'_4 is hence at most $2q/2^n$.

The same bound can be proved for the query $(5, x_5, y_5)$ symmetrically. With a union bound, the probability that BadAEqual occurs for either query adapted in AdaptNode(n) is at most $4q/2^n$.

Moreover, nodes in $(2,7)$-trees are all outer nodes. There are at most q outer nodes by Lemma 16, so the probability that BadAEqual occurs in an execution can be upper bounded by $4q^2/2^n$ with a union bound on the nodes. □

Lemma 54. *For $i = 4, 5$ and $x_i, x'_i \in F_i$ such that $x_i \ne x'_i$, we have $F_i(x_i) \ne F_i(x'_i)$.*

Proof. A query in position 4 or 5 can be defined by ReadTape or in AdaptNode(n) with n being a node in a $(2,7)$-tree. Assume without loss of generality that the query (i, x'_i) is defined later than (i, x_i).

If (i, x'_i) is defined by ReadTape, we have $x_i \in F_i$ when ReadTape(i, x'_i) is called. Then $F_i(x'_i) = f_i(x'_i) \ne F_i(x_i)$, since otherwise BadREqual occurs. If (i, x'_i) is defined in AdaptNode(n), let r be the root of the tree containing n. Then (i, x'_i, y'_i) is in the set $\mathcal{A}$ when GetAdapts(r) returns. If $x_i \in F_i$ is defined when AdaptTree(r) is called, we have $y'_i = F_i(x_i)$ and the first case of BadAEqual occurs. Otherwise (i, x_i) is also defined during AdaptTree(r) and $(i, x_i, y_i = y'_i) \in \mathcal{A}$, and the second case of BadAEqual occurs. □

[17] This lemma is implied by Lemma 27, unless the simulator aborts before adapting all queries in $\mathcal{A}$.

Lemma 55. *Let r be the root of a $(2,7)$-tree with τ nodes. In* AdaptTree(r)*, the probability of* $\mathsf{BadAMid} \wedge \neg\mathsf{BadAEqual}$ *is at most* $16q^3\tau/2^n$*.*

In a G_3*-execution, the probability that* BadAMid *occurs and* BadAEqual *doesn't occur is at most* $16q^4/2^n$*.*

Proof. Recall that the event BadAMid involves four queries $(4, x_4, y_4)$, $(5, x_5', y_5')$, $(4, u_4, v_4)$ and $(5, u_5', v_5')$, at least one of which is in $\mathcal{A}$.

First we prove $x_4 \neq u_4$. Assume by contradiction that $x_4 = u_4$, then $x_5' \neq u_5'$. Since $x_4 \oplus y_5' = u_4 \oplus v_5'$, we have $y_5' = v_5'$. If $(5, x_5', y_5') \in \mathcal{A}$ or $(5, u_5', v_5') \in \mathcal{A}$ (or both), BadAEqual occurs. Otherwise we have $F_5(x_5') = F_5(u_5')$, contradicting Lemma 54.

Then we prove that $x_5' = u_5'$ implies $y_5' = v_5'$: If $x_5' \in F_5$, by Lemma 48 we have $(5, x_5', y_5') \notin A$ and $(5, u_5', v_5') \notin A$. Thus $y_5' = F_5(x_5') = F_5(u_5') = v_5'$. On the other hand, if $x_5' \notin F_5$, we have $(5, x_5', y_5') \in A$ and $(5, u_5', v_5') \in A$. Since $x_5' = u_5'$ and there is a unique entry in $\mathcal{A}$ of the form $(5, x_5', *)$ (cf. Lemma 49), we have $y_5' = v_5'$. This implies $x_5' \neq u_5'$ since we already proved $x_4 \neq u_4$, which implies $x_4 \oplus y_5' \neq u_4 \oplus v_5'$ if $y_5' = v_5'$.

In the following discussion, if $(4, x_4, y_4)$ (resp. $(5, x_5', y_5')$, $(4, u_4, v_4)$ $(5, u_5', v_5')$) is in $\mathcal{A}$, we will use $\{x_h\}$ (resp. $\{x_h'\}$, $\{u_h\}$, $\{u_h'\}$) to represent the path in which it is adapted.

We note that for each adapted path $\{x_h\}$, the value $F_3(x_3) = f_3(x_3)$ is sampled in PrepareTree. By Lemma 46, $F_3(x_3)$ is newly sampled and is not used in another path. Thus $F_3(x_3)$ is distributed uniformly and is independent of existing queries as well as the queries about to be adapted in other paths (i.e., the only values that are *not* independent of $F_3(x_3)$ are x_4 and y_5, both of which are in the adapted path).

Consider the case where at least one of $(4, x_4, y_4)$ and $(4, u_4, v_4)$ is in $\mathcal{A}$. By symmetry we can assume $(4, x_4, y_4) \in \mathcal{A}$. We have $x_5' \neq x_5$, otherwise $x_4 \oplus y_5' = x_4 \oplus y_5 = x_6 \in F_6$ (x_6 is in the path $\{x_h\}$, which is ready to be adapted). If $u_5' \neq x_5$, none of u_4, x_5' and u_5' is in the path $\{x_h\}$, so $F_3(x_3)$ is independent of u_4, y_5' and v_5'. Then $x_4 = x_2 \oplus F_3(x_3)$ is also uniform and independent, and the probability that $x_4 \oplus y_5' = u_4 \oplus v_5'$ is $1/2^n$.

If $u_5' = x_5$, then $v_5' = y_5$ (as proved before). This implies $u_4 \oplus y_5' = x_4 \oplus v_5' = x_4 + y_5 = x_6$, which will be used in the following discussion. We consider the following possibilities of x_5' and u_4:

- If $u_4 \in F_4$ and $x_5' \in F_5$, we have $F_4(u_4) = v_4$ and $F_5(x_5') = y_5'$ before SampleTree(r) is called since no queries in positions 4 and 5 get defined during SampleTree(r) or PrepareTree(r). However, BadRCollide occurs when ReadTape$(7, x_7)$ is called by SampleTree(n), because $(6, x_6)$ is incident with an active 2chain $(4, u_4, x_5')$ and $x_6 = f_7(x_7) \oplus x_8$ where $x_8 \in F_8$. Hence, this case can never occur.
- If $(4, u_4, v_4) \in \mathcal{A}$ and $x_5' \neq u_5$, then since $u_4 = u_2 \oplus f_3(u_3)$ where $f_3(u_3)$ is independent of u_2, y_5' and x_6, the probability that $u_4 \oplus y_5' = f_3(u_3) \oplus u_2 \oplus y_5' = x_6$ is $1/2^n$.
- If $(4, u_4, v_4) \in \mathcal{A}$ and $x_5' = u_5$, then using the same argument before we have $(5, x_5', y_5') \in \mathcal{A}$ and $y_5' = v_5$. Then we have $x_6 = u_4 \oplus y_5' = u_4 \oplus v_5 = u_6$. Moreover, since $x_5 \neq x_5' = u_5$, the paths $\{x_h\}$ and $\{u_h\}$ are distinct. The query $(6, x_6) = (6, u_6)$ is sampled twice in PrepareTree of the two paths, which is impossible since the simulator would have aborted when ReadTape is called on the same query more than once.
- If $(5, x_5', y_5') \in \mathcal{A}$ and $u_4 \neq x_4'$,[18] we have $y_5' = x_4' \oplus x_6' = x_2' \oplus f_3(x_3') \oplus x_6'$ where $f_3(x_3')$ is independent of x_2', u_4, x_6 and x_6'. Similarly the probability of $u_4 \oplus y_5' = x_6$ is $1/2^n$.

[18] This case (and the next) overlaps with the previous ones.

– If $(5, x'_5, y'_5) \in \mathcal{A}$ and $u_4 = x'_4$, we have $x_6 = u_4 \oplus y'_5 = x'_4 \oplus y'_5 = x'_6$. The rest of the proof is similar to the third case.

Now consider the case where $x_4, u_4 \in F_4$. Then at least one of $(5, x'_5, y'_5)$ and $(5, u'_5, v'_5)$ is in $\mathcal{A}$, and without loss of generality let $(5, x'_5, y'_5) \in \mathcal{A}$. We have $y'_5 = x'_4 \oplus x'_6 = x'_2 \oplus f_3(x'_3) \oplus x'_6$ and $f_3(x'_3)$ is independent of x'_2, x'_6, x_4, u_4 and v'_5 (since $x'_5 \neq u'_5$). Thus the probability that $x_4 \oplus y'_5 = u_4 \oplus v'_5$ is $1/2^n$.

The above discussion shows that for all possible ways of choosing the four queries, the probability that BadAMid occurs for these queries is at most $1/2^n$.

At least one of the four queries should be in $\mathcal{A}$, which contains 2τ queries; each of the other three queries has a fixed position and can be either a defined query or a query in $\mathcal{A}$, where there are at most $2q$ choices (cf. Lemma 52). Therefore, the number of different ways to choose these queries is at most $2\tau \cdot (2q)^3 = 16q^3\tau$, and the lemma follows from a union bound.

Finally, let $\mathcal{T}_{2,7}$ be the set of $(2,7)$-trees and let $\tau(T)$ be the number of nodes in a tree T. Take a union bound over all $(2,7)$-trees, the probability of $\mathsf{BadAMid} \wedge \neg\mathsf{BadAEqual}$ in an execution is at most

$$\sum_{T \in \mathcal{T}_{2,7}} \frac{16q^3\tau(T)}{2^n} = \frac{16q^3}{2^n} \sum_{T \in \mathcal{T}_{2,7}} \tau(T) \le \frac{16q^4}{2^n}$$

where the inequality follows from Lemma 16 and the fact that nodes in $(2,7)$-trees are outer nodes. ☐

Lemma 56. *The probability that* BadA *occurs in an execution of* G_3 *is at most* $1760q^8/2^n$*.*

Proof. Since $\mathsf{BadA} = \mathsf{BadAHit} \vee \mathsf{BadAPair} \vee \mathsf{BadAEqual} \vee \mathsf{BadAMid}$, by a union bound we have

$$\begin{aligned}\Pr[\mathsf{BadA}] &\le \Pr[\mathsf{BadAHit}] + \Pr[\mathsf{BadAPair}] + \Pr[\mathsf{BadAEqual}] + \Pr[\neg\mathsf{BadAEqual} \wedge \mathsf{BadAMid}] \\ &\le \frac{200q^6}{2^n} + \frac{1760q^8 - 440q^6}{2^n} + \frac{4q^2}{2^n} + \frac{16q^4}{2^n} \le \frac{1760q^8}{2^n}\end{aligned}$$

where the second inequality follows from Lemmas 50, 51, 53 and 55. ☐

We say that an execution of G_3 is *good* if none of the bad events occurs.

Lemma 57. *An execution of* G_3 *is good with probability at least* $1 - 28392q^8/2^n$*.*

Proof. With a union bound on the results in Lemmas 40, 45 and 56, the probability that at least one of BadP, BadR and BadA occurs is at most

$$\frac{432q^6}{2^n} + \frac{26200q^8}{2^n} + \frac{1760q^8}{2^n} \le \frac{28392q^8}{2^n}.$$

Thus the probability of obtaining a good execution is at least $1 - 28392q^8/2^n$. ☐

5.4.2 Assertions don't Abort in G_3

Now we prove that assertions never fail in executions of G_3.

We recall that assertions appear in procedures F, ReadTape, Trigger, MakeNodeReady, and Adapt.

Lemma 58. *In an execution of* G_3*, the simulator doesn't abort in* Assert *called by* Trigger*.*

Proof. The counter *NumOuter* is increased only before an outer node is added in FindNewChildren, so the assertion fails only if the $(q+1)$th outer node is about to be added. We only need to prove that even without the assertions in Trigger at most q outer nodes are created in G_3.

When an outer node is created, the permutation query in its maximal path must be defined because of the call to CheckP^+ or CheckP^- in Trigger. Therefore each outer node can be associated with an entry in T.

Next we prove that the outer nodes are associated with distinct permutation queries in T. Assume by contradiction that the maximal paths of two outer nodes n_1 and n_2 contain the same permutation query $T(x_0, x_1) = (x_8, x_9)$. Let $x_2 = F_1(x_1) \oplus x_0$, $x_7 = F_8(x_8) \oplus x_9$. Without loss of generality, assume n_1 is created before n_2 and the origin of n_1 is 1 or 2.

If the origin of n_1 is 1, we already have $x_7 \in F_7$ and $x_8 \in F_8$ when n_1 is created. Thus when n_2 is created, its origin cannot be 7 or 8. If $n_2.\mathit{beginning} = (1, x_1) = n_1.\mathit{beginning}$, then $n_1.id = n_2.id = (7, x_7, x_8)$. By Lemmas 7 and 9 n_1 and n_2 have the same parent, which contradicts part (i) of Lemma 23. If the origin of n_2 is 2, then by observing FindNewChildren, n_2 can only be created when $(1, x_1)$ is defined. By Lemma 10, the query $(1, x_1)$ is defined when SampleTree(n_1) is called. But after the call and before a new node can be created, the maximal path of n_1 is completed and, in particular, AdaptTree(n_1) has been called and $(2, x_2)$ is defined. But n_2 can only be defined when $(2, x_2)$ is pending, which is impossible.

Next we consider the entries in the table T. Each of them is added when the permutation oracle is called, either by the simulator or by the distinguisher.

In an execution of G_3, the simulator makes new permutation queries only in MakeNodeReady(n) or, when the origin of n is 3 or 6, in PrepareTree(n). Moreover, if n is an outer node, the permutation query made in MakeNodeReady(n) already exists when n is created, because otherwise CheckP^+ or CheckP^- will return **false** and the node wouldn't be created in the first place. Therefore, new entries are added to T in MakeNodeReady(n) or PrepareTree(n) only if n is an inner node.

However, we are going to prove that if a permutation query is added (i.e., queried for the first time) during MakeNodeReady(n) or PrepareTree(n) where n is an inner node, no outer node contains the permutation query. Without loss of generality, assume the origin of n is 3 or 4. Let SimP^{-1} $(x_8, x_9) = (x_0, x_1)$ be the permutation query made in MakeNodeReady(n) or PrepareTree(n), which does not exist in T before.

Assume by contradiction that an outer node n' contains the permutation query $T(x_0, x_1) = (x_8, x_9)$ in its maximal path. n' must be created after the permutation query is added to T. Furthermore, when the permutation query is added, the queries $(8, x_8)$ and $(7, x_7)$ ($x_7 = F_8(x_8) \oplus x_9$ as usual) have been defined in MakeNodeReady(n) or PrepareTree(n) (or before). Thus $(8, x_8)$ and $(7, x_7)$ are defined when n' is created, implying that $n'.\mathit{beginning}$ equals $(1, x_1)$ or $(2, x_2)$.

If the origin of n is 3, after the call to PrepareTree(n) the queries $(1, x_1)$ and $(2, x_2)$ are adapted in the call to AdaptNode(n), before which no new outer node is created. If the origin of n is 4, after making the permutation query the simulator sets $n.\mathit{end} = (1, x_1)$. If $n'.\mathit{beginning} = (1, x_1)$, n' is a child of n, which contradicts Lemma 23 because $n'.id = (7, x_7, x_8)$ is contained by the maximal path of n. If $n'.\mathit{beginning} = (2, x_2)$, n' must be created after $(1, x_1)$ becomes defined, i.e., after SampleTree(n) is called (cf. Lemma 10). But after SampleTree(n) $(2, x_2)$ is adapted before n' can be created; since $n'.\mathit{beginning}$ should be pending when it is created, this is impossible.

Therefore, each outer node must contain a distinct permutation query made by the distinguisher. Since the distinguisher makes at most q permutation queries, at most q outer nodes are created. □

Lemma 59. *Let* $(i, x_i, y_i) \in \mathcal{A}$ *with* $i \neq 1$ *be adapted during* AdaptNode(n)*, and let* r *be the root of the tree containing* n*. When* AdaptTree(r) *is called, the query* (i, x_i) *is not incident with any active 2chain that is not contained by the maximal path of* n*.*

Proof. As discussed in the proof for Lemma 48, in the maximal path of n the adapted query (i, x_i) is adjacent to a newly sampled query $(i \pm 1, x_{i\pm1})$ (where "$\pm$" is "$-$" if $i = 4, 6$ or if $i = 2$ and n's origin is 1 or 4, and is "+" otherwise). Similar to the proof for Lemma 46, (i, x_i) is not incident to an active 2chain when the call ReadTape$(i \pm 1, x_{i\pm1})$ is made, otherwise BadRCollide occurs. The newly sampled query creates an active 2chain with which (i, x_i) is incident, but the 2chain is in the maximal path of n.

More queries are sampled after ReadTape$(i \pm 1, x_{i\pm1})$ and before AdaptTree(r), and we need to prove that the property still holds after the new queries are added.

