A shortened version of this paper appears under the same title in the proceedings of *Indocrypt 2012* (Steven Galbraith and Mridul Nandi, eds.), LNCS, Springer.

A Novel Permutation-based Hash Mode of Operation FP and The Hash Function SAMOSA

Souradyuti Paul[*] Ekawat Homsirikamol[†] Kris Gaj[2†]

Abstract

The contribution of the paper is two-fold. First, we design a novel permutation-based hash mode of operation FP, and analyze its security. The FP mode is derived by replacing the *hard-to-invert* primitive of the FWP mode – designed by Nandi and Paul, and presented at Indocrypt 2010 – with an *easy-to-invert* permutation; since easy-to-invert permutations with good cryptographic properties are *normally* easier to design, and are more efficient than the hard-to-invert functions, the FP mode is more suitable in practical applications than the FWP mode.

We show that any n-bit hash function that uses the FP mode is indifferentiable from a random oracle up to $2^{n/2}$ queries (up to a constant factor), if the underlying $2n$-bit permutation is free from any structural weaknesses. Based on our further analysis and experiments, we conjecture that the FP mode is resistant to all non-trivial generic attacks with work less than the brute force, mainly due to its large internal state. We compare the FP mode with other permutation-based hash modes, and observe that it displays the so far best security/rate trade-off.

To put this into perspective, our second contribution is a proposal for a concrete hash function SAMOSA using the new mode and the P-permutations of the SHA-3 finalist Grøstl. Based on our analysis we claim that the SAMOSA family cannot be attacked with work significantly less than the brute force. We also provide hardware implementation (FPGA) results for SAMOSA to compare it with the SHA-3 finalists. In our implementations, SAMOSA family consistently beats Grøstl, Blake and Skein in the throughput to area ratio. With more efficient underlying permutation, it seems possible to design a hash function based on the FP mode that can achieve even higher performances.

Keywords. Hash mode, indifferentiability, permutation, FPGA implementation

[*]University of Waterloo, Canada, and K. U. Leuven, Belgium, souradyuti.paul@uwaterloo.ca
[†]George Mason University, USA, {ekawat@gmail.com, kgaj@gmu.edu}

Contents

1 Introduction

1.1 Block-cipher-based hash modes

Iterative hash functions are generally built from two components: (1) a basic primitive C with finite domain and range, and (2) an iterative mode of operation H to extend the domain of C; the symbol H^C denotes the hash function based on the mode H which invokes C iteratively to compute the hash digest. Therefore, to design an efficient hash function one has to be innovative with both the mode H and the basic primitive C. Merkle-Damgård mode used with a secure block cipher was the most attractive choice to build a practical hash function; some examples are SHA-family [30], and MD5 [34]. The security of a hash function based on the Merkle-Damgård mode crucially relies on the fact that C is collision and preimage resistant. The compression function C achieves these properties when it is constructed using a secure block cipher [9]. However, several security issues changed this popular design paradigm in the last decade. The first concern is that the security of Merkle-Damgård mode of operation – irrespective of the strength of the primitive C – came under a host of generic attacks; the length-extension attack [13], expandable message attack [22], multi-collision attacks [20] and herding attack [10, 21] are some of them. Several strategies were discovered to thwart the above attacks. Lucks came up with the proposal of making the output of the primitive C at least twice as large as the hash output [26]; this proposal is outstanding since, apart from rescuing the security of the Merkle-Damgård mode, it is also simple and easy to implement. Another interesting proposal was HAIFA that includes a counter injected into the compression function C to rule out many of the aforementioned attacks [7]. Using the results of [9], it is easy to see that the Wide pipe and the HAIFA constructions are secure when the underlying primitive is a secure block cipher.

Despite the aforementioned foolproof design strategies, it turns out that using a block cipher as the basic primitive of a hash function may not be the best alternative, for several reasons. (1) A hash function does not need both the encryption and decryption functions of a block cipher; one of them could be avoided. (2) The key schedule of a block cipher often turns out to be weak [8]. (3) Furthermore, the key schedule weaknesses of a block cipher render invalid the very common ideal cipher assumption under which the security of block-cipher-based hash functions is usually based; note that an ideal cipher assumption is stronger than an ideal permutation assumption since, in the former case, an extra assumption is that a huge number of ideal permutations need to be *independent* too. (4) The amount of memory needed to implement a wide block cipher is larger due to the 'extra' key schedule than needed for an equally sized permutation.

1.2 Permutation-based hash modes

For the reasons described in the previous section, the popularity of permutation-based hash functions has been on the rise since the discovery of weaknesses in the Merkle-Damgård

Table 1: Indifferentiability bounds of permutation-based hash modes, where the hash size is n-bit in each case; π denotes the permutation, or one of many equally sized permutations. $\mathsf{FP}^{\textsc{Ext1}}$ is a natural variant of FP with parameters shown in the row. The ϵ is a small fraction due to the preimage attack on JH presented in [6]. Msg-blk denotes the message block.

Mode of operation	**Msg-blk** (ℓ)	**Size of π** (a)	**Rate** (ℓ/a)	**Indiff. bound** lower	upper	**# of independent permutations**
Hamsi [24]	$n/8$	$2n$	0.07	$n/2$	n	1
Luffa [5]	$n/3$	n	0.33	$n/4$	n	3
Sponge [4]	n	$3n$	0.33	n	n	1
Sponge [4]	n	$2n$	0.5	$n/2$	$n/2$	1
JH [29]	n	$2n$	0.5	$n/2$	$n(1-\epsilon)$	1
Grøstl [17]	n	$2n$	0.5	$n/2$	n	2
FP	$\boldsymbol{n}$	$\boldsymbol{2n}$	**0.5**	$\boldsymbol{n/2}$	$\boldsymbol{n}$	**1**
MD6 [14]	$6n$	$8n$	0.75	n	n	1
$\mathbf{FP^{Ext1}}$	$\boldsymbol{6n}$	$\boldsymbol{7n}$	**0.85**	$\boldsymbol{n/2}$	$\boldsymbol{n}$	**1**

mode. Sponge [3], Grøstl [17], JH [39], Luffa [11] and the Parazoa family [1] are some of them. We note that 9 out of 14 semi-finalist algorithms – and 3 out of 5 finalist algorithms – of the NIST SHA-3 hash function competition are based on permutations. Also, NIST selected Keccak as the winner of the SHA-3 competition, which is a permutation-based Sponge construction. Other notable example is MD6 [35]. In Table 1, we compare generic security and performance (measured in terms of rate) of various well known permutation-based hash modes.

1.3 Our contribution

1.3.1 The FP mode

Our first contribution is to give a proposal for a new hash mode of operation FP based on a single wide pipe permutation (see Figure 1). The FP mode is derived from the FWP (or Fast Wide Pipe) mode designed by Nandi and Paul at Indocrypt 2010 [31].[1] The difference between the FWP and the FP mode is simple: the FP mode is obtained when the underlying hard-to-invert function $f : \{0,1\}^{m+n} \rightarrow \{0,1\}^{2n}$ of the FWP mode is replaced by an easy-to-invert permutation $\pi : \{0,1\}^{2n} \rightarrow \{0,1\}^{2n}$. There are a number of practical reasons for switching from FWP to FP: (1) Easy-to-invert permutations are usually efficient, and such permutations with strong cryptographic properties are abundant in the literature (*e.g.* JH,

[1]FP is the shorthand for 'FWP with a permutation'.

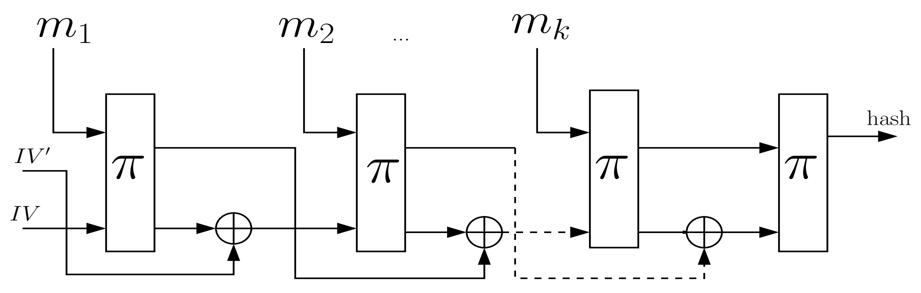


Figure 1: Diagram of the FP mode. The π is a permutation; all wires are n bits. See Figure 3(a) for the description.

Grøstl and Keccak permutations); (2) hard-to-invert functions are difficult to design, or they are less efficient.

