Indifferentiability of 10-Round Feistel Networks

Yuanxi Dai and John Steinberger

shusdtc@gmail.com, jpsteinb@gmail.com

Abstract. We prove that a (balanced) 10-round Feistel network is indifferentiable from a random permutation. In a previous seminal result, Holenstein et al. [17] had established indifferentiability of Feistel at 14 rounds. Our simulator achieves security $O(q^8/2^n)$, runtime $O(q^4)$ and query complexity $O(q^4)$, to be compared with security $O(q^{10}/2^n)$, runtime $O(q^4)$ and query complexity $O(q^4)$ for the 14-round simulator of Holenstein et al.

Our simulator is very similar to a 10-round simulator of Seurin [29] that was subsequently found to be flawed [17, 30]. Indeed, our simulator is essentially obtained by switching from a "FIFO" path completion order to a "LIFO" path completion order in Seurin's simulator. This apparently minor change results in a significant paradigm shift, including a conceptually simpler proof.

Table of Contents

1 Introduction . . . 2
2 Definitions and Main Result . . . 5
3 High-Level Simulator Overview . . . 7
4 Technical Simulator Overview and Pseudocode Description . . . 13
5 Proof of Indifferentiability . . . 19
5.1 Efficiency of the Simulator . . . 20
5.2 Transition from G_1 to G_2 . . . 33
5.3 Transition from G_2 to G_3 . . . 35
5.4 Bounding the Abort Probability in G_3 . . . 35
5.5 Transition from G_3 to G_4 . . . 55
5.6 Transition from G_4 to G_5 . . . 62
5.7 Concluding the Indifferentiability . . . 62
6 Pseudocode . . . 65

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 [33], Fiat and Shamir [16] (see proof in [26]) 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. [21], 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, 24, 25].) Coron et al. [7] are early adopters of the framework, and give additional insights.

Informally, given ideal primitives $\mathcal{Z}$ and $\mathcal{P}$, a construction $C^{\mathcal{P}}$ (where C is some stateless algorithm making queries to $\mathcal{P}$) is *indifferentiable* from $\mathcal{Z}$ if there exists a simulator S (a stateful, randomized algorithm) with oracle access to $\mathcal{Z}$ such that the pair $(C^{\mathcal{P}}, \mathcal{P})$ is statistically indistinguishable from the pair $(\mathcal{Z}, S^{\mathcal{Z}})$. 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 result.

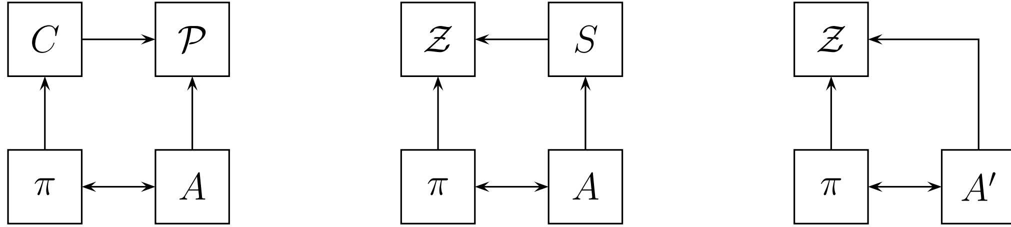

Fig. 1. The cliff notes of indifferentiability, after [7]: (left) adversary A interacts in a game with protocol π in which π calls a construction C that calls an ideal primitive $\mathcal{P}$ and in which A calls $\mathcal{P}$ directly; (middle) by indifferentiability, the pair $(C^{\mathcal{P}}, \mathcal{P})$ can be replaced with the pair $(\mathcal{Z}, S^{\mathcal{Z}})$, where $\mathcal{Z}$ 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 $C^{\mathcal{P}}$ has been replaced by the "ideal world" functionality $\mathcal{Z}$. Hence, a lack of attacks in the ideal world implies a lack of attacks in the real world.

For example, Coron et al. [7] already showed that random oracles can be instantiated from ideal ciphers in this sense, using constructions that are similar to the Merkle-Dåmgard iteration of a blockcipher in Davies-Meyer mode. Naturally, one would also like to consider the reverse direction, i.e., the implementation of an ideal cipher from a random function. This direction has revealed itself to be more challenging.

The natural candidate for a construction (and echoing the construction of pseudorandom permutations from pseudorandom functions by Luby and Rackoff [20]) is a (balanced, keyed) *Feistel network*. (Since all our Feistel networks are going to be balanced, we will subsequently drop this qualifier.) An unkeyed Feistel network is shown in Fig. 2. If each wire carries n bits, then the network implements a $2n$-bit to $2n$-bit permutation for any choice of the round functions $F_i : \{0,1\}^n \rightarrow \{0,1\}^n$. "Keyeing" the network means extending the domain of each F_i from n bits to $n + \kappa$ bits, where κ is the length of the key; if the F_i's are random, one then recovers 2^κ independent (though not necessarily "random") permutations from $2n$ to $2n$ bits, i.e., one recovers a blockcipher of key length κ and of word length $2n$ in which each key behaves independently from every other key.

Due to the trick of domain extension just mentioned, it suffices[1] to show that an unkeyed r-round Feistel network with random round functions is indifferentiable from a random permutation in order to implement a blockcipher from a random function. (The r independent round functions from n bits to n bits can be implemented from a single random function with slightly larger domain.) Consequently, much work has focused on the previous question.

The first paper directly in this line [9] shows that an r-round Feistel network cannot be indifferentiable from a random permutation for $r \leq 5$. The same paper also gives a proof that indifferentiability is achieved at $r = 6$. The second result, however, was found to have a serious

[1] A fact that authors tend to gloss over, however, is that there is an additional security loss—specifically, a factor q where q is the number of distinguisher queries—in going from the unkeyed to the keyed setting. Moreover this loss seems intrinsic: consider the case of a family of functions $F_1, \ldots, F_r : \{0,1\}^{n+\kappa} \rightarrow \{0,1\}^n$ for which there exists a (random) subset $T \subseteq \{0,1\}^\kappa$ of size $2^\kappa/1000$ such that for all $k \in T$, the functions $F_1(\cdot, k), \ldots, F_r(\cdot, k) : \{0,1\}^n \rightarrow \{0,1\}^n$ are identically zero, while for all $k \notin T$, the same functions are uniformly random and independent. Then if $r \geq 10$ (by the current paper's result), the r-round Feistel network permutation induced by selecting and fixing a random key is indifferentiable from a random permutation with high probability (specifically, probability 999/1000), while the keyed construction can be trivially differentiated from an ideal cipher in about 1000 queries.

flaw by Holenstein et al. [17], and who could indeed not repair the simulator at six rounds; instead, Holenstein et al. could only show that a 14-round Feistel network is indifferentiable from a random permutation. Holenstein et al. also found a flaw in the proof of indifferentiability for a 10-round simulator of Seurin [29] (a simplified alternative to the 6-round simulator of [9]) and Seurin himself subsequently found a clever attack against his own simulator, thus showing that the proof could not be patched [30]. On the other hand, it should be noted that the 14-round simulator of Holenstein et al. follows design principles that are very similar to Seurin's 10-round simulator; indeed, the 14-round simulator essentially consists of Seurin's 10-round simulator with four extra "buffer rounds" flanking the simulator's two "adapt zones".

Our purpose in this work is to recommence the "downward journey" in the number of rounds of Feistel that are known to be sufficient to achieve indifferentiability from a random permutation. Specifically, we show that 10 rounds suffice, while achieving basically the same security and query complexity as the 14-round simulator of Holenstein et al.[2] Our simulator is also closely based on Seurin's 10-round simulator, with the minor (but technically significant) difference that our simulator completes paths in LIFO fashion rather than in FIFO fashion: the last path detected is given priority for completion over paths that are already under completion. In particular, this change happens to circumvent Seurin's 10-round attack.

We note that the change from FIFO to LIFO path completion has deep structural repercussions for our proof, which ends up looking quite different from the indifferentiability proof of Holenstein et al. [17]. In our case, in particular, the concurrent completion of several paths unfolds in a highly structured manner that makes it easy to maintain a complete picture of the state of partially completed paths at any point in time. The soundness of our simulator ends up being quite a bit more intuitive than the simulator of [17].

Concurrent Work. Katz et al. [18] have also announced a 10-round indifferentiability result; the two teams have been in communication in order to coordinate the release of results, but as of going to print neither team has seen the technical details of the other's work.

We also expect the techniques of this paper to yield further improvements in security and round complexity for Feistel simulators. This is ongoing work.

Other Related Work. Even before [9] Dodis and Puniya [10] 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, however, [1] and [19] investigate the indifferentiability of key-alternating ciphers (with and without an idealized key scheduler, respectively). In a recent eprint note, Dodis et al. [11] investigate the indifferentiability of substitution-permutation networks, treating the S-boxes as independent idealized permutations; as we will mention later, our simulator is also partly inspired by theirs.

It should be noted 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

[2] Our query complexity is in fact quadratically lower, but this is because we apply an easy optimization (taken from [11]) that could equally well be applied to the simulator from [17].

to which the simulator belongs (the latter class being typically "probabilistic polynomial-time"). This limitation was first noted by Ristenpart et al. [27], who moreover give a interesting example where indifferentiability serves no use.

Finally, Feistel networks have been the subject of a very large body of work in the secret-key (or "indistinguishability") setting, such as in [20, 22, 23, 28] 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 intuitive overview of our simulator, focusing on high-level design principles. 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 of the proof.

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 $1 \leq i \leq r$. It is easy to see that the application

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

is a permutation, since x_{i-1} can be computed from x_i and x_{i+1} for $1 \leq i \leq r-1$. We denote the resulting permutation of $\{0,1\}^{2n}$ as

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

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

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 implies 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. For example, $\Psi[F_1, \ldots, F_r]$ 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 $\mathcal{P}$ as $C^{\mathcal{P}}$.

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(s) $\mathcal{Z}$ is notated as $S^{\mathcal{Z}}$.

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 $C^{\mathcal{P}}$, where $\mathcal{P}$ is a set of ideal primitives, is "as good as" an ideal primitive $\mathcal{Z}$ that has the same syntax (interface) as $C^{\mathcal{P}}$. In brief, there must exist a simulator S such that having oracle access to the pair $(C^{\mathcal{P}}, \mathcal{P})$ (often referred to as the "real world") is indistinguishable from the pair $(\mathcal{Z}, S^{\mathcal{Z}})$ (often referred to as the "simulated world").

In more detail we refer to the following definition, which is due to Maurer et al. [21].

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

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

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 $\mathcal{Z}$. Here t_S, q_S and ε are functions of q, and the probabilities are taken over the randomness in $\mathcal{P}$, $\mathcal{Z}$, S and (if any) in D.

As indicated, we allow S to depend on q.[3] The notation

$$D^{C^{\mathcal{P}},\mathcal{P}}$$

indicates that D has oracle access to $C^{\mathcal{P}}$ as well as to each of the primitives in the set $\mathcal{P}$. We also note that the oracle

$$S^{\mathcal{Z}}$$

offers one interface for D to query for each of the primitives in $\mathcal{P}$; however the simulator S is "monolithic" and treats each of these queries with knowledge of the others.

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

Informally, $C^{\mathcal{P}}$ is *indifferentiable* from $\mathcal{Z}$ 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 10-round Feistel network and $\mathcal{P}$ will be the set $\{F_1, \ldots, F_{10}\}$ of round functions, with each round function being an independent random function. Consequently, $\mathcal{Z}$ (matching $C^{\mathcal{P}}$'s syntax) will be a random permutation from $\{0,1\}^{2n}$ to $\{0,1\}^{2n}$, queriable (like $C^{\mathcal{P}}$) 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 $\{F_1, \ldots, F_{10}\}$ (where each F_i is an independent random function) plays the role of $\mathcal{P}$, the set of ideal primitives called by C.

[3] 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 [13]—for which the simulator crucially needs to know the number of distinguisher queries in advance.

Theorem 1. *The Feistel network $\Psi[F_1, \ldots, F_{10}]$ is (t_S, q_S, ε)-indifferentiable from a random $2n$-bit to $2n$-bit permutation with $t_S = O(q^4)$, $q_S = 16q^4$ and $\varepsilon = 6023424q^8/2^n$. Moreover, these bounds hold even if the distinguisher is allowed to make q queries to* each *of its 11 $(= 10 + 1)$ 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 34, 31 and 84 of sections 5.1, 5.1 and 5.7 respectively.

Miscellaneous Notations. Our pseudocode uses standard conventions from object-oriented programming, including consutrctors 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 uninitialized 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 $T(x) = \perp$.

3 High-Level Simulator Overview

In this section we try to convey the "design philosophy" of our simulator which, like [17], is a modification of a 10-round simulator by Seurin [29].

Round Function Tables. We recall that the simulator is responsible for 10 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. 3 and onwards), and which takes two inputs: an integer $i \in [10]$ and an input $x \in \{0,1\}^n$.

Correspondingly to these 10 interfaces, the simulator maintains 10 tables, notated $F_1, \ldots, F_{10}$, whose fields are initialized to $\perp$: initially, $F_i(x) = \perp$ for all $x \in \{0,1\}^n$, all $i \in [10]$. (Hence we note that F_i is no longer the name of a round function, but the name of a table, which should not cause confusion. The i-th round function is now $\mathrm{F}(i, \cdot)$.) 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 return y. Moreover, entries in the tables $F_1, \ldots, F_{10}$ are never 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. 6), and which plays the role of the ideal primitive $\mathcal{Z}$ 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 $(x_{10}, x_{11}) \in \{0,1\}^n \times \{0,1\}^n$. 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 $\mathrm{F}(1, \cdot)$, ..., $\mathrm{F}(10, \cdot)$ 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 $x_2, \ldots, x_{11}$ by setting

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

for $i = 1, \ldots, 10$, and finally check if $(x_{10}, x_{11}) = \mathrm{P}(x_0, x_1)$. (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

$$x_1, \ldots, x_{10}$$

queried by the distinguisher form a *completed path*. (The definition of a "completed path" will be made much more precise in the next section. The terminology that we use in this section is hand-wavy through and through.)

Moreover, the distinguisher need not complete paths in left-to-right fashion: it might choose, e.g., values x_4 and x_5 at random, and build a completed path outwards from those positions ("going left" for x_3, x_2 and x_1, "going right" for $x_6, \ldots, x_{10}$). Or it might, say, choose random values x_0, x_1, make the query

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

and then complete from both ends at once (making some queries on the left starting from x_0, x_1, and some queries on the right starting from x_{10}, x_{11}). Other possibilities can be imagined as well. Moreover, the distinguisher may reuse the same queries for several different paths.

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 chains a distinguisher is trying to complete, and "adapt" the values along chains such as to be 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 chain to one another; thus if, say,

$$x_1, x_2, x_5, x_6, x_7, x_8, x_9, x_{10}$$

are values such that

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

for all $i \in \{1, 2, 5, 6, 7, 8, 9, 10\}$, and such that

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

for $i \in \{6, 7, 8, 9\}$, and such that

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

where $x_0 := F_1(x_1) \oplus x_2$, $x_{11} := x_9 \oplus F_{10}(x_{10})$, and such that

$$F_3(x_3) = F_4(x_4) = \bot$$

where $x_3 := x_1 \oplus F_2(x_2)$, $x_4 := F_5(x_5) \oplus x_6$, then by making the assignments

$$F_3(x_3) \leftarrow x_2 \oplus x_4 \tag{1}$$

$$F_4(x_4) \leftarrow x_3 \oplus x_5 \tag{2}$$

the simulator turns $x_1, \dots, x_{10}$ into a completed path that is compatible with P. In such a case, we say that the simulator *adapts a path*. The values $F_3(x_3)$ and $F_4(x_4)$ 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 at the same position. 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.

Here it should be emphasized that our simulator, like previous simulators [9, 17, 29], 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 starting point is a 10-round simulator of Seurin [29], which nicely handles the problem of simulator termination.

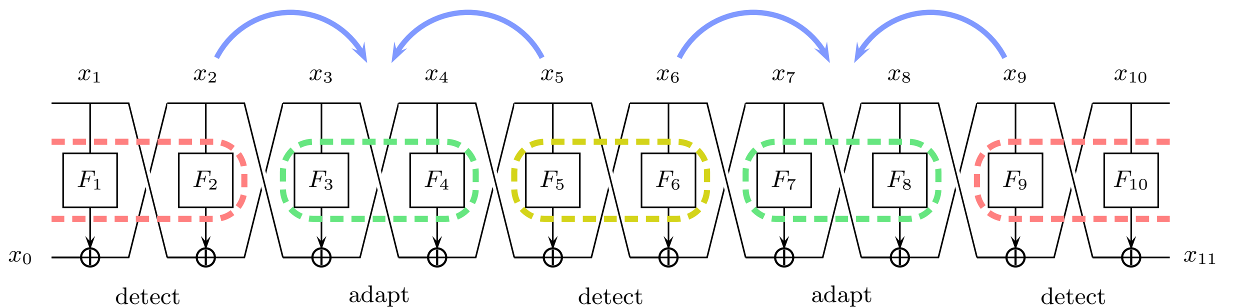

Fig. 2. A sketch of our 10-round simulator (and also Seurin's). Rounds 5 and 6 form one detect zone; rounds 1, 2, 9 and 10 form another detect zone; rounds 3 and 4 constitute the left adapt zone, 7 and 8 constitute the right adapt zone; blue arrows point from the position where a path is detected to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

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 $x_0 := F_1(x_1) \oplus x_2$, $x_{11} := x_9 \oplus F_{10}(x_{10})$.

Paths are adapted either at positions 3, 4 or else at positions 7, 8. (See 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 second type of path completion, 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 second type of path completion with negligibly low 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 (1), (2) 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. Other types of path completions are carried out analogously; for example, if $F_6(x_6) = \perp$ when the query

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

is made, then this query $\mathrm{F}(6, x_6)$ 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; and such a path completion proceeds 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 is (again) that $F_7(x_7^*) = F_8(x_8^*) = \bot$ when it comes time 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[4]) 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. 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 [17, 30].

Changes to Seurin's Simulator. For the following discussion, we will say that the table entries $F_2(x_2)$, $F_5(x_5)$ constitute the *endpoints* of a completed path $x_1, \ldots, x_{10}$ that is adapted at positions 3, 4; likewise, the table entries $F_6(x_6)$, $F_9(x_9)$ constitute the *endpoints* of a completed path $x_1, \ldots, x_{10}$ that is adapted at positions 7, 8. Hence, the endpoints are the two table entries that "flank" the adapted entries. Succinctly, our simulator's philosophy is to not sample the endpoints of a completed 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

$$F_2(x_2) = \bot$$

at the moment when the call $\mathrm{F}(2, x_2)$ is made. Instead of immediately lazy 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 the path is ready to be adapted. (Technically, "ready to be adapted" means, for a path that is adapted in positions 3 and 4, that both of the values x_2 and x_5 are known; for a path adapted in positions 7 and 8, that both of the values x_6, x_9 are known.) Moreover, and keeping the notations from the previous example, note that the query

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

will itself become a "pending query" at position 6 as long as there is at least one value x_5^* such that $F_5(x_5^*) \neq \bot$, 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 the corresponding path is ready to be 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. (Of course, one must show that cyclic dependencies don't arise except with negligible probability; this is done in the proof.)