If the tree rooted at r is a $(1, 4)$-tree, more queries in positions 1 and 4 are sampled by SampleTree. These queries are already pending before they are sampled, thus the activeness of 2chains is not changed. If $i = 2$, the query (i, x_i) can become incident with an active 2chain only if a query $(1, x'_1)$ becomes defined such that there exists an active 2chain $(0, x'_0, x'_1)$ with $x'_0 \oplus F_1(x'_1) = x_2$; however, then BadRCollide occurs in ReadTape$(1, x'_1)$ because the query $(2, x_2)$ is already incident with an active 2chain $(0, x_0, x_1)$. If $i = 3$, the proof is symmetric.

The case where the tree is a $(5, 8)$-tree is symmetric to the case of a $(1, 4)$-tree.

If the tree is a $(2, 7)$-tree, by symmetry we only consider $i = 4$. The queries sampled in SampleTree are in positions 2 and 7, which doesn't change the activeness of 2chains and cannot make a 2chain become incident with $(4, x_4)$. The queries sampled in PrepareTree are in positions 3 and 6 and are not pending before. We prove by contradiction and assume that $(4, x_4)$ becomes incident with an active 2chain that is not in the maximal path of n after a query $(i', x'_{i'})$ (with $i' \in \{3, 6\}$) is defined in PrepareTree. If $i' = 3$, then as the previous case, BadRCollide occurs when ReadTape$(3, x'_3)$ is called. If $i' = 6$, then $(4, x_4)$ must be incident to a 2chain $(5, x'_5, x'_6)$ for some $x'_5 \in F_5$. When ReadTape$(3, x_3)$ is called (in PrepareTree(n)), we have $x_2 \oplus f_3(x_3) = x_4 = F_5(x'_5) \oplus x'_6$ for $(6, x'_6) \in \mathit{ToPrep}$, $x_2 \in F_2$ and $x'_5 \in F_5$. Therefore BadRPrepare have occurred, leading to a contradiction.

If the tree is a $(3, 6)$-tree, then $i = 2$. The newly defined queries are in positions 7 and 8, which does not affect the incidence of $(2, x_2)$ with active 2chains (cf. the equivalent conditions in Lemma 36). However, new permutation queries are also defined in PrepareTree. Consider SimP(x'_8, x'_9) called in PrepareTree(n'): Since BadP didn't occur, the returned value (x'_0, x'_1) satisfies $x'_1 \notin F_1$. If a 2chain (u_0, x'_1) becomes active after the permutation query, since $x'_1 \notin F_1$, it cannot satisfy the conditions in Lemma 36. Thus, the conditions in Lemma 36 remain unsatisfied and $(2, x_2)$ is not incident with an active 2chain by the lemma. □

Lemma 60. *Let* n *and* n' *be distinct non-root nodes in a* $(2, 7)$*-tree rooted at* r*, and let* $(4, x_4)$ *and* $(5, x'_5)$ *be adapted in* AdaptNode(n) *and* AdaptNode(n') *respectively. Then the queries* $(3, F_4(x_4) \oplus x'_5)$ *and* $(6, x_4 \oplus F_5(x'_5))$ *are not active when* AdaptTree(r) *returns.*

Proof. Since the queries getting defined in AdaptTree(r) are in positions 4 and 5, we only need to prove the two queries are not active when AdaptTree(r) is called.

The queries $(3, x_3)$ and $(3, x'_3)$ are defined in PrepareTree(n) and PrepareTree(n') respectively. We have $x_3 \neq x'_3$ by Lemma 46. Then BadAPair occurs if the query $(6, x_4 \oplus F_5(x'_5))$ is active when AdaptTree(r) is called. Similarly we have $x_6 \neq x'_6$ and hence $(3, F_4(x_4) \oplus x'_5)$ is not active or BadAPair occurs. □

Lemma 61. *In an execution of* G_3*, the assertions in procedures* ReadTape *and* Adapt *always hold.*

Proof. ReadTape is called in SampleTree and PrepareTree, and Adapt is called in AdaptNode.

In a call to SampleTree(n), $n.end$ is sampled. By Lemma 10, the query $n.end$ is not defined when SampleTree(n) is called. Moreover, n is deleted from N before ReadTape is called, thus the query is not pending.

For ReadTape called in PrepareTree and Adapt called in AdaptNode, the queries being sampled or adapted are not pending or defined due to Lemmas 46 and 48. □

We are left with the assertions in MakeNodeReady and F. The procedure F is only called by the distinguisher and by MakeNodeReady.

A call to NewTree can be split into two phases: the *construction phase* consists of the first part of NewTree until GrowTree returns, and the *completion phase* consists of the next five instructions in NewTree, i.e., until AdaptTree returns. By extension, we say that a tree is in its *construction phase* or in its *completion phase* if the call to NewTree that created the tree is in the respective phase. The *phase* of the simulator is the phase of the tree being handled currently, i.e., is the phase of the last call to NewTree that has not yet returned.

A tree is *completed* if its completion phase is over, i.e., if AdaptTree(r) has returned, where r is the root of the tree. This is quasi-synonymous with a tree being *discarded*, where we recall that a tree is "discarded" when its root drops off the stack, i.e., when the call to NewTree in which the tree was created returns.

The simulator switches from the construction phase of a tree to the construction phase of another tree when a call to F causes a new tree to be created. The simulator will enter the construction phase of the new tree and will only resume the construction phase of the previous tree after the new tree is completed (and discarded). On the other hand, once the simulator enters the completion phase of a tree, it remains inside the completion phase of that tree until the phase is finished. In particular, at most one tree is in its completion phase at a time, and if the simulator is not in a completion phase then *no* tree is in its completion phase.

We note that calls to F and to MakeNodeReady do not occur when the simulator is in a completion phase, and, in particular, the assertions in these procedures take place when the simulator is not in a completion phase. This explains why for the following proof, we focus on properties that hold when the simulator is not in a completion phase.

Lemma 62. *For* $i = 3, 4$*, if* $x_i \in F_i$*,* $x_{i+1} \in F_{i+1}$*,* $x_{i+2} \in F_{i+2}$ *and* $F_{i+1}(x_{i+1}) = x_i \oplus x_{i+2}$*, there exists a node* n *whose maximal path contains* (i, x_i)*,* $(i+1, x_{i+1})$ *and* $(i+2, x_{i+2})$*.*

If $x_1 \in F_1$*,* $x_2 \in F_2$*,* $x_8 \in F_8$ *and* $T(x_0, x_1) = (x_8, *)$ *where* $*$ *is an arbitrary* n*-bit string and where* $x_0 = F_1(x_1) \oplus x_2$*, there exists a node* n *whose maximal path contains* x_1*,* x_2 *and* x_8*. Symmetrically, if* $x_7 \in F_7$*,* $x_8 \in F_8$*,* $x_1 \in F_1$ *and* $T^{-1}(x_8, x_9) = (*, x_1)$ *where* $*$ *is an arbitrary* n*-bit string and where* $x_9 = F_8(x_8) \oplus x_7$*, there exists a node* n *whose maximal path contains* x_7*,* x_8 *and* x_1*.*

Moreover, if the simulator is not in a completion phase, the node n *is completed.*

Proof. For the first part of the lemma, the proof for $i = 3$ is given; the case where $i = 4$ is symmetric. Let the three queries be $(3, x_3)$, $(4, x_4)$ and $(5, x_5)$.[19] We discuss the query that is defined latest

[19] The three queries are not necessarily in the maximal path of the same node n, i.e., we don't follow the convention that (i, x_i) denotes a query in the maximal path of n.

among these three queries.

First consider the case where the query is adapted. Assume that the latest query is defined in AdaptNode(n) for some node n in the tree rooted at r.

If neither of the other two queries is adapted in AdaptTree(r), they must have been defined when AdaptTree(r) is called. The latest query cannot be $(4, x_4)$, otherwise BadAHit occurs since $x_3 \in F_3$ and $x_5 \in F_5$. If the latest query is $(3, x_3)$, then $(4, x_4, x_5)$ is an active 2chain incident with $(3, x_3)$ when AdaptTree is called; by Lemma 59, x_4 and x_5 must be in the maximal path of n, so n is the node whose maximal path contains all three queries. If the latest query is $(5, x_5)$, the proof is similar to the previous case.

If at least two of the queries are adapted during AdaptTree(r), from Table 2 we can observe that r must be the root of a $(2, 7)$-tree (in other cases, at least one of the adapt positions is not in $\{3, 4, 5, 6\}$), and the adapted queries are in positions 4 and 5. If the two queries are adapted during the same call AdaptNode(n), the maximal path of n contains $(4, x_4)$ and $(5, x_5)$; since the node is successfully adapted, its maximal path also contains $(3, x_3 = F_4(x_4) \oplus x_5)$[20]. If $(4, x_4)$ and $(5, x_5)$ are adapted in AdaptNode(n) and AdaptNode(n') for $n \neq n'$, by Lemma 60 the query $(3, x_3 = F_4(x_4) \oplus x_5)$ is not active, leading to a contradiction.

Next we consider the case where the latest query is defined by ReadTape. The query cannot be $(4, x_4)$, otherwise BadRHit occurs since $F_4(x_4) = x_3 \oplus x_5$ for $x_3 \in F_3$, $x_5 \in F_5$. If the query is $(3, x_3)$ and is sampled in PrepareTree(n), then it is incident with an active 2chain $(4, x_4, x_5)$. By Lemma 46, the maximal path of n contains the 2chain and hence it contains all three queries.

We are left with the possibility that the latest query is $(3, x_3)$ or $(5, x_5)$, sampled in SampleTree(n) for some node n. We let the latest query be $(3, x_3)$ and the other case is similar. Let r be the root of the tree containing n. Consider the call GrowTree(r), where GrowTreeOnce(r) is called repeatedly. In the last iteration, *modified* is not set to **true** and thus no new node is added. In particular, no query becomes pending or defined in the last iteration. Moreover, the tree is a $(3, 6)$-tree and neither $(4, x_4)$ nor $(5, x_5)$ can be sampled during SampleTree(r). Therefore, we have $x_4 \in F_4$ and $x_5 \in F_5$ when the last call to GrowTreeOnce(r) occurs.

Consider FindNewChildren(n) called during the last call to GrowTreeOnce(r). When the triple (x_3, x_4, x_5) is checked, no child is added (as discussed before). Thus, we have either Equivalent($n.id, (3, x_3, x_4)$) = **true** or InChildren($n, (3, x_3, x_4)$) = **true**. In both cases, a node n' ($n' = n$ in the first case and $n' \in n.children$ in the second case) exists such that the maximal path of n' contains x_3 and x_4. Moreover, since n' is ready (recall that all existing nodes in a tree are ready when FindNewChildren is called) and is in a $(3, 6)$-tree, the endpoints of its maximal path are 3 and 6. Thus, the maximal path of n' can be extended to the endpoints and contains $x_5 = x_3 \oplus F_4(x_4)$.

The second part of the lemma is proved similarly, so we will omit some details. We give the proof for the first statement (i.e., the one about x_1, x_2 and x_8), and the proof for the second statement is almost symmetric (a difference is that $(7, x_7)$ may be queried in PrepareTree; this case can be handled using Lemma 46, as in the first part of the lemma).

We discuss which of the entries $F_1(x_1)$, $F_2(x_2)$ and $F_8(x_8)$ is defined latest. Note that the permutation query $T(x_0, x_1)$ (where $x_0 = F_1(x_1) \oplus x_2$ as usual) must be defined before $F_8(x_8)$, since otherwise BadPHit occurs. In particular, we have $(x_0, x_1) \in T$ when the latest query is defined.

If $F_1(x_1)$ is the latest query, it cannot be defined by ReadTape, or BadRHit occurs. If $F_1(x_1)$

[20] Note that the result cannot be obtained by simply extending the path, since in some pathological situation the maximal path is not "closed". Note that we can extend a *completed* path, which is closed by definition.

is adapted in AdaptNode(n), by assumption there exists x_9 such that $T^{-1}(x_8, x_9) = (x_0, x_1)$. By Lemma 48, $(8, x_8)$ and $(9, x_9)$ are in the maximal path of n. Since $(1, x_1)$ is adapted successfully, the path also contains $x_2 = F_1(x_1) \oplus x_0$.

If $F_2(x_2)$ is defined latest, then it is incident with an active 2chain $(0, x_0, x_1)$ when it is defined. If it is defined during SampleTree(n) for some node n, then similarly to the first part of the lemma, we can prove that the maximal path of n or one of n's children contains the three queries. If it is defined in AdaptNode(n), let r be the root of the tree containing n. If $(1, x_1)$ is not defined during AdaptTree(r), by Lemma 59, n's maximal path contains the 2chain $(0, x_0, x_1)$. If $(1, x_1)$ is also defined in AdaptNode(n), the maximal path of n contains both x_1 and x_2. In the above cases, since AdaptNode(n) is successful, the maximal path of n is completed and thus also contains x_8. If $(1, x_1)$ is defined in AdaptNode(n') where $n' \neq n$ is another node in the tree rooted at r, then since $T(x_8, x_9) = (x_0, x_1)$ and by Lemma 48, the maximal path of n' contains x_8 and x_9. The query $(2, x_2 = F_1(x_1) \oplus x_0)$ is adapted during the call AdaptNode(n'), which occurs before AdaptNode(n) is called, contradicting our assumption.

If $F_8(x_8)$ is defined last, it is incident with an active 2chain $(1, x_1, x_2)$ and the proof is similar to the previous case.

Finally, consider the last part of the lemma. We prove that if a node has not entered the completion phase, it cannot contain defined queries in the required positions (i.e., positions $\{3, 4, 5\}$, $\{4, 5, 6\}$, $\{1, 2, 8\}$ or $\{7, 8, 1\}$). Observe that each of the triples consist of three positions in either $\{1, 2, 7, 8\}$ or $\{3, 4, 5, 6\}$. From Table 2 we observe that when a node becomes ready, its maximal path contains at most two defined queries in each set of positions $\{1, 2, 7, 8\}$ and $\{3, 4, 5, 6\}$.

Therefore, the tree containing the node n must have entered the completion phase. Since the completion phase of a tree cannot be interrupted and the simulator is not in a completion phase, the completion phase of the tree must have finished and n has been completed. □

The above lemma implies that if the simulator is not in a completion phase, each triple of defined queries as in the lemma is contained in a completed path.

Lemma 63. *In an execution of* G_3*, the first assertion in* MakeNodeReady *always holds.*

Proof. The assertions in MakeNodeReady(n) occur right before $n.\mathit{end}$ is assigned. As in the pseudocode, let (j, x_j) be the query that is about to be assigned to $n.\mathit{end}$, where j equals the terminal of n. The first assertion asserts that the query (j, x_j) is not defined.

Note that when n is created, its maximal path already contains two defined queries as per FindNewChildren. More queries are made during the call to MakeNodeReady(n), which are also in n's maximal path according to Lemma 19. Let $\{x_h\}$ be the maximal path of n. Let i be the origin of n, and the positions of the defined queries are determined by i.

If $i = 1$, we have $j = 4$. The maximal path contains $(6, x_6)$ and $(5, x_5)$, both of which have been queried in MakeNodeReady(n). Assume by contradiction that $(j, x_j) = (4, x_4)$ is defined, then by Lemma 62, there exists a completed path containing the three queries $(4, x_4)$, $(5, x_5)$ and $(6, x_6)$. By extension, the completed path contains all queries in $\{x_h\}$. Thus $n.\mathit{beginning} = (1, x_1)$ is in a completed path and should be defined (recall that all the queries in a completed path are defined). However, the tree containing n hasn't entered the completion phase and hence SampleTree($n.\mathit{parent}$) hasn't been called. By Lemma 10, $(1, x_1) = n.\mathit{parent.end}$ cannot be defined, which leads to a contradiction.

The proof for $i = 2, 3, 4$ is similar, and we only give a sketch below. Again we will prove that if the (j, x_j) is defined, the maximal path of n contains a triple of defined queries that are contained by a completed path due to Lemma 62. The positions of the triple of queries are:

– If $i = 2$, then $j = 7$. Queries $(7, x_7)$, $(8, x_8)$ and $(1, x_1)$ are defined.
– If $i = 3$, then $j = 6$. Queries $(4, x_4)$, $(5, x_5)$ and $(6, x_6)$ are defined.
– If $i = 4$, then $j = 1$. Queries $(7, x_7)$, $(8, x_8)$ and $(1, x_1)$ are defined.

The completed path also contains $n.beginning$ by extension, thus $n.beginning$ is defined. However, SampleTree($n.parent$) hasn't been called, contradicting Lemma 10.

The cases $i = 5, 6, 7, 8$ are symmetric to $i = 4, 3, 2, 1$ respectively. □

Lemma 64. *In a call to* F *made by the distinguisher, the assertion in* F *holds.*

Proof. If F is called by the distinguisher, there is no pending query at the moment and the assertion trivially holds. □

The following group of lemmas build up to the proof that the second assertion in MakeNodeReady as well as the assertion in F called by MakeNodeReady do not abort.

We begin the analysis by giving some definitions that enable us to discuss all non-completed trees collectively.