On the other hand, easy-to-invert permutations – even though they are faster – have some drawbacks; the most crucial of them is that they allow the attacker to use reverse queries in addition to forward queries, and, as a result, make the adversary inherently more powerful. Therefore, a good deal of caution is required to design a hash mode of operation that uses permutations. We show that the FP mode based on an ideal permutation is indifferentiable from a random oracle up to approximately $2^{n/2}$ queries (forward and reverse together); this means that the FP mode is secure against *all* generic attacks – including (multi) collision, 2nd preimage, herding attacks – up to approximately $2^{n/2}$ queries, under the assumption that the underlying $2n$-bit permutation is structurally strong. Moving further, we performed experiments to implement our indifferentiability framework with randomly generated graphs using C programs, and our experiments strongly indicate that the indifferentiability security of the FP mode could be improved *close* to n bits. Another important feature of our work is that the security guarantee is based on only one assumption – like the Sponge and JH – that the underlying permutation should not display any structural weaknesses; note that the security of many permutation-based hash functions (*e.g.* Grøstl and Luffa) requires additional assumptions such as *independence* of several ideal permutations. In Figure 1, we compare the FP mode and a natural extension of it $\mathsf{FP}^{\textsc{Ext1}}$ with other permutation-based hash functions. It is noteworthy that the FP mode exhibits the best security/rate trade-off, when the internal permutation size is fixed.

1.3.2 Design and FPGA implementation of SAMOSA

Our second contribution is establishing the practical usefulness of the FP mode. As an example, we design a concrete hash function family SAMOSA.[2] It is based on the FP mode, where the internal primitives are the P-permutations of the Grøstl hash function. We provide security analysis of SAMOSA, demonstrating its resistance against any known

[2]SAMOSA is the name of an Indian food.

practical attacks.

As demonstrated by the AES and the SHA-3 competitions, the security of a cryptographic algorithm alone is not sufficient to make it stand out from among multiple candidates competing to become a new American or international standard. Excellent performance in software and hardware is necessary to make a cryptographic protocol usable across platforms and commercial products. Assuring good performance in hardware is typically more challenging, since hardware design requires involved and specialized training, and, as it turns out, that the majority of designer groups lack experience and expertise in that area.

In case of SAMOSA, the algorithm design and hardware evaluation have been performed side by side, leading to full understanding of all design decisions and their influence on hardware efficiency. In this paper, we present efficient high-speed architecture of SAMOSA, and show that this architecture outperforms the best known architecture of Grøstl in terms of the throughput to area ratio by a factor ranging between 24 and 51%. These results have been demonstrated using two representative FPGA families from two major vendors, Xilinx and Altera. As shown in [16], these results are also very likely to carry to any future implementations based on ASICs (Application Specific Integrated Circuits). Additionally, we demonstrate that SAMOSA consistently ranks above BLAKE, Skein and Grøstl in our FPGA implementations. Although it still loses to Keccak and JH, nevertheless, a relative distance to these algorithms substantially decreases compared to Grøstl, despite using the same underlying operations. This performance gain is accomplished without any known degradation of the security strength.

Additionally, SAMOSA's dependence on many AES operations makes it suitable for software implementations that use general-purpose processors with AES instruction sets, such as AES-NI. Finally, in both software and hardware, SAMOSA could be an attractive choice for applications where both confidentiality and authentication are required to share AES components. One such example is IPSec, protocol used for establishing Virtual Private Networks, which is one of the fundamental building blocks of secure electronic transactions over the Internet.

Although SAMOSA comes too late for the current SHA-3 competition, it still has a chance to contribute to better understanding of the security and performance bottlenecks of modern hash functions, and to find niche platforms and applications in which it may outperform the existing and upcoming standards.

1.4 Notation and convention

Throughout the paper we let n be a fixed integer. While representing a bit-string, we follow the convention of low-bit first (or little-endian bit ordering). For concatenation of strings, we use $a||b$, or just ab if the meaning is clear. The symbol $\langle n \rangle_m$ denotes the m-bit encoding of n. The symbol $|x|$ denotes the bit-length of the bit-string x, or sometimes the size of the set x. Let $x \overset{parse}{\rightarrow} x_1 x_2 \cdots x_k$ means parsing x into x_1, $x_2, \cdots, x_k$ such that

$|x_1| = |x_2| = \cdots = |x_{k-1}| = n$ and $|x_k| = |x| - |x_1 x_2 \cdots x_{k-1}|$. Let $Dom(T) = \{i \mid T[i] \neq \perp\}$ and $Rng(T) = \{T[i] \mid T[i] \neq \perp\}$. We write $\mathcal{A}^B$ to denote an Algorithm $\mathcal{A}$ with oracle access to B. Let $[c, d]$ be the set of integers between c and d inclusive, and $a[x, y]$ the bit-string between the x-th and y-th bit-positions of a. Finally, $\mathcal{U}[0, N]$ is the uniform distribution over the integers between 0 and N. The symbol $r \overset{\$}{\leftarrow} A$ denotes the operation of assigning r with a value sampled uniformly from the set A. The symbol λ denotes the empty string.

2 Definition of the FP Mode

Suppose $n \geq 1$. Let $\pi : \{0,1\}^{2n} \to \{0,1\}^{2n}$ be the $2n$-bit permutation used by the FP mode. The hash function FP^π is a mapping from $\{0,1\}^*$ to $\{0,1\}^n$. The diagram and description of the FP transform are given in Figures 1 and 3(a), where π is modeled as an ideal permutation. Below we define the padding function $\mathsf{pad}_n(\cdot)$.

Padding function $\mathsf{pad}_n(\cdot)$. It is an injective mapping from $\{0,1\}^*$ to $\cup_{i \geq 1}\{0,1\}^{ni}$, where the message $M \in \{0,1\}^*$ is mapped into a string $\mathsf{pad}_n(M) = m_1 \cdots\ m_{k-1}\ m_k$, such that $|m_i| = n$ for $1 \leq i \leq k$. The function $\mathsf{pad}_n(M) = M||1||0^t$ satisfies the above properties (t is the least non-negative integer such that $|M| + 1 + t = 0 \bmod n$). Note that $k = \left\lceil \frac{|M|+1}{n} \right\rceil$.

In addition to the injectivity of $\mathsf{pad}_n(\cdot)$, we will also require that there exists a function $\mathsf{dePad}_n(\cdot)$ that can efficiently compute M, given $\mathsf{pad}_n(M)$. Formally, the function $\mathsf{dePad}_n\colon \cup_{i\geq 1} \{0,1\}^{in} \to \{\perp\} \cup \{0,1\}^*$ computes $\mathsf{dePad}_n(\mathsf{pad}_n(M)) = M$, for all $M \in \{0,1\}^*$, and otherwise $\mathsf{dePad}_n(\cdot)$ returns $\perp$. The padding rule described above satisfies this property also.

3 Indifferentiability Framework: An Overview

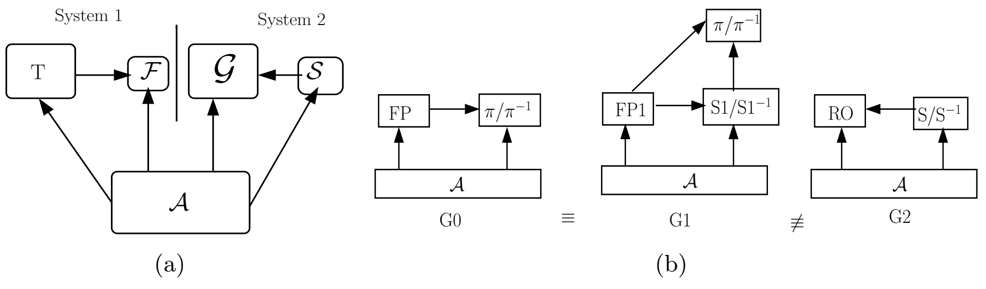


Figure 2: (a) Indifferentiability framework formalized in Definition 3.1; (b) Schematic diagram for the security games – description provided in Sections 7, 8, and 9 – used in the indifferentiability framework for FP. The arrows show the directions in which the queries are submitted. System 1 = $(T, \mathcal{F})$ = G0 = $(\mathsf{FP}, \pi, \pi^{-1})$, and System 2 = $(\mathcal{G}, \mathcal{S})$ = G2 = (RO, S, S^{-1}). See Section 4 for description.

We first define the indifferentiability security framework, and briefly discuss its significance.