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

Next, to describe the "batch adaptation" of paths, say (for now) that a table entry

$$F_j(x_j)$$

for $j \in \{2,5,6,9\}$ is a *pending query* if $F_j(x_j) = \bot$ and the call $\mathrm{F}(j, x_j)$ has been made and has triggered at least one path completion. Moreover, say that two pending queries are *linked* if they are the two endpoints of the same yet-to-be-completed path (in which case we say that the path is "ready to be adapted", as described above). Moreover a pending query $F_j(x_j)$ is *stable* if it does not currently[5] trigger a path completion. (Paths that are already ready to be adapted do not count as potential triggers.)

Pending queries can be represented by a graph, with a node for each pending query and an edge for each linked pair of pending queries. Thus, each edge corresponds to a path that is ready to be adapted and vice-versa. If we say that a vertex corresponding to pending query $F_j(x_j)$ is in "shore j", then edges only exist between vertices in shores 2 and 5 on the one hand and between vertices in shores 6 and 9 on the other hand. Moreover, one can show that, with high probability, each connected component is a tree. We say that a tree is *stable* if all its nodes are stable.

To picture the evolution of this graph over time, when the distinguisher initially makes a query $\mathrm{F}(j, x_j)$ the graph is empty, because there are no pending queries. If $j \notin \{2,5,6,9\}$ then the simulator lazy samples the value $F_j(x_j)$ if does not already exist, and simply returns $F_j(x_j)$ to the distinguisher. Otherwise, right after the query is made, the graph contains at most one node, namely the pending query $F_j(x_j)$, which has triggered one or more path completions if the graph is nonempty. From there, the simulator "grows"[6] a tree containing this node; the tree spans shores 2, 5 if $j \in \{2,5\}$ and spans shores 6, 9 if $j \in \{6,9\}$. At any point the growth of a tree spanning shores 2, 5 may be interrupted by the apparition of a tree spanning shores 6 and 9, and vice-versa. Hence a "stack of trees" (alternating between trees of shore 2, 5 and trees of shore 6, 9) is created, where only the last (topmost) tree on the stack is being grown at any time. The topmost tree is also the only tree that potentially becomes *stable*, with trees lower down in the stack being unstable by virtue of still being under construction. The proof also shows that new trees do not "collide" with older trees as they grow.

If and when the topmost tree on the stack becomes stable, the simulator adapts the paths corresponding to edges in this tree all at once. This happens in two stages: first the simulator lazy samples the values of all pending queries (a.k.a. nodes) in the tree; then, for each path (a.k.a. edge) the simulator adapts last two queries on the path as in (1), (2) (or using the analogous equations for F_7, F_8). This two-step process is what we refer to as "batch adaptation". After the tree has been adapted, it disappears and the simulator resumes work on the next topmost tree in the stack.

Structural vs. Conceptual Changes. Of the two main changes to Seurin's simulator just described it should be noted that the first (i.e., LIFO path completion) is crucial to the correctness of our simulator, whereas the second (i.e., batch adaptations) is only a conceptual convenience, not necessary for correctness. Indeed, one way or another every non-null value

$$F_j(x_j)$$

[5] Here "currently" emphasizes that the query does not trigger a path completion *with respect to the current contents of the tables* $\{F_i\}_{i=1}^{10}$ as opposed to with respect to the "old" contents of the same tables at the point in time when the query $\mathrm{F}(j, x_j)$ was originally made.

[6] If this seems nebulous, consider that a distinguisher can make, for example, an arbitrary set of queries to the functions $\mathrm{F}(1,\cdot)$, $\mathrm{F}(6,\cdot)$, $\mathrm{F}(7,\cdot)$, $\ldots$, $\mathrm{F}(10,\cdot)$ without triggering any path completions. Then a query to $\mathrm{F}(2,\cdot)$ or to $\mathrm{F}(5,\cdot)$ may cause a large chain reaction of path completions.

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 made, as in Seurin's original simulator. This approach ends up being correct, but is conceptually less convenient, since 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 [11], and which indeed formed part of the inspiration for our own work. In [11], however, batch adaptations are purely made for conceptual convenience.

Finally, readers seeking even more concrete insights can consult Seurin's attack against his own 10-round simulator [30] and check this attack fails when the simulator is switched to LIFO path completion.

The Termination Argument. For completeness, we also briefly reproduce Seurin's (by now classic) termination argument.

The basic idea is that each path of the second type—that is, paths detected at position 2 or 9—is associated to a previously existing P-query, and one can show that this P-query is, with high probability, first made by the distinguisher. Since the distinguisher only has q queries total, this already implies that the number of path completions of the second type is at most q.

Secondly, path completions of the first type do not actually add any entries to either of the tables F_5 or F_6. Hence, only two mechanisms add entries to the tables F_5 and F_6: queries directly made by the distinguisher and path completions of the second type. Each of these accounts for at most q table entries, so that the tables F_5, F_6 do not exceed size $2q$. This implies that the number of path completions of the first type is at most $(2q)^2$ and the total number of all path completions is at most $4q^2 + q$.

4 Technical Simulator Overview and Pseudocode Description

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. 3–5, and more specifically by the pseudocode for game G_1, which is the simulated world. In Fig. 3, in particular, one finds the function F (to be called with an argument $(i, x) \in [10] \times \{0, 1\}^n$), 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. 6, which is also part of game G_1.

Our pseudocode uses *explicit random tapes*, similarly to [17]. On the one hand there are tapes $f_1, ..., f_{10}$ where f_i is a table of 2^n random n-bit values for each $1 \leq i \leq 10$, i.e., $f_i(x)$ is a uniform independent random n-bit value for each $1 \leq i \leq 10$ 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 maitains a table $F_i : \{0, 1\}^n \to \{0, 1\}^n$ for the i-th round function, $1 \leq i \leq 10$. Initially, $F_i(x) = \perp$ for all $1 \leq i \leq 10$ 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 $\mathrm{F}(i, x)$ occurs and $F_i(x) \neq \perp$, the call simply returns $F_i(x)$.

The permutation oracle $\mathrm{P}/\mathrm{P}^{-1}$ 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. 6 for details (briefly, however, the tables T/T^{-1} remember the values on which $\mathrm{P}/\mathrm{P}^{-1}$ have already been called). These tables serve no tangible purpose in G_1, where $\mathrm{P}/\mathrm{P}^{-1}$ 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.

If a call $\mathrm{F}(i, x)$ occurs and $F_i(x) = \perp$ and $i \notin \{2, 5, 6, 9\}$, the simulator sets $F_i(x) \leftarrow f_i(x)$ and returns this value. Otherwise, if $i \in \{2, 5, 6, 9\}$ and if $F_i(x) = \perp$, the pair (i, x) becomes a "pending query", as formally defined 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 $\mathrm{F}(\cdot, \cdot)$ 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 $(i, x) \in [10] \times \{0, 1\}^n$. 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. 3.) Nodes are arranged into *trees*. A node n is the *root* of its tree if and only if $n.parent = \mathbf{null}$. 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., *beginning, end* $\in \{[10] \times \{0, 1\}^n, \mathbf{null}\}$. In fact

$$beginning,\ end \in \{\{2, 5, 6, 9\} \times \{0, 1\}^n, \mathbf{null}\}$$

more precisely.

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

A query (i, x) is *pending* if and only if $F_i(x) = \perp$ and there exists a node n such that $n.end = (i, x)$. In particular, one can observe from the pseudocode that when a call $\mathrm{F}(i, x)$ occurs such that

$F_i(x) = \bot$ and such that $i \in \{2, 5, 6, 9\}$, a call 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.

Intuitively, a query (i, x) is pending if $F_i(x) = \bot$ but the simulator has already decided to assign a value to $F_i(x)$ during that query cycle. A query can only be pending for $i = 2$, 5, 6 or 9, since a pending query is the *end* field of some node (see the remark above about the limited positions at which *beginning* and *end* appear).

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 $a.end, b.end \neq$ **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 $n.beginning = (i, x_i) \neq$ **null**, $n.end = (j, x_j) \neq$ **null** then
$$\{i, j\} \in \{\{2, 5\}, \{6, 9\}\}.$$
3. Each tree has at most one non-ready node, i.e., at most one node n with $n.end =$ **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 $F_i(x) = \bot$ 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 $n.beginning$, if $n.beginning \neq$ **null**. The *terminal* of a node n is the position of $n.end$, if $n.end \neq$ **null**. (Thus, as per the second bullet above, if the origin is 2 the terminal is 5 and vice-versa, whereas if the origin is 6 the terminal is 9 and vice-versa.)

A *2chain* is a triple of the form $(i, x_i, x_{i+1}) \in \{0, 1, \ldots, 10\} \times \{0, 1\}^n \times \{0, 1\}^n$. 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. The position of *id* is $i - 1$ if the node has origin $i \in \{2, 6\}$; the position of *id* is i if the node has origin $i \in \{5, 9\}$.

Intuitively, each node that is ready is associated to a path from its origin to its terminal, and the *id* contains the first two queries on the path; indeed the first two queries are enough to uniquely determine the path (provided the relevant table values are present).

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 GrowSubTree (one or more times)

- GrowSubTree calls FindNewChildren (one or more times) and also calls GrowSubTree (zero or more times)
- FindNewChildren calls AddChild (zero or more times)
- AddChild calls MakeNodeReady (one call to MakeNodeReady per call to AddChild)
- 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 can be seen by direct inspection of the pseudocode, 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 procedure FindNewChildren) is now discussed in more detail.

Firstly we must define *equivalence* of 2chains. This definition relies on the functions Val^+, Val^-, which we invite the reader to consult at this point. (See Fig. 4.) Briefly, a 2chain $(1, x_1, x_2)$ is *equivalent* to a 2chain $(5, x_5, x_6)$ if and only if $\mathrm{Val}^-(1, x_1, x_2, j) = x_j$ for $j = 5, 6$ or, equivalently, if and only if $\mathrm{Val}^+(5, x_5, x_6, j) = x_j$ for $j = 1, 2$. A 2chain $(5, x_5, x_6)$ is *equivalent* to a 2chain $(9, x_9, x_{10})$ if and only if $\mathrm{Val}^+(9, x_9, x_{10}, j) = x_j$ for $j = 5, 6$ or, equivalently, if and only if $\mathrm{Val}^-(5, x_5, x_6, j) = x_j$ for $j = 9, 10$. Moreover any 2chain is *equivalent* to itself. (Equivalence is defined in these specific cases only, and, in particular, we do not bother to extend the notion transitively, but which in any case would make no difference.) 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) be a pending query. We will consider four cases according to the value of i. Let n be the node such that $n.end = (i, x)$. (We remind that such a node n exists and is unique; existence follows by definition of *pending*, uniqueness is argued within the proof.)

If $i = 2$, let $x_2 := x$. A value $x_1 \in \{0,1\}^n$ is a *trigger* for $(i, x) = (i, x_2)$ if $F_1(x_1) \neq \perp$, if $F_{10}(x_{10}) \neq \perp$ where $(x_{10}, x_{11}) := \mathrm{P}(x_0, x_1)$ where $x_0 := F_1(x_1) \oplus x_2$, if $F_9(x_9) \neq \perp$ where $x_9 := F_{10}(x_{10}) \oplus x_{11}$, and finally if 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 $i = 9$, let $x_9 := x$. A value $x_{10} \in \{0,1\}^n$ is a *trigger* for $(i, x) = (i, x_9)$ if $F_{10}(x_{10}) \neq \perp$, if $F_1(x_1) \neq \perp$ where $(x_0, x_1) := \mathrm{P}^{-1}(x_{10}, x_{11})$ where $x_{11} := F_{10}(x_{10}) \oplus x_9$, if $F_2(x_2) \neq \perp$ where $x_2 := x_0 \oplus F_1(x_1)$, and finally if the 2chain $(9, x_9, x_{10})$ is not equivalent to $n.id$ and not equivalent to $c.id$ for any existing child c of n.

If $i = 5$, let $x_5 := x$. A value x_6 is a *trigger* for the pending query $(i, x) = (i, x_6)$ if $F_6(x_6) \neq \perp$ and if $(5, x_5, x_6)$ is not equivalent to $n.id$ and not equivalent to $c.id$ for any existing child c of n.

If $i = 6$, let $x_6 := x$. A value x_5 is a *trigger* for the pending query $(i, x) = (i, x_6)$ if $F_5(x_5) \neq \perp$ and if $(5, x_5, x_6)$ is not equivalent to $n.id$ and not equivalent to $c.id$ for any existing child c of n.

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[7] $n.end$. For each

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

trigger y that FindNewChildren identifies, it creates a new child c for n; the *id* of c is set to $(i-1, y, x)$ if $i \in \{2, 6\}$ and to (i, x, y) if $i \in \{5, 9\}$. After creating c, FindNewChildren calls MakeNodeReady(c).

As a subtlety, one should observe that certain values y that are *not* triggers before a call to MakeNodeReady might be triggers *after* the call. However one can also observe that FindNewChildren will in any case be called again on node n by virtue of having returned *child_added* = **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*[8] 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}, $\ldots, x_{11}, x_0, x_1, \ldots, x_j$ 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 $h \notin \{i, j, 0, 11\}$; if $i > j$, then $i \leq 10$, $j \geq 1$, and $T(x_0, x_1) = (x_{10}, x_{11})$; if $i < j$, then $0 \leq i < j \leq 11$.

We notate the partial path as $\{x_h\}_{h=i}^{j}$ regardless of whether $i < j$ or $i > j$, with the understanding that x_{11} 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 ($h \neq 11$) 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 *proper* if $i, j \in [10]$, if $x_i \notin F_i$, $x_j \notin F_j$, and if $(i, j) \in \{(2, 6), (5, 9), (5, 2), (6, 2), (9, 5), (9, 6)\}$ (the latter technical requirement is clarified in the proof, and needn't be scrutinized now).

A *completed path* is a $(0, 11)$-partial path $\{x_h\}_{h=0}^{11}$ such that $T(x_0, x_1) = (x_{10}, x_{11})$.

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 whether the origin of *node* is $i = 2, 5, 6$ or 9. If $i = 2$ then *node.id* $= (1, u_1, u_2)$ for some values u_1, u_2, where $(i, u_2) =$ *node.beginning*. Starting with $j = 1$, MakeNodeReady executes the instructions

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

until $j = 5$. One can note (from the pseudocode of Prev) that after each call of the form Prev(j, u_1, u_2) with $j \neq 0$, $F_j(u_1) \neq \perp$. (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 $x_5, x_6, \ldots, x_1, x_2$ with endpoints $(i, j) = (2, 5)$ and with $(1, x_1, x_2) =$ *node.id*.

We also have $F_2(x_2) = \perp$ by item 4 above and, if MakeNodeReady doesn't abort, $F_5(x_5) = \perp$ as well when MakeNodeReady returns. In particular, $x_5, x_6, \ldots, x_1, x_2$ is a proper $(5, 2)$-partial path when MakeNodeReady returns, containing *node.id*.

For $i = 5$, MakeNodeReady similarly creates a $(5, 2)$-partial path $x_5, x_6, \ldots, x_1, x_2$ such that $(5, x_5, x_6) =$ *node.beginning*, by repeated calls to Next. Here the partial path is also proper when MakeNodeReady returns, and likewise contains *node.id*.

[8] 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.

The cases $i = 6$ and $i = 9$ are symmetric, respectively, to the cases $i = 5$ and $i = 2$.

In summary, when MakeNodeReady(*node*) returns one has $\mathit{node.beginning} \neq \mathbf{null}$, $\mathit{node.end} \neq \mathbf{null}$, and there exists a proper (i, j)-partial path $x_i, x_{i+1}, \dots, x_{11}, x_0, \dots, x_j$ containing *node.id* such that $(i, j) \in \{(5, 2), (9, 6)\}$ and such that

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

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_5 (resp. F_6) are nodes with terminal 6 (resp. 5). Likewise, the only nodes whose stability is potentially affected by a change to the table F_2 (resp. F_9) are nodes with terminal 9 (resp. 2). Given that all the nodes in the tree either have terminals $i \in \{2, 5\}$ or terminals $i \in \{6, 9\}$, 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 higher up in the stack is potentially affected.)

After SampleTree(*root*) returns, AdaptTree(*root*) is called, which "adapts" the partial path associated[9] to each non-root node of the tree into a completed path. In more detail, if the endpoints of the partial paths are 2 and 5 then F_3 and F_4 are adapted (by a call to the procedure 'Adapt') as in equations (1) and (2); if the endpoints of the partial paths are 6 and 9 then F_7 and F_8 are adapted, via similar assignments.

Further Pseudocode Details: The Tables $T_{\mathrm{sim}}/T_{\mathrm{sim}}^{-1}$. In order to reduce its query complexity, and following an idea of [11], 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. 3), 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 [17], 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. 5), 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 $T_{\mathrm{sim}}/T_{\mathrm{sim}}^{-1}$ 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 $\mathrm{P}/\mathrm{P}^{-1}$.

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

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 34, 31 and 84 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. 3–6 while game G_5 is given by the pseudocode of Fig. 7. Every game offers the same interface to the distinguisher, consisting of functions F, P and P^{-1}.

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

In G_2: The simulator's procedure CheckP (Fig. 4) used by the simulator in FindNewChildren (Fig. 3) "peeks" at the table T and returns $\perp$ if $(x_0, x_1) \notin T$; this modification ensures that a call to 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^{-1}_{sim}. (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^{-1}_{sim} may be proper subsets of the tables T, T^{-1} (see Lemma 21). On the other hand, the change to CheckP may result in "false negatives" being returned by CheckP.)

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 Fig. 8.

In G_4: The only difference occurs in the implementation of the oracles P, P^{-1} (see Fig. 6). In G_4, these oracles no longer rely on the random permutation table $p : \{0,1\}^{2n} \to \{0,1\}^{2n}$, but instead evaluate a 10-round Feistel network using the random tapes $f_1, \ldots, f_{10}$ as round functions.

In G_5: This is the real world, meaning that $\mathrm{F}(i, x)$ 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

$$\Delta_D(\mathrm{G}_i, \mathrm{G}_j) = \Pr_{\mathrm{G}_i}[D^{F,P,P^{-1}} = 1] - \Pr_{\mathrm{G}_j}[D^{F,P,P^{-1}} = 1] \tag{3}$$

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 $\Delta_D(\mathrm{G}_1, \mathrm{G}_5)$ 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 34 and 31 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 $\mathrm{F}(i, \cdot)$ for each $i \in [10]$ 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 $F(i, x_i)$ for $i = 1, 2, \ldots, 10$ where $x_i = F(i-1, x_{i-1}) \oplus x_{i-2}$ for $i = 2, 3, \ldots, 10$, 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, 17, 19] 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 [11, 17], 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 $F_i(x_i) \neq \perp$. 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 $x_0, \ldots, x_{11}$ such that $x_{i+1} = x_{i-1} \oplus F_i(x_i)$ for $1 \leq i \leq 10$ and such that $T(x_0, x_1) = (x_{10}, x_{11})$.