Definition 17. The *tree stack* is a list of trees $(\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_\ell)$ consisting of all trees such that SampleTree(r_i) hasn't been called yet, where r_i is the root of $\mathcal{T}_i$, and where $\mathcal{T}_i$ is created before $\mathcal{T}_j$ for $i < j$.

A new tree is created when F calls NewTree and NewTree creates a new root node. Since a tree $\mathcal{T}_i$ with root r_i is removed from the tree stack when SampleTree(r_i) is called in NewTree, and since only the last call to NewTree on the stack can be in its completion phase, $\mathcal{T}_\ell$ will be the first to be removed from the tree stack. Hence the tree stack behaves in LIFO fashion, as indicated by its name.

If the simulator is in a construction phase and a tree rooted at r is not in the tree stack then the tree rooted at r must be completed. Indeed, the call SampleTree(r) has occurred by definition, so AdaptTree(r) must already have occurred and returned, given that the simulator is not in a completion phase.

Definition 18. A node n is *is in the tree stack* if n is in a tree $\mathcal{T}_i$ in the tree stack.

Lemma 65. *Assume the simulator is not in a completion phase. Then a query (i, x_i) is pending if and only if $(i, x_i) = n.end$ for some node n in the tree stack.*

Proof. Recall that a query is pending if and only if there exists a node n such that $n.end$ equals the query, and the query hasn't been defined. We only need to prove that $n.end$ is defined if and only if n is not in a tree in the tree stack.

If a tree rooted at r is not in the tree stack, then SampleTree(r) has been called. Moreover, as the simulator is not in a completion phase, SampleTree(r) has returned and thus the *end* of each node in the tree has been sampled.

On the other hand, SampleTree(r_i) hasn't been called for the roots r_i of trees in the tree stack, thus by Lemma 10 the *end* of the nodes in the tree stack are not defined. □

Lemma 66. *Let $(\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_\ell)$ be the tree stack. If $\ell \geq 1$ the tree $\mathcal{T}_1$ is created by a distinguisher query to* F*. Moreover $\mathcal{T}_{i+1}$ is created during the call to* MakeNodeReady(n_i)*, where n_i is the unique non-ready node in $\mathcal{T}_i$, for $1 \leq i < \ell$.*

Proof. The first tree to be created during a query cycle obviously comes from a distinguisher query to F, since if the distinguisher query to F does not cause a call to NewTree the simulator returns an answer immediately to the distinguisher. Moreover, this tree is only removed from the tree stack when the first call to NewTree enters its completion phase, after which no more calls to NewTree occur, since the simulator returns an answer to the distinguisher once the first call to NewTree returns.

The simulator calls F only in MakeNodeReady. Whenever a new tree is created, the simulator will not call MakeNodeReady on nodes in the old tree until the new tree is completed. Therefore $\mathcal{T}_{i+1}$ must be created in MakeNodeReady(n) for some n in $\mathcal{T}_i$, since $\mathcal{T}_i$ is the newest tree that hasn't been completed at the moment when $\mathcal{T}_{i+1}$ is created. Moreover, a call to F is made in MakeNodeReady(n) only when n is not ready. By Lemma 20, there is at most one non-ready node in a tree. Therefore, n must be the unique non-ready node in tree $\mathcal{T}_i$ at the moment when $\mathcal{T}_{i+1}$ is created.

Later, nodes may be added to $\mathcal{T}_{i+1}$ (and more trees may be added to the tree stack), but the root of $\mathcal{T}_{i+1}$ never changes and the state of $\mathcal{T}_i$ doesn't change until after $\mathcal{T}_{i+1}$ leaves the tree stack. This completes the lemma. □

For the rest of the proof ℓ will denote the number of trees in the tree stack. The above lemma implies that each tree $\mathcal{T}_i$ for $i < \ell$ in the tree stack contains a non-ready node. The non-ready node is a leaf, because non-ready nodes cannot have children. Thus each tree in the tree stack (except possibly $\mathcal{T}_\ell$) contains a unique non-ready leaf, where the uniqueness is by Lemma 20.

Lemma 67. *If the origin of a node n is 2, 3, 6 or 7, then the calls to* F *made by* MakeNodeReady(n) *are for queries that are already defined.*

Proof. Table 2 shows the positions of queries to F made during MakeNodeReady(n) and the positions of queries that are already defined.

This lemma is an observation from Table 2. □

Lemma 68. *Let $(\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_\ell)$ be the tree stack. If $\mathcal{T}_i$ is a $(2, 7)$-tree or a $(3, 6)$-tree, then $i = \ell$, i.e., $\mathcal{T}_i$ must be the last tree in the tree stack.*

Proof. By Lemma 66, the tree $\mathcal{T}_{i+1}$ is created when F is called during MakeNodeReady(n_i), where n_i is a node in $\mathcal{T}_i$. By Lemma 67, if the origin of n_i is 2, 3, 6 or 7, F is only called on defined queries during MakeNodeReady(n_i), so NewTree is never called. Thus, $\mathcal{T}_{i+1}$ can never be created if $\mathcal{T}_i$ is a $(2, 7)$- or $(3, 6)$-tree, and T_i must be the last tree in the tree stack. □

In the following discussion, we will focus on a point in time when the second assertion in MakeNodeReady(n) aborts or when F called by MakeNodeReady(n) aborts. In such a case n must be a node in $\mathcal{T}_\ell$ since a tree is always "put on hold" while a new tree is created and completed. Thus n must be the unique non-ready leaf of $\mathcal{T}_\ell$ and, in particular, the last tree on the tree stack has a non-ready leaf.

We let r_i denote the root of $\mathcal{T}_i$ and n_i denote the unique non-ready leaf in $\mathcal{T}_i$, $1 \leq i \leq \ell$.

Next we reiterate the formal definition of a full partial path (already given in Section 4) and introduce the notion of a *proper* partial path. Every proper path is also full.

Definition 19. An (i, j)-partial path $\{x_h\}_{h=i}^{j}$ is *full* if $1 \le i, j \le 8$ and if $x_i \notin F_i$ and $x_j \notin F_j$. Moreover, a full (i, j)-partial path is *proper* if $(i, j) \in \{(5,1), (4,1), (4,8), (1,5), (8,5), (8,4)\}$. A proper partial path is an *outer proper partial path* if $i > j$, and is an *inner proper partial path* if $i < j$.

Observe that an inner proper partial path must be a $(1,5)$-partial path or a $(4,8)$-partial path.

Lemma 69. *A 2chain is contained in at most one full partial path.*

Proof. This is easy to see, but we provide a detailed proof for completeness.

Let $\{x_h\}_{h=i}^{j}$ be a partial path containing the 2chain (k, x_k, x_{k+1}). Then the sequence $x_{k+2}, \dots, x_j$ (where, as usual, x_9 is followed by x_0) is uniquely determined by (k, x_k, x_{k+1}) and j, and the sequence $x_{k-1}, \dots, x_i$ (where, as usual, x_0 is followed by x_9) is uniquely determined by (k, x_k, x_{k+1}) and i. Also, we have $F_h(x_h) \ne \perp$ for $h \ne i, j, 0, 9$ by the definition of a partial path. The full partial path containing (k, x_k, x_{k+1}) (if it exists) is thus uniquely determined by the additional requirement that $x_i \notin F_i$, $x_j \notin F_j$. □

Definition 20. The queries (i, x_i) and (j, x_j) are called the *endpoint queries* of the partial path $\{x_h\}_{h=i}^{j}$.

Definition 21. An *oriented partial path* is a pair $R = (P, \sigma)$ where $P = \{x_h\}_{h=i}^{j}$ is a partial path and where $\sigma \in \{+, -\}$. The *starting point* of R is (i, x_i) if $\sigma = +$ and is (j, x_j) if $\sigma = -$. The *ending point* of R is (j, x_j) if $\sigma = +$ and is (i, x_i) if $\sigma = -$.

Definition 22. A *path cycle* is a sequence of oriented full partial paths $((P_1, \sigma_1), \dots, (P_t, \sigma_t))$, $t \ge 2$, such that:

1. Adjacent paths in the cycle are distinct, i.e., $P_s \ne P_{s+1}$ for all $1 \le s \le t$, where $(P_{t+1}, \sigma_{t+1}) \coloneqq (P_1, \sigma_1)$.
2. The ending point of (P_s, σ_s) is the starting point of (P_{s+1}, σ_{s+1}) for $1 \le s \le t$.

Definition 23. A path cycle $((P_1, \sigma_1), \dots, (P_t, \sigma_t))$ is a $(3,6)$-*cycle* (resp. $(7,2)$-*cycle*) if for $1 \le s \le t$, P_s is a $(3,6)$-full path (resp. $(7,2)$-full path).

Definition 24. A path cycle $((P_1, \sigma_1), \dots, (P_t, \sigma_t))$ is a *proper cycle* if P_s is a proper path for $1 \le s \le t$, and not both P_s and P_{s+1} are proper inner partial paths for $1 \le s \le t$, where $P_{t+1} \coloneqq P_1$.

Next we prove that if abortion occurs in the second assertion in MakeNodeReady or in the assertion in F, there exists a $(3,6)$-cycle, a $(7,2)$-cycle or a proper cycle.

Lemma 70. *For $i < \ell$, the endpoint queries of n_i's maximal path are n_i.beginning and r_{i+1}.end.*

Proof. By Lemma 66, $\mathcal{T}_{i+1}$ is created during the call to MakeNodeReady(n_i). The query $r_{i+1}.end = (j, x_j)$ is issued by MakeNodeReady(n_i) and thus is in the maximal path of n_i by Lemma 19. Since $\mathcal{T}_{i+1}$ has not entered the completion phase, $r_{i+1}.end$ is still pending. So it must be an endpoint query of the maximal path.

Moreover, the fact that n_i.*beginning* is also an endpoint of the maximal path follows directly by Definition 9. □

Lemma 71. *For $i < \ell$, if the origin of n_i is 1 (resp. 4, 5, 8) and if T_{i+1} is not a $(3,6)$- or $(2,7)$-tree, then the position of r_{i+1}.end is 5 (resp. 8, 1, 4).*

Proof. We can observe from Table 2 that when the origin of n is 1 (resp. 4, 5, 8) the only call to F that issues a (possibly) new query in positions other than 2, 3, 6, 7 is the call in position 5 (resp. 8, 1, 4). These positions are colored red in the table. ☐

Lemma 72. *Suppose $\mathcal{T}_\ell$ is not a $(3,6)$-tree or a $(2,7)$-tree. If $\mathcal{T}_i$ is a $(1,4)$-tree (resp. $(5,8)$-tree) and $i < \ell$, then $\mathcal{T}_{i+1}$ is a $(5,8)$-tree (resp. $(1,4)$-tree).*

Proof. This is a direct consequence of Lemma 71. ☐

Lemma 73. *When the simulator aborts in a call to* $\mathrm{F}(i, x_i)$ *by* MakeNodeReady(n), *we have $n = n_\ell$ and the origin of n is 1, 4, 5 or 8. Moreover, when the origin of n is 1 (resp. 4, 5, 8), i equals 5 (resp. 8, 1, 4).*

Proof. That $n = n_\ell$ follows from the fact that MakeNodeReady is only called on a node in the latest tree in the tree stack, and the node is not ready when abortion occurs.

Abortion occurs in the call $\mathrm{F}(i, x_i)$ if and only if (i, x_i) is pending. By Lemma 67, if the origin of n is 2, 3, 6 or 7, the calls to F by MakeNodeReady(n) must be on defined queries. These calls return immediately and don't abort. Thus the origin of n must be 1, 4, 5 or 8, which implies that T_ℓ is a $(1,4)$- or $(5,8)$-tree. By Lemma 68, there is no $(3,6)$- or $(2,7)$-tree in the tree stack, so queries in positions 2, 3, 6 or 7 are not pending as per Lemma 65.

Since (i, x_i) is pending, we must have $i \in \{1, 4, 5, 8\}$. Table 2 summarizes the queries to F by MakeNodeReady(n), where queries in positions 1, 4, 5 or 8 that are not necessarily defined are colored red. We can observe that if the origin of n is 1 (resp. 4, 5, 8), i equals 5 (resp. 8, 1, 4). ☐

Lemma 74. *Suppose the second assertion in* MakeNodeReady *or the assertion in* F *fails. If T_ℓ is a $(3,6)$-tree, the maximal path of each non-root node in T_ℓ is a $(3,6)$-full partial path; if T_ℓ is a $(2,7)$-tree, the maximal path of each non-root node in T_ℓ is a $(7,2)$-full partial path; if T_ℓ is a $(1,4)$- or $(5,8)$-tree, the maximal path of each non-root node in the tree stack is a proper partial path.*

Moreover, the endpoint queries of the aforementioned full partial paths are pending.

Proof. As a preliminary to the proof, we remind that pending queries are undefined.

Let n be a non-root node in the tree stack. By Lemma 65, the *end* of ready nodes in the tree stack are all pending. Since $n.beginning = n.parent.end$, $n.beginning$ is pending.

In particular, if n is ready, the endpoint queries of its maximal path are $n.beginning$ and $n.end$, both of which are pending; moreover, by the positions in Table 2, the maximal path of n is a $(3,6)$-full partial path if n is in a $(3,6)$-tree, is a $(7,2)$-full partial path if n is in a $(2,7)$-tree, and is a proper partial path if n is in a $(1,4)$- or $(5,8)$-tree. Therefore the statement holds for ready nodes, and in the following discussion we only consider non-ready leaves.

Consider the case when T_ℓ is a $(3,6)$-tree, and let n be the non-ready leaf of T_ℓ (i.e., $n = n_\ell$). By Lemma 73, abortion cannot occur in a call to F and hence must occur in the second assertion in MakeNodeReady. The statement follows by part 4 of Lemma 19 since the query which was going to be assigned to $n.end$ is pending by the abort condition.

The case when T_ℓ is a $(2,7)$-tree can be proved similarly.

Finally, consider the case when T_ℓ is a $(1,4)$- or $(5,8)$-tree. In this case the statement should be proved for all non-root, non-ready nodes in the tree stack (whereas the previous cases only concerns nodes in T_ℓ).

By Lemma 68 and the assumption, the tree stack contains no $(3,6)$- or $(2,7)$-tree. If $n = n_k$ for

$k < \ell$ then the statement follows by Lemmas 70 and 71 and because $n_k.\mathit{beginning}$ and $r_{k+1}.\mathit{end}$ are both pending.

If $n = n_\ell$, we discuss which abort condition is triggered. If the second assertion in MakeNodeReady(n) fails, then the proof is similar to the first case (of a $(3,6)$-tree). Otherwise the abortion occurs in a call $\mathrm{F}(h, x_h)$ made by MakeNodeReady(n), and in which case (h, x_h) is necessarily pending. The statement follows by Lemma 19 (which tells us that (h, x_h) is in the maximal path of n) and Lemma 73 (which tells us the possible origins of n and the corresponding values of h). ☐

Lemma 75. *If a node's maximal path is an inner proper path, then the node has origin* 4, 5 *and moreover the node is not ready.*

Proof. Recall that an inner proper path must be a $(1,5)$- or $(4,8)$-full partial path. Thus if the maximal path of a node is an inner proper path, its origin must be 1, 4, 5 or 8.

In particular, the maximal path of a ready node is not an inner path, which establishes the last part of the statement.

When a node n with origin 1 (resp. 8) is created, its maximal path contains $n.\mathit{id}$, which includes queries a query in position 8 (resp. 1). But inner paths don't contain queries in both positions 1 and 8, which establishes the first part of the statement. ☐

Lemma 76. *If the second assertion in* MakeNodeReady *or the assertion in* F *fails, then the nodes in the tree stack have distinct maximal paths.*

Proof. We assume by contradiction that there exists two distinct nodes m_1 and m_2 whose maximal paths are identical.

First we prove that neither m_1 nor m_2 is ready. Assume by contradiction that m_1 is ready, then the two endpoint queries of the maximal path of m_1 are $m_1.\mathit{beginning}$ and $m_1.\mathit{end}$. $m_2.\mathit{beginning}$ is also an endpoint query of m_2's maximal path. Since the maximal paths are identical, we have $m_1.\mathit{beginning} = m_2.\mathit{beginning}$ or $m_1.\mathit{end} = m_2.\mathit{beginning}$. In the former case, by Lemma 9 we have $m_1.\mathit{parent}.\mathit{end} = m_2.\mathit{parent}.\mathit{end}$ and furthermore by Lemma 7 we have $m_1.\mathit{parent} = m_2.\mathit{parent}$. In the latter case, by Lemma 9 we know $m_1 = m_2.\mathit{parent}$. However, the maximal path of m_1 contains both queries in $m_2.\mathit{id}$, contradicting Lemma 23 in both cases. Similarly we can prove that m_2 is not ready.

Since the only non-ready node in T_i is n_i, we have $m_1 = n_i$ and $m_2 = n_j$, where we assume $i < j$ without loss of generality.