Definition 3.1 (Indifferentiability framework) *[13] An interactive Turing machine (ITM) T with oracle access to an ideal primitive $\mathcal{F}$ is said to be $(t_{\mathcal{A}}, t_{\mathcal{S}}, \sigma, \varepsilon)$-indifferentiable from an ideal primitive $\mathcal{G}$ if there exists a simulator $\mathcal{S}$ such that, for any distinguisher $\mathcal{A}$, the following equation is satisfied:*

$$\mathbf{Adv}^{T,\mathcal{F}}_{\mathcal{G},\mathcal{S}}(\mathcal{A}) \overset{def}{=} \left|\Pr[\mathcal{A}^{T,\mathcal{F}} = 1] - \Pr[\mathcal{A}^{\mathcal{G},\mathcal{S}} = 1]\right| \leq \varepsilon.$$

The simulator $\mathcal{S}$ is an ITM which has oracle access to $\mathcal{G}$ and runs in time at most $t_{\mathcal{S}}$. The distinguisher $\mathcal{A}$ runs in time at most $t_{\mathcal{A}}$. The number of queries used by $\mathcal{A}$ is at most σ. Here ε is a negligible function in the security parameter of T. See Figure 2(a) for a pictorial representation. $\mathbf{Adv}^{T,\mathcal{F}}_{\mathcal{G},\mathcal{S}}(\mathcal{A})$ denotes the advantage of adversary $\mathcal{A}$ in distinguishing $(T, \mathcal{F})$ from $(\mathcal{G}, \mathcal{S})$.

The significance of the framework is as follows. Suppose, an ideal primitive $\mathcal{G}$ is indifferentiable from an algorithm T based on another ideal primitive $\mathcal{F}$. In such a case, any cryptographic system $\mathcal{P}$ based on $\mathcal{G}$ is as secure as $\mathcal{P}$ based on $T^{\mathcal{F}}$ (i.e., $\mathcal{G}$ replaces $T^{\mathcal{F}}$ in $\mathcal{P}$). For a more detailed explanation, we refer the reader to [27]. Some limitations of the indifferentiability framework have recently been discovered in [15] and [33]. They offer a deep insight into the framework; nevertheless, the observations are *not known* to affect the security of the indifferentiable hash functions in any meaningful way.

An oracle, a system, and a game. An oracle is an algorithm (accessed by another oracle or algorithm) which, given an input as an appropriately defined query, responds with an output. For example, in Figure 2(a), T, $\mathcal{F}$, $\mathcal{G}$ and $\mathcal{S}$ are oracles. A system is a set of oracles (*e.g.* System 1 = $(T, \mathcal{F})$, System 2 = $(\mathcal{G}, \mathcal{S})$ in Figure 2(a)). A game is the interaction of a system with an adversary. We refrain from providing a formal definition of a game, since such formalization will not be necessary in our analysis.

4 Main Theorem: Birthday Bound for FP Mode

Let $\mathsf{RO}\colon \{0,1\}^* \to \{0,1\}^n$ and $\pi\colon \{0,1\}^{2n} \to \{0,1\}^{2n}$ be a random oracle and an ideal permutation. Our indifferentiability framework uses three systems G0 = $(\mathsf{FP}, \pi, \pi^{-1})$, G1 = $(\mathsf{FP1}, S1, S1^{-1})$, and G2 = (RO, S, S^{-1}) (see Figure 2(b)). The correspondence between the entities of Figures 2(a) and 2(b) are as follows: $\mathcal{G} = \mathsf{RO}$, $T = \mathsf{FP}$ and $\mathcal{F} = (\pi, \pi^{-1})$. The description of $\mathsf{FP1}$, S, S^{-1}, $\mathsf{S1}$, and $\mathsf{S1}^{-1}$ will be provided in Sections 7, 8, and 9. Now we state our main theorem using Definition 3.1.

Theorem 4.1 (Main Theorem) *The hash function FP^{π} (or $\mathsf{FP}^{\pi,\pi^{-1}}$) is $(t_{\mathcal{A}}, t_{\mathcal{S}}, \sigma, \varepsilon)$-indifferentiable from RO, where $t_{\mathcal{A}} = \infty$, $t_{\mathcal{S}} = \mathcal{O}(\sigma^5)$, and $\sigma \leq K2^{n/2}$, where K is a fixed constant derived from ε.*

In the next few sections, we will prove Theorem 4.1 by breaking it into several components. First, we briefly describe what the theorem means: it says that no adversary with unbounded running time can mount a *nontrivial generic* attack on the hash function FP^{π} using at most $K2^{n/2}$ queries. The parameter K is an increasing function in ε, and is constant for all $n > 0$ for a fixed ε. To reduce the notation complexity, we shall derive the indifferentiability bound assuming $\varepsilon = 0.5$ for which, we shall derive, $K = 1/\sqrt{56}$.

Outline of the Proof. Our proof of Theorem 4.1 uses the blueprint developed in [28] and [29] dealing with indifferentiability security analysis of FWP and JH modes of operation. However, to make the paper self-contained, we write out the proof from scratch. The proof consists of the following two components (see Definition 3.1): (1) Construction of a simulator $\mathcal{S} = (\mathsf{S}, \mathsf{S}^{-1})$ with the worst-case running time $t_{\mathcal{S}} = \mathcal{O}\big(\sigma^5\big)$. This is done in Section 8. (2) Showing that, for any adversary $\mathcal{A}$ with unbounded running time,

$$\Big|\Pr\big[\mathcal{A}^{G0} \Rightarrow 1\big] - \Pr\big[\mathcal{A}^{G2} \Rightarrow 1\big]\Big| \leq \frac{28\sigma^2}{2^n}, \tag{1}$$

where the systems $G0 = (\mathsf{FP}, \pi, \pi^{-1})$ and $G2 = (\mathsf{RO}, \mathsf{S}, \mathsf{S}^{-1})$. The systems G1 and G2 are called the main systems. Proof of (1) is, again, composed of proofs of the following three (in)equations:

- In Sections 8 and 9, we will concretely define the simulator pair $(\mathsf{S}, \mathsf{S}^{-1})$ and a new intermediate system G1. Using them we will show in Section 10,

$$\Pr\big[\mathcal{A}^{G0} \Rightarrow 1\big] = \Pr\big[\mathcal{A}^{G1} \Rightarrow 1\big]. \tag{2}$$

- In Sections 11 and 12, we will appropriately define a set of events BAD_i and GOOD_i in the system G1, and will establish that

$$\Big|\Pr\big[\mathcal{A}^{G1} \Rightarrow 1\big] - \Pr\big[\mathcal{A}^{G2} \Rightarrow 1\big]\Big| \leq \sum_{i=1}^{\sigma} \Pr\big[\mathsf{BAD}_i \mid \mathsf{GOOD}_{i-1}\big]. \tag{3}$$

- In Section 13, we complete proof of (1) by establishing that

$$\sum_{i=1}^{\sigma} \Pr\big[\mathsf{BAD}_i \mid \mathsf{GOOD}_{i-1}\big] \leq \frac{28\sigma^2}{2^n} \tag{4}$$

  where

$$\sum_{i=1}^{\sigma} \Pr\big[\mathsf{BAD}_i \mid \mathsf{GOOD}_{i-1}\big] \leq \varepsilon = 0.5.^{3}$$

[3]Setting $\frac{28\sigma^2}{2^n} \leq \varepsilon = 1/2$, we get $\sigma \leq \frac{1}{\sqrt{56}} 2^{n/2}$. Therefore, $K = 1/\sqrt{56}$.

5 Organization of the paper

5.1 Proof of main theorem

Sections 6 to 13 are devoted to complete the proof of Theorem 4.1. The three parts of the proof – *i.e.,* proving (2), (3), and (4) – are done in Sections 10, 12, and 13. These sections make use of the results that are developed in the following sections: Section 6 defines the data structures used by all systems and proofs; Sections 7, 8, 9, and 11 give detailed description of all the systems.

5.2 Design and implementation of SAMOSA

In Section 14, we propose a new concrete hash function named SAMOSA, and provide its security analysis. Finally, in Section 15, we give FPGA hardware implementation results for SAMOSA.

6 Data Structures

The systems G0, G1, and G2 have been mentioned in Section 4 (see schematic diagram in Figure 2(b)). The pseudocode of them is given in Figures 3(a), 5, and 3(b). In this section we describe several data structures used by these systems.

6.1 Objects used in pseudocode

6.1.1 Oracles

The main component of a system is the set of oracles that receive queries from the adversary. In Figure 2(b), any algorithm that receives a query is an oracle. Note that, except the adversary $\mathcal{A}$, each rectangle denotes an oracle.

The systems use a total of 9 oracles. The oracles FP, $\mathsf{FP1}$, and RO are mappings from $\{0,1\}^*$ to $\{0,1\}^n$. The oracle S is a mapping from $\{0,1\}^{2n}$ to $\{0,1\}^{2n}$. The permutations π, π^{-1}, $\mathsf{S1}$, and $\mathsf{S1}^{-1}$ are all defined on $\{0,1\}^{2n}$, while S^{-1} is a mapping from $\{0,1\}^{2n}$ to $\{0,1\}^{2n} \cup \{\perp\}$. Instruction-by-instruction description of these oracles and the used subroutines are provided in the subsequent sections.

6.1.2 Global and local Variables

The oracles described above will use several global and local variables. The local variables are re-initialized every new invocation of the system, while the global data structures maintain their states across queries. The tables D_l, D_s and D_π are global variables initialized with $\perp$. The graphs T_π and T_s are also global variables which initially contain only the root node (IV, IV'). Other than them, all other variables are local, and they are initialized with $\perp$.