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 $F_i(x_i) \neq \perp$. A node n is *completed* if there exists a completed path $x_0, x_1, \ldots, x_{11}$ 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 $r.id = \textbf{null}$ (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 (Fig. 3) and AddChild (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}). Moreover, in games G_1, G_2 and G_3, 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 10-round Feistel network built from the round functions (random tapes) $f_1, \ldots, f_{10}$ 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$(i, x_i) =$ **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 F(i, x_i). Between the time when 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 F(i, x_i) occurs. By Lemmas 5 and 6, when F(i, x_i) is called, we have IsPending$(i, x_i) =$ **true** or $x_i \in F_i$. But F(i, x_i) aborts if IsPending$(i, x_i) =$ **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 IsPending$(i, x_i) =$ **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, both $n.beginning$ and $n'.end$ are not 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. *The end of a node must be in position 2, 5, 6 or 9. Moreover, queries in these positions only become defined in calls to* SampleTree.

Proof. The *end* of a node is only assigned in NewTree and MakeNodeReady. NewTree(i, x_i) is only called by F(i, x_i) for $i \in \{2, 5, 6, 9\}$. When the *end* of a node is assigned a query (j, x_j) in MakeNodeReady, we have $j = \text{Terminal}(i)$, while the output of Terminal is 2, 5, 6 or 9.

A query can be defined a call to procedures F, Adapt, and SampleTree. F calls ReadTape only if $i \notin \{2, 5, 6, 9\}$, and Adapt is only called on queries in positions 3, 4, 7 and 8. Therefore, queries in positions 2, 5, 6 and 9 must be defined (if at all) in SampleTree. □

Lemma 11. *For every node n, the query $n.end$ is not defined until* SampleTree(n) *is called.*

Proof. By Lemma 10, $n.end$ is in position 2, 5, 6 or 9, and must be sampled in a call to SampleTree(n') for some node n'. The query defined in SampleTree(n') is $n'.end$. By Lemma 7, if $n' \neq n$, $n'.end \neq n.end$. Therefore, the query $n.end$ must be defined inside of SampleTree(n). □

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 only need to prove that a node is removed from N if and only if it becomes sampled. A node n is deleted from N only in the procedure SampleTree. ReadTape is called on $n.end$ before the deletion, so the node is sampled when the deletion occurs. Moreover, by Lemma 11, the query $n.end$ can only be defined in SampleTree(n), following which the deletion occurs immediately.

Therefore, the set N always equals the set of nodes that are ready but not sampled, except for the gaps between the two lines when 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 $i \in [10]$.

For any $i \in [10]$, 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 2 or 9, and is an *inner node*if its origin is 5 or 6.

The names imply by which detect zone a path is triggered: an inner node is associated with a path triggered by the middle detect zone; an outer node is associated with a path triggered by the 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}, $\ldots, x_{11}, x_0, x_1, \ldots, x_j$ if $i > j$, satisfying the following properties: $i \neq j$ and $0 \leq i, j \leq 11$; $x_h \in F_h$ and $x_{h-1} \oplus F_h(x_h) = x_{h+1}$ for subscripts h such that $h \notin \{i, j, 0, 11\}$; if $i > j$, we also require $(i, j) \neq (11, 0)$, $T(x_0, x_1) = (x_{10}, x_{11})$ if $1 \leq j < i \leq 10$, $T(x_0, x_1) = (*, x_{11})$ if $i = 11$, and $T^{-1}(x_{10}, x_{11}) = (x_0, *)$ if $j = 0$.

As can be noted, the only difference with the definition given in Section 4 is that the cases $i = 11$ 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

$$h \in \{i, i + 1, \ldots, 11\} \cup \{0, 1, \ldots, j\}$$

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 $(11, x_{11})$ 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 $0 \leq \ell \leq 10$) 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 $\mathrm{G}_2, \mathrm{G}_3, \mathrm{G}_4$ (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* $x_i, \dots, x_j$[10]*, 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* $x_i, \dots, x_j$*.*

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_{10}, x_{11} 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 $j - i + 13$ 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 $h = 11$.

We note that a partial path cannot have length more than 12, because Definition 7 doesn't allow "self-overlapping" paths.

Definition 9. The *left maximal path* of a 2chain $(\ell, x_\ell, x_{\ell+1})$ is the longest (i,j)-partial path containing $(\ell, x_\ell, x_{\ell+1})$ such that $j = \ell + 1$. Similarly, the *right maximal path* of the 2chain is the longest (i,j)-partial path containing $(\ell, x_\ell, x_{\ell+1})$ such that $i = \ell$.

Thus, a maximal path can have length at most 12, 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. *Each 2chain has a unique left maximal path and a unique right maximal path.*

Proof. We give a proof for the left maximal path, and the right maximal path is symmetric. Since the partial paths have length at most 12, there exists a unique maximum length for the $(i, \ell+1)$-partial paths containing the 2chain. Moreover, the partial path of the maximum length is unique by Lemma 17. □

Definition 10. Let n be a non-root node. The *maximal path* of n is the left maximal path of $n.id$ if n's origin is 2 or 6, and is the right maximal path of $n.id$ if n's origin is 5 or 9.

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. *If* $n.id = (i, x_i, x_{i+1})$*, then* (i, x_i) *and* $(i+1, x_{i+1})$ *are in the maximal path of* n*.*
2. *After* $\text{F}(i, x_i)$ *is called in* MakeNodeReady(n)*, the query* (i, x_i) *is in the maximal path of* n*.*
3. *After* $\text{SimP}(x_0, x_1)$ *is called in* MakeNodeReady(n)*, both* $(0, x_0)$ *and* $(1, x_1)$ *are in the maximal path of* n*; after the call returns with value* (x_{10}, x_{11})*,* $(10, x_{10})$ *and* $(11, x_{11})$ *are in the maximal path of* n*. Symmetrically for a call to* SimP^{-1}*.*

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

4. *The query that is assigned to $n.end$ is in the maximal path of n (even if the assignment doesn't occur because an assertion fails).*

Proof. In the following we assume that the origin of n is 2 or 6. The other two 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 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 10 and 11) 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 $(x_{10}, x_{11}) = T(x_0, x_1)$, which are in the same partial path as x_0 and x_1, given that the partial path has origin 2 or 6 (i.e., that we are not overextending the partial path past length 12).

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. □

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 NotChild, 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 case where the terminal of n is 2 for example: the origin of n is

5, and the position of $n.id$ is also 5. A call to Equivalent($n.id, (1, x_1, x_2)$) is made in FindNewChildren(n), in which $\mathrm{Val}^+(n.id, 1)$ and $\mathrm{Val}^+(n.id, 2)$ are called. Since n is ready, by Lemma 19 we know $n.end$ is in the right maximal path of $n.id$, i.e., $\mathrm{Val}^+(n.id, 2) = n.end \neq \perp$. This also implies $\mathrm{Val}^+(n.id, 1) \neq \perp$. The other cases are similar.

The call to AdaptNode(n) occurs after SampleTree(n), and both $n.beginning$ and $n.end$ are defined at this point. 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 $\{x_h\}_{h=0}^{11}$ 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 NotChild 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** only 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. *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 11, $n.end$ 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 25. *If a non-root node n is an inner node or a ready outer node, its maximal path contains pending or defined queries in positions 5 and 6. Moreover, for any two distinct nodes n_1 and n_2 each of which is an inner node or a ready outer node, their maximal paths contain different pairs of queries in positions 5 and 6.*

Proof. If n is an inner node, then $n.id$ contains queries in positions 5 and 6. One can observe from the pseudocode of FindNewChildren that both of these queries are pending or defined, with at least one of the two queries being defined. (The latter fact will be useful later in the proof of this lemma.)

If n is a ready outer node, we show the proof when n's origin is 2. In the last iteration in MakeNodeReady(n), F$(6, x_6)$ is called and $(6, x_6)$ is defined; when n becomes ready, we have $n.end = (5, x_5)$. The above queries are in the maximal path of n by Lemma 19, and they remain pending or defined for the rest of the execution (Lemma 14). Therefore, the maximal path contains pending or defined queries in positions 5 and 6, with the query in position 6 being defined.

Now we prove the second part of the lemma. Assume by contradiction that the maximal paths of n_1 and n_2 both contain queries $(5, x_5)$ and $(6, x_6)$. By the above observations at least one of these two queries is defined; without loss of generality, assume $(5, x_5)$ becomes defined after $(6, x_6)$, and in particular, that $(6, x_6)$ is defined at the moment. The following discussion also relies on the fact that the queries in $n.id$ and the queries made in MakeNodeReady(n) are in the maximal path of n, as per Lemma 19.

If both n_1 and n_2 are inner nodes, then $n_1.id = n_2.id = (5, x_5, x_6)$ and $n_1.beginning = n_2.beginning = (5, x_5)$. Indeed, the *beginning* of a node is defined later than the other query in the *id*, because when the node is created in FindNewChildren, the *beginning* is pending (cf. Lemma 24) while the other query is defined. By Lemma 9, we have $n_1.parent.end = n_2.parent.end = (5, x_5)$. Since the nodes have distinct *end* by Lemma 7, n_1 and n_2 have the same parent. However, this is impossible because by Lemma 23, a parent can't have two children with the same *id*.

If n_1 and n_2 are both ready outer nodes then we claim that $n_1.end = n_2.end = (5, x_5)$, which will violate Lemma 7. Indeed, before the *end* of the node is assigned a query in MakeNodeReady the query cannot be pending or defined (otherwise the simulator aborts before the assignment). If $end = (6, x_6)$, then $(5, x_5)$ is defined in the call to Next$(4, x_4, x_5)$, at which point $(6, x_6)$ is not defined. This contradicts the assumption that $(6, x_6)$ becomes defined earlier. Thus $n_1.end = n_2.end = (5, x_5)$, which contradicts Lemma 7.

The only possibility left is the case where one of the two nodes is an inner node and the other is a ready outer node. Wlog, let n_1 be the inner node. The analysis of the two previous cases shows that $n_1.beginning = n_2.end = (5, x_5)$. Thus, n_2 is the unique node whose *end* equals $(5, x_5)$, and it must be the parent of n_1. However, because $n_1.id = (5, x_5, x_6)$ and the maximal path of n_2 also contains $(5, x_5)$ and $(6, x_6)$, this contradicts Lemma 23. □

Lemma 26. *The simulator creates at most $4q^2 - q$ inner nodes in an execution.*

Proof. We can upper bound the number of inner nodes by upper bounding the number of pending or defined queries in positions 5 and 6, i.e., the sizes of $\tilde{F}_5$ and $\tilde{F}_6$.

Other than the queries issued by the distinguisher, a query can only be added to $\tilde{F}_i$ in a call to MakeNodeReady or AdaptNode. Specifically, for $i \in \{5, 6\}$, a query is added only in MakeNodeReady(n) where n is an outer node. Indeed, AdaptNode only defines queries at positions 3, 4, 7 and 8, while MakeNodeReady(n) creates no new queries at positions 5 or 6 if n is an inner node, as the queries in $n.id$ are already defined or pending when n is created.

For each outer node n, at most two queries in positions 5 and 6 become pending or defined in the call to MakeNodeReady(n): one of them is queried in Next or Prev, and the other becomes pending when the node becomes ready. Therefore, if an outer node n is not ready, this node accounts for at most one query added to $\tilde{F}_5$ and $\tilde{F}_6$.

Let Outer_1 and Outer_2 be the number of outer nodes that are ready and not ready, respectively. By Lemma 16, we have $\mathsf{Outer}_1 + \mathsf{Outer}_2 \leq q$. Since the distinguisher makes at most q queries in each position, we have

$$|\tilde{F}_5| + |\tilde{F}_6| \leq 2q + 2 \cdot \mathsf{Outer}_1 + \mathsf{Outer}_2 \leq 3q + \mathsf{Outer}_1. \tag{4}$$

By Lemma 25, the maximal paths of inner nodes and ready outer nodes contain distinct pairs of pending or defined queries in positions 5 and 6. Therefore, there exists an injection from the set of inner nodes and ready outer nodes to $\tilde{F}_5 \times \tilde{F}_6$. Let Inner be the number of inner nodes, then

$$\mathsf{Inner} + \mathsf{Outer}_1 \leq |\tilde{F}_5| \cdot |\tilde{F}_6| \leq \left(\left(|\tilde{F}_5| + |\tilde{F}_6|\right)/2\right)^2 \tag{5}$$

Combining equations (4) and (5), we have

$$\mathsf{Inner} \leq \left((3q + \mathsf{Outer}_1)/2\right)^2 - \mathsf{Outer}_1 = (1/4) \cdot \mathsf{Outer}_1^2 + (3q-1) \cdot \mathsf{Outer}_1 + (3q/2)^2$$

Since the right-hand side of the above inequality is monotone increasing with respect to Outer_1 for $q \geq 1$ and since $\mathsf{Outer}_1 \leq q$, we have $\mathsf{Inner} \leq 4q^2 - q$. □

Lemma 27. *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 26 and 16, The total number of non-root nodes is upper bounded by $(4q^2 - q) + q = 4q^2$. □

Lemma 28. *At any point of an execution, the number of pending or defined queries satisfies $|\tilde{F}_i| \leq 2q$ for $i \in \{5, 6\}$, $|\tilde{F}_i| \leq 4q^2$ for $i \in \{1, 2, 9, 10\}$, and $|\tilde{F}_i| \leq 4q^2 + q$ for $i \in \{3, 4, 7, 8\}$.*

Proof. We will use the analysis in Lemma 26. A query is added to $\tilde{F}_i$ for $i \in \{5, 6\}$ only if the query is made by the distinguisher or if the query is in the maximal path of an outer node. The distinguisher makes at most q queries in each position. By Lemma 16, there are at most q outer nodes. Therefore, the sizes of $\tilde{F}_5$ and $\tilde{F}_6$ are upper bounded by $q + q = 2q$.

The queries in $\tilde{F}_i$ for $i \in \{1, 2, 9, 10\}$ are added by distinguisher queries or if the query is in the maximal path of an inner node (similarly to the proof of Lemma 26, the queries in positions $1, 2, 9$ and 10 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 26, therefore each set contains at most $4q^2$ queries.

The queries positions $3, 4, 7$ and 8 cannot be pending. They can become defined in MakeNodeReady(n) or AdaptNode(n). It is easy to check that for each non-root node n, at most one query in each of these positions becomes defined in the two procedures: F is called on two of the positions in MakeNodeReady(n), and Adapt is called on the other two positions in AdaptNode(n). The distinguisher also makes at most q queries in each position, and there are at most $4q^2$ non-root nodes, thus the size of $\tilde{F}_i$ is upper bounded by $4q^2 + q$ for $i \in \{3, 4, 7, 8\}$. □

Lemma 29. *We have $|F_i| \leq 2q$ for $i \in \{5, 6\}$, $|F_i| \leq 4q^2$ for $i \in \{1, 2, 9, 10\}$, and $|F_i| \leq 4q^2 + q$ for $i \in \{3, 4, 7, 8\}$. 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 28.

In G_2, G_3 and G_4, the CheckP procedure doesn't 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) is the one queried in CheckP before n is added (which preexists in T even before the call to CheckP occurs). Thus the simulator makes new permutation queries only in MakeNodeReady(n) for inner nodes n. By Lemma 26, 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 30. *Consider an execution of* G_1*. If the simulator calls* $\text{SimP}(x_0, x_1) = (x_{10}, x_{11})$ *or* $\text{SimP}^{-1}(x_{10}, x_{11}) = (x_0, x_1)$ *(note that the answer must be returned because* SimP *and* SimP^{-1} *don't abort in* G_1*), we have* $x_1 \in F_1$ *and* $x_2 := F_1(x_1) \oplus x_0 \in \tilde{F}_2$ *at the end of the current query cycle.*

Proof. The simulator makes permutation queries in the CheckP procedure and in the MakeNodeReady procedure.

If the permutation query 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) when the origin of n is 2 or 9. 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 9, $x_1 \in F_1$ and $x_2 \in F_2$.

Now consider a call to MakeNodeReady(n). If n is an outer node, the permutation made in MakeNodeReady(n) has been queried in CheckP before the node is added. If n is an inner node, the origin of n is 5 or 6.

If n's origin is 6, MakeNodeReady(n) calls Prev and make queries in positions 5, 4, 3, 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.

Otherwise if n's origin is 5, MakeNodeReady(n) makes queries via Next. Since the procedures SimP^{-1} and $\text{F}(1, x_1)$ does not abort in G_1, when SimP^{-1} is called in Next, the next two calls to Next will successfully return. Therefore, another call to Next will be made, in which $\text{F}(2, x_2)$ is called. Unless $(2, x_2)$ is already pending or defined, NewTree$(2, x_2)$ will be called and $(2, x_2)$ will become pending immediately. Thus $(1, x_1)$ must be defined and $(2, x_2)$ must be pending or defined at the end of the query cycle. ☐

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

Theorem 31. *In the simulated world* G_1*, the simulator makes at most* $16q^4$ *queries to the permutation oracle.*

Proof. The simulator queries the permutation oracle via the wrapper functions SimP and SimP^{-1}. The functions maintain tables T_{sim} and T_{sim}^{-1}, consisting of previously made permutation queries. When SimP or SimP^{-1} is called, they first check whether the query is in the tables; if so, the table entry is directly returned without actually querying the permutation oracle. Therefore, the permutation oracle is queried only when a query is made for the first time, and we only need to upper bound the number of distinct permutation queries issued by the simulator (note that a permutation query and its inverse are considered as the same query).

By Lemma 30, if a permutation query $\text{SimP}(x_0, x_1) = (x_{10}, x_{11})$ or $\text{SimP}^{-1}(x_{10}, x_{11}) = (x_0, x_1)$ is issued by the simulator, we have $x_1 \in F_1$ and $x_2 := F_1(x_1) \oplus x_0 \in \tilde{F}_2$ at the end of the current query

cycle. Note that each pair of x_1 and x_2 determines a unique permutation query $(F_1(x_1) \oplus x_2, x_1)$. Thus, the number of distinct permutation queries issued by the simulator is at most

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

where the inequality is due to Lemmas 28 and 29. ☐

Lemma 32. *The* NewTree *procedure is called at most* $4q^2 + 4q$ *times, i.e., at most* $4q^2 + 4q$ *root nodes are created in an execution.*

Proof. The NewTree procedure is only called in F, when the argument is an undefined query in position $2, 5, 6$, or 9 (we will call these positions the trigger positions, in this proof only).

The distinguisher can make q queries to F in each position, therefore at most $4q$ queries in trigger positions are made by the distinguisher.

F is also called by the simulator in MakeNodeReady. We claim that in a call to MakeNodeReady(n), F is called on at most one undefined query in trigger positions. Indeed, if the origin of n is 2, F is called on queries in positions $1, 10, 9, 8, 7$, and 6, where the queries in positions $1, 10$, and 9 are defined before the node is created. Therefore, only the query in position 6 can be an undefined query in a trigger position. If the origin of n is 5, F is called on queries in positions $6, 7, 8, 9, 10$, and 1, where the query in position 6 is defined before the node is created. Only the query in position 9 can be an undefined query in a trigger position. The cases where the origin of n is 9 or 6 are symmetric to the above cases. By Lemma 27, there are at most $4q^2$ non-root nodes, and MakeNodeReady is called once on each of them.