By Lemma 74, the maximal paths of nodes in $(3,6)$-trees, in $(2,7)$-trees and in $(1,4)$- and $(5,8)$-trees are of different types. If the maximal paths of m_1 and m_2 are $(3,6)$- or $(7,2)$-full paths, then both m_1 and m_2 must be in T_ℓ by Lemma 68. However, $m_1 = n_i$ is not in T_ℓ since $i < j \leq \ell$, which is a contradiction.

Therefore, the maximal paths of m_1 and m_2 must be proper paths and the origins of m_1 and m_2 are 1, 4, 5 or 8.

For $i < \ell$, the maximal path of n_i is a $(5,1)$-, $(4,8)$-, $(1,5)$- or $(8,4)$-full path, by Lemmas 71 and 70. We note that if two partial paths are identical, their endpoints must be identical[21]. If $j = \ell$, then the maximal path of n_ℓ must also be in the aforementioned positions. In this case abortion cannot occur in the second assertion of MakeNodeReady(n), otherwise the maximal path of n_ℓ is a $(4,1)$- or $(8,5)$-full path.

[21] To wit, an (i,j)-partial path and an (i',j')-partial path have *identical endpoints* if and only if $i = i'$ and $j = j'$.

By Lemmas 71 and 73, the endpoints of the maximal paths of n_i and n_j are determined[22] by the origins of n_i and n_j, and their endpoints are identical only if the origins are the same (this also holds when $j = \ell$, because the positions in Lemma 73 are the same as in Lemma 71).

Since n_i and n_j are in different trees, they must have different parent nodes, and by Lemma 7 we have $n_i.beginning \neq n_j.beginning$. But $n_i.beginning$ and $n_j.beginning$ are in their respective maximal paths, and the two queries are in the same position since the origins are the same. Thus the maximal paths contain different queries in the position and cannot be identical. ☐

Lemma 77. *If the second assertion in* MakeNodeReady *or the assertion in* F *fails, there exists a* $(3, 6)$*-cycle, a* $(7, 2)$*-cycle or a proper cycle (cf. Definitions 23, 24).*

Proof. By Lemma 64, the assertion in F never fails if called by the distinguisher. Thus we only need to consider calls to F by MakeNodeReady.

As usual, let $(\mathcal{T}_1, \ldots, \mathcal{T}_\ell)$ be the tree stack when the simulator aborts, and let r_i and n_i denote the root and the non-ready leaf respectively in $\mathcal{T}_i$ for $i = 1, \ldots, \ell$. Then the abortion occurs in MakeNodeReady(n_ℓ) or in a call to F made by MakeNodeReady(n_ℓ), as discussed before Lemma 71.

When the second assertion in MakeNodeReady or the assertion in F fails, both endpoint queries of the maximal path of n_ℓ are pending by Lemma 74. Let (h, x_h) be the query which causes the assertion to fail, and which is therefore one of the endpoint queries of the maximal path of n_ℓ (the other endpoint query being $n_\ell.beginning$). By Lemma 65 there exists a node n' in the tree stack such that $n'.end = (h, x_h)$. Let $\mathcal{T}_k$ be the tree containing n'.

In each tree $\mathcal{T}_i$, there exists a unique route from n_i to r_i. Let τ_i be the sequence of nodes in the route except the last node r_i. Note that $n_i \neq r_i$, therefore τ_i contains at least one node n_i.

Moreover, in the tree $\mathcal{T}_k$, there exists a unique route from n_k to n'. Let γ be the sequence of nodes in this route, and let n_{top} be the highest node in the sequence (i.e., n_{top} is the node in the sequence closest to the root). Let γ_1 be the prefix of γ consisting of nodes to the left of n_{top}, and let γ_2 be the suffix of γ consisting of nodes to the right of n_{top}, with neither sub-sequence containing n_{top}.

Because n_k is a non-ready leaf while n' is ready, we have $n_k \neq n'$ and γ contains at least two nodes. The leaf n_k can only be adjacent to its parent, thus $n_k \neq n_{\text{top}}$. Thus n_k must be in the prefix γ_1 since it is the first node in γ, so γ_1 is not empty. (However, γ_2 may be empty if $n_{\text{top}} = n'$. This is also the only case in which $n' \notin \gamma_1 \cup \gamma_2$.) Moreover, if the root r_k is in γ, then we must have $n_{\text{top}} = r_k$. This implies that neither γ_1 nor γ_2 may contain r_k, i.e., the nodes in γ_1 and γ_2 are non-root nodes.

For each non-root node n we define the following two oriented partial paths:

- Let n^+ denote the *positive oriented path* of n, whose partial path equals the maximal path of n and whose starting point equals $n.beginning$;
- Let n^- denote the *negative oriented path* of n, whose partial path equals the maximal path of n and whose ending point equals $n.beginning$.

Moreover, for a sequence τ of non-root nodes, let τ^+ and τ^- be the sequences of positive and negative oriented paths of the nodes respectively. We claim that the concatenated sequence

$$(\tau_\ell^-, \tau_{\ell-1}^-, \ldots, \tau_{k+1}^-, \gamma_1^-, \gamma_2^+) \tag{5}$$

[22] In more detail, Lemmas 71 and 73 imply that if the origin of $n \in \{n_i, n_j\}$ is 1, 4, 5, 8 respectively, then the maximal path of n is a $(5, 1)$-, $(4, 8)$-, $(1, 5)$- and $(8, 4)$-partial path, respectively.

is a path cycle satisfying the requirements of the lemma.

Each oriented path in (5) contains the maximal path of a non-root node n in the tree stack. By Lemma 74, these maximal paths are full partial paths.

The sequence is of length at least 2: if $k < \ell$, both τ_ℓ and γ_1 contain at least one node; otherwise $k = \ell$, and it suffices to show that $n' \neq n_\ell$ and that n' is not the parent of n_ℓ; the former follows from the fact that n' is ready whereas n_ℓ is not, while the latter follows from the fact that $n'.\mathit{end} = (h, x_h) \neq n_\ell.\mathit{beginning}$.

By Lemma 76, the maximal paths of non-root nodes in the tree stack are distinct. Since each node appears in (5) at most once, the partial paths in the cycle are distinct and property 1 of Definition 22 holds.

If the origin of n_ℓ is 3 or 6, then h equals 6 or 3. T_ℓ is the only $(3,6)$-tree in the stack, so n' must also be a node in T_ℓ (i.e., $k = \ell$). Thus all paths in (5) are maximal paths of the nodes in T_ℓ, which are $(3,6)$-full paths by Lemma 74. Similarly we can prove if the origin of n_ℓ is 2 or 7, the paths in (5) are $(7,2)$-full paths.

If the origin of n_ℓ is 1, 4, 5 or 8, then T_ℓ is a $(1,4)$- or $(5,8)$-tree. By Lemma 74, the maximal paths of all nodes in the tree stack are proper partial paths. We remind that for (5) to be a proper cycle, it should not contain two consecutive inner proper paths. For convenience, we will call this property "property 3" in the following proof. (Property 3 also holds for a $(3,6)$- or $(7,2)$-cycle, since such a cycle contains no inner proper paths.) Both property 2 (of a path cycle) and property 3 concerns two adjacent paths in (5). In the following discussion, we will prove the two properties for each pair of adjacent paths.

Let $t \geq 2$ be the length of (5). Let $R_s = (P_s, \sigma_s)$ and $R_{s+1} = (P_{s+1}, \sigma_{s+1})$ be adjacent oriented paths in (5), with $s + 1 = 1$ if $s = t$, and let m_s and m_{s+1} be the nodes corresponding to R_s and R_{s+1}. We will distinguish between the following four cases: (case 1) m_s is not the last node of τ_i, γ_1 or γ_2, (case 2) m_s is the last node of τ_i, (case 3) m_s is the last node of γ_1, and (case 4) m_s is the last node of γ_2.

Case 1. If m_s is in τ_i or γ_1 and is not the last node in that sequence then m_{s+1} is in the same sequence and is the parent of m_s since these sequences represent a route towards the root (or towards n_{top}). Moreover we have $R_s = m_s^-$ and $R_{s+1} = m_{s+1}^-$ so the ending point of R_s and the starting point of R_{s+1} are $m_s.\mathit{beginning} = m_{s+1}.\mathit{end}$.

Only ready nodes have children, so m_{s+1} is ready. By Lemma 75, P_{s+1} is not an inner proper path, and property 3 follows.

Similarly, if m_s is in γ_2 and is not the last node of γ_2, m_{s+1} is also in γ_2 and is a child of m_s. We have $R_s = m_s^+$ and $R_{s+1} = m_{s+1}^+$, and the proof is symmetric to the previous case.

Case 2. If m_s is the last node of τ_i then its parent is r_i; furthermore, $m_{s+1} = n_{i-1}$ (i.e., the non-ready leaf in T_{i-1}) and $R_s = m_s^-$, $R_{s+1} = n_{i-1}^-$. The ending point of m_s^- is $m_s.\mathit{beginning}$ and, by Lemma 71, the starting point of n_{i-1}^- is $r_i.\mathit{end} = m_s.\mathit{beginning}$. This establishes property 2 of a path cycle.

For property 3, if the origin of $m_{s+1} = n_{i-1}$ is 1, 4, 5 or 8, the position of $r_i.\mathit{end} = m_s.\mathit{beginning}$ is 5, 8, 1 or 4 respectively. Either way at most one of the origins of m_s, m_{s+1} is 4 or 5, thus at most one of P_s and P_{s+1} is an inner proper path by Lemma 75.

Case 3. If m_s is the last node in γ_1 and γ_2 is not empty, then m_{s+1} is the first node in γ_2. Both m_s and m_{s+1} are children of n_{top}, so we have $m_s.\mathit{beginning} = m_{s+1}.\mathit{beginning} = n_{\text{top}}.\mathit{end}$. The *beginning* of the two nodes are the ending point of m_s^- and the starting point of m_{s+1}^+ respectively, thus property 2 holds. Since n_k is the unique non-ready node in $\mathcal{T}_k$ and $n_k \in \gamma_1$, the node $m_{s+1} \in \gamma_2$ is ready and, by Lemma 75, P_{s+1} is not an inner proper path.

On the other hand, if γ_2 is empty, then m_s is the last node of (5) and $m_{s+1} = m_1 = n_\ell$ and $n_{\text{top}} = n'$. The ending point of m_s^- is $m_s.beginning = n'.end = (h, x_h)$, which is in the maximal path of n_ℓ. More precisely, since this query is pending, it is the starting point of n_ℓ^- (while the ending point of n_ℓ^- is $n_\ell.beginning$).

Next we prove that the maximal paths of m_s and n_ℓ can't both be inner proper paths. By Lemma 75, the paths are inner proper paths only if both m_s and n_ℓ have origins 4 or 5. If n_ℓ has origin 4 or 5, then no matter whether the abortion occurs in the second assertion of MakeNodeReady or in the call to F, the query (h, x_h) is in position 1 or 8, i.e., the origin of m_s is $h \in \{1, 8\}$. Thus, m_s cannot be an inner proper path at the same time.

Case 4. If m_s is the last path in γ_2 (assuming γ_2 is non-empty), then $m_s = n'$ and $m_{s+1} = n_\ell$. The ending point of n'^+ is $n'.end = (h, x_h)$, which is also the starting point of n_ℓ^-. Since n' is ready, its maximal path is not an inner proper path by Lemma 75, so property 3 holds. □

Next we will prove that the aforementioned types of path cycles *never* exist in executions of G_3. Note that a path cycle can only be created when the tables are modified. The procedures that modify the tables are P, P^{-1}, ReadTape and Adapt. We will go through these procedures one-by-one and prove that none of them may create such a path cycle, provided that such a path cycle didn't previously exist.

Lemma 78. *In an execution of* G_3*, no* $(3,6)$-*,* $(7,2)$- *or proper cycle is created during a call to* P *or* P^{-1}.

Proof. We prove the lemma for a call to P, with the argument being symmetric for a call to P^{-1}.

The paths in a $(3,6)$-cycle don't contain permutation queries, thus such cycles are not created after a call to P.

Suppose an entry $T(x_0, x_1) = (x_8, x_9)$ is added in a call to P. We must have $x_8 \notin F_8$, otherwise BadPHit occurs. Thus, the path containing the permutation query cannot be a $(7,2)$-path and hence no $(7,2)$-cycle is created. Moreover, if a proper path p contains the permutation query, the proper path must be a $(8,5)$- or $(8,4)$-full path. This implies that $(8, x_8)$ is an endpoint query of p.

Assume a proper cycle is created after the call to P, then by definition, one of the paths adjacent to p also has $(8, x_8)$ as an endpoint query. Let the path be p'. It does not contain the permutation query, otherwise p and p' both contain the 2chain $(8, x_8, x_9)$ and are identical by Lemma 69, violating property 1 of Definition 22. Therefore the proper path p' already exists when the call to P is issued. The path p' is a $(4,8)$-, $(8,5)$- or $(8,4)$-full partial path, so $(8, x_8)$ is incident with an active 2chain when $P(x_0, x_1)$ is called. But then BadPCollide occurs in the call. □

Lemma 79. *In an execution of* G_3*, no* $(3,6)$-*,* $(7,2)$- *or proper cycle is created during a call to* ReadTape.

Proof. Consider a call ReadTape(i, x_i). For any path cycle created during the call, at least one of the partial paths in the cycle contains the query (i, x_i). Let $\{x_h\}_{h=s}^t$ denote a partial path in the cycle that contains x_i. Since $x_i \in F_i$, it cannot be in an endpoint of the path. Moreover, (i, x_i) must be adjacent to an endpoint of the path; otherwise $(i-1, x_{i-1}, x_i)$ is left active and (i, x_i, x_{i+1}) is right active (since neither x_{i-1} nor x_{i+1} is an endpoint query), and BadRHit occurs when ReadTape is called.

Without loss of generality, assume $i-1$ is an endpoint of the path, i.e., $s=i-1$. Observe that the length of a $(3,6)$-, $(7,2)$- or proper path is at least 4, so $i+1$ is not an endpoint of the path and hence the 2chain (i,x_i,x_{i+1}) is right active. On the other hand, an adjacent path in the path cycle, which we can denote $\{x'_h\}_{h=s'}^{t'}$, also contains the endpoint query x_{i-1}. If $\{x'_h\}_{h=s'}^{t'}$ also contains x_i, by Lemma 69, the two paths are identical, violating the definition of a path cycle. Therefore $\{x'_h\}_{h=s'}^{t'}$ cannot contain x_i, and exists before ReadTape(i,x_i) is called. But BadRCollide occurs when ReadTape(i,x_i) is called, because the 2chain (i,x_i,x_{i+1}) is right active and $(i-1,x_{i-1}) = (i-1,f_i(x_i)\oplus x_{i+1})$ is incident with an active 2chain (contained in the path $\{x'_h\}_{h=s'}^{t'}$). □

Finally we are left with the Adapt procedure.

Lemma 80. *In an execution of* G$_3$*, no* $(7,2)$*-cycle is created during a call to* Adapt*.*

Proof. A $(7,2)$-full path only contains defined queries in positions 1 and 8. The procedure Adapt is never called on queries in position 8. It is called on queries in position 1 during AdaptNode(n) if the origin of n is 3 or 6. Let x_i denote the queries in the maximal path of n, and in particular the adapted query is $(1,x_1)$.

Assume that a $(7,2)$-full path $\{x'_h\}_{h=7}^{2}$ is created after the call to Adapt. We have $T^{-1}(x'_8,x'_9) = (x'_0,x'_1)$, which implies $x'_8=x_8$ and $x'_9=x_9$ by Lemma 48. Thus $x'_7=F_8(x'_8)\oplus x'_9=F_8(x_8)\oplus x_9 = x_7$. However, the query $(7,x_7)$ has been defined in PrepareTree(n), while $(7,x'_7)$ is an endpoint query of a $(7,2)$-full path and should be undefined, leading to a contradiction. Therefore, no $(7,2)$-full path is created, and hence no $(7,2)$-cycle is created. □

For $(3,6)$- and proper cycles, we will not prove the result for each individual adaptation; instead, we will consider the adaptations that occur in a call to AdaptTree(r) all at once, where r is a root node.

In the following discussion, we will use the same notations and shorthands as in Definition 16. E.g., $\mathcal{A}$ denotes the set of adapted queries in AdaptTree (constructed by GetAdapts), and $\{x_h\}$ denotes the partial path associated to a node.

Lemma 81. *In an execution of* G$_3$*, no* $(3,6)$*-cycle is created during a call to* AdaptTree*.*

Proof. The defined queries contained by $(3,6)$-full paths are in positions 4 and 5. From Table 2 we can see that queries in positions 4 and 5 are only adapted when the origin of a node is 2 or 7. Thus we only need to consider a call to AdaptTree(r) where r is the root of a $(2,7)$-tree.