6.1.3 Query and round: definitions

In Figure 2(b), an arrow denotes a query. The submitter and receiver algorithms of a query are denoted by the rectangles attached to the head and the tail of the arrow.

Long query. Any query submitted to FP, $\mathsf{FP1}$, or RO is a *long query.* A long query and its response are stored in the table D_l.

Short query. Queries submitted to S, $\mathsf{S1}$ are s-queries, and those submitted to S^{-1} and $\mathsf{S1}^{-1}$ are s^{-1}-queries. The s- and s^{-1}-queries and their responses are stored in table D_s. Similarly, queries submitted to π, and π^{-1} are π- and π^{-1}-queries; these queries and their responses are stored in table D_π. Each of the above queries is classified as *short query.* Note that, for G0, $D_s = \emptyset$; for G1, $D_\pi \supseteq D_s$; and, for G2, $D_\pi = \emptyset$.

Fresh and old queries. The *current* short query can also be of two disjoint types: (1) an *old* query, which is already present in the relevant database (*e.g.* for G1, when an adversary submits an s-query which is an intermediate π-query of a previously submitted long query); or (2) a *fresh* query, which is so far not present in the relevant database.

Message block. In order to compare the time complexities of the oracles FP, $\mathsf{FP1}$ and RO on a uniform scale, we recall the notion of a *message block.* A long query M – irrespective of the oracle – is assumed to be a sequence of k message blocks $m_1, m_2, \cdots m_k$, where $\mathsf{pad}_n(M) = m_1 m_2 \cdots m_k$. Note that, for FP and $\mathsf{FP1}$, every message block m_i corresponds to a π-query $x||m_i$ for some bit-string x. However, it is not known how the RO processes the message blocks of a long query M. We assume that the RO processes the message blocks sequentially, and that the time taken to process a message block is the same for all FP, $\mathsf{FP1}$ and RO.

Round (and query). The time interval to process a short query or a message block is defined as a *round.* We assume that each round takes an equal amount of time. To simplify the analysis, henceforth, unless otherwise specified, a query would mean either a short query or a message block.

Rules of the game. An adversary *never* re-submits an identical query. Moreover, an s-query (or s^{-1}-query) is also *not* submitted, if it matches with the output of a previous s^{-1}-query (or s-query).

6.2 Graph theoretic objects used in proof of main theorem

In addition to objects defined in the section above, we will use the following notions for a rigorous mathematical analysis of our results.

Suppose, $\pi\colon \{0,1\}^{2n} \to \{0,1\}^{2n}$ is an ideal permutation, and D is a finite set of pairs of the form $(x, \pi(x))$.

6.2.1 Reconstructible message

From the high level, M is a *reconstructible message* for the set D, if D contains all the π-queries and responses $(x, \pi(x))$, required to compute $\mathsf{FP}^{\pi}(M)$.
More formally, M is a *reconstructible* message for D, if, for all $0 \le i \le k-1$, $(y_i m_{i+1}, \pi(y_i m_{i+1})) \in D$, where $\mathsf{pad}_n(M) = m_1 m_2 \cdots m_k$ and $y_{i+1}y'_{i+1} = \pi(y_i m_{i+1}) \oplus (y'_i||0)$.

6.2.2 (Full) Reconstruction graph

To put it loosely, a *reconstruction graph* stores *reconstructible messages* on its branches. A *full reconstruction graph* stores *all reconstructible messages.* We now define them formally.
A weighted digraph $T = (V, E)$ is defined by the set of nodes V, and the set of weighted edges E. A weighted edge $(v, w, v') \in E$ is an ordered triple, such that $v, v' \in V$, and w is the weight of the ordered pair (v, v').

Definition 6.1 (Reconstruction graph) *Suppose a weighted digraph $T = (V, E)$ is such that V is a set of $2n$-bit strings, and, for all $(a, b, c) \in E$, the weight b is an n-bit string.*

The graph T is called a reconstruction *graph for D if, for every $(y_1y'_1,\ m_2,\ y_2y'_2) \in E$, the following equation holds: $y_2y'_2 = \pi(y_1m_2) \oplus (y'_1||0)$ (all variables are n bits each), where $(y_1m_2, \pi(y_1m_2)) \in D$. (An example of reconstruction graph is given in Figure 4, which will be discussed in detail in the subsequent sections.)*

A branch B of a reconstruction graph T, rooted at $y_0y'_0 = IVIV'$, is *fertile*, if $\mathsf{dePad}_n(m_1\ m_2 \cdots\ m_k) \ne \perp$, where $\{m_1,\ m_2,\ \cdots,\ m_k, y'_k\}$ is the sequence of weights on the branch B. The y'_k is computed following the recursion: $y_{i+1}y'_{i+1} = \pi(y_i m_{i+1}) \oplus (y'_i||0)$, for all $0 \le i \le k-1$.
Remark: Each fertile branch of a reconstruction graph corresponds to exactly one reconstructible message.

Definition 6.2 (Full reconstruction graph) *A reconstruction graph T (for the set D) is full, if, for each reconstructible message M (for D), T contains a fertile branch B that corresponds to M.*

6.2.3 View

Very loosely, the data structure view records the history of the interaction between a system and an adversary. Let x_i and y_i be the i-th query from the adversary and the corresponding response from the system. The view of the system after j queries is the sequence of queries and responses $\{(x_1y_1), \ldots, (x_jy_j)\}$.

7 Main System G0

Following the definition provided in Section 2, the system G0 implements the FP hash function using the ideal permutations π and π^{-1}. See Figure 3(a).

```
FP(M)
01. If $M \in Dom(D_l)$ then
        return $D_l[M]$;
02. $m_1 m_2 \ldots m_k := \mathsf{pad}_n(M)$;
03. $y_0 := IV$, $y'_0 := IV'$;
04. for $(i := 1, 2, \ldots k)$
      $y_i y'_i := \pi(y_{i-1} || m_i) \oplus (y'_{i-1} || 0)$;
05. $r := \pi(y_k || y'_k)$;
06. $D_l[M] := r[n, 2n-1]$;
07. return $D_l[M]$;

π(x)
11. If $x \notin Dom(D_\pi)$
      then $D_\pi[x] \xleftarrow{\$} \{0,1\}^{2n} \setminus Rng(D_\pi)$;
12. return $D_\pi[x]$;

π^{-1}(r)
21. If $r \notin Rng(D_\pi)$
      then $D_\pi^{-1}[r] \xleftarrow{\$} \{0,1\}^{2n} \setminus Dom(D_\pi)$;
22. return $D_\pi^{-1}[r]$;
```

(a) System G0 = (FP, π, π^{-1}). For all i, $|m_i| = |y_i| = |y'_i| = |r/2| = n$.

```
RO(M)
001. If $M \in Dom(D_l)$ then
         return $D_l[M]$;
002. $D_l[M] \xleftarrow{\$} \{0,1\}^n$;
003. return $D_l[M]$;

S^{-1}(r)
300. $x \xleftarrow{\$} \{0,1\}^{2n}$;
301. if $x \in Dom(D_s)$ then Abort;
302. $D_s[x] := r$;
303. FullGraph($D_s$);
304. return $x$;

S(x)
100. $r \xleftarrow{\$} \{0,1\}^{2n}$;
101. if $r \in Rng(D_s)$ then Abort;
102. $\mathcal{M}$ := MessageRecon($x, T_s$);
103. if $|\mathcal{M}| = 1$ then
       $r[n, 2n-1]$ := RO($M$);
104. $D_s[x] := r$;
105. FullGraph($D_s$);
106. return $r$;

MessageRecon(x, T_s)
201. $\mathcal{M}'$ := FindBranch($x, T_s$);
202. $\mathcal{M} := \{\mathsf{dePad}(X) \mid X \in \mathcal{M}'\}$;
203. return $\mathcal{M}$;
```

(b) System G2 = (RO, S, S^{-1}).

Figure 3: The main systems G0 and G2

8 Main System G2

See Figure 3(b) for the pseudocode. The random oracle RO defined in Section 4 is implemented through lazy sampling. The only remaining part is to construct the simulator-pair $(\mathsf{S}, \mathsf{S}^{-1})$. Our design strategy for the simulator-pair is fairly straightforward and simple. Before going into the details, we first provide a high level intuition.

8.1 Intuition for the simulator pair $(\mathsf{S}, \mathsf{S}^{-1})$

The purpose of the simulator pair $(\mathsf{S}, \mathsf{S}^{-1})$ is two-pronged: (1) to output values that are indistinguishable from the output from the ideal permutation (π, π^{-1}), and (2) to respond in such a way that $\mathsf{FP}^{\pi}(M)$ and $\mathsf{RO}(M)$ are identically distributed. It will easily follow that as long as the simulator-pair $(\mathsf{S}, \mathsf{S}^{-1})$ is able to output values satisfying the above conditions, no adversary can distinguish between G0 and G2.