To summarize, F is called on undefined queries in trigger positions for at most $4q + 4q^2$ times, which is also an upper bound for the number of calls to NewTree. ☐

Finally we upper bound the time complexity of the simulator.

Theorem 33. *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 or calls other procedures. Note that the following discussion considers the running time inside a procedure, i.e., the running time of called procedures are 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, 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 $10q$ times by the distinguisher. NewTree is only called in F, and IsPending is called in F and MakeNodeReady. By Lemmas 27 and 32, there are at most $4q^2$ non-root nodes and at most $4q^2 + 4q$ 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, i.e., the iteration takes $O(q^2)$ time. The other procedures run in constant time. Therefore, the total running time of the aforementioned procedures is $O(q^2) \cdot O(q^2) = O(q^4)$.

We still need to upper bound the time spent in procedures GrowTree, GrowSubTree, FindNewChildren and NotChild. All these queries are called in the life cycle of a call to GrowTree.

Consider a call to GrowTree($root$) where $root$ is the root of a tree that has τ nodes after GrowTree returns. GrowTree repeatedly calls GrowSubTree 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 GrowSubTree($root$) is called at most τ times. GrowSubTree 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 of GrowTree, and a total of τ^2 times for each tree.

For $i \in \{2, 9\}$, FindNewChildren traverses three tables F_9, F_{10}, F_1 or F_1, F_2, F_{10}. By Lemma 29, the number of triples in the tables is at most $(4q^2)^3 = 64q^6$. Moreover, for each (x_1, x_2), CheckP(x_1, x_2, x_9, x_{10}) outputs **true** for a unique (x_9, x_{10}). In the iteration, CheckP is called on at most $(4q^2)^2 = 16q^4$ different values of (x_1, x_2), so it returns **true** in at most $16q^4$ iterations and the procedures Equivalent and NotChild are called at most $16q^4$ times. Equivalent runs in constant time and NotChild runs in $O(q^2)$ time (since there are at most $O(q^2)$ nodes). On the other hand, for $i \in \{5, 6\}$, FindNewChildren traverses one table F_5 or F_6 of size at most $2q$. The running time of each iteration, as in the previous case, is upper bounded by $O(q^2)$. Therefore, in both cases, each call to FindNewChildren runs in time $O(q^6)$ (including time spent in NotChild).

By Lemmas 27 and 32, the total number of nodes in the trees is at most $4q^2 + 4q^2 + 4q = O(q^2)$, i.e., $\sum \tau = O(q^2)$. The time complexity of the above four procedures 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 simulator can also be optimized to run in $O(q^4)$ time by maintaining hash tables for tree-growing procedures (which costs $O(q^4)$ extra space).

The following proof gives a sketch of the optimization. We assume that hash table entries can be accessed in constant time.

Theorem 34. *The simulator of game* G_1 *can be implemented to run in time* $O(q^4)$*.*

Proof Sketch. From the proof of Theorem 33, the running time of the simulator is $O(q^4)$ except for the time inside GrowTree. Therefore we only need to speed up the tree-growing process.

In our pseudocode, every time FindNewChildren(n) is called, the simulator traverses all tuples of defined queries in certain positions to look for undetected triggered paths, even if the tuple of queries have been checked in previous iterations. Recall that the process has to be repeated because during the completion of other paths more queries are defined, spawning more triggered paths that didn't exist in the previous call to FindNewChildren(n). To improve the running time, the simulator can maintain a table Trigger for these newly triggered path, so that FindNewChildren can add the paths without going through all defined queries.

The table Trigger has an entry for each pending query (i, x_i) (i.e., the entry Trigger(i, x_i) is created when (i, x_i) becomes pending and is deleted as soon as (i, x_i) becomes defined). Each entry Trigger(i, x_i) is a list containing paths triggered by the pending query $(i, x_i) = n.end$ that haven't been added as a child of n. A triggered path is represented by a 2chain (j, x_j, x_{j+1}) which will be the id of the corresponding node (so the 2chain will be the second parameter of the call to AddChild). For example, Trigger$(2, x_2)$ is a list of 2chains $(1, x_1, x_2)$ such that FindNewChildren should call AddChild$(n, (1, x_1, x_2))$ for each 2chain.

We sketch how the table $\mathsf{Trigger}$ is maintained. Whenever a query $n.end = (i, x_i)$ becomes pending, the corresponding entry $\mathsf{Trigger}(i, x_i)$ is added. The simulator initializes the entry by enumerating all defined queries next to (i, x_i) and checks if the two queries are in a triggered path. For example, if $i = 2$, the simulator goes through $x_1 \in F_1$ and adds $(1, x_1, x_2)$ to $\mathsf{Trigger}(2, x_2)$ if $(1, x_1, x_2)$ is not equivalent to $n.id$, if $(x_0, x_1) \in T$ where $x_0 := F_1(x_1) \oplus x_2$, if $x_{10} \in F_{10}$ where $(x_{10}, x_{11}) := T(x_0, x_1)$, and if $x_9 \in F_9$ where $x_9 := F_{10}(x_{10}) \oplus x_{11}$.[11] The aforementioned checking serves the same purpose as the calls to CheckP and Equivalent in the pseudocode of FindNewChildren. On the other hand, if $i = 5$, $(5, x_5, x_6)$ is added to $\mathsf{Trigger}(i, x_i)$ for every $x_6 \in F_6$ unless $(5, x_5, x_6) = n.id$. Since the size of each F_i is $O(q^2)$ and at most $O(q^2)$ queries become pending throughout an execution, the total cost of initializing entries in $\mathsf{Trigger}$ is $O(q^4)$.

To efficiently update $\mathsf{Trigger}$, the simulator also maintains a list of pending queries $\mathsf{Pending}_i$ for each position i. Entries in $\mathsf{Trigger}$ are updated whenever a query becomes defined. Like the initialization, the simulator checks queries in an adjacent position to find newly triggered paths. For example, consider when a query $(1, x_1)$ is defined. The simulator checks all queries $x_2 \in F_2$ and extends the path to see whether the corresponding $x_{10} \in F_{10}$ and $x_9 \in \mathsf{Pending}_9$; if so, adds $(9, x_9, x_{10})$ to $\mathsf{Trigger}(9, x_9)$. The simulator also checks all $x_2 \in \mathsf{Pending}_2$ to see whether the corresponding $x_{10} \in F_{10}$ and $x_9 \in F_9$; if so, it adds $(1, x_1, x_2)$ to $\mathsf{Trigger}(2, x_2)$. Since there are at most $O(q^2)$ defined queries in each position, the update takes $O(q^2)$ time for each defined query. The total running time for updating is $O(q^4)$.

With the table $\mathsf{Trigger}$, the procedure FindNewChildren(n) can be simplified as follows: Let $(i, x_i) := n.end$. The simulator reads the list $\mathsf{Trigger}(i, x_i)$; if the list is non-empty, it runs AddChild(n, c) (and increments *NumOuter* if $i \in \{2, 9\}$) for each $c \in \mathsf{Trigger}(i, x_i)$ and sets *child_added* to **true**.

Note that there is no need to call NotChild or Equivalent, because each path is added to $\mathsf{Trigger}$ exactly once (when the triggering query (i, x_i) becomes pending or when the last of the other queries in the detect zone becomes defined, whichever occurs later) and the 2chain equivalent to $n.id$ has already been checked against when $\mathsf{Trigger}(i, x_i)$ is initialized.

The procedure FindNewChildren is called at most $O(q^2)$ times on each node and there are $O(q^2)$ nodes, thus the running time is $O(q^4)$ excluding the time for handling the triggered paths. It takes $O(1)$ time to add a node for each triggered path, and there are at most $O(q^2)$ triggered paths, so handling the triggered paths in FindNewChildren incurs, overall, only an extra additive term of $O(q^2)$. □

5.2 Transition from $\mathbf{G_1}$ to $\mathbf{G_2}$

MODIFICATIONS IN G_2. The game G_2 differs from G_1 in two places: the procedure 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, Val^+ and Val^- are also used in games G_3 and G_4.

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

[11] This already saves a factor of $O(q^4)$ compared to enumerating all tuples of x_9, x_{10} and x_1.

Note that the CheckP procedure is called by FindNewChildren and is responsible for determining whether a quadruple of queries (x_1, x_2, x_9, x_{10}) is 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 is not 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 35. *We have*

$$\Delta_D(\mathrm{G}_1, \mathrm{G}_2) \leq 512q^8/2^{2n}.$$

Proof. This proof uses the divergence technique from Lemma 40 in [11].

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

Consider the executions of G_1 and G_2 with the same random tapes $f_1, \ldots, f_{10}, p$. We say the two executions *diverge* if in a call to $\mathrm{CheckP}(x_1, x_2, x_9, x_{10})$ we have $p(x_0, x_1) = (x_{10}, x_{11})$ and $(x_0, x_1) \notin T$ in the execution of G_2, where x_0, x_1, x_{10}, x_{11} are defined as in the pseudocode of CheckP (i.e., $x_0 = F_1(x_1) \oplus x_2$ and $x_{11} = x_9 \oplus F_{10}(x_{10})$). It is easy to check that a call to 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, P, P^{-1}, PSim, PSim^{-1}, Val^+ and Val^-. CheckP always returns 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, 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 $(x_0, x_1) \notin T$ and $p(x_0, x_1) = (x_{10}, x_{11})$. We compute the probability of the above event in G_2.

We will upper bound the probability that divergence occurs in each CheckP call, 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 29, 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, and it equals (x_{10}, x_{11}) with probability at most $1/(2^{2n} - 4q^2)$. In both cases, the probability that divergence occurs in the CheckP call is upper bounded by $1/(2^{2n} - 4q^2)$.

The procedure CheckP is only called in FindNewChildren, and its arguments correspond to four queries that are pending or defined in positions $1, 2, 9, 10$. By Lemma 28, the number of pending or defined queries in each of these positions is at most $4q^2$, and CheckP is called on at most $(4q^2)^4 = 256q^8$ distinct arguments.

If CheckP is called multiple times with the same argument (x_1, x_2, x_9, x_{10}) 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

$$256q^8 \cdot \frac{1}{2^{2n} - 4q^2} \leq 256q^8 \cdot \frac{1}{2^{2n} - 2^{2n}/4} \leq \frac{512q^8}{2^{2n}}$$

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

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 $512q^8/2^{2n}$, 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, 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 $\Delta_D(G_2, G_3)$; 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 36. *We have*

$$\Delta_D(G_2, G_5) \leq \Delta_D(G_3, G_5).$$

Proof. By the definition of advantage in equation (3), we have

$$\Delta_D(G_i, G_5) = \Pr_{G_i}[D^{F,P,P^{-1}} = 1] - \Pr_{G_5}[D^{F,P,P^{-1}} = 1].$$

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, and a 2chain is *active* if it is both left active and right active as defined below:

Definition 11. 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 $i \leq 9$ and the query $(i+1, x_{i+1})$ is active, or if $i = 10$ and $(x_i, x_{i+1}) \in T^{-1}$.

The procedures IsLeftActive, IsRightActive, and IsActive in the pseudocode check whether a 2chain is left active, right active, and active, respectively.

The explicit definitions of the bad events BadP, BadR and BadA given below in definitions 12, 14 and 17 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 defined query in position 1 or 10 (depending on the direction of the permutation query):

Definition 12. 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 $x_{10} \in F_{10}$ for $(x_{10}, x_{11}) = p(x_0, x_1)$. Similarly, BadP occurs in $\mathrm{P}^{-1}(x_{10}, x_{11})$ if at the beginning of the procedure, $(x_{10}, x_{11}) \notin T^{-1}$ and $x_1 \in F_1$ for $(x_0, x_1) = p^{-1}(x_{10}, x_{11})$.

Lemma 37. *The probability that* BadP *occurs in an execution of* G_3 *is at most* $32q^4/2^n$*.*

Proof. As in Lemma 35 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 29 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 $x_{10} \in F_{10}$ where x_{10} is the first half of the tape entry $p(x_0, x_1) = (x_{10}, x_{11})$. By Lemma 29 we have $|F_{10}| \leq 4q^2$, and there are at most 2^n possible values for x_{11}. Therefore, BadP occurs when (x_{10}, x_{11}) equals one of the $4q^2 \cdot 2^n$ 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 $2^n \cdot 4q^2/(2^{2n} - 4q^2)$.

The same bound can be proved symmetrically for a call to $\mathrm{P}^{-1}(x_{10}, x_{11})$ with $(x_{10}, x_{11}) \notin T^{-1}$.

Each call to $\mathrm{P}(x_0, x_1)$ with $(x_0, x_1) \notin T$ or to $\mathrm{P}^{-1}(x_{10}, x_{11})$ with $(x_{10}, x_{11}) \notin T^{-1}$ adds an entry to the table T. By Lemma 29, 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

$$4q^2 \cdot \frac{2^n \cdot 4q^2}{2^{2n} - 4q^2} = \frac{16q^4}{2^n - 4q^2/2^n}.$$

Since $q \leq 2^{n-2}$, $4q^2/2^n < 2^{n-1}$ and $16q^4/(2^n - 4q^2/2^n) < 32q^4/2^n$. □

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 events BadR and BadA.

Definition 13. 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 $\mathrm{Val}^+(j, x_j, x_{j+1}, i) = x_i$ or $\mathrm{Val}^-(j, x_j, x_{j+1}, i) = x_i$.

Lemma 38. *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 $\mathrm{Val}^+(i-2, x_{i-2}, x_{i-1}, i) = x_i$;*
- *$i \in \{0, 1\}$ and there exists an active 2chain $(10, x_{10}, x_{11})$ such that $\mathrm{Val}^+(10, x_{10}, x_{11}, i) = x_i$;*
- *$i \leq 9$ and there exists an active 2chain $(i+1, x_{i+1}, x_{i+2})$ such that $\mathrm{Val}^-(i-2, x_{i-2}, x_{i-1}, i) = x_i$;*
- *$i \in \{10, 11\}$ and there exists an active 2chain $(0, x_0, x_1)$ such that $\mathrm{Val}^-(0, x_0, x_1, i) = x_i$.*

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 $(10, x'_{10}, x'_{11})$ is active by looking separately at the cases $k = 10$ and $k < 10$, and that $\mathrm{Val}^+(10, x'_{10}, x'_{11}, i) = x_i$. $\square$

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. It aborts if the following event occurs:

Definition 14. Let ReadTape be called with argument (i, x_i) such that $x_i \notin F_i$. Then we define the following two events:

- $\mathsf{BadRHit}$ 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.
- $\mathsf{BadRCollide}$ 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.

Moreover, we let $\mathsf{BadR} = \mathsf{BadRHit} \vee \mathsf{BadRCollide}$.

In order to upper bound the probability of BadR, 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 11, 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 $i \leq 9$, $x_{i+1} \in \tilde{F}_{i+1}$.

We extend the definition of sets $\tilde{F}_i$ for $i = 0, 11$ 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 $\tilde{F}_{11}$ is the set values of x_{11} such that $(10, x_{10}, x_{11})$ is right active for some x_{10}. Or, equivalently:

$$\tilde{F}_0 := \{x_0 : \exists x_1 \text{ s.t. } T(x_0, x_1) \neq \bot\}, \text{ and}$$
$$\tilde{F}_{11} := \{x_{11} : \exists x_{10} \text{ s.t. } T^{-1}(x_{10}, x_{11}) \neq \bot\}.$$

In particular we have $|\tilde{F}_0| \leq |T|$ and $|\tilde{F}_{11}| \leq |T|$.

Lemma 39. *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 $1 \leq i \leq 10$. This lemma follows from the definition of left active, right active, and from the definition of the sets $\tilde{F}_i$ for $0 \leq i \leq 11$. □

We note that Lemma 39 is not if-and-only-if; for example, if x_0, x_1 are values such that $x_0 \in \tilde{F}_0$ and $T(x_0, x_1) = \bot$, then $(0, x_0, x_1)$ is not left active. (However, the first part of Lemma 39 is if-and-only-if for $1 \leq i \leq 10$, and symmetrically, the second part is if-and-only-if for $0 \leq i \leq 9$.)

Lemma 40. *We have $|\tilde{F}_i| \leq 4q^2$ for $i \in \{0, 11\}$, and $|\tilde{F}_i| \leq 4q^2 + q$ for all $\tilde{F}_i$.*

Proof. By Lemma 29, we have $|\tilde{F}_0| \leq |T| \leq 4q^2$ and $|\tilde{F}_{11}| \leq |T| \leq 4q^2$. The second statement then follows by Lemma 28. □

Definition 15. Let $\mathcal{C}_i$ denote the set of x_i such that (i, x_i) is incident with an active 2chain.

Lemma 41. *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 38, 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

$$j = \begin{cases} i - 2 & \text{if } i \geq 2 \\ 10 & \text{if } i \leq 1 \end{cases}$$

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

$$j = \begin{cases} i + 1 & \text{if } i \leq 9 \\ 0 & \text{if } i \geq 10 \end{cases}$$

such that $\text{Val}^-(j, x_j, x_{j+1}, i) = x_i$. Moreover, the total number of active 2chains in each position is at most $(4q^2 + q)^2$ by Lemma 40. □

Lemma 42. BadRHit *occurs in a call to* $\text{ReadTape}(i, x_i)$ *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 39, 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 40, 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. $\mathsf{BadRCollide}$ *occurs in a call to* $\mathrm{ReadTape}(i, x_i)$ *with probability at most* $4(4q^2+q)^3/2^n$.

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

Consider the first part of $\mathsf{BadRCollide}$. If $(i-1, x_{i-1}, x_i)$ is left active, we must have $x_{i-1} \in \tilde{F}_{i-1}$ by Lemma 39. 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 40 and 41, we have

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

Symmetrically, the same bound can be proved for the second part of $\mathsf{BadRCollide}$. Thus $\mathsf{BadRCollide}$ occurs for at most $2 \cdot 2(4q^2 + q)^3 = 4(4q^2 + q)^3$ values of $f_i(x_i)$. $\square$

Lemma 44. *In an execution of* G_3, BadR *occurs with probability at most* $21000q^8/2^n$.

Proof. Every time $\mathrm{ReadTape}(i, x_i)$ 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 29.

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

$$40q^2 \cdot \left(\frac{(4q^2+q)^2}{2^n} + \frac{4(4q^2+q)^3}{2^n}\right) \leq \frac{21000q^8}{2^n}.$$

$\square$

$(2,5)$-Trees and $(6,9)$-Trees. At this point it will be useful to establish some terminology for distinguishing trees that have nodes with origin/terminal 2 and 5 from trees that have nodes with origin/terminal 6 and 9. Indeed:

Lemma 45. *A ready node with origin 2 (resp. 5, 6, 9) has terminal 5 (resp. 2, 9, 6).*

Proof. This is obvious from MakeNodeReady. $\square$

Moreover, recall that a non-root node's origin is the terminal of its parent (Lemma 9). In particular, it follows from Lemma 45 that if $r.end$ has position 2 or 5 (resp. 6 or 9) where r is the root of a tree, then $n.end$ has position 2 or 5 (resp. 6 or 9) for every ready node n in the tree rooted at r.