Assume by contradiction that a $(3,6)$-cycle is created during AdaptTree(r), then at least one $(3,6)$-full path in the cycle contains an adapted query. Let the path be $\{x_h\}_{h=3}^{6}$, and by the definition of a cycle, one of its adjacent paths in the cycle $\{x'_h\}_{h=3}^{6}$ has $x'_6=x_6$. Moreover, the two paths are not identical, and $x_6=x'_6\notin F_6$ since they are $(3,6)$-full paths. However, the queries $(4,x_4)$, $(5,x_5)$, $(4,x'_4)$ and $(5,x'_5)$ are either defined at the beginning of AdaptTree(r) or are adapted during the call (where at least one of $(4,x_4)$ and $(5,x_5)$ is adapted), with $x_4\oplus F_5(x_5)=x'_4\oplus F_5(x'_5)\notin F_6$ and $(x_4,x_5)\neq(x'_4,x'_5)$, so BadAMid occurs and the simulator should have aborted when AdaptTree(r) is called. □

Lemma 82. *In an execution of* G$_3$*, if no proper cycle has existed before a call to* AdaptTree*, no proper cycle is created during the call.*

Proof. Consider a call to AdaptTree(r) where r is a root node. If a proper cycle is created in the call, one of the proper partial paths in the cycle must contain an adapted query. Let $P = \{u_h\}$ be a proper path in the proper cycle that contains an adapted query $(i, x_i, y_i) \in \mathcal{A}$ (with $x_i = u_i$). Let the query be adapted in the call AdaptNode(n), and let (h, x_h) denote queries in the maximal path of n.

If $i = 1$ the proper path P contains a defined query in position 1 and must be a $(8, 5)$- or $(8, 4)$-full path. Thus $T^{-1}(u_8, u_9) = (u_0, x_1)$, and by Lemma 48 we have $u_8 = x_8$ and $u_9 = x_9$. The node n is in a $(3, 6)$-tree, and $(8, x_8)$ is sampled in PrepareTree(n) before $(1, x_1)$ is adapted. Thus $(8, x_8)$ is defined and cannot be an endpoint query of a full path, a contradiction! Therefore P cannot contain an adapted query in position 1.

Note that an adapted query cannot be in position 8.

If P contains exactly one adapted query $(i, x_i, y_i) \in \mathcal{A}$ $(1 < i < 8)$, we discuss the position of the query. If both $(i - 1, u_{i-1})$ and $(i + 1, u_{i+1})$ are defined, then $y_i = u_{i-1} \oplus u_{i+1}$ and BadAHit occurs. Thus (i, x_i) is next to an endpoint query of P; without loss of generality let $(i - 1, u_{i-1})$ be undefined. Note that a proper path contains at least three consecutive defined queries (which is easy to check from Definition 19). Because (i, x_i) is the only adapted queries, queries $(i + 1, u_{i+1})$ and $(i + 2, u_{i+2})$ are defined when AdaptTree(r) is called, and the query (i, x_i) is incident with the active 2chain $(i + 1, u_{i+1}, u_{i+2})$. By Lemma 59 the 2chain must be contained in the maximal path of n. Since P is a full path, by extension P contains all defined queries in the maximal path of n. Since AdaptNode(n) has been called, P contains at least 6 defined queries, which is too many for P to be a proper path.

Therefore, P must contain two adapted queries. The tree rooted at r cannot be a $(3, 6)$-tree, where the adapted queries are in positions 1 and 2 and P can only contain one in position 2. The tree cannot be a $(2, 7)$-tree either: the adapted queries are in positions 4 and 5, but we can observe that one of the endpoints of a proper path is 4 or 5, i.e., one of $(4, u_4)$ and $(5, u_5)$ is not defined. Without loss of generality we assume r is the root of a $(1, 4)$-tree, with the proof for a $(5, 8)$-tree being symmetric.

The queries in $\mathcal{A}$ are in positions 2 and 3. Let P contain $(2, x_2, y_2) \in A$ and $(3, x'_3, y'_3) \in \mathcal{A}$, adapted in paths $\{x_h\}$ and $\{x'_h\}$ respectively. The two paths cannot be identical, otherwise by extension $\{u_h\}$ is identical to $\{x_h\}$ and cannot be a proper path. For the same reason, we have $u_1 \neq x_1$ and $u_4 \neq x'_4$. If P is a $(8, 4)$- or $(8, 5)$-full path, $(0, u_0, u_1)$ is an active 2chain with which $(2, x_2)$ is incident. This contradicts Lemma 59 since $u_1 \neq x_1$. Thus P must be a $(1, 5)$-full path. The above argument applies to all proper paths containing queries in $\mathcal{A}$, i.e., if a proper path contains at least one query in $\mathcal{A}$, it must be a $(1, 5)$-full path.

The query $u_4 \in F_4$ is not adapted and has been defined since AdaptTree(r) is called. Note that $u_4 = x_2 \oplus y'_3$; if $x_1 \neq x'_1$, BadAPair occurs for the pair $(2, x_2, y_2)$ and $(3, x'_3, y'_3)$. Thus we must have $x_1 = x'_1$.

Now consider the path adjacent to P in the proper cycle that also contains the endpoint query $(1, u_1)$. Let $P' = \{u'_h\}$ denote the path, then $u'_1 = u_1$ and P' is a proper path. The path P' cannot contain a query in $\mathcal{A}$, otherwise it is also a $(1, 5)$-full path (as proved above), but a proper cycle cannot contain two adjacent inner proper paths (cf. Definition 24). Thus P' exists when AdaptTree(r) is called, so the query $(1, u'_1) = (1, u_1)$ is incident with an active 2chain (contained in P'). Since $u_1 = y_2 \oplus x'_3$, BadAPair occurs for the pair $(2, x_2, y_2)$ and $(3, x'_3, y'_3)$ if $x_4 \neq x'_4$. Hence we must have $x_4 = x'_4$.

From the above discussion, we have $x_1 = x'_1$ and $x_4 = x'_4$. Now we consider the point in time right before SampleTree(r) was called: Since SampleTree(n) and SampleTree(n') haven't been called, $\{x_h\}$ and $\{x'_h\}$ are distinct proper $(4,1)$-paths, and $(\{x_h\},+)$ and $(\{x'_h\},-)$ form a proper cycle of length 2. This contradicts the assumption that no path cycle existed before AdaptTree(r) is called! □

Lemma 83. *The simulator does not abort in good executions of* G_3.

Proof. Bad events don't occur in good executions, so the simulator doesn't abort in CheckBadP, CheckBadR or CheckBadA.

By Lemmas 78 through 82, none of the procedures P, P^{-1}, ReadTape and Adapt may create the first $(3,6)$-, $(7,2)$- or proper cycle. Since these are the only procedures that modify the tables, no query cycle can be created in any execution of G_3. By Lemma 77, the assertion in F and the second assertion in MakeNodeReady never fail. Moreover, by Lemmas 58, 61 and 63, the other assertions of the simulator don't fail.

Therefore, no abortion occurs in good executions of G_3. □

Lemma 84. *The probability that an execution of* G_3 *aborts is at most* $28392q^8/2^n$.

Proof. This directly follows by Lemmas 57 and 83. □

5.5 Transition from G_3 to G_4

With the result in the previous section, we can prove the indistinguishability of G_3 and G_5. We will upper bound $\Delta_D(\mathrm{G}_3, \mathrm{G}_4)$ and $\Delta_D(\mathrm{G}_4, \mathrm{G}_5)$, and use a triangle inequality to complete the transition. Our upper bound on $\Delta_D(\mathrm{G}_3, \mathrm{G}_4)$ holds only if D completes all paths (see Definition 1), which means that our final upper bound on $\Delta_D(\mathrm{G}_1, \mathrm{G}_5)$ holds only if D completes all paths. However, an additional reduction (see Theorem 97) implies the general case, at the cost of doubling the number of distinguisher queries. We also remind that lemmas marked with (*) are only hold under the assumption that D completes all paths.

The general idea for the following section is similar to the randomness mapping in [18], but since (and following [1]) we didn't replace the random permutation with a two-sided random function in intermediate games, the computation is slightly different. We also adapt a trick from [12] that ensures the probability of abortion in G_3 is not counted twice in the transition from G_3 to G_5, saving a factor of two overall.

Footprints. In the following discussion, we will rename the random tapes used in G_4 as g_1, g_2, $\ldots, g_8$ (all of which are random oracle tapes), in contrast to the tapes $f_1, f_2, \ldots, f_8$ used in G_3. The permutation tape p is only used in G_3, so need not be renamed.

We will use the notion of a *footprint* (from [1]) to characterize an execution of G_3 or G_4. Basically, the footprint of an execution is the subset of the random tapes that are actually used. Note that the footprint is defined with respect to the fixed distinguisher D.

Definition 25. A *partial random tape* is a table $\tilde{f}$ of size 2^n such that $\tilde{f}(x) \in \{0,1\}^n \cup \{\perp\}$ for each $x \in \{0,1\}$. A *partial random permutation tape* is a pair of tables $\tilde{p}$, $\tilde{p}^{-1}$ of size 2^{2n} such that $\tilde{p}(u), \tilde{p}^{-1}(v) \in \{0,1\}^{2n} \cup \{\perp\}$ for all u, $v \in \{0,1\}^{2n}$, such that $\tilde{p}^{-1}(\tilde{p}(u)) = u$ for all u such that $\tilde{p}(u) \neq \perp$, and such that $\tilde{p}(\tilde{p}^{-1}(v)) = v$ for all v such that $\tilde{p}^{-1}(v) \neq \perp$.

We note that random (permutation) tapes—in the sense used so far—can be viewed as special cases of partial random (permutation) tapes, namely, they are partial tapes with no $\perp$ entries. We also note that $\tilde{p}$ determines $\tilde{p}^{-1}$ and vice-versa in the above definition, so that we may use either $\tilde{p}$ or $\tilde{p}^{-1}$ to designate the pair $\tilde{p}/\tilde{p}^{-1}$.

Definition 26. A random tape f_i *extends* a partial random tape $\tilde{f}_i$ if $f_i(x) = \tilde{f}_i(x)$ for all $x \in \{0,1\}^n$ such that $\tilde{f}_i(x) \neq \perp$. A random permutation p *extends* a partial random permutation tape $\tilde{p}$ if $p(u) = \tilde{p}(u)$ for all $u \in \{0,1\}^{2n}$ such that $\tilde{p}(u) \neq \perp$. We also say that f_i (resp. p) is *compatible* with $\tilde{f}_i$ (resp. $\tilde{p}$) if f_i (resp. p) extends $\tilde{f}_i$ (resp. $\tilde{p}$).

We use the term *partial tape* to refer either to a partial random tape or to a partial random permutation tape.

Definition 27. Given an execution of G_3 with random tapes $f_1, f_2, \ldots, f_8, p$, the *footprint* of the execution is the set of partial tapes $\tilde{f}_1, \tilde{f}_2, \ldots, \tilde{f}_8, \tilde{p}$ consisting of entries of the corresponding tapes that are accessed at some point during the execution. (For the case of $\tilde{p}$, an access to $p(u)$ also counts as an access to $p^{-1}(p(u))$ and vice-versa.) Similarly, for an execution of G_4 with random tapes $g_1, g_2, \ldots, g_8$, the *footprint* is the set of partial tapes $\tilde{g}_1, \tilde{g}_2, \ldots, \tilde{g}_8$, with $\tilde{g}_i$ containing the entries of g_i that are accessed at some point during the G_4-execution.

Note that Definition 27 exclusively refers to the state of tape accesses at the *end* of an execution: we do not consider footprints as they evolve over time; rather, and given the fixed distinguisher D, the footprint is a deterministic function of the initial random tapes $f_1, \ldots, f_8$, p in G_3 or $g_1, \ldots, g_8$ in G_4.

Note that for the fixed distinguisher D, some combinations of partial tapes cannot be obtained as footprints. We thus let FP_3 and FP_4 denote the set of obtainable footprints in G_3 and G_4 respectively. For $i = 3$, 4, let $\Pr_{\mathrm{G}_i}[\omega]$ denote the probability of obtaining the footprint $\omega \in \mathsf{FP}_i$ in an execution of G_i.

We say that a set of random tapes is *compatible* with a footprint ω if each random tape is compatible with the corresponding partial tape in ω.

Lemma 85. *For $i = 3, 4$ and $\omega \in \mathsf{FP}_i$, an execution of G_i has footprint ω if and only if the random tapes are compatible with ω.*

Proof. Let $\mathcal{T} = (f_1, f_2, \ldots, f_8, p)$ if $i = 3$, $\mathcal{T} = (g_1, g_2, \ldots, g_8)$ if $i = 4$.

The "only if" direction is trivial: If the footprint of the execution with tapes $\mathcal{T}$ is ω, then by definition, ω consists of partial tapes that are compatible with the tapes in $\mathcal{T}$.

For the "if" direction, consider an arbitrary $\omega \in \mathsf{FP}_i$. There exist random tapes $\mathcal{T}'$ such that the execution of G_i with $\mathcal{T}'$ has footprint ω. During the execution with $\mathcal{T}'$, only entries in ω are read. If we run in parallel the executions of G_i with $\mathcal{T}$ and with $\mathcal{T}'$, the two executions can never diverge: as long as they don't diverge, the tape entries read in both executions exist in ω and hence are answered identically in the two execution. This implies that the executions with $\mathcal{T}'$ and $\mathcal{T}$ are identical and should have identical footprints. □

A corollary of Lemma 85 is that for $\omega \in \mathsf{FP}_i$, $\Pr_{\mathrm{G}_i}[\omega]$ equals the probability that the random tapes are compatible with ω. Let $|\tilde{f}| = |\{x \in \{0,1\}^n : \tilde{f}(x) \neq \perp\}|$, $|\tilde{p}| = |\{u \in \{0,1\}^{2n} : \tilde{p}(u) \neq \perp\}|$. Then the probability that random tapes in G_3 are compatible with a footprint $\omega = (\tilde{f}_1, \ldots, \tilde{f}_8, \tilde{p}) \in \mathsf{FP}_3$

is

$$\left(\prod_{i=1}^{8}\frac{1}{2^{n|\tilde{f}_i|}}\right)\left(\prod_{\ell=0}^{|\tilde{p}|-1}\frac{1}{2^{2n}-\ell}\right)=\Pr_{\mathrm{G}_3}[\omega], \tag{6}$$

by elementary counting. Similarly, the probability that random tapes in G_4 are compatible with $\omega = (\tilde{g}_1, \ldots, \tilde{g}_8) \in \mathsf{FP}_4$ is

$$\prod_{i=1}^{8}\frac{1}{2^{n|\tilde{g}_i|}}=\Pr_{\mathrm{G}_4}[\omega]. \tag{7}$$

Let $\Pr_{\mathrm{G}_i}[\mathcal{S}]$ denote the probability that one of the footprints in a set $\mathcal{S} \subseteq \mathsf{FP}_i$ is obtained. As every execution corresponds to a unique footprint, the events of obtaining different footprints are mutually exclusive, so

$$\Pr_{\mathrm{G}_i}[\mathcal{S}] = \sum_{\omega\in\mathcal{S}}\Pr_{\mathrm{G}_i}[\omega].$$

Since the distinguisher D is deterministic, we can recover a G_i-execution from a footprint $\omega \in \mathsf{FP}_i$ by simulating the execution, answering tape queries using entries in ω. We say a footprint is *non-aborting* if the corresponding execution does not abort. Let $\mathsf{FP}_3^* \subseteq \mathsf{FP}_3$ and $\mathsf{FP}_4^* \subseteq \mathsf{FP}_4$ be the set of all non-aborting footprints of G_3 and G_4 respectively.

Randomness Mapping. The heart of the randomness mapping is an injection $\zeta : \mathsf{FP}_3^* \to \mathsf{FP}_4^*$ such that executions with footprints ω and $\zeta(\omega)$ have the same output. Moreover, $\Pr_{\mathrm{G}_3}[\omega]$ will be close to $\Pr_{\mathrm{G}_4}[\zeta(\omega)]$.

Definition 28. The injection $\zeta : \mathsf{FP}_3^* \to \mathsf{FP}_4^*$ is defined as follows: for $\omega = (\tilde{f}_1, \ldots, \tilde{f}_8, \tilde{p}) \in \mathsf{FP}_3^*$, $\zeta(\omega) = (\tilde{g}_1, \ldots, \tilde{g}_8)$ where

$$\tilde{g}_i = \{(x, y) \in \{0,1\}^n \times \{0,1\}^n : F_i(x) = y\}$$

and where F_i refers to the table F_i at the end of the execution of G_3 with footprint ω.

Since we can recover an execution using its footprint, the states of the tables F_i at the end of the execution, as well as the output of the distinguisher, are determined by the footprint. Thus, the mapping ζ is well-defined. We still need to prove that ζ is an injection and that $\zeta(\omega) \in \mathsf{FP}_4^*$ (i.e., $\zeta(\omega)$ is a footprint of G_4 and is non-aborting).

We start by showing that answers to permutation queries in G_3 are compatible with the Feistel construction of the tables F_i.