To achieve (1), the simulator S, for a distinct input x, should output a random value, such that the distributions of $\mathsf{S}(x)$ and $\pi(x)$ are close. Similarly, the simulator S^{-1}, for a distinct input r, should give outputs such that the random variables $\mathsf{S}^{-1}(r)$ and $\pi^{-1}(r)$ follow statistically close distributions.

To achieve (2), the simulator-pair needs to generate reconstructible messages from the set D_s. To accomplish this, it needs to do the following:
• Assessing the power of the adversary: To asses the adversarial power, the simulator-pair (S, S^{-1}) maintains the *full reconstruction graph* T_s for the set D_s that contains all s-, s^{-1}-queries and responses; this helps the simulator keep track of all 'FP-mode-compatible' messages (more formally, all *reconstructible* messages) that can be formed using the elements of D_s. This is accomplished by a special subroutine FullGraph. The pictorial representation of the reconstruction graph T_s is given in Figure 4.
• Adjusting the elements of the tables D_l and D_s: Whenever a new reconstructible message M is found, the simulator makes this crucial adjustment: it assigns $\mathsf{FP}^{\mathsf{S}}(M) := \mathsf{RO}(M)$. It is fairly intuitive that, if S and π produce outputs according to statistically close distributions, then the distributions of $\mathsf{FP}^{\mathsf{S}}(M)$ and $\mathsf{FP}^{\pi}(M)$ are also close. Since $\mathsf{FP}^{\mathsf{S}}(M) = \mathsf{RO}(M)$, the distributions of $\mathsf{RO}(M)$ and $\mathsf{FP}^{\pi}(M)$ are also close. This is accomplished by the subroutine MessageRecon.

8.2 Detailed description of the simulator pair $(\mathsf{S}, \mathsf{S}^{-1})$

We first describe the two most important parts of the simulator-pair: the subroutines FullGraph and MessageRecon.

FullGraph(D_s). This routine builds the full reconstruction graph T_s using all the s-, s^{-1}-queries and responses stored in D_s. Hence the name FullGraph. We do not provide the pseudocode for this subroutine, since its operation is straight-forward and brute force: every invocation FullGraph constructs the graph T_s by searching through the elements in D_s, then creating all possible nodes, and finally connecting them to create the graph T_s. In Appendix B, we compute the running time of FullGraph on the i-th query to be $\mathcal{O}(i^4)$.

MessageRecon(x, T_s). The graph T_s is already the full reconstruction graph for the set D_s. Given the current s-query x, this subroutine derives all new *reconstructible* messages M, such that, in the computation of $\mathsf{FP}^{\mathsf{S}}(M)$, the final input to S is x, and all other intermediate inputs to S are old s-queries.

To determine such messages M, first, $\mathsf{FindBranch}(x, T_s)$ collects all branches between the nodes (IV, IV') and x; then, it selects the sequence of weights $X = m_1 m_2 \cdots m_k$ for all such branches. Finally it returns a set $\{M = \mathsf{dePad}(X)\}$ for all X. If no such $M \neq \perp$ is found, then the subroutine returns the empty set.

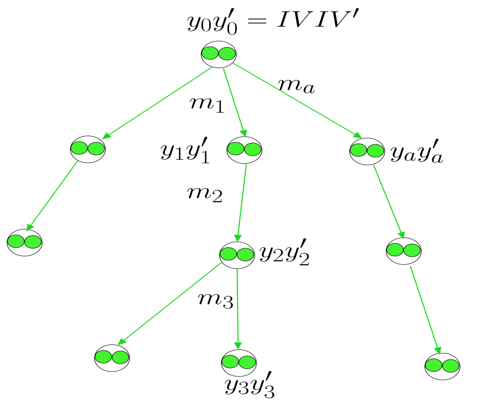


Figure 4: The reconstruction graph T_s (or T_π) updated by $\mathsf{FullGraph}$ of G2 (or $\mathsf{PartialGraph}$ of G1).

With the definition of the above subroutines, we now describe how S and S^{-1} respond to queries.

An s-query and response (for S): For an s-query, the simulator S assigns a uniformly sampled $2n$-bit value to r; if r matches an old range point in D_s then the round is aborted.[4] Then the subroutine $\mathsf{MessageRecon}(x, T_s)$ is invoked which returns a set of reconstructible messages $\mathcal{M}$. If $|\mathcal{M}| = 1$ then the RO is invoked on $M \in \mathcal{M}$, and the value is assigned to $r[n, 2n-1]$. Finally, the graph T_s is updated by $\mathsf{FullGraph}$, before r is returned. In Appendix B, we show that the worst-case running time of S after σ queries is $\mathcal{O}(\sigma^5)$.

An s^{-1}-query and response (for S^{-1}): For an s^{-1}-query, the simulator S^{-1} assigns a uniformly sampled $2n$-bit value to x; if x matches an old domain point in D_s then the round is aborted. Finally, the graph T_s is updated by $\mathsf{FullGraph}$, before x is returned. In Appendix B, we show that the worst-case running time of S^{-1} after σ queries is $\mathcal{O}(\sigma^5)$.

[4]This adjustment is required to preserve the permutativity property of D_s.

```
$\underline{\mathbf{FP1}(M)}$
001. $m_1 m_2 \cdots m_{k-1} m_k := \mathsf{pad}_n(M)$;
002. $y_0 := IV$, $y'_0 := IV'$;
003. for $(i := 1, \cdots, k)$ {
004.     $r := \pi(y_{i-1} m_i)$;
005.     $y_i y'_i := r \oplus (y'_{i-1} || 0)$;
006.     if $y_{i-1} m_i$ is fresh then
             $\mathsf{PartialGraph}(y_{i-1} m_i, r, T_\pi)$;}
007. if Type3 then $\boxed{\mathsf{BAD} := \mathrm{True}}$;
008. $r := \pi(y_k y'_k)$;
009. if Type0-b then $\boxed{\mathsf{BAD} := \mathrm{True}}$;
010. if $y_k y'_k$ is fresh then
         $\mathsf{PartialGraph}(y_k y'_k, r, T_\pi)$;
011. $D_l[M] := r[n, 2n-1]$;
012. return $D_l[M]$;

$\underline{\mathbf{MessageRecon}(x, T_s)}$
201. $\mathcal{M}' := \mathsf{FindBranch}(x, T_s)$;
202. $\mathcal{M} := \{\mathsf{dePad}(X) \mid X \in \mathcal{M}'\}$;
203. return $\mathcal{M}$;

$\underline{\boldsymbol{\pi}(x)}$
301. if $x \notin Dom(D_\pi)$ then
         $D_\pi[x] \xleftarrow{\$} \{0,1\}^{2n} \setminus Rng(D_\pi)$;
302. return $D_\pi[x]$;

$\underline{\boldsymbol{\pi}^{-1}(r)}$
501. If $r \notin Rng(D_\pi)$ then
         $D_\pi^{-1}[r] \xleftarrow{\$} \{0,1\}^{2n} \setminus Dom(D_\pi)$;
502. return $D_\pi^{-1}[r]$;

$\underline{\mathbf{S1}(x)}$
100. If Type2 then $\boxed{\mathsf{BAD} := \mathrm{True}}$;
101. $r := \pi(x)$;
102. if Type0-a then $\boxed{\mathsf{BAD} := \mathrm{True}}$;
103. $\mathcal{M} := \mathsf{MessageRecon}(x, T_s)$;
104. if $|\mathcal{M}| = 1 \wedge M \notin Dom(D_l)$ then
         $D_l[M] := r[n, 2n-1]$;
105. $D_s[x] := r$;
106. if $x$ is fresh then $\mathsf{PartialGraph}(x, r, T_\pi)$;
107. return $r$;

$\underline{\mathbf{S1}^{-1}(r)}$
601. if Type4 then $\boxed{\mathsf{BAD} := \mathrm{True}}$;
602. $x := \pi^{-1}(r)$;
603. if Type0-c then $\boxed{\mathsf{BAD} := \mathrm{True}}$;
604. if Type1-c then $\boxed{\mathsf{BAD} := \mathrm{True}}$;
605. $D_s[x] := r$;
606. if $r$ is fresh then $\mathsf{PartialGraph}(x, r, T_\pi)$;
607. return $x$;

$\underline{\mathbf{PartialGraph}(x, r, T_\pi)}$
401. $x \overset{parse}{\to} y_c m$; $r \overset{parse}{\to} y^* y'$;
402. $\mathsf{C} := \mathsf{ContactPoints}(y_c, T_\pi)$;
403. $\mathsf{E} := \{(y_c y'_c, m, y y') | y := y^* \oplus y'_c, y_c y'_c \in \mathsf{C}\}$;
404. for $\forall e \in \mathsf{E}$ {
     $\mathsf{AddEdge}(e, T_\pi)$;
     if Type1-a$\vee$Type1-b then $\boxed{\mathsf{BAD} := \mathrm{True}}$;}
```

Figure 5: System G1. $|m_i| = |m| = |y_i| = |y'_i| = |y_c| = |y'_c| = |y| = |y'| = |y^*| = |r/2| = n$, for all i.