Definition 16. A tree is called a $(2,5)$-*tree* if its root has terminal 2 or 5; a tree is a $(6,9)$-*tree* if its root has terminal 6 or 9.

By the above remarks, every ready node of a $(2,5)$-tree has terminal 2 or 5, and every ready node of a $(6,9)$-tree has terminal 6 or 9.

Bad Adapt. The CheckBadA procedure is called once per tree. The call is made after SampleTree(r) is called and before AdaptTree(r) is called for a tree with root r, i.e., right after the pending queries in the tree have been sampled.

Recall that by Lemmas 11 and 19, before SampleTree(r) is called, each node n in the tree is associated to a (unique) proper partial path containing $n.id$ and whose endpoints are the origin and terminal of the node. Each such proper partial path thus has the form $\{x_h\}_{h=5}^{2}$ in the case of a

$(2,5)$-tree, and has the form $\{x_h\}_{h=9}^{6}$ in the case of a $(6,9)$-tree. 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 $(i,j) \in \{(2,5),(6,9)\}$ are the endpoints of the path, for each such partial path. After the call to SampleTree, thus, the partial path can be extended by one query in each direction, meaning that each node in the tree can be associated to a unique $(4,3)$-partial path $\{x_h\}_{h=4}^{3}$ containing $n.id$ in the case of a $(2,5)$-tree and to a unique $(8,7)$-partial path $\{x_h\}_{h=8}^{7}$ in the case of a $(6,9)$-tree. We will refer to these partial paths as, respectively, the $(4,3)$-*partial path associated to* n (for $(2,5)$-trees) and the $(8,7)$-*partial path associated to* n (for $(6,9)$-trees).

The $(4,3)$- (resp. $(8,7)$-) partial path associated to n thus becomes well-defined after SampleTree(r) returns, where r is the root of the tree containing n.

Focusing for concreteness on the case of a $(2,5)$-tree, AdaptTree assigns

$$F_3(x_3) \leftarrow x_2 \oplus x_4$$
$$F_4(x_4) \leftarrow x_3 \oplus x_5$$

for each non-root node n, where $\{x_h\}_{h=4}^{3}$ is the $(4,3)$-partial path associated to n. (See the procedures AdaptTree, AdaptNode and Adapt.) We say that the queries $(3,x_3)$ and $(4,x_4)$ are *adapted* in the call to AdaptTree. The assignments to F_3 and F_4 are also called *adaptations*. Thus, two adaptations occur per non-root node in the three.

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 $(2,5)$-tree with associated $(4,3)$-partial path $\{x_h\}_{h=4}^{3}$, and letting

$$y_3 = x_2 \oplus x_4$$
$$y_4 = x_3 \oplus x_5$$

be the future values of $F_3(x_3)$ and $F_4(x_4)$ respectively, GetAdapts will create the two instances of **Adapt** with the following settings:

$$(\textit{query, value, left, right}) = ((3,x_3), y_3, x_2, x_5),$$
$$(\textit{query, value, left, right}) = ((4,x_4), y_4, x_2, x_5).$$

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.

Lemma 46. *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. For simplicity we let $i = 3$. Other cases can be proved similarly.

The Lemma is equivalent to the statement that for a given query $(3, x_3)$, there is at most one non-root node n in the tree rooted at r whose associated partial path $\{x_h\}_{h=4}^{3}$ contains $(3, x_3)$. By contradiction, assume that $n' \neq n$ is a second non-root node whose associated partial path $\{x'_h\}_{h=4}^{3}$ contains $(3, x_3)$, i.e., $x'_3 = x_3$.

If $x_2 = x'_2$ then $(x_2, x_3) = (x'_2, x'_3)$ and, by extension, $\{x_h\}_{h=4}^{3} = \{x'_h\}_{h=4}^{3}$. Since $n.end \neq n'.end$ by Lemma 7 we must have $n.beginning = n'.end$ and $n.end = n'.beginning$, but this contradicts Lemma 9. Hence $x_2 \neq x'_2$.

The values $F_2(x_2)$ and $F_2(x'_2)$ are sampled in SampleTree. We can assume wlog that $F_2(x_2)$ is sampled before $F_2(x'_2)$ since $x_2 \neq x'_2$. But then BadRCollide occurs in ReadTape$(2, x'_2)$, since the 2chain $(1, x'_1, x'_2)$ is left active and the query $(3, x'_3)$ where $x'_3 = x'_1 \oplus f_2(x'_2)$ is incident with the active 2chain $(1, x_1, x_2)$. The simulator would have aborted in the procedure, leading to a contradiction. ☐

By Lemma 46 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 $\{x_h\}_{h=4}^{3}$ or $\{x_h\}_{h=8}^{7}$ 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 $\{x_h\}_{h=4}^{3}$ (for $(2, 5)$-trees) or for $\{x_h\}_{h=8}^{7}$ (for $(6, 9)$-trees).

Definition 17. Let r be a root node, and consider the point in the execution after GetAdapts(r) is called. Then we define the following two 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 $(i, x_i, y_i) \in \mathcal{A}$, there exist $x'_{i-1} \in \tilde{F}_{i-1}$ and $x'_{i+1} \in \tilde{F}_{i+1}$ such that $y_i = x'_{i-1} \oplus x'_{i+1}$.
- 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.

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

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). By Lemma 11, the *end* of the nodes in the tree haven't been defined and are sampled uniformly at random in SampleTree. When we use the notations $\{x_h\}_{h=4}^{3}$ or $\{x_h\}_{h=8}^{7}$ (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_4, x_3 or x_8, x_7 (depending) of the path are random variables defined over the coins read by SampleTree. By extension, each $(i, x_i, y_i) \in \mathcal{A}$ is a random variable over the same set of coins.

Also note that the sets $\tilde{F}_i$ are not changed during the call to SampleTree(r) (though some pending queries become defined), therefore the sets are fixed before SampleTree(r) is called and are independent of the randomly sampled queries.

On the other hand, the sets $\mathcal{C}_i$ (Definition 15) are affected by SampleTree(r) and should therefore also be seen as random variables. In more detail, Lemma 39 implies that changing a query from pending to defined at position $i \in \{2, 5, 6, 9\}$, can only affect $\mathcal{C}_{i-1}$ and $\mathcal{C}_{i+1}$. (Note indeed that such

a change does not affect the set of active 2chains, as the tables $\tilde{F}_j$ do not change.) Thus if r is the root of a $(2,5)$-tree then $\mathcal{C}_1$ and $\mathcal{C}_3$ are affected by the new entries added to F_2 (and only by those entries), while $\mathcal{C}_4$ and $\mathcal{C}_6$ are affected by the new entries added to F_5 (and only by those entries). The other $\mathcal{C}_i$'s are not affected by the call to SampleTree(r), and can be seen as static sets. Symmetric observations hold for $(6,9)$-trees.

Lemma 47. *Let n be a non-root node in the tree rooted at r, then the probability that* BadAHit *occurs for a query adapted in n is at most $2(4q^2+q)^2/2^n$.*

Proof. First consider an adapted query in position 3. Let $(3,x_3,y_3)\in\mathcal{A}$ be adapted in the path $\{x_h\}$. We have $y_3=x_2\oplus x_4=x_2\oplus f_5(x_5)\oplus x_6$, where $f_5(x_5)$ is uniformly distributed and x_2 and x_6 are fixed before SampleTree(r) is called. Both $\tilde{F}_2$ and $\tilde{F}_4$ contain at most $4q^2+q$ queries by Lemma 40, and are independent of $f_5(x_5)$. Therefore, the probability that $x_2\oplus f_5(x_5)\oplus x_6$ equals a value in $\tilde{F}_2\oplus\tilde{F}_4$ is at most $(4q^2+q)^2/2^n$. Similarly, the same bound can be proved for adapted queries in other positions.

Since two queries are adapted per node, the lemma follows by a union bound. ☐

Lemma 48. *Let n_1 and n_2 be non-root nodes in the tree rooted at r. The probability that* BadAPair *occurs for the position-i query adapted in n_1 and the position-$(i+1)$ query adapted in n_2 is at most $(8q^2+2q+4(4q^2+q)^2)/2^n$.*

Proof. We have $i=3$ if r is the root of a $(2,5)$-tree, or $i=7$ if r is the root of a $(6,9)$-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.

We have $x_i=x_{i-2}\oplus f_{i-1}(x_{i-1})$ 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 $x_{i-1}\neq u_{i-1}$, 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 value is also independent of $\tilde{F}_{i+2}$ and of $\mathcal{C}_{i+2}$, as discussed before Lemma 47. 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 40 and 41.

By a symmetric argument, the same bound can be proved for the event that $x_{i+2}\neq u_{i+2}$ and $(i-1,y_i,u_{i+1})$ is active or is incident with an active 2chain. The lemma follows by a union bound on these two events. ☐

Lemma 49. *The probability that* BadA *occurs in an execution of* G_3 *is at most $1960q^8/2^n$.*

Proof. By Lemma 27, there are at most $4q^2$ non-root nodes in an execution. By Lemmas 47 and 48, and with a union bound over all non-root nodes in an execution, the probability that either BadAHit or BadAPair occurs in an execution of G_3 is at most

$$4q^2\cdot\frac{2(4q^2+q)^2}{2^n}+(4q^2)^2\cdot\frac{8q^2+2q+4(4q^2+q)^2}{2^n}\le\frac{1960q^8}{2^n}$$

☐

We say that an execution of G_3 is *good* if none of the bad events occurs. (Note that a good execution is not defined as "one that doesn't abort"; however, the two notions are shown to be equivalent for G_3 in Lemma 70.)

Lemma 50. *An execution of* G_3 *is good with probability at least* $1 - 22992q^8/2^n$.

Proof. With a union bound on the results in Lemmas 37, 44 and 49, the probability that at least one of BadP, BadR and BadA occurs is upper bounded by

$$\frac{32q^4}{2^n} + \frac{21000q^8}{2^n} + \frac{1960q^8}{2^n} \leq \frac{22992q^8}{2^n}.$$

Thus the probability of obtaining a good execution is at least $1 - 22992q^8/2^n$. □

5.4.2 Assertions don't Abort in G_3

Now we prove that assertions never fail in executions of G_3. All lemmas of this section apply to G_3 (which is sometimes reminded) except for Lemma 53, which applies both to G_3 and G_4.

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

Lemma 51. *In an execution of* G_3*, the simulator doesn't abort in* Assert *called by* FindNewChildren.

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

When an outer node is created, the permutation query in its maximal path has been defined because of the call to CheckP in FindNewChildren. Therefore each outer node can be associated with an entry in T.

Next we prove that the outer nodes are associated with distinct entries in T. Assume by contradiction that two outer nodes n_1 and n_2 are associated to the same permutation query $T(x_0, x_1) = (x_{10}, x_{11})$, and let $x_2 = F_1(x_1) \oplus x_0$, $x_9 = F_{10}(x_{10}) \oplus x_{11}$. When n_1 or n_2 is created in FindNewChildren, one of the queries $(2, x_2)$ and $(9, x_9)$ is pending and the other is defined, and the pending one is the *end* of its parent. Assuming without loss of generality that $(2, x_2)$ becomes defined after $(9, x_9)$, we have $n_1.parent.end = n_2.parent.end = (2, x_2)$. By Lemma 7, the *end* of nodes are distinct and hence $n_1.parent = n_2.parent$. But then we have $n_1.id = n_2.id = (1, x_1, x_2)$ for the siblings, contradicting property (i) in Lemma 23.

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

In an execution of G_3, the simulator makes permutation queries only in MakeNodeReady(n). Moreover, if n is an outer node, the permutation query made in MakeNodeReady(n) already exists when n is created, because otherwise CheckP will return **false**. Therefore, new entries are added to T in MakeNodeReady(n) only if n is an inner node.

Next we prove that no outer node corresponds to an entry in T added by MakeNodeReady(n) where n is an inner node. Assume $n.beginning = (5, x_5)$ without loss of generality, and let $\mathrm{SimP}^{-1}(x_{10}, x_{11}) = (x_0, x_1)$ be the permutation query made in MakeNodeReady(n), which does not exist in T before.

When the entry $T(x_0, x_1) = (x_{10}, x_{11})$ is added, the queries $(10, x_{10})$ and $(9, x_9)$ for $x_9 = F_{10}(x_{10}) \oplus x_{11}$ have been defined in MakeNodeReady(n) (or before). Moreover, after making the permutation query, the simulator immediately calls $\mathrm{F}(1, x_1)$ (which cannot abort) and then sets

$n.end = (2, x_2)$ for $x_2 = x_0 \oplus F_1(x_1)$ (or aborts if $(2, x_2)$ is already defined or pending). During the above process, no new node is created. Hence if an outer node n' is associated to the permutation query $T(x_0, x_1) = (x_{10}, x_{11})$, it must be created after MakeNodeReady(n) successfully returns.

Since $(9, x_9)$ is defined and the *beginning* of a newly created node is pending, we must have $n'.beginning = (2, x_2)$. Then $n.end = n'.beginning$, which implies $n'.parent = n$ by Lemma 9. However, $n'.id = (1, x_1, x_2)$ and the maximal path of n contains both queries $(1, x_1)$, and $(2, x_2)$, contradicting property (ii) of Lemma 23.

Therefore, the permutation queries associated to the outer nodes must be added to T by a distinguisher query. Since the distinguisher makes at most q permutation queries, at most q outer nodes are created. □

Lemma 52. *In an execution of* G$_3$*, the assertions in procedures* ReadTape *and* Adapt *always hold.*

Proof. In a call to ReadTape made in F, the assertion always holds because F calls ReadTape only if $x \notin F_i$.

ReadTape is also called in SampleTree(n), where $n.end$ is sampled. By Lemma 11, the query $n.end$ is not defined when SampleTree(n) is called. ReadTape is called at the beginning of SampleTree, therefore the query is not defined when ReadTape is called.

Now consider Adapt(i, x_i, y_i) called in AdaptNode(n). The arguments of the call are determined in SampleTree(r), where r is the root of the tree containing n. We will give the proof for $i = 3$, with the proof for other positions being similar.

Let $\{x_h\}_{h=4}^{3}$ be the path associated to the node n, where $x_3 = F_2(x_2) \oplus x_1$. When $F_2(x_2) = f_2(x_2)$ is sampled in SampleTree(r), the query $(1, x_1)$ is already defined. If $(3, x_1 \oplus f_2(x_2))$ is defined at this point, BadRHit occurs and the simulator aborts. Thus the query $(3, x_3)$ is not defined when SampleTree(r) is called.

No query in position 3 gets defined during SampleTree(r). By Lemma 46 and since $\mathcal{A}$ contains all queries to be adapted in AdaptTree(r), the query $(3, x_3)$ is not adapted between the time SampleTree(r) returns and Adapt(i, x_i, y_i) is called. Therefore, the query $(3, x_3)$ is not defined when Adapt($3, x_3, y_3$) is called, i.e., the assertion must hold. □

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 three 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 53. *In* G_3 *or* G_4*, if* $x_5 \in F_5$ *and* $x_6 \in F_6$*, there exists a node* n *whose maximal path contains* x_5 *and* x_6*.*

Similarly, if $x_1 \in F_1$*,* $x_2 \in F_2$*,* $x_9 \in F_9$ *and* $x_{10} \in F_{10}$ *such that* $T(x_0, x_1) = (x_{10}, x_{11})$ *for* $x_0 = F_1(x_1) \oplus x_2$ *and* $x_{11} = x_9 \oplus F_{10}(x_{10})$*, there exists a node* n *whose maximal path contains* x_1*,* x_2*,* x_9 *and* x_{10}*.*

Proof. By Lemma 10, $(5, x_5)$ becomes defined in a call SampleTree(n) where n is a node such that $n.end = (5, x_5)$ and, likewise, $(6, x_6)$ becomes defined in a call SampleTree(m) where m is a node such that $m.end = (6, x_6)$. Let r be the root of the tree containing n and let s be the root of the tree containing m. Since n is in a $(2, 5)$-tree and m is in a $(6, 9)$-tree, $r \neq s$. Thus we can assume wlog that SampleTree(s) is called before SampleTree(r). In particular, since calls to SampleTree aren't nested, SampleTree(s) has already returned when SampleTree(r) is called.

Now consider the last call to GrowSubTree(r) in GrowTree(r). The calls to FindNewChildren don't add any new nodes during this call, otherwise the variable *modified* will be set to **true**, and it wouldn't be the last call to GrowSubTree(r). Therefore, no change (to any data structure) is made during the last call to GrowSubTree(r). In particular, when the call occurs, n is already in the tree rooted at r and SampleTree(s) has already been called and returned.

When FindNewChildren(n) is called during the last call to GrowSubTree(r), and since no children are added, when $x_6 \in F_6$ is checked we must have either Equivalent($n.id$, $(5, x_5, x_6)$) = **true** or NotChild(n, $(5, x_5, x_6)$) = **false**. In both cases, a node n' (either $n' = n$ or $n' \in n.children$) exists in the tree rooted at r such that the maximal path of n' contains $(5, x_5, x_6)$. Therefore, a node exists whose maximal path contains $(5, x_5, x_6)$.

Since the lemma considers a point in time when SampleTree(r) and AdaptTree(r) have returned (as otherwise the simulator is in a completion phase), a call AdaptNode(n') has occurred and returned in AdaptTree(r), so the queries in the maximal path of n' form a completed path by Lemma 22. Since the maximal path of n' contains $(5, x_5, x_6)$, this completes the proof for this case.

Now consider the second statement. We first prove that the last query being defined is either $(2, x_2)$ or $(9, x_9)$. By contradiction, assume that $(1, x_1)$ is defined after the other three queries (the case for $(10, x_{10})$ is symmetric). Note that $(1, x_1)$ can only be sampled in a call to ReadTape$(1, x_1)$. When ReadTape$(1, x_1)$ is called, the query $T(x_0, x_1)$ mustn't have been added to T, otherwise BadRHit occurs since $(0, x_0, x_1)$ is left active and $(1, x_1, x_2)$ is right active. In this case, however, BadP occurs when $T(x_0, x_1) = (x_{10}, x_{11})$ is added in a call to $P(x_0, x_1)$ or $P^{-1}(x_{10}, x_{11})$, since $x_{10} \in F_{10}$ and $x_1 \in F_1$ at the moment.

The rest of the proof is similar to the first statement, and we only give a sketch here. By Lemma 10, $(2, x_2)$ and $(9, x_9)$ become respectively defined in calls SampleTree(r) and SampleTree(s) where $r \neq s$ are the roots of $(2, 5)$- and $(6, 9)$-trees respectively. Wlog, we assume that SampleTree(s) is called (and returns) before SampleTree(r). Then the queries $(1, x_1)$, $(9, x_9)$ and $(10, x_{10})$ are defined

when the last call to GrowSubTree(r) occurs. Since no new node is added during this call, there is a node n' in the tree rooted at r whose maximal path contains $(1, x_1, x_2)$. ☐

(Subsequent lemmas of this section only relate to G_3.)

Lemma 54. *The following statements hold if the simulator is not in a completion phase.*

If $x_5 \in F_5$ and $x_6 \in F_6$, there exists a completed path $\{x_h\}_{h=0}^{11}$ containing x_5 and x_6.