Lemma 86. (*) *At the end of a non-aborting execution in* G_3 *or* G_4*, a permutation query* $T(x_0, x_1) = (x_8, x_9)$ *exists in* T *if and only if there exists a non-root node whose maximal path contains* x_0*,* x_1*,* x_8 *and* x_9*.*

Proof. By Lemma 22, at the end of a non-aborting execution, each non-root node corresponds to a completed path formed by the queries in its maximal path. Therefore, if the maximal path contains x_0, x_1, x_8 and x_9, then we have $T(x_0, x_1) = (x_8, x_9)$ due to the definition of a completed path.

To prove the "only if" direction, consider an arbitrary entry $T(x_0, x_1) = (x_8, x_9)$. If the entry is added by a simulator query, then it must be added during a call to MakeNodeReady(n) and, by Lemma 19, the values x_0, x_1, x_8 and x_9 are in the maximal path of n. Otherwise the entry is added by a distinguisher query. Since the distinguisher completes all paths, the distinguisher calls $\mathrm{F}(i, x_i)$

for $i \in \{1, 2, \dots, 5\}$, where $x_i := x_{i-2} \oplus \mathrm{F}(i-1, x_{i-1})$ for $2 \le i \le 5$. In particular, the queries $(3, x_3)$, $(4, x_4)$ and $(5, x_5)$ are defined before the end of the execution. By Lemma 62, there exists a node whose maximal path contains x_3, x_4 and x_5. The path also contains x_0 and x_1 (by definition of a completed path), as well as x_8 and x_9 (since $T(x_0, x_1) = (x_8, x_9)$ and by definition of a completed path). □

In the following lemma, we will prove that an execution of G_3 with footprint ω has the same output as an execution of G_4 with footprint $\zeta(\omega)$. Note that the simulators of G_3 and G_4 are not identical, thus the two executions cannot be "totally identical". Nonetheless, we can run an execution of G_3 and an execution of G_4 in parallel, and say they are *identical* if neither execution aborts, if the tables are identical anytime during the executions, and if calls to procedures that return a value return the same value in the two executions (note that some procedure calls only occur in G_3, but none of them return a value). In particular, if two executions of G_3 and G_4 are identical, then the answers to distinguisher queries are identical in the two executions and thus the deterministic distinguisher outputs the same value.

Lemma 87. (*) *The executions of* G_3 *and* G_4*, with footprints* ω *and* $\zeta(\omega)$ *respectively, are identical.*

Proof. Let $\omega = (\tilde{f}_1, \dots, \tilde{f}_8, \tilde{p}) \in \mathsf{FP}_3^*$. First we prove that $\zeta(\omega) \in \mathsf{FP}_4$, i.e., $\zeta(\omega)$ is the footprint of some execution of G_4. We arbitrarily extend the partial tapes in $\zeta(\omega) = (\tilde{g}_1, \dots, \tilde{g}_8)$ into a set of full random tapes $\lambda = (g_1, \dots, g_8)$. We will prove that the execution of G_4 with tapes λ has footprint $\zeta(\omega)$.

Consider an execution of G_3 with footprint ω, and an execution of G_4 with random tapes λ. We will prove that the two executions are identical as defined before this lemma. Note that the only differences between G_3 and G_4 are in the calls to CheckBadR and CheckBadA in G_3, in the permutation oracles P and P^{-1}, and in the different random tapes. Since $\omega \in \mathsf{FP}_3^*$, the execution of G_3 does not abort. Moreover, the procedures CheckBadP, CheckBadR and CheckBadA don't modify the global variables, therefore they can be ignored without affecting the execution. Now we prove by induction that as long as the executions are identical until the last line, they remain identical after the next line is executed. We only need to consider the case where the next line of code is different in G_3 and G_4.

If the next line reads a tape entry $f_i(x_i)$ in G_3, this must occur in a call to ReadTape and the entry will be written to $F_i(x_i) = f_i(x_i)$. By Lemma 3 the entry is never overwritten, so we have $F_i(x_i) = f_i(x_i)$ at the end of the execution and hence $\tilde{g}_i(x_i) = f_i(x_i)$. Moreover, g_i is an extension of $\tilde{g}_i$, which implies that the entry read in G_4 is $g_i(x_i) = f_i(x_i)$.

If the next line calls P or P^{-1}(issued by the distinguisher or by the simulator), the call outputs an entry of T. If the entry pre-exists before the call, then by the induction hypothesis, the output is identical in the two executions. Otherwise, the entry does not pre-exist in either execution, and a new entry will be added in both executions. We only need to prove that the same entry is added in both executions.

Let $T(x_0, x_1) = (x_8, x_9)$ be the new entry added by the call to P or P^{-1} in the G_3-execution. By Lemma 86, there exists a node whose maximal path contains x_0, x_1, x_8, x_9. By Lemma 22, the queries are in a completed path, which implies $\mathrm{Val}^+(0, x_0, x_1, i) = x_i$ for $i = 8, 9$. As discussed above, the defined queries also exist in g_i. Because in G_4 the call to P and P^{-1} is answered according to the Feistel network of g_i, the new entry in the G_4-execution is also $T(x_0, x_1) = (x_8, x_9)$.

By induction, we can prove that the two executions are identical. Furthermore, we can observe from the above argument that an entry $g_i(x_i)$ is read in G_4 if and only if the corresponding table entry $F_i(x_i)$ is defined in G_3: The calls to ReadTape are identical in the two executions, thus the query defined in G_3 is the same as the tape entry read in G_4. Entries of g_i read by P and P^{-1} in the G_4-execution are in a completed path in the G_3-execution and thus are defined. The queries defined by Adapt in the G_3-execution must be read in G_4 when the corresponding permutation query is being answered for the first time. Therefore, the footprint of the G_4-execution with tapes λ is $\zeta(\omega)$.

The G_4-execution does not abort by the definition of identical executions, so $\zeta(\omega) \in \mathsf{FP}_4^*$. □

Lemma 88. (*) *The mapping ζ defined in Definition 28 is an injection from FP_3^* to FP_4^*.*

Proof. By Lemma 87, for any $\omega \in \mathsf{FP}_3^*$, the G_4-execution with footprint $\zeta(\omega)$ is identical to the G_3-execution with footprint ω. In particular, neither execution aborts and thus $\zeta(\omega) \in \mathsf{FP}_4^*$.

That the executions are identical also implies that ζ is injective: Given $\zeta(\omega)$, the execution of G_4 can be recovered. In particular, we have the state of tables F_i and T at the end of the execution, which we denote by $\Sigma = (F_1, \ldots, F_{10}, T)$. Since the execution of G_3 with footprint ω is identical, the state of tables at the end of the execution is also Σ. We note that all tape entries read in a G_3-execution will be added to the corresponding table (entries of f_i are added to F_i, and entries of p are added to T). Thus ω can only contain entries in Σ.[23] Assume $\omega' \in \mathsf{FP}_3^*$ is also a preimage of $\zeta(\omega)$ under ζ, i.e., $\zeta(\omega') = \zeta(\omega)$. Similarly ω' only contains entries in Σ. In both executions with footprints ω and ω', tape queries receive answers compatible with Σ and the two executions can never diverge. This implies that the executions are identical and the footprints $\omega = \omega'$. Therefore, $\zeta(\omega)$ has a unique preimage $\omega \in \mathsf{FP}_3^*$, i.e., ζ is injective. □

Lemma 89. (*) *At the end of a non-aborting execution of* G_3*, the size of T equals the number of non-root nodes created throughout the execution.*

Proof. We only need to prove that maximal paths of different non-root nodes contain distinct (x_0, x_1) pairs, then by Lemma 86, there is a one-one correspondence between non-root nodes and permutation queries in T, implying that the numbers are equal.

By contradiction, assume that the maximal paths of two nodes both contain x_0 and x_1. By Lemma 22, the queries in the maximal paths of the nodes form two completed paths. Since a completed path can be determined by two queries in consecutive positions, the completed paths of the two nodes are identical. However, this is impossible in a non-aborting execution: After one of the nodes is completed, all queries in the completed path are defined. When AdaptNode is called on the other node (which must occur by the end of the execution), the queries to be adapted are defined and abortion will occur in the call to Adapt. □

Lemma 90. (*) *Let $\omega = (\tilde{f}_1, \ldots, \tilde{f}_8, \tilde{p}) \in \mathsf{FP}_3^*$ and $\zeta(\omega) = (\tilde{g}_1, \ldots, \tilde{g}_8) \in \mathsf{FP}_4^*$. Then*

$$\sum_{i=1}^{8} |\tilde{g}_i| = \sum_{i=1}^{8} |\tilde{f}_i| + 2 \cdot |\tilde{p}|.$$

Proof. Consider an execution of G_3 with footprint ω, and in the following discussion let F_i and T denote the state of the tables at the end of the execution. By the definition of the mapping ζ, g_i consists of entries in F_i, so the left-hand side of the equality equals the sum of $|F_i|$.

[23] More accurately, ω only contains entries in the *corresponding* tables in Σ, where F_i corresponds to f_i and T corresponds to p. We will abuse notations and not mention the transformation explicitly.

The queries in F_i are added exactly once, by either ReadTape or Adapt. We split F_i into two sub-tables F_i^R and F_i^A consisting of queries added by ReadTape and Adapt respectively. Let $F^A = \bigcup_i(\{i\} \times F_i^A)$ be the set of adapted queries in all positions (note that elements of F^A also include the position of the query).

In the execution of G_3, f_i are only read by ReadTape, and it is easy to see that $f_i(x_i)$ is read if and only if $x_i \in F_i^R$, which implies $|\tilde{f}_i| = |F_i^R|$.

The queries in F^A are adapted in Adapt called by AdaptNode. Two queries are adapted for each non-root node. By Lemma 89, the number of non-root nodes equals the size of T at the end of a non-aborting execution. Moreover, entries in T are only added by P and P^{-1}, and each entry $T(x_0, x_1) = (x_8, x_9)$ exists if and only if $p(x_0, x_1) = (x_8, x_9)$ is read. Thus $|T| = |\tilde{p}|$ and the number of adapted queries is $|F^A| = 2 \cdot |T| = 2 \cdot |\tilde{p}|$.

Putting everything together, we have

$$\sum_{i=1}^{8} |\tilde{g}_i| = \sum_{i=1}^{8} |F_i| = \sum_{i=1}^{8} |F_i^R| + |F^A| = \sum_{i=1}^{8} |\tilde{f}_i| + 2 \cdot |\tilde{p}|.$$

□

Lemma 91. (*) *For every $\omega \in \mathsf{FP}_3^*$, we have*

$$\Pr_{G_4}[\zeta(\omega)] \geq \Pr_{G_3}[\omega] \cdot (1 - 16q^4/2^{2n})$$

Proof. Let $\omega = (\tilde{f}_1, \ldots, \tilde{f}_8, \tilde{p}) \in \mathsf{FP}_3^*$, then by Lemma 88, $\zeta(\omega) = (\tilde{g}_1, \ldots, \tilde{g}_8) \in \mathsf{FP}_4^*$. By equations (6) and (7), we have

$$\begin{aligned}
\Pr_{G_4}[\zeta(\omega)] / \Pr_{G_3}[\omega] &= 2^{-n \sum |\tilde{g}_i|} \Bigg/ \left(2^{-n \sum |\tilde{f}_i|} \cdot \prod_{\ell=0}^{|\tilde{p}|-1} \frac{1}{2^{2n} - \ell} \right) \\
&= 2^{-n(\sum |\tilde{f}_i| + 2|\tilde{p}|)} \cdot 2^{n \sum |\tilde{f}_i|} \cdot \prod_{\ell=0}^{|\tilde{p}|-1} (2^{2n} - \ell) \\
&= 2^{-2n \cdot |\tilde{p}|} \cdot \prod_{\ell=0}^{|\tilde{p}|-1} (2^{2n} - \ell) \\
&\geq \left(\frac{2^{2n} - |\tilde{p}|}{2^{2n}} \right)^{|\tilde{p}|} \qquad (8)
\end{aligned}$$

where the second equality uses Lemma 90.

Note that each entry in $\tilde{p}$ corresponds to a distinct permutation query in T. By Lemma 28, we have $|T| \leq 4q^2$, so $|\tilde{p}| \leq 4q^2$. Since (8) is monotone decreasing with respect to $|\tilde{p}|$, we have

$$\left(\frac{2^{2n} - |\tilde{p}|}{2^{2n}} \right)^{|\tilde{p}|} \geq \left(\frac{2^{2n} - 4q^2}{2^{2n}} \right)^{4q^2} \geq 1 - \frac{16q^4}{2^{2n}}$$

and the lemma follows. □

Lemma 92. (*) *We have*

$$\mathit{\Delta}_D(\mathrm{G}_3, \mathrm{G}_4) \leq \Pr_{\mathrm{G}_4}[\mathsf{FP}_4^*] - \Pr_{\mathrm{G}_3}[\mathsf{FP}_3^*] + \frac{16q^4}{2^{2n}}.$$

Proof. Let $D^{\mathrm{G}_3}(\omega)$ denote the output of D in an execution of G_3 with footprint $\omega \in \mathsf{FP}_3$, and let $D^{\mathrm{G}_4}(\omega)$ denote the output of D in an execution of G_4 with footprint $\omega \in \mathsf{FP}_4$.

Recall that by assumption D outputs 1 when it sees abortion. Also note that abortion occurs in an execution of G_3 (resp. G_4) if and only if the footprint is not in FP_3^* (resp. FP_4^*). For $i \in \{3, 4\}$ we have

$$\Pr_{\mathrm{G}_i}[D^{\mathrm{F},\mathrm{P},\mathrm{P}^{-1}} = 1] = 1 - \Pr_{\mathrm{G}_i}[\mathsf{FP}_i^*] + \sum_{\omega \in \mathsf{FP}_i^*, D^{\mathrm{G}_i}(\omega)=1} \Pr_{\mathrm{G}_i}[\omega]. \tag{9}$$

By Lemma 87, executions with footprints ω and $\zeta(\omega)$ have the same output; by Lemma 88, ζ is injective. So $\zeta(\omega)$ is distinct for distinct ω and $\{\zeta(\omega) : \omega \in \mathsf{FP}_3^*,\ D^{\mathrm{G}_3}(\omega) = 1\}$ is a subset of $\{\omega : \omega \in \mathsf{FP}_4^*,\ D^{\mathrm{G}_4}(\omega) = 1\}$. Thus we have

$$\begin{aligned}\sum_{\omega \in \mathsf{FP}_4^*, D^{\mathrm{G}_4}(\omega)=1} \Pr_{\mathrm{G}_4}[\omega] &\geq \sum_{\omega \in \mathsf{FP}_3^*, D^{\mathrm{G}_3}(\omega)=1} \Pr_{\mathrm{G}_4}[\zeta(\omega)] \\ &\geq (1 - \frac{16q^4}{2^{2n}}) \sum_{\omega \in \mathsf{FP}_3^*, D^{\mathrm{G}_3}(\omega)=1} \Pr_{\mathrm{G}_3}[\omega] \end{aligned} \tag{10}$$

where the second inequality is due to Lemma 91.

Furthermore, combining (4) and (9), we have

$$\begin{aligned}\mathit{\Delta}_D(\mathrm{G}_3, \mathrm{G}_4) &= \Pr_{\mathrm{G}_3}[D^{\mathrm{F},\mathrm{P},\mathrm{P}^{-1}} = 1] - \Pr_{\mathrm{G}_4}[D^{\mathrm{F},\mathrm{P},\mathrm{P}^{-1}} = 1] \\ &= \Pr_{\mathrm{G}_4}[\mathsf{FP}_4^*] - \Pr_{\mathrm{G}_3}[\mathsf{FP}_3^*] + \sum_{\omega \in \mathsf{FP}_3^*, D^{\mathrm{G}_3}(\omega)=1} \Pr_{\mathrm{G}_3}[\omega] - \sum_{\omega \in \mathsf{FP}_4^*, D^{\mathrm{G}_4}(\omega)=1} \Pr_{\mathrm{G}_4}[\omega] \\ &\leq \Pr_{\mathrm{G}_4}[\mathsf{FP}_4^*] - \Pr_{\mathrm{G}_3}[\mathsf{FP}_3^*] + \left(1 - \left(1 - \frac{16q^4}{2^{2n}}\right)\right) \sum_{\omega \in \mathsf{FP}_3^*, D^{\mathrm{G}_3}(\omega)=1} \Pr_{\mathrm{G}_3}[\omega] \\ &\leq \Pr_{\mathrm{G}_4}[\mathsf{FP}_4^*] - \Pr_{\mathrm{G}_3}[\mathsf{FP}_3^*] + \frac{16q^4}{2^{2n}} \end{aligned}$$

where the first inequality follows by (10), and the second inequality uses the fact that the sum of probabilities of obtaining a subset of footprints is at most 1. □

5.6 Transition from G_4 to G_5

Lemma 93. *At the end of a non-aborting execution of* G_4*, the tables* F_i *are consistent with the tapes* g_i*.*

Proof. The entries of F_i added by ReadTape are read from g_i and thus are consistent with g_i.