9 Intermediate system G1

The pseudocode is provided in Figure 5. For the sake of clear understanding, we first discuss the motivation for designing this system.

9.1 Motivation for G1

The main motivation for constructing a new system G1 is that it is difficult to compare between the executions of the systems G0 and G2, instruction by instruction. The difficulty arises from the fact that G2 has a graph T_s, and two extra subroutines FullGraph, and MessageRecon, while G0 has no such graphs or subroutines. To get around this difficulty, we reduce G0 to an equivalent system G1 by endowing it with additional memory for constructing a similar graph T_π, and supplying it with the additional subroutines MessageRecon and PartialGraph. These additional components do not result in any difference in the input and output distributions of the systems G0 and G1 for any adversary (this result is formalized in Proposition 10.1); therefore, in the indifferentiability framework, G0 can be replaced by G1.

Even though G1 and G2 now appear 'close', there are still important differences. The most crucial of them is that, in the former case, the long queries are processed as a sequence of π-queries; therefore, current s- and s^{-1}-queries of G1 *may* match old π-queries and responses, while such events are not possible for G2. This difference comes with two implications:

1. The reconstruction graph T_π in G1 is built using s-, s^{-1}-, π-queries, and their responses stored in the table D_π; in case of G2, the reconstruction graph T_s is built using *only* s-, s^{-1}-queries and responses stored in D_s. This difference can be identified by separating out, from T_π, the maximally connected subgraph T_s built from all the s-, s^{-1}-queries and responses stored in D_s in G1. Now the reconstruction graph T_s in both systems are comparable.

2. In G1, the reconstruction graph T_π *may* not be *full* for the set D_π, since the subroutine PartialGraph adds only a few nodes – rather than all nodes – to T_π every round; by contrast, the reconstruction graph T_s – built by the subroutine FullGraph – for G2 is necessarily *full* for the set D_s. In Section 11, we identify a set of events in the system G1, and then, in Section 12, show that, if those events do not occur, then the reconstruction graphs in both the systems are full.

9.2 Detailed description of G1

Now we describe G1 in detail. For the moment, we postpone the description of the Type0,1, 2, 3 and 4 events until Section 11, since they do not impact the output and the global data structures of G1. We first discuss the subroutines used by the oracles $\mathsf{FP1}$, $\mathsf{S1}$ and $\mathsf{S1}^{-1}$.

$\mathsf{PartialGraph}(x, r, T_\pi)$. This subroutine is invoked whenever a fresh π- and π^{-1}-query – with $r = \pi(x)$ – is encountered. The subroutine updates the reconstruction graph T_π with (x, r) in the following way: First, the subroutine $\mathsf{ContactPoints}(y_c = x[0, n-1])$ is invoked, that returns a set C containing all nodes in T_π with y_c being least significant n bits. The size of C determines the number of fresh nodes to be added to T_π in the current iteration. Using the members of C and the new pair (x, r), new weighted edges are constructed, stored in E, and added to T_π using the subroutine $\mathsf{AddEdge}$. See Figure 4 for a pictorial description. Note that the reconstruction graph T_π may not be *full* for the elements in D_π; hence the name $\mathsf{PartialGraph}$.

$\mathsf{MessageRecon}(x, T_s)$: This subroutine has been described already in the context of G2, that determines new *reconstructible* messages. Note that the graph T_s is the maximally connected subgraph of T_π with the root-node (IV, IV'), generated by the s-, s^{-1}-queries and responses stored in D_s; x is the current s-query.

Now we describe how the oracles $\mathsf{S1}$, $\mathsf{S1}^{-1}$, and $\mathsf{FP1}$ respond to queries.

An s-query and response (for $\mathsf{S1}$): For the s-query x, $\mathsf{S1}$ computes $\pi(x)$. Then the subroutine $\mathsf{MessageRecon}(x, T_s)$ is called which returns a set of reconstructible messages $\mathcal{M}$. If $|\mathcal{M}| = 1$, and the $M \in \mathcal{M}$ is not a previous long query then $D_l[M]$ is assigned the value of $\pi(x)[n, 2n-1]$. Before finally returning r, the subroutine $\mathsf{PartialGraph}$ is called with input (x, r), if it is fresh, to update the existing graph T_π.

An s^{-1}-query and response (for $\mathsf{S1}^{-1}$): For an s^{-1}-query r, x is assigned the value of $\pi^{-1}(r)$. Finally, $D_s[x]$ and T_π are updated, and x is returned.

A long query and response (for FP1): $\mathsf{FP1}$ mimics FP, while updating the graph T_π using the subroutine $\mathsf{PartialGraph}$, whenever a *fresh* π-query is generated. $D_l[M]$ is assigned $r[n, 2n-1]$, where r is the output from the final π call. Finally, $r[n, 2n-1]$ is returned.

10 First Part of Main Theorem: Proof of (2)

From the definitions of systems G0 and G1 – in Sections 7 and 9 – we are well equipped to prove (2).

Proposition 10.1 *For any distinguishing adversary $\mathcal{A}$,*

$$\Pr\big[\mathcal{A}^{G0} \Rightarrow 1\big] = \Pr\big[\mathcal{A}^{G1} \Rightarrow 1\big].$$

PROOF. From the description of $\mathsf{S1}$ and $\mathsf{S1}^{-1}$, we observe that, for all $x \in \{0,1\}^{2n}$, $\mathsf{S1}(x) = \pi(x)$ and $\mathsf{S1}^{-1}(x) = \pi^{-1}(x)$. Likewise, from the descriptions of $\mathsf{FP1}$ and FP, for all $M \in \{0,1\}^*$, $\mathsf{FP1}(M) = \mathsf{FP}(M)$. □

11 Type0, 1, 2, 3, and 4 of System G1

In this section, we concretely define the Type0, Type1, Type2, Type3 and Type4 events of the system G1 (see Figure 5). Informally they will be called 'bad' events, since these events set the variable BAD in G1. We first provide the motivation for these events.

11.1 Motivation

We recall that the adversary submits s-, s^{-1}- and long queries to the system G1 and receives responses, and based on the history of query-response pairs, known as view – she then tries to distinguish G1 from G2. Intuitively, those events are called 'bad', for which the outputs from the π and π^{-1} oracles of G1 can be predicted by the adversary with probability better than when interacting with G2. These events primarily involve various forms of collision, occurring in the graph T_π, allowing the adversary to generate non-trivial reconstructible messages. Secondly, we need to catch the events where current queries match old queries too. One can intuit that these events help the adversary in distinguishing G1 from G2. It is also important to note that, if T_π is not a *full reconstruction graph* then the adversary can also use this fact to compel G1 to produce outputs different from those from G2 (since G2 always maintains the full reconstruction graph T_s).

Next sections deal with concrete definitions of these events, keeping the above motivation in mind.

11.2 Classifying elements of D_π, branches of T_π, and π/π^{-1}-queries

The Type0 to Type4 events depend on the elements in D_π, the branches of T_π, and the types of π- and π^{-1}-queries. In the following sections we first classify them.

11.2.1 Elements of D_π: six types

The query-response pairs of D_π are classified according to its known and $\mathsf{unknown}$ parts. The known part of a query-response pair is the part that is present in the view of the system G1, or it can be derived from the view with probability 1; the $\mathsf{unknown}$ part is $n$ot present in the view, and it cannot be derived from the view with probability 1. We observe that there are six types of such a pair, and we denote them by Q0, Q1, Q2, Q3, Q4 and Q5 in Figure 6(a); the head and tail nodes – each $2n$ bits – denote the input to, and the output from the query. Two-sided arrowhead indicates that the corresponding input-output pair is generated from either a π-or a π^{-1}-query. The *red* and *green* circles – each n bits – denote unknown and known parts.

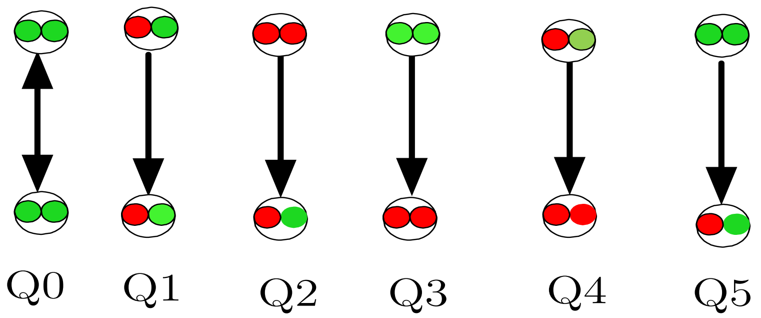
11.2.2 Branches of T_π: four types

The branches of T_π can be classified into four types, as shown in Figure 6(b). A branch B is: type I, if the final query is Q1, Q2 or Q5; type II, if the final query is Q3 or Q4; type III, if the final query is Q0, and if one of the intermediate queries is Q1, Q2, Q3, Q4 or Q5; type IV, if all queries are Q0. The first three types are called *red* branch. The fourth type is called *green* branch.