Similarly, if $x_1 \in F_1$, $x_2 \in F_2$, $x_9 \in F_9$ and $x_{10} \in F_{10}$ such that $T(x_0, x_1) = (x_{10}, x_{11})$ for $x_0 = F_1(x_1) \oplus x_2$ and $x_{11} = x_9 \oplus F_{10}(x_{10})$, there exists a completed path $\{x_h\}_{h=0}^{11}$ containing x_1, x_2, x_9 and x_{10}.

Proof. Consider the first statement. By Lemma 53, there exists a node n whose maximal path contains x_5 and x_6. We claim that either $n.beginning$ or $n.end$ is in $\{(5, x_5), (6, x_6)\}$, depending on the origin of n. In all cases, one of the two queries is the *end* of n or n's parent, thus is the *end* of a node in the tree containing n. Let r denote the root of the tree containing n. Since the queries are both defined and by Lemma 11, SampleTree(r) must has been called; and since the lemma considers a point when SampleTree(r) and AdaptTree(r) have returned (as otherwise the simulator is in a completion phase), a call AdaptNode(n) has occurred and returned in AdaptTree(r), so the queries in the maximal path of n form a completed path by Lemma 22. Since the maximal path of n contains $(5, x_5, x_6)$, this completes the proof in this case.

The second statement is proved in a similar manner. There exists a node containing queries $(9, x_9)$ and $(2, x_2)$, and one of the queries must be the *end* of either n or $n.parent$. Let r be the root of the tree containing n. Since the queries are defined, calls to SampleTree(n) and to AdaptTree(n) have returned. Therefore the queries in the maximal path of n form a completed path, which contains the four queries required by the problem. ☐

Lemma 55. *The first assertion in* MakeNodeReady *always holds.*

Proof. The assertions in MakeNodeReady(n) occur right before $n.end$ is assigned. As in the pseudocode, let (j, x_j) be the query that is about to be assigned to $n.end$. Thus j is the terminal of n.

First consider $j = 2$. The origin of n is 5 and $F(9, x_9)$, $F(10, x_{10})$ and $F(1, x_1)$ are called in MakeNodeReady(n). By Lemma 19, the queries $(9, x_9)$, $(10, x_{10})$, $(1, x_1)$ and $(2, x_2)$ are in the maximal path of n. If $x_2 \in F_2$, then all four queries are defined. By Lemma 54 (MakeNodeReady is only called in the construction phase), they are contained in a completed path. By extension, the completed path also contains $(5, x_5)$ in the maximal path of n, which equals $n.beginning = n.parent.end$. However, since SampleTree hasn't been called on nodes in the tree containing n and $n.parent$, the query $n.parent.end$ is not defined by Lemma 11. This contradicts the definition of a completed path, in which each query should be defined.

The case where $j = 5$ can be proved similarly: if $x_5 \in F_5$, since $x_6 \in F_6$ and by Lemma 54, the queries $(5, x_5)$ and $(6, x_6)$ are in a completed path when the first assertion of MakeNodeReady is reached. The completed path also contains $n.beginning$, which is not defined at the moment by Lemma 11, leading to a contradiction.

The cases $j = 6, 9$ are symmetric to the above cases. ☐

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

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

Definition 18. 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 19. A node n is *is in the tree stack* if n is in a tree $\mathcal{T}_i$ in the tree stack.

Lemma 56. *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 11 the *end* of the nodes in the tree stack are not defined. □

Lemma 57. *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.

In the following discussion, we will focus on a point in time when 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 $\mathcal{T}_i$, $1 \leq i \leq \ell$.

Definition 20. An (i,j)-partial path $\{x_h\}_{h=i}^{j}$ is *proper* if $(i,j) \in \{(2,6)$, $(5,9)$, $(5,2)$, $(6,2)$, $(9,5)$, $(9,6)\}$ and $x_i \notin F_i$, $x_j \notin F_j$. Moreover, a proper (i,j)-partial path is a *proper outer partial path* if $i > j$, and is a *proper inner partial path* if $i < j$.

(Proper partial paths were already defined in Section 4, but the rest Definition 20 is new.) Observe that a proper inner partial path must be a $(2,6)$-partial path or a $(5,9)$-partial path.

Lemma 58. *A 2chain is contained in at most one proper 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 $x_{k+1}, \ldots, x_j$ (where, as usual, x_{11} is followed by x_0) is uniquely determined by (k, x_k, x_{k+1}) and j, and the sequence $x_k, x_{k-1}, \ldots, x_i$ (where, as usual, x_0 is followed by x_{11}) is uniquely determined by (k, x_k, x_{k+1}) and i. Also, we have $F_h(x_h) \neq \perp$ for $h \neq i, j, 0, 11$ by the definition of a partial path. The proper 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 21. The queries (i, x_i) and (j, x_j) are called the *endpoint queries* of the partial path $\{x_h\}_{h=i}^{j}$.

Definition 22. 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 23. A *path cycle* is a sequence of oriented proper partial paths $((P_1, \sigma_1), \ldots, (P_t, \sigma_t))$, $t \geq 2$, such that:

1. Adjacent paths in the cycle are distinct, i.e., $P_s \neq P_{s+1}$ for all $1 \leq s \leq t$, where $(P_{t+1}, \sigma_{t+1}) := (P_1, \sigma_1)$.
2. The ending point of (P_s, σ_s) is the starting point of (P_{s+1}, σ_{s+1}) for $1 \leq s \leq t$.
3. Not both P_s and P_{s+1} are inner proper partial paths for $1 \leq s \leq t$.

Next we prove that if abortion occurs in the second assertion in MakeNodeReady or in the assertion in F, there must exists a path cycle in the sense of Definition 23.

Lemma 59. *For $i < \ell$, if the origin of n_i is 2 (resp. 5, 6, 9), then the position of $r_{i+1}.end$ is 6 (resp. 9, 2, 5). Moreover the endpoint queries of the maximal path of n_i are $n_i.beginning$ and $r_{i+1}.end$.*

Proof. As per the discussion in Lemma 32, when the origin of n is 2 (resp. 5, 6, 9) the only call $\mathrm{F}(j, x_j)$ made in MakeNodeReady(n) that can create a new tree is the one with $j = 6$ (resp. 9, 2, 5). The reason is that the other calls are either in positions other than 2, 5, 6 or 9 or else the corresponding query has already been defined.

By Lemma 57, $\mathcal{T}_{i+1}$ is created during the call to MakeNodeReady(n_i). Therefore $r_{i+1}.end = (j, x_j)$ where j is determined by the origin of n_i as above. The fact that (j, x_j) is in the maximal path of n_i follows by Lemma 19. Since SampleTree(r_{i+1}) hasn't been called by definition of the tree stack, $r_{i+1}.end$ is pending by Lemma 11 and, being undefined, must be an endpoint of the maximal path of n_i.

Finally, the fact that $n_i.beginning$ is also an endpoint of the maximal path follows directly by Definition 10. □

Lemma 60. *If $\mathcal{T}_i$ is a $(2,5)$-tree (resp. $(6,9)$-tree) and $i < \ell$, then $\mathcal{T}_{i+1}$ is a $(6,9)$-tree (resp. $(2,5)$-tree).*

Proof. This is a direct consequence of the first part of Lemma 59. □

Lemma 61. *When the simulator aborts in a call to* $\mathrm{F}(i, x_i)$ *by* MakeNodeReady(n), *we have $n = n_\ell$ and if the origin of n_ℓ is 2 (resp. 5, 6, 9) then $i = 6$ (resp. 9, 2, 5).*

Proof. The first statement 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.

The second statement is proved similarly to Lemma 59: Abortion occurs only if (i, x_i) is pending, which implies that $i \in \{2, 5, 6, 9\}$ and that (i, x_i) is not defined when n is created. With the same argument as in the proof for Lemma 32, when the origin of n is 2 (resp. 5, 6, 9) only the call with $i = 6$ (resp. 9, 2, 5) may abort. □

Lemma 62. *If the second assertion in* MakeNodeReady *or the assertion in* F *fails, then the maximal path of each non-root node in the tree stack is a proper partial path. Moreover, the endpoint queries of the proper partial path 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 56, 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 statement follows by Lemmas 19 and 45 because $n.end$ and $n.beginning$ are both pending. Thus we can assume that n is not ready, in which case n is the non-ready leaf of a tree in the tree stack.

If $n = n_k$ for $k < \ell$ then the statement follows by Lemma 59 and because $n_k.beginning$ and $r_{k+1}.end$ are both pending.

If $n = n_\ell$ then the abortion occurs during the call to MakeNodeReady(n) (possibly within a subcall to F). If the second assertion in MakeNodeReady(n) fails, then 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.

Otherwise the abortion occurs in a call $\text{F}(h, x_h)$ made by MakeNodeReady(n), and in which case (h, x_h) is necessarily pending. The statement then follows by Lemma 19 (which tells us that (h, x_h) is in the maximal path of n) and Lemma 61 (which tells us the value of h). □

Lemma 63. *If a node's maximal path is an inner path, then the node has origin 5 or 6 and moreover the node is not ready.*

Proof. Recall that an inner path must be a proper $(2, 6)$- or $(5, 9)$-partial path. 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 2 or 9 is created, its maximal path contains $n.id$, which includes a query in position 1 or 9. But inner paths don't contain queries in position 1 or 9, by definition, which establishes the first part of the statement. □

Lemma 64. *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. Observe that the maximal path of a ready non-root node n has length 10 (and the two endpoint queries are $n.beginning$ and $n.end$), while the maximal path of a non-ready leaf $n_i \neq n_\ell$ is strictly shorter. Therefore, we consider the following two cases.

If the maximal paths of both m_1 and m_2 have length 10, then at least one of them is ready (since the only possible non-ready node with maximal path of length 10 is n_ℓ), and we assume wlog that m_1 is ready. If $m_1.beginning = m_2.beginning$, then their parents have the same *end* and by Lemma 7, we have $m_1.parent = m_2.parent$. By statement (i) of Lemma 23, the *id* of sibling nodes must contain at least one different query, and hence their maximal paths are different. If $m_1.beginning \neq m_2.beginning$ then we must have $m_2.beginning = m_1.end$. By Lemma 9, m_1 is the parent of m_2; then by statement (ii) of Lemma 23, the maximal paths of m_1 and m_2 are not identical.

Now we consider the case where both m_1 and m_2 are of length strictly shorter than 10. Then neither m_1 nor m_2 is ready, i.e., they must be the non-ready leaves: let $m_1 = n_i$ and $m_2 = n_j$ for $1 \leq i < j \leq \ell$. In particular, if $j = \ell$, the abortion must occur in a call to F, otherwise the maximal path of $m_2 = n_\ell$ has length 10. We note that if two partial paths are identical, their endpoints must be identical[12]. By Lemmas 59 and 61, the endpoints of the maximal paths of n_i and n_j are determined[13] 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 61 are the same as in Lemma 59).

Since n_i and n_j are in different trees, they must have different parent nodes, and by Lemma 7 we have $n_i.beginning \neq n_j.beginning$. By Lemma 59 and since $i < \ell$, the endpoint queries of the maximal path of n_i are $n_i.beginning$ and $r_{i+1}.end$, so we must have $r_{i+1}.end = n_j.beginning$. By Lemma 59 again, however, this implies that n_j does not have the same origin as n_i. As remarked above, however, this implies that the maximal paths of n_i and n_j have different endpoints, and hence cannot be the same. □

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

[13] In more detail, Lemmas 59 and 61 imply that if the origin of $n \in \{n_i, n_j\}$ is 2, 5, 6, 9 respectively, then the maximal path of n is a $(6, 2)$-, $(5, 9)$-, $(2, 6)$- and $(9, 5)$-partial path, respectively.

Lemma 65. *If the second assertion in* MakeNodeReady *or the assertion in* F *fails, there exists a path cycle.*

Proof. If F is called by the distinguisher, since there is no pending query at the moment, the assertion trivially holds. We only need to consider calls to F issued by MakeNodeReady in the following proof.

As usual, let $(\mathcal{T}_1, \dots, \mathcal{T}_\ell)$ 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 $i = 1, \dots, \ell$. Then the abortion occurs in MakeNodeReady(n_ℓ) or in a call to F made by MakeNodeReady(n_ℓ), as discussed before Lemma 59.

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 62. 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 $n_\ell.\mathit{beginning}$). By Lemma 56 there exists a node n' in the tree stack such that $n'.\mathit{end} = (h, x_h)$. 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 $n.\mathit{beginning}$;
- Let n^- denote the *negative oriented path* of n, whose partial path equals the maximal path of n and whose ending point equals $n.\mathit{beginning}$.

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

$$(\tau_\ell^-, \tau_{\ell-1}^-, \dots, \tau_{k+1}^-, \gamma_1^-, \gamma_2^+) \tag{6}$$

is a path cycle.

Each oriented path in (6) contains the maximal path of a non-root node n in the tree stack. By Lemma 62, these maximal paths are proper 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 64, the maximal paths of non-root nodes in the tree stack are distinct. Since each node appears in (6) at most once, the partial paths in the cycle are distinct and property 1 of Definition 23 holds. In the following we therefore focus on the remaining two properties.

Let $t \geq 2$ be the length of (6). Let $R_s = (P_s, \sigma_s)$ and $R_{s+1} = (P_{s+1}, \sigma_{s+1})$ be adjacent oriented paths in (6), 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}). Only ready nodes have children, so m_{s+1} is ready and P_{s+1} is an outer path by Lemma 63, giving property 3 of Definition 23. 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 $m_s.\mathit{beginning} = m_{s+1}.\mathit{end}$.

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, $m_{s+1} = n_{i-1}$ 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 59, the starting point of n_{i-1}^- is $r_i.\mathit{end} = m_s.\mathit{beginning}$. This establishes property 2 of a path cycle.

If the origin of $m_{s+1} = n_{i-1}$ is 2 (resp. 5, 6, 9), the position of $r_i.\mathit{end} = m_s.\mathit{beginning}$ is 6 (resp. 9, 2, 5). Either way at most one of the origins of m_s, m_{s+1} is 2 or 9, thus at most one of P_s and P_{s+1} is an inner path by Lemma 63.

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 points 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 63, P_{s+1} is an outer path.

On the other hand, if γ_2 is empty, then m_s is the last node of (6) and $m_{s+1} = m_1 = n_\ell$ and $n_{\text{top}} = n'$. The ending point of m_s^- is $m_s.\mathit{beginning} = n'.\mathit{end} = (h, x_h)$, 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 $n_\ell.\mathit{beginning}$).

Next we prove that the maximal paths of m_s and n_ℓ can't both be inner paths. By Lemma 20, the paths are inner paths only if both m_s and n_ℓ have origins 5 or 6. If n_ℓ has origin 5 or 6, 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 2 or 9, i.e., the origin of m_s is $h \in \{2, 9\}$. Thus, m_s cannot be an inner 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 $n'.\mathit{end} = (h, x_h)$, which is also the starting point of n_ℓ^-. Since n' is ready, its maximal path is an outer path by Lemma 63 and property 3 holds. $\square$

Next we will prove that 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 a path cycle, provided that a path cycle didn't previously exist.

Lemma 66. *In an execution of* G_3*, no path 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}.

Suppose an entry $T(x_0, x_1) = (x_{10}, x_{11})$ is added in a call to P. We must have $x_{10} \notin F_{10}$, otherwise CheckBadP aborts and the entry is not added. If a path cycle is created, one of the proper partial paths in the cycle must be an outer proper partial path that uses the new permutation query. However, since $x_{10} \notin F_{10}$, this contradicts Definition 20, which does not allow endpoints at position 10. □

Lemma 67. *In an execution of* G_3*, no path cycle is created during a call to* ReadTape*.*

Proof. Consider a call $\mathrm{ReadTape}(i, x_i)$. If a path cycle is created, at least one of the proper partial paths in the cycle contains the query (i, x_i). Let $\{x_h\}_{h=s}^{t}$ denote a proper partial path containing 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), so BadRHit would occur (and ReadTape would abort) before $F_i(x_i)$ becomes defined.

Without loss of generality, assume $i-1$ is an endpoint of the path, i.e., $s = i-1$. Because the length of a proper partial path is at least 5, $i+1$ 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 58, 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 $\mathrm{ReadTape}(i, x_i)$ is called. The endpoint query of a proper partial path is incident with an active 2chain. Indeed, it can be checked from the definition that a proper partial path must contain an active 2chain, which is incident with the endpoint queries of the path. However, BadRCollide occurs when $\mathrm{ReadTape}(i, x_i)$ 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 a pre-existing active 2chain. □

Finally we are left with the Adapt procedure. We will not prove the result for each individual adaptation; instead, we will consider the adaptations that occur in a call to $\mathrm{AdaptTree}(r)$ all at once, where r is a root node.

In the following discussion, we will use the same notations and shorthands as in Definition 17. E.g., $\mathcal{A}$ denotes the set of adapted queries in AdaptTree (constructed by GetAdapts), while $\{x_h\}$ denotes the partial path associated to a node, which either has the form $\{x_h\}_{h=4}^{3}$ (for $(2,5)$-trees) or $\{x_h\}_{h=8}^{7}$ (for $(6,9)$-trees).

We start by giving a useful observation about adapted queries.

Lemma 68. *When* $\mathrm{AdaptTree}(r)$ *is called, for every adapted query* $(i, x_i, y_i) \in \mathcal{A}$ *adapted in path* $\{x_h\}$*, there does not exist a pair* $(x'_{i+1}, x'_{i+2}) \in \tilde{F}_{i+1} \times \tilde{F}_{i+2}$*, such that* $x'_{i+1} \neq x_{i+1}$ *and such that* $x_i = F_{i+1}(x'_{i+1}) \oplus x'_{i+2}$*. The statement also holds if we replace all "+"s with "−"s.*

Proof. Assume by contradiction that a pair $(x'_{i+1}, x'_{i+2}) \in \tilde{F}_{i+1} \times \tilde{F}_{i+2}$ satisfying the properties exist.

Note that only pending queries become defined in $\mathrm{SampleTree}(r)$, so $\mathrm{SampleTree}(r)$ does not change any of the tables $\tilde{F}_h$. Thus the queries $(i+1, x'_{i+1})$ and $(i+2, x'_{i+2})$ are already defined or pending when $\mathrm{SampleTree}(r)$ is called. Note that $x_i = F_{i\pm1}(x_{i\pm1}) \oplus x_{i\pm2}$, where "±" is "+" when $i \in \{4, 8\}$ and is "−" when $i \in \{3, 7\}$, and where $F_{i\pm1}(x_{i\pm1})$ is sampled in $\mathrm{SampleTree}(r)$ while $(i \pm 2, x_{i\pm2})$ is already defined when $\mathrm{SampleTree}(r)$ is called.

If $(i+1, x'_{i+1})$ is defined when $\mathrm{ReadTape}(i\pm1, x_{i\pm1})$ is called, then the query $(i, x_i) = (i, F_{i\pm1}(x_{i\pm1}) \oplus x_{i\pm2})$ is incident with an active 2chain $(i+1, x'_{i+1}, x'_{i+2})$ and BadRCollide occurs. Otherwise if

$(i+1, x'_{i+1})$ is not defined when ReadTape$(i \pm 1, x_{i\pm1})$ is called, since it is defined when AdaptTree(r) is called, it is also sampled in SampleTree(r) after ReadTape$(i \pm 1, x_{i\pm1})$ returns. Thus when ReadTape$(i+1, x'_{i+1})$ is called, the query $(i, x_i) = (i, F_{i+1}(x'_{i+1}) \oplus x'_{i+2})$ is incident with an active 2chain $(i \pm 1, x_{i\pm1}, x_{i\pm2})$ and BadRCollide occurs.