For the entries added by Adapt, we prove the claim by induction on the number of calls to AdaptNode. Consider a call to AdaptNode(n), assuming that the entries added during the previous calls to AdaptNode are consistent with the tapes. When AdaptNode(n) is called, queries in the maximal path of n are defined except the queries to be adapted; in particular, its maximal path contains x_0, x_1, x_8, x_9 such that $T(x_0, x_1) = (x_8, x_9)$. The entry of T is added by P or P^{-1}, and from the pseudocode of G_4, we observe that there exists $x_2, x_3, \ldots, x_7$ such that $x_i = g_{i-1}(x_{i-1}) \oplus x_{i-2}$ for $i = 2, 3, \ldots, 9$. By the induction hypothesis, pre-existing queries in F_i are compatible with tapes g_i. Furthermore, when the call to AdaptNode(n) occurs the maximal path of n contains $x_0, x_1, \ldots, x_9$, and all these queries except the two queries to be adapted are defined. Note that $g_i(x_i) = x_{i-1} \oplus x_{i+1}$ also holds for each (i, x_i) to be adapted. By the pseudocode of AdaptNode, we can see that the queries adapted during the call to AdaptNode(n) are compatible with g_i. □

Lemma 94. *In a non-aborting execution of* G_4*, the distinguisher queries are answered identically to an execution of* G_5 *with the same random tapes. In particular, the distinguisher outputs the same value in the two executions.*

Proof. The permutation oracles in the two executions are identical and are independent to the state of tables, the answers to the permutation queries are identical in the two executions.

In G_4, calls to F return the corresponding entry in F_i. By Lemma 93, the tables F_i at the end of a G_4-execution are compatible with tapes g_i, and so are the answers of calls to F. In G_5, F directly returns the entry of g_i, which is the same as the answer in G_4. □

Lemma 95. *We have*

$$\Delta_D(\mathrm{G}_4, \mathrm{G}_5) \leq 1 - \Pr_{\mathrm{G}_4}[\mathsf{FP}_4^*].$$

Proof. By Lemma 94, if random tapes $g_1, \ldots, g_8$ result in a non-aborting execution of G_4, the execution of G_5 with the same random tapes have the same output. Therefore, the probabilities of outputting 1 with such tapes cancel out. The distinguisher only gains advantage in aborting executions of G_4, whose probability is $1 - \Pr_{\mathrm{G}_4}[\mathsf{FP}_4^*]$. □

5.7 Concluding the Indifferentiability

Now we can put everything together and give the indistinguishability between G_1 and G_5.

Lemma 96. (*) *The advantage of D in distinguishing* G_1 *and* G_5 *is at most* $24185q^8/2^n$*.*

Proof. We have

$$\begin{aligned}
\Delta_D(\mathrm{G}_1, \mathrm{G}_5) &\leq \Delta_D(\mathrm{G}_1, \mathrm{G}_2) + \Delta_D(\mathrm{G}_2, \mathrm{G}_5) \\
&\leq \Delta_D(\mathrm{G}_1, \mathrm{G}_2) + \Delta_D(\mathrm{G}_3, \mathrm{G}_5) \\
&\leq \Delta_D(\mathrm{G}_1, \mathrm{G}_2) + \Delta_D(\mathrm{G}_3, \mathrm{G}_4) + \Delta_D(\mathrm{G}_4, \mathrm{G}_5) \\
&\leq \frac{500q^8}{2^{2n}} + (\Pr_{\mathrm{G}_4}[\mathsf{FP}_4^*] - \Pr_{\mathrm{G}_3}[\mathsf{FP}_3^*] + \frac{16q^4}{2^{2n}}) + (1 - \Pr_{\mathrm{G}_4}[\mathsf{FP}_4^*]) \\
&\leq \frac{516q^8}{2^{2n}} + 1 - \Pr_{\mathrm{G}_3}[\mathsf{FP}_3^*] \\
&\leq \frac{516q^8}{2^{2n}} + \frac{28392q^8}{2^n} \\
&\leq \frac{28908q^8}{2^n}
\end{aligned}$$

where the second inequality is due to Lemma 35, the fourth inequality uses Lemmas 34, 92 and 95, and the second-to-last inequality is due to Lemma 84. □

Lemma 96 only holds if D completes all paths, because it relies on Lemma 92, which requires the same assumption. (This is what the '(*)' indicates, as we recall.) Our last step, thus, is to derive a bound that holds for all q-query distinguishers.

Theorem 97. *Any distinguisher that issues at most q queries to each of the round functions and at most q queries to the permutation oracle cannot distinguish the simulated world from the real world with advantage more than* $7400448q^8/2^n$.

Proof. Let D be an arbitrary distinguisher that issues at most q queries to each of its oracles. From D we can construct a distinguisher D^* that completes all paths, makes at most $2q$ queries in each position, and such that $\Delta_D(\mathrm{G}_1, \mathrm{G}_5) = \Delta_{D^*}(\mathrm{G}_1, \mathrm{G}_5)$. To wit, D^* starts by simulating D until D has finished its queries; assuming that the game has not aborted yet, D^* then completes all paths as in Definition 1, with respect to D's queries to $\mathrm{P}/\mathrm{P}^{-1}$. Since D has issued at most q queries to $\mathrm{P}/\mathrm{P}^{-1}$, D^* makes at most q extra queries in each position, for a total of at most $2q$ queries in each position. After doing this (or after the game aborts during this second phase, potentially) D^* outputs D's value, regardless of the result of the extra queries. Hence D^*'s output is always the same as D's, and the two distinguishers have the same advantage.

By Lemma 96, moreover, which applies to an *arbitrary* distinguisher making at most q queries to each oracle and completing all paths, D^* advantage at distinguishing G_1 and G_5 is at most $28908(2q)^8/2^n = 7400448q^8/2^n$. □

References

1. Elena Andreeva, Andrey Bogdanov, Yevgeniy Dodis, Bart Mennink, and John P. Steinberger. On the indifferentiability of key-alternating ciphers. In *Advances in Cryptology—CRYPTO 2013* (volume 1), pages 531–550.
2. Mihir Bellare and Phillip Rogaway, Random oracles are practical: A paradigm for designing efficient protocols, In Proceedings of the 1st ACM Conference on Computer and Communications Security (1993), pages 62–73.
3. Guido Bertoni, Joan Daemen, Michael Peeters, and Gilles Van Assche. On the indifferentiability of the sponge construction. In Smart [32], pages 181–197.
4. Ran Canetti, Security and composition of multi-party cryptographic protocols. Journal of Cryptology, 13(1): 143–202, 2000.
5. Ran Canetti, Universally composable security: A new paradigm for cryptographic protocols. In Proc. 42nd IEEE Symposium on Foundations of Computer Science (FOCS), pages 136–145, 2001.
6. Shan Chen and John Steinberger. Tight Security Bounds for Even-Mansour Ciphers, EUROCRYPT 2014, LNCS 8441, pp. 327–350, 2014.
7. Jean-Sébastien Coron, Yevgeniy Dodis, Cécile Malinaud, and Prashant Puniya. Merkle-damgård revisited: How to construct a hash function. In Victor Shoup, editor, *Advances in Cryptology—CRYPTO 2005*, volume 3621 of *LNCS*, pages 430–448. Springer-Verlag, 14–18 August 2005.
8. Jean-Sébastien Coron, Yevgeniy Dodis, Avradip Mandal, and Yannick Seurin. A domain extender for the ideal cipher. To appear in the *Theory of Cryptography Conference* (TCC), February 2010.
9. Jean-Sébastien Coron, Jacques Patarin, and Yannick Seurin. The random oracle model and the ideal cipher model are equivalent. In Wagner [33], pages 1–20.
10. Yuanxi Dai and John Steinberger, Indifferentiability of 10-Round Feistel Networks. Technical Report 2015/874, IACR eprint archive, 2015.
11. Yevgeniy Dodis and Prashant Puniya, On the Relation Between the Ideal Cipher and the Random Oracle Models. Proceedings of TCC 2006, 184–206.
12. Yevgeniy Dodis, Liu Tianren, John Steinberger and Martijn Stam, On the Indifferentiability of Confusion-Diffusion Networks. Technical Report 2015/XXX, IACR eprint archive, 2015.

13. Yevgeniy Dodis, Leonid Reyzin, Ronald L. Rivest, and Emily Shen. Indifferentiability of permutation-based compression functions and tree-based modes of operation, with applications to md6. In Orr Dunkelman, editor, *Fast Software Encryption: 16th International Workshop, FSE 2009*, volume 5665 of *Lecture Notes in Computer Science*, pages 104–121. Springer-Verlag, 22–25 February 2009.
14. Yevgeniy Dodis, Thomas Ristenpart, John Steinberger, and Stefano Tessaro, To Hash or Not to Hash Again? (In)differentiability Results for H^2 and HMAC. Advances in CryptologyCRYPTO 2012, LNCS 7417, pp. 348–366.
15. Hörst Feistel. Cryptographic coding for data-bank privacy. IBM Technical Report RC-2827, March 18 1970.
16. Hörst Feistel, William A. Notz, J. Lynn Smith. Some Cryptographic Techniques for Machine-to-Machine Data Communications. IEEE proceedings, **63**(11), pages 1545–1554, 1975.
17. Amos Fiat and Adi Shamir, How to Prove Yourself: Practical Solutions to Identification and Signature Problems. In Andrew M. Odlyzko, editor, Advances in Cryptology CRYPTO 86, volume 263 of LNCS, pages 186–194.
18. Thomas Holenstein, Robin Künzler, and Stefano Tessaro. The equivalence of the random oracle model and the ideal cipher model, revisited. In Lance Fortnow and Salil P. Vadhan, editors, *Proceedings of the 43rd ACM Symposium on Theory of Computing, STOC 2011, San Jose, CA, USA, 6-8 June 2011*, pages 89–98. ACM, 2011.
19. Dana Dachman-Soled, Jonathan Katz and Aishwarya Thiruvengadam, 10-Round Feistel is Indifferentiable from an Ideal Cipher. IACR eprint archive, 2015.
20. Rodolphe Lampe and Yannick Seurin. How to construct an ideal cipher from a small set of public permutations. ASIACRYPT 2013, LNCS 8269, 444–463. Springer, 2013.
21. M. Luby and C. Rackoff. How to construct pseudorandom permutations and pseudorandom functions. SIAM Journal on Computing, 17(2):373–386, April 1988.
22. Ueli M. Maurer, Renato Renner, and Clemens Holenstein. Indifferentiability, impossibility results on reductions, and applications to the random oracle methodology. In Moni Naor, editor, *First Theory of Cryptography Conference — TCC 2004*, LNCS 2951, 21–39. Springer, 2004.
23. M. Naor and O. Reingold, On the construction of pseudorandom permutations: Luby-Rackoff revisited, J. of Cryptology, 1999. Preliminary Version: STOC 1997.
24. Jacques Patarin, Security of balanced and unbalanced Feistel Schemes with Linear Non Equalities. Technical Report 2010/293, IACR eprint arxiv.
25. Birgit Pfitzmann and Michael Waidner, Composition and integrity preservation of secure reactive systems. In 7th ACM Conference on Computer and Communications Security, pages 245–254. ACM Press, 2000.
26. Birgit Pfitzmann and Michael Waidner, A model for asynchronous reactive systems and its application to secure message transmission. Technical Report 93350, IBM Research Division, Zürich, 2000.
27. David Pointcheval and Jacques Stern, Security Proofs for Signature Schemes. EUROCRYPT 1996, LNCS 1070, pp. 387–398.
28. Thomas Ristenpart and Hovav Shacham and Thomas Shrimpton, Careful with Composition: Limitations of the Indifferentiability Framework. EUROCRYPT 2011, LNCS 6632, pp. 487–506.
29. Phillip Rogaway, Viet Tung Hoang, On Generalized Feistel Networks. EUROCRYPT 2010, LNCS 6223, pp. 613–630, Springer, 2010.
30. Yannick Seurin, *Primitives et protocoles cryptographiques à sécurité prouvée*. PhD thesis, Université de Versailles Saint-Quentin-en-Yvelines, France, 2009.
31. Yannick Seurin, A Note on the Indifferentiability of the 10-Round Feistel Construction. Available from `yannickseurin.free.fr/pubs/Seurin_note_ten_rounds.pdf`.
32. Nigel P. Smart, editor. *Advances in Cryptology - EUROCRYPT 2008*, LNCS 4965. Springer-Verlag, 2008.
33. David Wagner, editor. *Advances in Cryptology - CRYPTO 2008*, LNCS 5157. Springer-Verlag, 2008.
34. R. Winternitz. A secure one-way hash function built from DES. *Proceedings of the IEEE Symposium on Information Security and Privacy*, pp. 88–90. IEEE Press, 1984.

G_1, G_2, G_3, G_4:
Global variables:
  Tables $F_1, \ldots, F_8$
  Permutation tables T_{sim}, T_{sim}^{-1}, T, T^{-1}
  Set of nodes N // Nodes that are ready
  Counter $NumOuter$ // Initialized to 0
Random oracle tapes: $f_1, \ldots, f_8$

class Node
  Node *parent*
  Set of **Node** *children*
  2chain *id*
  Queries *beginning*, *end*
  constructor Node(p, C)
    self.*parent* ← p
    self.*children* ← $\emptyset$
    self.*id* ← C
    self.*beginning* ← **null**
    if ($p \neq$ **null**) **then**
      self.*beginning* ← p.*end*
    self.*end* ← **null**

private procedure Assert(*fact*)
  if ¬*fact* **then abort**

private procedure SimP(x_0, x_1)
  if $(x_0, x_1) \notin T_{\text{sim}}$ **then**
    $(x_8, x_9) \leftarrow \text{P}(x_0, x_1)$
    $T_{\text{sim}}(x_0, x_1) \leftarrow (x_8, x_9)$
    $T_{\text{sim}}^{-1}(x_8, x_9) \leftarrow (x_0, x_1)$
  return $T_{\text{sim}}(x_0, x_1)$

private procedure SimP^{-1}(x_8, x_9)
  if $(x_8, x_9) \notin T_{\text{sim}}^{-1}$ **then**
    $(x_0, x_1) \leftarrow \text{P}^{-1}(x_8, x_9)$
    $T_{\text{sim}}(x_0, x_1) \leftarrow (x_8, x_9)$
    $T_{\text{sim}}^{-1}(x_8, x_9) \leftarrow (x_0, x_1)$
  return $T_{\text{sim}}^{-1}(x_8, x_9)$

public procedure F(i, x)
  if $x \in F_i$ **then return** $F_i(x)$
  Assert(¬IsPending(i, x_i))
  return NewTree(i, x)

private procedure NewTree(i, x_i)
  root ← **new Node**(**null**, **null**)
  root.*end* ← (i, x_i)
  N.**add**(*root*)
  GrowTree(*root*)
  SampleTree(*root*)
  SetToPrep(*root*) // G_3
  PrepareTree(*root*)
  CheckBadA(*root*) // G_3
  AdaptTree(*root*)
  return $F_i(x_i)$

private procedure ReadTape(i, x_i)
  Assert($x_i \notin F_i$ **and** ¬IsPending(i, x_i))
  CheckBadR(i, x_i) // G_3
  $F_i(x_i) \leftarrow f_i(x_i)$
  return $F_i(x_i)$

private procedure GrowTree(*root*)
  do *modified* ← GrowTreeOnce(*root*)
  while *modified*

private procedure GrowTreeOnce(*node*)
  modified ← FindNewChildren(*node*)
  forall *c* **in** *node*.*children* **do**
    modified ← *modified* **or** GrowTreeOnce(c)
  return *modified*

private procedure FindNewChildren(*node*)
  (i, x) ← *node*.*end*
  added ← **false**
  if $i = 1$ **then forall** (x_7, x_8) **in** (F_7, F_8) **do**
    added ← *added* **or** Trigger($7, x_7, x_8, x$, *node*)
  if $i = 8$ **then forall** (x_1, x_2) **in** (F_1, F_2) **do**
    added ← *added* **or** Trigger($1, x_1, x_2, x$, *node*)
  if $i = 2$ **then forall** (x_8, x_1) **in** (F_8, F_1) **do**
    added ← *added* **or** Trigger($1, x_1, x, x_8$, *node*)
  if $i = 7$ **then forall** (x_8, x_1) **in** (F_8, F_1) **do**
    added ← *added* **or** Trigger($7, x, x_8, x_1$, *node*)
  if $i = 3$ **then forall** (x_4, x_5) **in** (F_4, F_5) **do**
    added ← *added* **or** Trigger($3, x, x_4, x_5$, *node*)
  if $i = 6$ **then forall** (x_4, x_5) **in** (F_4, F_5) **do**
    added ← *added* **or** Trigger($4, x_4, x_5, x$, *node*)
  if $i = 4$ **then forall** (x_5, x_6) **in** (F_5, F_6) **do**
    added ← *added* **or** Trigger($4, x, x_5, x_6$, *node*)
  if $i = 5$ **then forall** (x_3, x_4) **in** (F_3, F_4) **do**
    added ← *added* **or** Trigger($3, x_3, x_4, x$, *node*)
  return *added*

private procedure Trigger(i, x_i, x_{i+1}, u, *node*)
  if $i = 7$ **then**
    if ¬CheckP$^+$(x_i, x_{i+1}, u) **then return false**
  else if $i = 1$ **then**
    if ¬CheckP$^-$(x_i, x_{i+1}, u) **then return false**
  else // $i = 3, 4$
    if $F_{i+1}(x_{i+1}) \neq x_i \oplus u$ **then return false**
  if Equivalent(*node*.*id*, (i, x_i, x_{i+1})) **or**
   InChildren(*node*, (i, x_i, x_{i+1})) **then return false**
  if $i \in \{1, 7\}$ **then** Assert(++$NumOuter \leq q$)
  new_child ← **new Node**(*node*, (i, x_i, x_{i+1}))
  node.*children*.**add**(*new_child*)
  MakeNodeReady(*new_child*)
  return true

private procedure IsPending(i, x_i)
  forall n **in** N **do**
    if $(i, x_i) = n$.*end* **then return true**
  return false