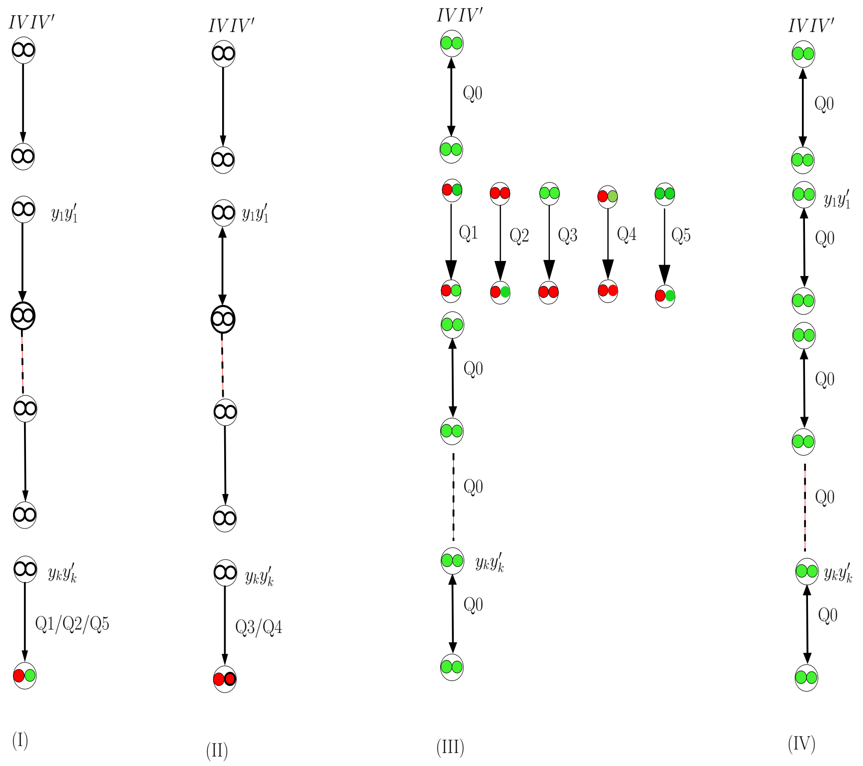
11.2.3 The π- and π^{-1}-queries: nine types

We observe that – based on the types described in the sections above – the current π- and π^{-1}-query can be categorized into the following classes.

1. Current π-query is an s-query. This can be of two types.
   (a) The π-query is fresh.
   (b) The π-query is one of six types of elements in D_π described in Section 11.2.
2. Current π^{-1}-query is an s^{-1}-query. This can be of two types.
   (a) The π^{-1}-query is fresh.
   (b) The π-query is one of six types of elements in D_π.
3. Current π-query is an intermediate π-query for the current long query. This is of three types.
   (a) Current long query is present on a *red* branch – as defined in Section 11.2 – of the graph T_π. The π-query in this case is necessarily one of six types stored in D_π; we divide it into two cases.
      i. The π-query is the final one.
      ii. The π-query is a non-final one.
   (b) Current long query is present on a *green* branch of the graph T_π. The π-query in this case is also one of six types stored in D_π.
   (c) Current long query is *not* present on a branch of the graph T_π. We divide the π-query into two types.
      i. The π-query is fresh.
      ii. The π-query is one of six types of elements in D_π.



(a) Q0, Q1, Q2, Q3, Q4, and Q5 denote six types of π/π^{-1}-query and response.



(b) Several types of a branch in T_π. (I), (II) and (II) are called *red* branch. (IV) is called *green* branch.

Figure 6: Several types of old π/π^{-1}-queries and branches in T_π.

11.3 Type0 and Type1 on Fresh queries

11.3.1 Intuition

We address the classes 1a, 2a, and 3ci of Section 11.2.3 together, since they are connected by the fact that the π- or π^{-1}-query is fresh. It is straightforward to notice that, if the outputs of the fresh queries are uniformly distributed, then distinguishing between G1 and G2 is difficult: Type0 events are designed to measure the degree to which the outputs of the π- and π^{-1}-queries are uniformly distributed.

The second scenario is when the adversary is able to generate a *non-trivial* reconstructible message, for distinguishing G1 from G2. This is possible, if the fresh π-query causes a node collision in the graph T_π, or if it causes an old query to be attached to a fresh node, or if an s^{-1}-query can be attached to a node of T_π. Type1 events cover these events. In addition, we require that the absence of these events make the graph T_π a full reconstruction graph. Detailed descriptions are below.

11.3.2 Type0: distance from the uniform

Type0 event occurs when the output of a fresh π/π^{-1}-query is *distinguishable* from the uniform distribution $\mathcal{U}[0, 2^{2n} - 1]$. A Type0 event can be of three types: event Type0-a occurs when a fresh π-query is an s-query; event Type0-b occurs when a fresh π-query is the final π-query of a long query; event Type0-c occurs when an s^{-1}-query is a fresh π^{-1}-query.

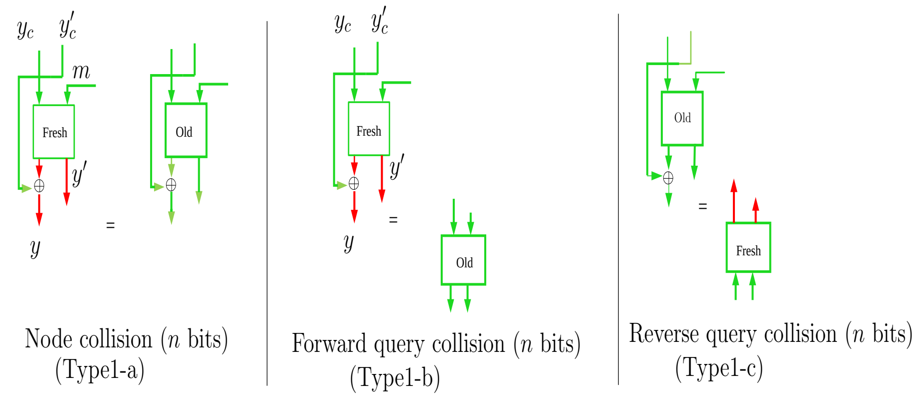

y_c
y'_c
m
Fresh
Old
y'
y
=
Node collision (n bits)
(Type1-a)
Forward query collision (n bits)
(Type1-b)
Reverse query collision (n bits)
(Type1-c)

Figure 7: Type1 events of G1. All arrows are n bits each. Red arrow denotes fresh n bits of output from the ideal permutation π/π^{-1}. The symbol "=" denotes n-bit equality.

11.3.3 Type1: collision in T_π

There are three types of Type1 events (see Figure 7). The purpose is to ensure that, if they do not occur then (1) no *non-trivial* reconstructible message can be generated by the adversary, (2) the growth of T_π every round is "small", and (3) T_π is a *full reconstruction graph* for the set D_π.

- Type1-a. Suppose yy' is a fresh $2n$-bit node generated when a fresh π-query is attached to T_π. This event occurs when yy' collides with another node in T_π; this collision can be used to generate at least two reconstructible messages in the next rounds – one of them can be used to distinguish G1 from G2. It is important to note that, even though we are interested in $2n$-bit node collision, Type1 event captures collision on the least significant n bits of the nodes. Therefore, it includes a bigger set of events than necessary. This is done to bound the growth of the graph; more precisely, it allows at most one fresh node to be added in the next round, if this event does not occur.

- Type1-b. Suppose yy' is a fresh node as defined above. This event occurs if yy' collides with any element in $Dom(D_\pi)$; like before, this event can also be used to form a *non-trivial* reconstructible message. In a similar manner as Type1-a, we define Type1-b event when y collides with the least significant n bits of any element in $Dom(D_\pi)$, and, as a result, it covers more events than required. Exactly like the Type1-a event, this is used to bound the growth of the graph, that is, it ensures that no new nodes can be added to T_π in the present round, if this event does not occur.

- Type1-c. This event occurs when the output of the current s^{-1}-query collides with any node in T_π, and thereby, the absence of this event precludes the formation of a reconstructible message. Like the previous two types, we define this event when a node and the output of the s^{-1} query collide on the least significant n bits. The absence of this event ensures that the s^{-1}-query is *not* added to T_π.