Therefore no matter which one of $(i+1, x'_{i+1})$ and $(i \pm 1, x_{i\pm1})$ is sampled first, BadRCollide occurs when the other is sampled. Thus such x'_{i+1} and x'_{i+2} cannot exist when AdaptTree(r) is called.

The proof is the same if the "+"s are replaced with "−"s. □

Lemma 69. *In an execution of* G$_3$*, if no path cycle has existed before a call to* AdaptTree*, no path cycle is created during the call.*

Proof. Consider a call to AdaptTree(r) where r is a root node. If a path cycle is created in the call, one of the proper partial paths in the cycle must contain an adapted query. Let $(i, x_i, y_i) \in \mathcal{A}$ be such a query and let the query be adapted in the path $\{x_h\}$. Without loss of generality, assume r is the root of a $(2,5)$-tree. Then queries in $\mathcal{A}$ are in position 3 or 4, and in particular $i \in \{3, 4\}$.

Let $P = \{x'_h\}$ be the proper path in the cycle that contains x_i, i.e., $x'_i = x_i$. First we claim that P does not contain two consecutive queries in a path being completed: Otherwise, by extending the path in the feistel way, we can prove that

If P also contains x_{i-1} or x_{i+1}, it contains two consecutive queries in the path $\{x_h\}$ being completed. Then the queries in P are also in the path being completed; but since SampleTree has been called on the path, queries in positions 2, 5, 6 and 9 of the path are all defined. Since P is proper, its endpoints must be in these positions and so its endpoint queries are defined, contradicting the definition of a proper partial path. Thus we have $x'_{i-1} \neq x_{i-1}$ and $x'_{i+1} \neq x_{i+1}$ (if the positions are in the path P).

If P does not contain another query adapted in AdaptTree(r), then the other queries in P have been defined when AdaptTree(r) is called. If P contains a query in position $i-2$, the query x'_{i-2} is not active since otherwise $x'_{i-1} \neq x_{i-1}$ and $F_{i-1}(x'_{i-1}) = x'_{i-2} \oplus x_i$, violating Lemma 68. Similarly, if x'_{i+2} exists in the path, $x'_{i+2} \notin \tilde{F}_{i+2}$. Then P contains at most three defined queries. Since a proper partial path contains at least three defined queries (which is easy to check by definition), P must contain three defined queries $(i-1, x'_{i-1})$, (i, x_i) and $(i+1, x'_{i+1})$. The queries $(i-1, x'_{i-1})$ and $(i+1, x'_{i+1})$ are defined when AdaptTree(r) is called and $y_i = x'_{i-1} \oplus x'_{i+1}$, so BadAHit occurs and simulator aborts before AdaptTree(r) is called.

Therefore, P must contain two adapted queries. Similarly, the two queries cannot be adapted in the same path, because otherwise P contains two consecutive queries in a path being completed. Let the two adapted queries be $(3, x_3, y_3) \in \mathcal{A}$ and $(4, u_4, v_4) \in \mathcal{A}$, adapted in paths $\{x_h\}$ and $\{u_h\}$ respectively. Note that the defined queries in P other than $(3, x_3)$ and $(4, u_4)$ are defined before AdaptTree is called. As before, if P contains x'_1 or x'_6, they must not be active, or Lemma 68 is violated for $(3, x_3)$ and $(4, u_4)$ respectively. Thus P only contains queries in positions 1 through 6, and since it is a proper partial path, it can only be a proper $(2,6)$-partial path. This proves that if a proper partial path contains a query in $\mathcal{A}$, it must be an inner path.

Moreover, we have $x'_5 \in F_5$ since $(5, x'_5)$ is not an endpoint query. This should hold when AdaptTree(r) is called since no queries in position 5 become defined in AdaptTree. Note that $x'_5 = x_3 \oplus v_4$, then if $x_2 \neq u_2$, BadAPair occurs for the pair $(3, x_3, y_3)$ and $(4, u_4, v_4)$ and the simulator aborts before AdaptTree(r) is called. Thus we must have $x_2 = u_2$.

Now consider the path adjacent to P in the cycle that also contains the endpoint query $(2, x'_2)$. Let $\tilde{P} = \{x''_h\}$ denote the path, then $x''_2 = x'_2$ and the path is a proper partial path. The path $\tilde{P}$ cannot contain a query in $\mathcal{A}$, otherwise it is also an inner path (as proved above), violating property 3 of Definition 23. This implies that the non-endpoint queries in $\tilde{P}$ are defined when AdaptTree(r) is called, i.e, the proper partial path $\tilde{P}$ exists at that moment. As discussed in the proof for Lemma 67, the endpoint query of a proper partial path is incident with an active 2chain. Thus $(2, x'_2)$ is incident with an active 2chain when AdaptTree(r) is called. Since $x'_2 = y_3 \oplus u_4$, BadAPair occurs for the pair $(3, x_3, y_3)$ and $(4, u_4, v_4)$ if $x_5 \neq u_5$. Hence AdaptTree(r) can be called only when $x_5 = u_5$.

From the above discussion, we have $x_2 = u_2$ and $x_5 = u_5$. However, when SampleTree(r) is called, both $\{x_h\}$ and $\{u_h\}$ are proper $(5, 2)$-partial paths, and $(\{x_h\}, +)$ and $(\{u_h\}, -)$ form a path cycle of length 2. This contradicts the assumption that no path cycle has existed before AdaptTree(r) is called! □

Lemma 70. *The simulator does not abort in good executions of* G_3.

Proof. In good executions, bad events don't occur and the simulator doesn't abort in CheckBadP, CheckBadR or CheckBadA.

By Lemmas 66, 67 and 69, the first query cycle cannot be created in P, P^{-1}, ReadTape or Adapt, which are the only procedures that modify the tables. This implies that no query cycle exists in any execution of G_3. By Lemma 65 the assertion in F and the second assertion in MakeNodeReady never fail. Moreover, by Lemmas 51, 52 and 55, the other assertions don't fail, either.

Therefore, no abortion occurs in good executions of G_3. □

Lemma 71. *The probability that an execution of* G_3 *aborts is at most* $22992q^8/2^n$.

Proof. This directly follows by Lemmas 50 and 70. □

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 $\Delta_D(G_3, G_4)$ and $\Delta_D(G_4, G_5)$, and use a triangle inequality to complete the transition. Our upper bound on $\Delta_D(G_3, G_4)$ holds only if D completes all paths (see Definition 1), which means that our final upper bound on $\Delta_D(G_1, G_5)$ holds only if D completes all paths. However, an additional reduction (see Theorem 84) 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 [17], 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 [11] 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, $\dots, g_{10}$ (all of which are random oracle tapes), in contrast to the tapes $f_1, f_2, \dots, f_{10}$ 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 24. A *partial random tape* is a table $\tilde{f}$ of size 2^n such that $\tilde{f}(x) \in \{0,1\}^n \cup \{\bot\}$ 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 $\tilde{p}(u), \tilde{p}^{-1}(v) \in \{0,1\}^{2n} \cup \{\bot\}$ for all u, $v \in \{0,1\}^{2n}$, such that $\tilde{p}^{-1}(\tilde{p}(u)) = u$ for all u such that $\tilde{p}(u) \neq \bot$, and such that $\tilde{p}(\tilde{p}^{-1}(v)) = v$ for all v such that $\tilde{p}^{-1}(v) \neq \bot$.

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 $\bot$ 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 25. 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 $\tilde{f}_i(x) \neq \bot$. 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 $\tilde{p}(u) \neq \bot$. 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 26. Given an execution of G_3 with random tapes $f_1, f_2, \ldots, f_{10}, p$, the *footprint* of the execution is the set of partial tapes $\tilde{f}_1, \tilde{f}_2, \ldots, \tilde{f}_{10}, \tilde{p}$ 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 $g_1, g_2, \ldots, g_{10}$, the *footprint* is the set of partial tapes $\tilde{g}_1, \tilde{g}_2, \ldots, \tilde{g}_{10}$, with $\tilde{g}_i$ containing the entries of g_i that are accessed at some point during the G_4-execution.

Note that Definition 26 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 $f_1, \ldots, f_{10}, p$ in G_3 or $g_1, \ldots, g_{10}$ 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 $\Pr_{G_i}[\omega]$ 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 72. *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 $z = (f_1, f_2, \ldots, f_{10}, p)$ if $i = 3$, $z = (g_1, g_2, \ldots, g_{10})$ if $i = 4$.

The "only if" direction is trivial: If the footprint of the execution with tapes z is ω, then by definition, ω consists of partial tapes that are compatible with the tapes in z.

For the "if" direction, consider an arbitrary $\omega \in \mathsf{FP}_i$. There exist random tapes z' such that the execution of G_i with z' has footprint ω. During the execution with z', only entries in ω are read. If we run in parallel the executions of G_i with z and with z', 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 z' and z are identical and should have identical footprints. ☐

A corollary of Lemma 72 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 $\omega = (\tilde{f}_1, \ldots, \tilde{f}_{10}, \tilde{p}) \in \mathsf{FP}_3$ is

$$\left(\prod_{i=1}^{10} \frac{1}{2^{n|\tilde{f}_i|}}\right)\left(\prod_{\ell=0}^{|\tilde{p}|-1} \frac{1}{2^{2n} - \ell}\right) = \Pr_{\mathrm{G}_3}[\omega], \tag{7}$$

by elementary counting. Similarly, the probability that random tapes in G_4 are compatible with $\omega = (\tilde{g}_1, \ldots, \tilde{g}_{10}) \in \mathsf{FP}_4$ is

$$\prod_{i=1}^{10} \frac{1}{2^{n|\tilde{g}_i|}} = \Pr_{\mathrm{G}_4}[\omega]. \tag{8}$$

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 and answering tape queries using entries in ω. We say a footprint is *non-aborting* if the corresponding execution does not abort. Let $\mathsf{FP}_3^\star \subseteq \mathsf{FP}_3$ and $\mathsf{FP}_4^\star \subseteq \mathsf{FP}_4$ 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 $\zeta : \mathsf{FP}_3^\star \to \mathsf{FP}_4^\star$ 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 27. The injection $\zeta : \mathsf{FP}_3^\star \to \mathsf{FP}_4^\star$ is defined as follows: for $\omega = (\tilde{f}_1, \ldots, \tilde{f}_{10}, \tilde{p}) \in \mathsf{FP}_3^\star$, $\zeta(\omega) = (\tilde{g}_1, \ldots, \tilde{g}_{10})$ 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 $\zeta(\omega) \in \mathsf{FP}_4^\star$ (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 73. (*) *At the end of a non-aborting execution in* G_3 *or* G_4*, a permutation query* $T(x_0, x_1) = (x_{10}, x_{11})$ *exists in* T *if and only if there exists a non-root node whose maximal path contains* x_0*,* x_1*,* x_{10} *and* x_{11}*.*

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_{10} and x_{11}, then we have $T(x_0, x_1) = (x_{10}, x_{11})$ due to the definition of a completed path.

To prove the "only if" direction, consider an arbitrary entry $T(x_0, x_1) = (x_{10}, x_{11})$. 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_{10} and x_{11} 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 $i \in \{1, 2, \ldots, 6\}$, where $x_i := x_{i-2} \oplus \mathrm{F}(i-1, x_{i-1})$ for $2 \leq i \leq 6$. In particular, the queries $(5, x_5)$ and $(6, x_6)$ are defined before the end of the execution. By Lemma 53, there exists a node whose maximal path contains x_5 and x_6. The path also contains x_0 and x_1 (by definition of a completed path), as well as x_{10} and x_{11} (since $T(x_0, x_1) = (x_{10}, x_{11})$ 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 74. (*) *The executions of* G_3 *and* G_4*, with footprints* ω *and* $\zeta(\omega)$ *respectively, are identical.*

Proof. Let $\omega = (\tilde{f}_1, \ldots, \tilde{f}_{10}, \tilde{p}) \in \mathsf{FP}_3^\star$. 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 $\zeta(\omega) = (\tilde{g}_1, \ldots, \tilde{g}_{10})$ into a set of full random tapes $\lambda = (g_1, \ldots, g_{10})$. 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 $\omega \in \mathsf{FP}_3^\star$, 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 $T(x_0, x_1) = (x_{10}, x_{11})$ be the new entry added by the call to P or P^{-1} in the G_3-execution. By Lemma 73, there exists a node whose maximal path contains x_0, x_1, x_{10}, x_{11}. By Lemma 22, the queries are in a completed path, which implies $\text{Val}^+(0, x_0, x_1, i) = x_i$ for $i = 10, 11$. 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 $T(x_0, x_1) = (x_{10}, x_{11})$.

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 $\zeta(\omega) \in \mathsf{FP}_4^\star$. □

Lemma 75. (*) *The mapping ζ defined in Definition 27 is an injection from $\mathsf{FP}_3^\star$ to $\mathsf{FP}_4^\star$.*

Proof. By Lemma 74, for any $\omega \in \mathsf{FP}_3^\star$, 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 $\zeta(\omega) \in \mathsf{FP}_4^\star$.

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.[14] Assume $\omega' \in \mathsf{FP}_3^\star$ 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 $\omega \in \mathsf{FP}_3^\star$, i.e., ζ is injective. □

Lemma 76. (*) *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 73, 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. □

[14] 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.

Lemma 77. (*) *Let* $\omega = (\tilde{f}_1, \ldots, \tilde{f}_{10}, \tilde{p}) \in \mathsf{FP}_3^\star$ *and* $\zeta(\omega) = (\tilde{g}_1, \ldots, \tilde{g}_{10}) \in \mathsf{FP}_4^\star$. *Then*

$$\sum_{i=1}^{10} |\tilde{g}_i| = \sum_{i=1}^{10} |\tilde{f}_i| + 2 \cdot |\tilde{p}|.$$

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 $|F_i|$.

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 76, 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 $T(x_0, x_1) = (x_{10}, x_{11})$ exists if and only if $p(x_0, x_1) = (x_{10}, x_{11})$ 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

$$\sum_{i=1}^{10} |\tilde{g}_i| = \sum_{i=1}^{10} |F_i| = \sum_{i=1}^{10} |F_i^R| + |F^A| = \sum_{i=1}^{10} |\tilde{f}_i| + 2 \cdot |\tilde{p}|.$$

□

Lemma 78. (*) *For every* $\omega \in \mathsf{FP}_3^\star$, *we have*

$$\Pr_{\mathrm{G}_4}[\zeta(\omega)] \geq \Pr_{\mathrm{G}_3}[\omega] \cdot (1 - 25q^4/2^{2n})$$

Proof. Let $\omega = (\tilde{f}_1, \ldots, \tilde{f}_{10}, \tilde{p}) \in \mathsf{FP}_3^\star$, then by Lemma 75, $\zeta(\omega) = (\tilde{g}_1, \ldots, \tilde{g}_{10}) \in \mathsf{FP}_4^\star$. By equations (7) and (8), we have

$$\begin{aligned}
\Pr_{\mathrm{G}_4}[\zeta(\omega)] / \Pr_{\mathrm{G}_3}[\omega] &= 2^{-n \sum |\tilde{g}_i|} \Big/ \left(2^{-n \sum |\tilde{f}_i|} \cdot \prod_{\ell=0}^{|\tilde{p}|-1} \frac{1}{2^{2n} - \ell} \right) \\
&= 2^{-n(\sum |\tilde{f}_i| + 2|\tilde{p}|)} \cdot 2^{n \sum |\tilde{f}_i|} \cdot \prod_{\ell=0}^{|\tilde{p}|-1} (2^{2n} - \ell) \\
&= 2^{-2n \cdot |\tilde{p}|} \cdot \prod_{\ell=0}^{|\tilde{p}|-1} (2^{2n} - \ell) \\
&\geq \left(\frac{2^{2n} - |\tilde{p}|}{2^{2n}} \right)^{|\tilde{p}|}
\end{aligned} \tag{9}$$

where the second equality uses Lemma 77.

Note that each entry in $\tilde{p}$ corresponds to a distinct permutation query in T. By Lemma 29, we have $|T| \leq 5q^2$, so $|\tilde{p}| \leq 5q^2$. Since (9) is monotone decreasing with respect to $|\tilde{p}|$, we have

$$\left(\frac{2^{2n} - |\tilde{p}|}{2^{2n}}\right)^{|\tilde{p}|} \geq \left(\frac{2^{2n} - 5q^2}{2^{2n}}\right)^{5q^2} \geq 1 - \frac{25q^4}{2^{2n}}$$

and the lemma follows. □

Lemma 79. (*) *We have*

$$\Delta_D(\mathrm{G}_3, \mathrm{G}_4) \leq \Pr_{\mathrm{G}_4}[\mathsf{FP}_4^\star] - \Pr_{\mathrm{G}_3}[\mathsf{FP}_3^\star] + \frac{25q^4}{2^{2n}}.$$

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 $\mathsf{FP}_3^\star$ (resp. $\mathsf{FP}_4^\star$). For $i \in \{3, 4\}$ we have

$$\Pr_{\mathrm{G}_i}[D^{\mathrm{F,P,P}^{-1}} = 1] = 1 - \Pr_{\mathrm{G}_i}[\mathsf{FP}_i^\star] + \sum_{\omega \in \mathsf{FP}_i^\star, D^{\mathrm{G}_i}(\omega)=1} \Pr_{\mathrm{G}_i}[\omega]. \tag{10}$$

By Lemma 74, executions with footprints ω and $\zeta(\omega)$ have the same output; by Lemma 75, ζ is injective. So $\zeta(\omega)$ is distinct for distinct ω and $\{\zeta(\omega) : \omega \in \mathsf{FP}_3^\star, D^{\mathrm{G}_3}(\omega) = 1\}$ is a subset of $\{\omega : \omega \in \mathsf{FP}_4^\star, D^{\mathrm{G}_4}(\omega) = 1\}$. Thus we have

$$\begin{aligned}\sum_{\omega \in \mathsf{FP}_4^\star, D^{\mathrm{G}_4}(\omega)=1} \Pr_{\mathrm{G}_4}[\omega] &\geq \sum_{\omega \in \mathsf{FP}_3^\star, D^{\mathrm{G}_3}(\omega)=1} \Pr_{\mathrm{G}_4}[\zeta(\omega)] \\ &\geq (1 - \frac{25q^4}{2^{2n}}) \sum_{\omega \in \mathsf{FP}_3^\star, D^{\mathrm{G}_3}(\omega)=1} \Pr_{\mathrm{G}_3}[\omega] \end{aligned} \tag{11}$$

where the second inequality is due to Lemma 78.