Fig. 4. The simulator

```
private procedure CheckP+(x_7, x_8, x_1)
    x_9 ← x_7 ⊕ F_8(x_8)
    if (x_8, x_9) ∉ T^{-1} then return false        // G_2, G_3, G_4
    (x'_0, x'_1) ← SimP^{-1}(x_8, x_9)
    return x'_1 = x_1

private procedure CheckP−(x_1, x_2, x_8)
    x_0 ← x_2 ⊕ F_1(x_1)
    if (x_0, x_1) ∉ T then return false        // G_2, G_3, G_4
    (x'_8, x'_9) ← SimP(x_0, x_1)
    return x'_8 = x_8

private procedure Equivalent(C_1, C_2)
    if C_1 = null then return false
    (i, x_i, x_{i+1}), (j, x'_j, x'_{j+1}) ← C_1, C_2
    if i = j then return x_i = x'_j and x_{i+1} = x'_{j+1}
    if (i, j) ∈ {(7, 4), (1, 7), (3, 1), (4, 3)} then
        return x'_j = Val−(C_1, j) and
            x'_{j+1} = Val−(C_1, j + 1)
    if (i, j) ∈ {(4, 7), (7, 1), (1, 3), (3, 4)} then
        return x'_j = Val+(C_1, j) and
            x'_{j+1} = Val+(C_1, j + 1)

private procedure InChildren(node, C)
    forall n in node.children do
        if Equivalent(n.id, C) then return true
    return false

private procedure MakeNodeReady(node)
    (i, x) ← node.beginning
    (j, u_1, u_2) ← node.id
    if i ∈ {1, 2, 5, 6} then
        while j ≠ Terminal(i) do
            (u_1, u_2) ← Prev(j, u_1, u_2)
            j ← j − 1 mod 9
        x_j ← u_1
    else
        while j + 1 ≠ Terminal(i) do
            (u_1, u_2) ← Next(j, u_1, u_2)
            j ← j + 1 mod 9
        j ← j + 1
        x_j ← u_2
    Assert(x_j ∉ F_j)
    Assert(¬IsPending(j, x_j))
    node.end ← (j, x_j)
    N.add(node)

private procedure Terminal(i)
    if i = 1 then return 4
    if i = 2 then return 7
    if i = 3 then return 6
    if i = 4 then return 1
    if i = 5 then return 8
    if i = 6 then return 3
    if i = 7 then return 2
    if i = 8 then return 5
```

```
private procedure Next(i, x_i, x_{i+1})
    if i = 8 then
        (x_0, x_1) ← SimP^{-1}(x_i, x_{i+1})
        return (x_0, x_1)
    else
        x_{i+2} = x_i ⊕ F(i + 1, x_{i+1})
        return (x_{i+1}, x_{i+2})

private procedure Prev(i, x_i, x_{i+1})
    if i = 0 then
        (x_8, x_9) ← SimP(x_i, x_{i+1})
        return (x_8, x_9)
    else
        x_{i−1} = F(i, x_i) ⊕ x_{i+1}
        return (x_{i−1}, x_i)

private procedure SampleTree(node)
    N.delete(node)
    ReadTape(node.end)
    forall c in node.children do
        SampleTree(c)

private procedure PrepareTree(node)
    (i, x_i) ← node.end
    if i ∈ {2, 7} and node.id ≠ null then
        ReadTape(3, Val+(node.id, 3))
        ReadTape(6, Val−(node.id, 6))
    if i ∈ {3, 6} and node.id ≠ null then
        ReadTape(7, Val+(node.id, 7))
        ReadTape(8, Val+(node.id, 8))
        SimP^{-1}(Val+(node.id, 8), Val+(node.id, 9))
    forall c in node.children do
        PrepareTree(c)

private procedure AdaptTree(root)
    forall c in root.children do
        AdaptNode(c)

private procedure AdaptNode(node)
    (i, x_i), C ← node.beginning, node.id
    (m, n) ← AdaptPositions(i)
    x_{m−1}, x_m ← Val+(C, m − 1), Val+(C, m)
    x_n, x_{n+1} ← Val−(C, n), Val−(C, n + 1)
    Adapt(m, x_m, x_{m−1} ⊕ x_n)
    Adapt(n, x_n, x_m ⊕ x_{n+1})
    forall c in node.children do
        AdaptNode(c)

private procedure Adapt(i, x_i, y_i)
    Assert(x_i ∉ F_i and ¬IsPending(i, x_i))
    F_i(x_i) ← y_i

private procedure AdaptPositions(i)
    if i ∈ {1, 4} then return (2, 3)
    if i ∈ {5, 8} then return (6, 7)
    if i ∈ {2, 7} then return (4, 5)
    if i ∈ {3, 6} then return (1, 2)
```

Fig. 5. The simulator (continued)

private procedure $\mathrm{Val}^+(i, x_i, x_{i+1}, k)$
  if $k \in \{i, i+1\}$ **then return** x_k
  $j \leftarrow i + 1$
  $U, U^{-1} \leftarrow T_{\mathrm{sim}}, T_{\mathrm{sim}}^{-1}$
  $U, U^{-1} \leftarrow T, T^{-1}$ // $\mathrm{G}_2, \mathrm{G}_3, \mathrm{G}_4$
  while $j \neq k$ **do**
    if $j < 9$ **then**
      if $x_j \notin F_j$ **then return** $\perp$
      $x_{j+1} \leftarrow x_{j-1} \oplus F_j(x_j)$
      $j \leftarrow j + 1$
    else
      if $(x_8, x_9) \notin U^{-1}$ **then return** $\perp$
      $(x_0, x_1) \leftarrow U^{-1}(x_8, x_9)$
      if $k = 0$ **then return** x_0
      $j \leftarrow 1$
  return x_k

private procedure $\mathrm{Val}^-(i, x_i, x_{i+1}, k)$
  if $k \in \{i, i+1\}$ **then return** x_k
  $j \leftarrow i$
  $U, U^{-1} \leftarrow T_{\mathrm{sim}}, T_{\mathrm{sim}}^{-1}$
  $U, U^{-1} \leftarrow T, T^{-1}$ // $\mathrm{G}_2, \mathrm{G}_3, \mathrm{G}_4$
  while $j \neq k$ **do**
    if $j > 0$ **then**
      if $x_j \notin F_j$ **return** $\perp$
      $x_{j-1} \leftarrow F_j(x_j) \oplus x_{j+1}$
      $j \leftarrow j - 1$
    else
      if $(x_0, x_1) \notin U$ **then return** $\perp$
      $(x_8, x_9) \leftarrow U(x_0, x_1)$
      if $k = 9$ **then return** x_9
      $j \leftarrow 8$
  return x_k

Fig. 6. The simulator (continued)

$\mathrm{G}_1, \mathrm{G}_2, \mathrm{G}_3$:
Random permutation tape: p

public procedure $\mathrm{P}(x_0, x_1)$
  if $(x_0, x_1) \notin T$ **then**
    $(x_8, x_9) \leftarrow p(x_0, x_1)$
    CheckBadP($(8, x_8)$) // G_3
    $T(x_0, x_1) \leftarrow (x_8, x_9)$
    $T^{-1}(x_8, x_9) \leftarrow (x_0, x_1)$
  return $T(x_0, x_1)$

public procedure $\mathrm{P}^{-1}(x_8, x_9)$
  if $(x_8, x_9) \notin T^{-1}$ **then**
    $(x_0, x_1) \leftarrow p^{-1}(x_8, x_9)$
    CheckBadP($(1, x_1)$) // G_3
    $T(x_0, x_1) \leftarrow (x_8, x_9)$
    $T^{-1}(x_8, x_9) \leftarrow (x_0, x_1)$
  return $T^{-1}(x_8, x_9)$

G_4:

public procedure $\mathrm{P}(x_0, x_1)$
  if $(x_0, x_1) \notin T$ **then**
    for $i \leftarrow 2$ **to** 9 **do**
      $x_i \leftarrow x_{i-2} \oplus f_{i-1}(x_{i-1})$
    $T(x_0, x_1) \leftarrow (x_8, x_9)$
    $T^{-1}(x_8, x_9) \leftarrow (x_0, x_1)$
  return $T(x_0, x_1)$

public procedure $\mathrm{P}^{-1}(x_8, x_9)$
  if $(x_8, x_9) \notin T^{-1}$ **then**
    for $i \leftarrow 7$ **to** 0 **do**
      $x_i \leftarrow x_{i+2} \oplus f_{i+1}(x_{i+1})$
    $T(x_0, x_1) \leftarrow (x_8, x_9)$
    $T^{-1}(x_8, x_9) \leftarrow (x_0, x_1)$
  return $T^{-1}(x_8, x_9)$

Fig. 7. Permutation oracles

G_5:
Variables:
Random tapes: $f_1, \ldots, f_{10}$

public procedure $\mathrm{F}(i, x)$
  return $f_i(x)$

public procedure $\mathrm{P}(x_0, x_1)$
  for $i \leftarrow 2$ **to** 11 **do**
    $x_i \leftarrow x_{i-2} \oplus f_{i-1}(x_{i-1})$
  return (x_{10}, x_{11})

public procedure $\mathrm{P}^{-1}(x_{10}, x_{11})$
  for $i \leftarrow 7$ **to** 0 **do**
    $x_i \leftarrow x_{i+2} \oplus f_{i+1}(x_{i+1})$
  return (x_0, x_1)

Fig. 8. The real world

```
G3:
Variables:
   Sets A, ToPrep, ToAdapt^(4), ToAdapt^(5)

class Adapt
   Query query
   String value, left, right
   constructor Adapt(i, x_i, y_i, l, r)
      self.query ← (i, x_i)
      self.value ← y_i
      self.left ← l        // Left edge
      self.right ← r       // Right edge

private procedure CheckBadP(i, x_i)
   if x_i ∈ F_i then abort

private procedure SetToPrep(node)
   (i, x_i) ← node.end
   if i ∉ {2, 7} then return
   if node.id ≠ null then
      ToPrep.add((3, Val^+(node.id, 3)))
      ToPrep.add((6, Val^-(node.id, 6)))
   forall c in node.children do
      SetToPrep(c)

private procedure CheckBadR(i, x_i)
   CheckEqual(i, f_i(x_i))
   CheckBadlyHit(i, x_i, f_i(x_i))
   CheckRCollide(i, x_i, f_i(x_i))
   if (i, x_i) ∈ ToPrep then ToPrep.delete((i, x_i))

private procedure CheckBadA(root)
   A, ToAdapt^(4), ToAdapt^(5) ← ∅, ∅, ∅
   GetAdapts(root)
   forall a in A do
      (i, x_i), y_i ← a.query, a.value
      CheckBadlyHit(i, x_i, y_i)
      if i ∈ {4, 5} then CheckEqual(i, y_i)
   forall a, b in A × A do
      CheckAPair(a, b)
   CheckAMid(root)

private procedure ActiveQueries(i)
   P ← ∅
   forall n in N do
      (j, x_j) ← n.end
      if j = i then P.add(x_j)
   // For the BadRPrepare event
   forall (j, x_j) in ToPrep do
      if j = i then P.add(x_j)
   return P ∪ F_i

private procedure IsRightActive(i, x_i, x_{i+1})
   if i ≤ 9 then
      return x_{i+1} ∈ ActiveQueries(i + 1)
   else return (x_i, x_{i+1}) ∈ T^{-1}

private procedure IsLeftActive(i, x_i, x_{i+1})
   if i ≥ 1 then
      return x_i ∈ ActiveQueries(i)
   else return (x_i, x_{i+1}) ∈ T

private procedure IsIncident(i, x_i)
   if i ≥ 2 then j ← i − 2 else j ← 10
   forall u_j, u_{j+1} in {0,1}^n × {0,1}^n do
      if IsLeftActive(j, u_j, u_{j+1}) and
         Val^+(j, u_j, u_{j+1}, i) = x_i then return true
   if i ≤ 9 then j ← i + 1 else j ← 0
   forall u_j, u_{j+1} in {0,1}^n × {0,1}^n do
      if IsRightActive(j, u_j, u_{j+1}) and
         Val^-(j, u_j, u_{j+1}, i) = x_i then return true
   return false

private procedure CheckEqual(i, y_i)
   forall x_i in F_i do
      if F_i(x_i) = y_i then abort

private procedure CheckBadlyHit(i, x_i, y_i)
   forall x_{i−1} in {0,1}^n do
      x_{i+1} ← x_{i−1} ⊕ y_i
      if IsRightActive(i, x_i, x_{i+1}) and
         IsLeftActive(i − 1, x_{i−1}, x_i) then abort

private procedure CheckRCollide(i, x_i, y_i)
   forall x_{i−1} in {0,1}^n do
      if IsLeftActive(i − 1, x_{i−1}, x_i) and
         IsIncident(i + 1, x_{i−1} ⊕ y_i) then abort
   forall x_{i+1} in {0,1}^n do
      if IsRightActive(i, x_i, x_{i+1}) and
         IsIncident(i − 1, y_i ⊕ x_{i+1}) then abort

private procedure CheckAPair(a, b)
   (i, x_i), y_i ← a.query, a.value
   (j, u_j), v_j ← b.query, b.value
   if i = j and i ∈ {4, 5} then
      // Check the second part of BadAEqual
      if x_i ≠ u_j and y_i = v_j then abort
   if j ≠ i + 1 then return
   if a.left ≠ b.left then
      if x_i ⊕ v_j ∈ ActiveQueries(i + 2) then abort
      if IsIncident(i + 2, x_i ⊕ v_j) then abort
   if a.right ≠ b.right then
      if y_i ⊕ u_j ∈ ActiveQueries(i − 1) then abort
      if IsIncident(i − 1, y_i ⊕ u_j) then abort
```

Fig. 9. The abort-checking procedures in G_3

private procedure CheckAMid($root$)
  $(i, x_i) \leftarrow root.end$
  if $i \notin \{4, 5\}$ **then return**
  $S \leftarrow \emptyset$
  forall $x_4, x_5 \in \text{ActiveQueries}(4), F_5$ **do**
    S.**add**$(x_4 \oplus F_5(x_5))$
  forall $x_4, x_5 \in \mathit{ToAdapt}^{(4)}, F_5$ **do**
    $x_6 \leftarrow x_4 \oplus F_5(x_5)$
    if $x_6 \notin F_6$ **and** $x_6 \in S$ **then abort**
    if S.**add**(x_6)
  forall $x_4, x_5 \in \text{ActiveQueries}(4) \cup \mathit{ToAdapt}^{(4)}$, $\mathit{ToAdapt}^{(5)}$ **do**
    $x_6 \leftarrow x_4 \oplus \text{GetAdaptVal}(x_5)$
    if $x_6 \notin F_6$ **and** $x_6 \in S$ **then abort**
    S.**add**(x_6)

private procedure GetAdapts($node$)
  if $node.id \neq$ **null**
    $(i, x_i), (j, x_j) \leftarrow node.beginning, node.end$
    $C \leftarrow node.id$
    $m, n \leftarrow \text{AdaptPositions}(i)$
    $x_{m-1}, x_m \leftarrow \text{Val}^+(C, m-1), \text{Val}^+(C, m)$
    $x_n, x_{n+1} \leftarrow \text{Val}^-(C, n), \text{Val}^-(C, n+1)$
    $y_m, y_n \leftarrow x_{m-1} \oplus x_n, x_m \oplus x_{n+1}$
    $\mathcal{A}$.**add**(**new Adapt**$(m, x_m, y_m, x_{m-1}, x_{n+1})$)
    $\mathcal{A}$.**add**(**new Adapt**$(n, x_n, y_n, x_{m-1}, x_{n+1})$)
    if $m = 4$ **then**
      $\mathit{ToAdapt}^{(4)}$.**add**(x_m), $\mathit{ToAdapt}^{(5)}$.**add**(x_n)
  forall c **in** $node.children$ **do**
    GetAdapts(c)

private procedure GetAdaptVal(i, x_i)
  forall a **in** $\mathcal{A}$ **do**
    if $(i, x_i) = a.query$ **then return** $a.value$

Fig. 10. The abort-checking procedures in G_3 (continued)