Furthermore, combining (3) and (10), we have

$$\begin{aligned}\Delta_D(\mathrm{G}_3, \mathrm{G}_4) &= \Pr_{\mathrm{G}_3}[D^{\mathrm{F,P,P}^{-1}} = 1] - \Pr_{\mathrm{G}_4}[D^{\mathrm{F,P,P}^{-1}} = 1] \\ &= \Pr_{\mathrm{G}_4}[\mathsf{FP}_4^\star] - \Pr_{\mathrm{G}_3}[\mathsf{FP}_3^\star] + \sum_{\omega \in \mathsf{FP}_3^\star, D^{\mathrm{G}_3}(\omega)=1} \Pr_{\mathrm{G}_3}[\omega] - \sum_{\omega \in \mathsf{FP}_4^\star, D^{\mathrm{G}_4}(\omega)=1} \Pr_{\mathrm{G}_4}[\omega] \\ &\leq \Pr_{\mathrm{G}_4}[\mathsf{FP}_4^\star] - \Pr_{\mathrm{G}_3}[\mathsf{FP}_3^\star] + \left(1 - \left(1 - \frac{25q^4}{2^{2n}}\right)\right) \sum_{\omega \in \mathsf{FP}_3^\star, D^{\mathrm{G}_3}(\omega)=1} \Pr_{\mathrm{G}_3}[\omega] \\ &\leq \Pr_{\mathrm{G}_4}[\mathsf{FP}_4^\star] - \Pr_{\mathrm{G}_3}[\mathsf{FP}_3^\star] + \frac{25q^4}{2^{2n}}\end{aligned}$$

where the first inequality follows by (11), 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 80. *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. Since the node n is ready when AdaptNode is called, its maximal path contains x_0, x_1, x_{10}, x_{11} such that $T(x_0, x_1) = (x_{10}, x_{11})$. The entry of T is added by P or P^{-1}, and from the pseudocode of G_4, we observe that there exists $x_2, x_3, \ldots, x_9$ such that $x_i = g_{i-1}(x_{i-1}) \oplus x_{i-2}$ for $i = 2, 3, \ldots, 11$. By the induction hypothesis, pre-existing queries in F_i are compatible with tapes g_i. Furthermore, because n is ready when the call to AdaptNode(n) occurs, the maximal path of n contains $x_0, x_1, \ldots, x_{11}$, 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 81. *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 80, 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 82. *We have*

$$\Delta_D(G_4, G_5) \le 1 - \Pr_{G_4}[\mathsf{FP}_4^\star].$$

Proof. By Lemma 81, if random tapes $g_1, \ldots, g_{10}$ 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 $1 - \Pr_{G_4}[\mathsf{FP}_4^\star]$. □

5.7 Concluding the Indifferentiability

Now we can put pieces together and conclude the indistinguishability of G_1 and G_5.

Lemma 83. (*) $\Delta_D(G_1, G_5) \le 23529q^8/2^n$.

Proof. Since the advantage satisfies the triangle inequality (which is in fact an equality under our definition of advantage, but nevermind), we have

$$\begin{aligned}
\Delta(\mathrm{G}_1,\mathrm{G}_5) &\leq \Delta(\mathrm{G}_1,\mathrm{G}_2) + \Delta(\mathrm{G}_2,\mathrm{G}_5)\\
&\leq \Delta(\mathrm{G}_1,\mathrm{G}_2) + \Delta(\mathrm{G}_3,\mathrm{G}_5)\\
&\leq \Delta(\mathrm{G}_1,\mathrm{G}_2) + \Delta(\mathrm{G}_3,\mathrm{G}_4) + \Delta(\mathrm{G}_4,\mathrm{G}_5)\\
&\leq \frac{512q^8}{2^{2n}} + (\Pr_{\mathrm{G}_4}[\mathsf{FP}_4^\star] - \Pr_{\mathrm{G}_3}[\mathsf{FP}_3^\star] + \frac{25q^4}{2^{2n}}) + (1 - \Pr_{\mathrm{G}_4}[\mathsf{FP}_4^\star])\\
&\leq \frac{537q^8}{2^{2n}} + 1 - \Pr_{\mathrm{G}_3}[\mathsf{FP}_3^\star]\\
&\leq \frac{537q^8}{2^{2n}} + \frac{22992q^8}{2^n}\\
&\leq \frac{23529q^8}{2^n}
\end{aligned}$$

where the second inequality is due to Lemma 36, the fourth inequality uses Lemmas 35, 79 and 82, and the second-to-last inequality is due to Lemma 71. □

As indicated, Lemma 83 only holds for q-query distinguishers D that complete all paths, as described at the start of Section 5. Our next (final) theorem drops this limitation.

Theorem 84. $\Delta_D(\mathrm{G}_1,\mathrm{G}_5) \leq 6023424q^8/2^n$.

Proof. Recall that D is an arbitrary, information-theoretic, deterministic distinguisher that makes at most q queries to each of its oracles.

As discussed in the proof overview, there exists a deterministic distinguisher D^* that makes at most $2q$ queries to each of its oracles, that completes all paths, and such that $\Delta_D(\mathrm{G}_1,\mathrm{G}_5) \leq \Delta_{D^*}(\mathrm{G}_1,\mathrm{G}_5)$. Then D^* fulfills the conditions of Lemma 83 with $2q$ substituted for q. Hence the theorem follows from the fact that $23529 \cdot 2^8 = 6023424$. □

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 [31], 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 [32], pages 1–20.
10. Yevgeniy Dodis and Prashant Puniya, On the Relation Between the Ideal Cipher and the Random Oracle Models. Proceedings of TCC 2006, 184–206.
11. Yevgeniy Dodis, Liu Tianren, John Steinberger and Martijn Stam, On the Indifferentiability of Confusion-Diffusion Networks, eprint archive report XXX.
12. 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.
13. 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.
14. Hörst Feistel. Cryptographic coding for data-bank privacy. IBM Technical Report RC-2827, March 18 1970.
15. 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.
16. 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.
17. 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.
18. Dana Dachman-Soled, Jonathan Katz and Aishwarya Thiruvengadam, 10-Round Feistel is Indifferentiable from an Ideal Cipher. IACR eprint archive, 2015.
19. 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.
20. M. Luby and C. Rackoff. How to construct pseudorandom permutations and pseudorandom functions. SIAM Journal on Computing, 17(2):373–386, April 1988.
21. 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.
22. M. Naor and O. Reingold, On the construction of pseudorandom permutations: Luby-Rackoff revisited, J. of Cryptology, 1999. Preliminary Version: STOC 1997.
23. Jacques Patarin, Security of balanced and unbalanced Feistel Schemes with Linear Non Equalities. Technical Report 2010/293, IACR eprint arxiv.
24. 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.
25. 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.
26. David Pointcheval and Jacques Stern, Security Proofs for Signature Schemes. EUROCRYPT 1996, LNCS 1070, pp. 387–398.
27. Thomas Ristenpart and Hovav Shacham and Thomas Shrimpton, Careful with Composition: Limitations of the Indifferentiability Framework. EUROCRYPT 2011, LNCS 6632, pp. 487–506.
28. Phillip Rogaway, Viet Tung Hoang, On Generalized Feistel Networks. EUROCRYPT 2010, LNCS 6223, pp. 613–630, Springer, 2010.
29. Yannick Seurin, *Primitives et protocoles cryptographiques à sécurité prouvée*. PhD thesis, Université de Versailles Saint-Quentin-en-Yvelines, France, 2009.
30. Yannick Seurin, A Note on the Indifferentiability of the 10-Round Feistel Construction. Available from `yannickseurin.free.fr/pubs/Seurin_note_ten_rounds.pdf`.
31. Nigel P. Smart, editor. *Advances in Cryptology - EUROCRYPT 2008*, LNCS 4965. Springer-Verlag, 2008.
32. David Wagner, editor. *Advances in Cryptology - CRYPTO 2008*, LNCS 5157. Springer-Verlag, 2008.
33. 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.

```
G1, G2, G3, G4:
Global variables:
    Tables F1, ..., F10
    Permutation table Tsim, Tsim^-1
    Set of nodes N
    Counter NumOuter
Random oracle tapes: f1, ..., f10

class Node
    Node parent
    Set of Node children
    2chain id
    Queries beginning, end
    constructor Node(p, c)
        self.parent ← p
        self.children ← ∅
        self.id ← c
        self.beginning ← null
        if (p ≠ null) then
            self.beginning ← p.end
        self.end ← null

private procedure SimP(x0, x1)
    if (x0, x1) ∉ Tsim then
        (x10, x11) ← P(x0, x1)
        Tsim(x0, x1) ← (x10, x11)
        Tsim^-1(x10, x11) ← (x0, x1)
    return Tsim(x0, x1)

private procedure SimP^-1(x10, x11)
    if (x10, x11) ∉ Tsim^-1 then
        (x0, x1) ← P^-1(x10, x11)
        Tsim(x0, x1) ← (x10, x11)
        Tsim^-1(x10, x11) ← (x0, x1)
    return Tsim^-1(x10, x11)

private procedure Assert(fact)
    if ¬fact then abort

public procedure F(i, xi)
    if xi ∈ Fi then return Fi(xi)
    Assert(¬IsPending(i, xi))
    if i ∈ {2, 5, 6, 9} then
        return NewTree(i, xi)
    else return ReadTape(i, xi)

private procedure ReadTape(i, xi)
    Assert(xi ∉ Fi)
    CheckBadR(i, xi) // G3
    Fi(xi) ← fi(xi)
    return Fi(xi)
```

```
private procedure NewTree(i, xi)
    root ← new Node(null, null)
    root.end ← (i, xi)
    N.add(root)
    GrowTree(root)
    SampleTree(root)
    CheckBadA(root) // G3
    AdaptTree(root)
    return Fi(xi)

private procedure GrowTree(root)
    do
        modified ← GrowSubTree(root)
    while modified

private procedure GrowSubTree(node)
    modified ← FindNewChildren(node)
    forall c in node.children do
        modified ← modified or GrowSubTree(c)
    return modified

private procedure IsPending(i, xi)
    forall n in N do
        if (i, xi) = n.end then return true
    return false

private procedure FindNewChildren(node)
    (i, x) ← node.end
    child_added ← false
    if i = 2 then forall (x9, x10, x1) in (F9, F10, F1) do
        if CheckP(x1, x, x9, x10) and
        ¬Equivalent(node.id, (1, x1, x)) and
        NotChild(node, (1, x1, x)) then
            Assert(++NumOuter ≤ q)
            AddChild(node, (1, x1, x))
            child_added ← true
    if i = 9 then forall (x1, x2, x10) in (F1, F2, F10) do
        if CheckP(x1, x2, x, x10) and
        ¬Equivalent(node.id, (9, x, x10)) and
        NotChild(node, (9, x, x10)) then
            Assert(++NumOuter ≤ q)
            AddChild(node, (9, x, x10))
            child_added ← true
    if i = 5 then forall x6 in F6 do
        ¬Equivalent(node.id, (5, x, x6)) and
        NotChild(node, (5, x, x6)) then
            AddChild(node, (5, x, x6))
            child_added ← true
    if i = 6 then forall x5 in F5 do
        ¬Equivalent(node.id, (5, x5, x)) and
        NotChild(node, (5, x5, x)) then
            AddChild(node, (5, x5, x))
            child_added ← true
    return child_added
```

Fig. 3. First part of pseudocode for games G_1–G_4. Game G_1 implements the simulator. Lines commented with '// G_i' appear in game G_i only.

```
private procedure AddChild(parent, id)
    new_child ← new Node(parent, id)
    parent.children.add(new_child)
    MakeNodeReady(new_child)

private procedure CheckP(x_1, x_2, x_9, x_10)
    if x_1 ∉ F_1 or x_10 ∉ F_10 then return false
    x_0 ← F_1(x_1) ⊕ x_2
    x_11 ← x_9 ⊕ F_10(x_10)
    if (x_0, x_1) ∉ T then return false      // G_2, G_3, G_4
    return SimP(x_0, x_1) = (x_10, x_11)

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 = 9 and j = 5 or i = 5 and j = 1 then
        return x'_j = Val^+(C_1, j) and
            x'_{j+1} = Val^+(C_1, j + 1)
    if i = 5 and j = 9 or i = 1 and j = 5 then
        return x'_j = Val^-(C_1, j) and
            x'_{j+1} = Val^-(C_1, j + 1)

private procedure NotChild(node, C)
    forall n in node.children do
        if Equivalent(n.id, C) then return false
    return true

private procedure MakeNodeReady(node)
    (i, x) ← node.beginning
    (j, u_1, u_2) ← node.id
    if i ∈ {2, 6} then
        while j ≠ Terminal(i) do
            (u_1, u_2) ← Prev(j, u_1, u_2)
            j ← j − 1 mod 11
        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 11
        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 = 2 then return 5
    if i = 5 then return 2
    if i = 6 then return 9
    if i = 9 then return 6

private procedure Next(i, x_i, x_{i+1})
    if i = 10 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_10, x_11) ← SimP(x_i, x_{i+1})
        return (x_10, x_11)
    else
        x_{i−1} = F(i, x_i) ⊕ x_{i+1}
        return (x_{i−1}, x_i)

private procedure SampleTree(node)
    ReadTape(node.end)
    N.delete(node)
    forall c in node.children do
        SampleTree(c)

private procedure AdaptTree(root)
    forall c in root.children do
        AdaptNode(c)

private procedure AdaptNode(node)
    C ← node.id
    (i, x_i) ← node.beginning
    (j, x_j) ← node.end
    m ← min{i, j} + 1
    x_{m−1}, x_m ← Val^+(C, m − 1), Val^+(C, m)
    x_{m+1}, x_{m+2} ← Val^-(C, m + 1), Val^-(C, m + 2)
    Adapt(m, x_m, x_{m−1} ⊕ x_{m+1})
    Adapt(m + 1, x_{m+1}, x_m ⊕ x_{m+2})
    forall c in node.children do
        AdaptNode(c)

private procedure Adapt(i, x_i, y_i)
    Assert(x_i ∉ F_i)
    F_i(x_i) ← y_i
```

Fig. 4. Second part of games G_1, G_2, G_3 and G_4.

```
private procedure Val⁺(i, x_i, x_{i+1}, k)
   if k ∈ {i, i+1} then return x_k
   j ← i + 1
   U, U^{-1} ← T_sim, T_sim^{-1}
   U, U^{-1} ← T, T^{-1}      // G2, G3, G4
   while j ≠ k do
      if j < 11 then
         if x_j ∉ F_j then return ⊥
         x_{j+1} ← x_{j-1} ⊕ F_j(x_j)
         j ← j + 1
      else
         if (x_10, x_11) ∉ U^{-1} then return ⊥
         (x_0, x_1) ← U^{-1}(x_10, x_11)
         if k = 0 then return x_0
         j ← 1
   return x_k
```

```
private procedure Val⁻(i, x_i, x_{i+1}, k)
   if k ∈ {i, i+1} then return x_k
   j ← i
   U, U^{-1} ← T_sim, T_sim^{-1}
   U, U^{-1} ← T, T^{-1}      // G2, G3, G4
   while j ≠ k do
      if j > 0 then
         if x_j ∉ F_j then return ⊥
         x_{j-1} ← F_j(x_j) ⊕ x_{j+1}
         j ← j − 1
      else
         if (x_0, x_1) ∉ U then return ⊥
         (x_10, x_11) ← U(x_0, x_1)
         if k = 11 then return x_11
         j ← 10
   return x_k
```

Fig. 5. Third part of games G_1, G_2, G_3 and G_4.

```
G1, G2, G3:
Variables:
   Permutation table T, T^{-1}
Random permutation tape: p

public procedure P(x_0, x_1)
   if (x_0, x_1) ∉ T then
      (x_10, x_11) ← p(x_0, x_1)
      CheckBadP(10, x_10) // G3
      T(x_0, x_1) ← (x_10, x_11)
      T^{-1}(x_10, x_11) ← (x_0, x_1)
   return T(x_0, x_1)

public procedure P^{-1}(x_10, x_11)
   if (x_10, x_11) ∉ T^{-1} then
      (x_0, x_1) ← p^{-1}(x_10, x_11)
      CheckBadP(1, x_1) // G3
      T(x_0, x_1) ← (x_10, x_11)
      T^{-1}(x_10, x_11) ← (x_0, x_1)
   return T^{-1}(x_10, x_11)
```

```
G4:
Variables:
   Table with two-way access T

public procedure P(x_0, x_1)
   if (x_0, x_1) ∉ T then
      for i ← 2 to 11 do
         x_i ← x_{i-2} ⊕ f_{i-1}(x_{i-1})
      T(x_0, x_1) ← (x_10, x_11)
      T^{-1}(x_10, x_11) ← (x_0, x_1)
   return T(x_0, x_1)

public procedure P^{-1}(x_10, x_11)
   if (x_10, x_11) ∉ T^{-1} then
      for i ← 9 to 0 do
         x_i ← f_{i+1}(x_{i+1}) ⊕ x_{i+2}
      T(x_0, x_1) ← (x_10, x_11)
      T^{-1}(x_10, x_11) ← (x_0, x_1)
   return T^{-1}(x_10, x_11)
```

Fig. 6. Permutation oracles for G_1, G_2, G_3 (at left) and G_4 (at right).

```
G5:
Variables:
Random tapes: f_1, ..., f_10

public procedure F(i, x_i)
   return f_i(x_i)
```

```
public procedure P(x_0, x_1)
   for i ← 2 to 11 do
      x_i ← x_{i-2} ⊕ f_{i-1}(x_{i-1})
   return (x_10, x_11)

public procedure P^{-1}(x_10, x_11)
   for i ← 9 to 0 do
      x_i ← f_{i+1}(x_{i+1}) ⊕ x_{i+2}
   return (x_0, x_1)
```

Fig. 7. Game G_5 (the real world).

```
G₃:
Variables: Set 𝒜      // Set of adapted queries

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 CheckBadR(i, x_i)
    CheckBadlyHit(i, x_i, f_i(x_i))
    CheckRCollide(i, x_i, f_i(x_i))

private procedure CheckBadA(root)
    𝒜 ← ∅
    GetAdapts(root)
    forall a in 𝒜 do
        (i, x_i), y_i ← a.query, a.value
        CheckBadlyHit(i, x_i, y_i)
        forall b in 𝒜 do
            CheckAPair(a, b)

private procedure GetAdapts(node)
    if node.id ≠ null
        (i, x_i), (j, x_j) ← node.beginning, node.end
        C ← node.id
        m, n ← min{i, j}, max{i, j}
        x_{m+1}, x_{n-1} ← Val⁺(C, m + 1), Val⁻(C, n − 1)
        y_{m+1}, y_{n-1} ← x_m ⊕ x_{n-1}, x_{m+1} ⊕ x_n
        𝒜.add(new Adapt(m + 1, x_{m+1}, y_{m+1}, x_m, x_n))
        𝒜.add(new Adapt(n − 1, x_{n-1}, y_{n-1}, x_m, x_n))
    forall c in node.children do
        GetAdapts(c)

private procedure ActiveQueries(i)
    P ← ∅
    forall n in N do
        (j, x_j) ← n.end
        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} ∈ F_{i+1} or IsPending(i + 1, x_{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 ∈ F_i or IsPending(i, x_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 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) then
            if 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}) then
            if 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 j ≠ i + 1 then return
    if (a.left, a.right) = (b.left, b.right) 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. 8. The abort-checking procedures for G_3.