Remark: Our conservative choice of Type1 events, eventually, degrades the indifferentiability bound of FP. The bound of $n/2$ bits of this paper seems likely to be improved by relaxing the above conditions. We experimented with a smaller set of events than the ones mentioned above, and obtained an indifferentiability bound very close to n bits. However, constructing a theoretical proof of that appears to be an involved task.

11.4 Type2, Type3 and Type4 on Old queries

11.4.1 Intuition

Now we deal with the classes 1b, 2b, 3a, 3b and 3cii of Section 11.2.3. All of them address the issue when the current queries match old ones.

The classes 1b or 2b happen when an s- or s^{-1}-query matches one of six types of old elements stored in D_π; these events can potentially help the adversary in distinguishing between G1 and G2, and we identify class 1b as Type2, and class 2b as Type4 events; the case by case analysis of the events will follow in a while.

The remaining classes are now 3a, 3b and 3cii, when the adversary submits a long query – say M – to the oracle $\mathsf{FP1}$, and it is found that M is already present on some (fertile) branch of the graph T_π (3a and 3b), or it is not present *at all* on any branch of T_π (3c). The class 3c necessarily includes a fresh π-query, and this scenario has already been considered in Type0, and Type1; one can also see that class 1b (or Type2 events) already included the class 3cii. So we skip them here. The other classes – 3a and 3b – are crucial now, and they represent when M corresponds to an already present *red* or *green* branch of T_π (definitions in Section 11.2). We ignore the classes 3b, and 3aii, since they do not help the adversary in distinguishing systems.

So now we focus on the class 3ai, which deals with the final π-query of a *red* branch. Depending on the type of branch, the adversary tries to predict the most significant n bits of the final π-query (*i.e.,* the hash output) with non-trivial probability; she succeeds only for Type3 events that will be discussed shortly.

11.4.2 Type2

Recall that a query-response pair in D_π can be of six types: Q0 to Q5. Type2 event is divided into several cases depending on the type of the current s-query.

Type2-Q1, Type2-Q2, and Type2-Q4 events occur, if the s-query is type Q1, Q2 and Q4 respectively.

Type2-Q3 event occurs, if the s-query is type Q3, and if the most significant n bits are distinguishable from the uniform distribution.

Type2-Q5. We observe that a Q5 query can be located in two different types of branch in T_π, as shown in Figures 8(b)(I) and (II).

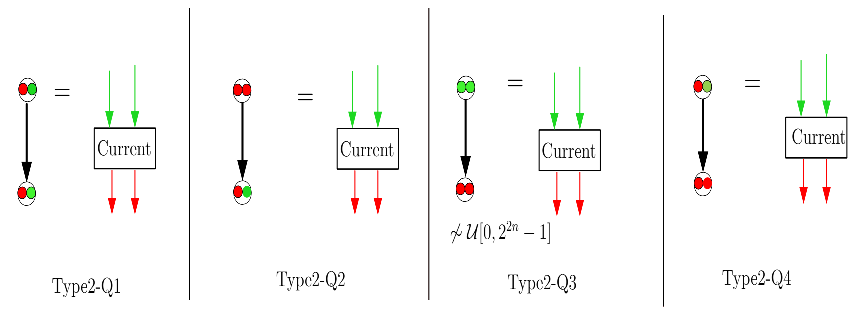
- **Type2-Q5-1** occurs if the current s-query is Q5, and is located in a type I branch, and if the least significant n bits are distinguishable from random.
- **Type2-Q5-2** occurs if the current s-query is Q5, and is located in a type II branch.

11.4.3 Type3

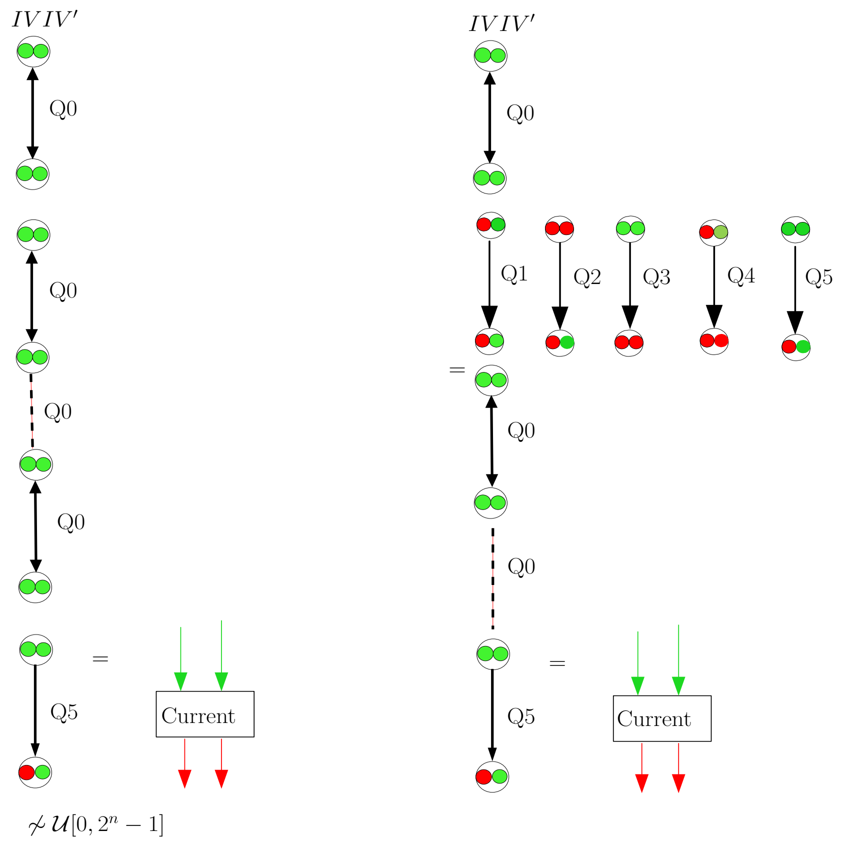
In this case, we consider the final π-query of a *red* branch as the current query. Several types of *red* branch – (I), (II), and (III) – are shown in Figure 6(b)(I) to (III).

There are three types of Type3 event: (Type3-a) if the current long query M with $m_1 m_2 \cdots m_k = \mathsf{pad}_n(M)$ forms a *red* branch of type (I).[5] (Type3-b) if M is a *red* branch of

[5]Observe that this case implies a node collision in T_π, since the $y_k y'_k$ is the final π-query for two distinct long queries, the current M and also an old one. Therefore, if Type1 event did not occur in the previous



(a) Type2-Q1 to Type2-Q4 events.



(I) Type2-Q5-1 event

(II) Type2-Q5-2 event

(b) Type2-Q5 events; they are subdivided into Type2-Q5-1 and Type2-Q5-2. The branches in (I) and (II) represent long queries.

Figure 8: Several types of Type2 events.

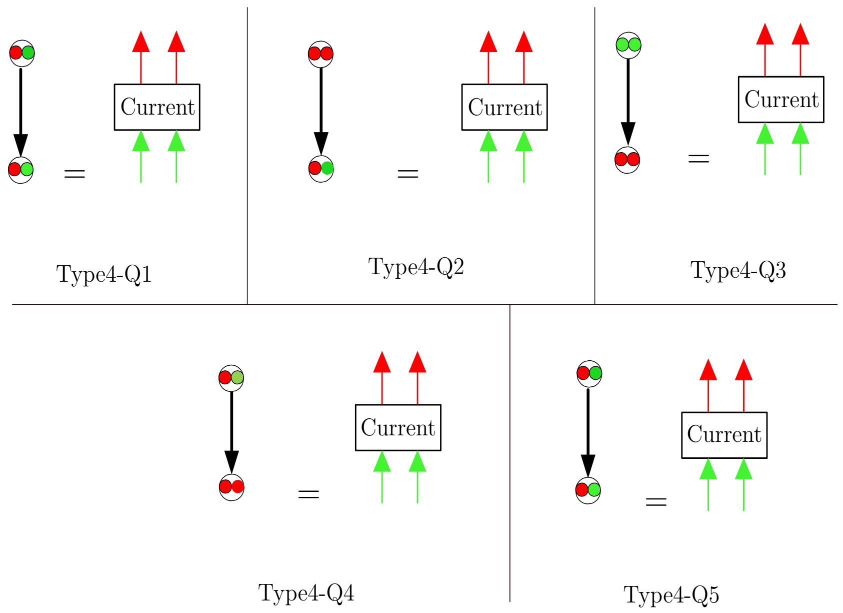


Figure 9: Several types of Type4 events: Type4-Q1 to -Q5

type (II), and if the most significant n bits of output can be distinguished from the uniform distribution $\mathcal{U}[0, 2^n - 1]$. (Type3-c) if M is a *red* branch of type (III).

11.4.4 Type4

This event is shown in Figure 9. The Type4 event occurs, if the current s^{-1}-query is equal to the output of an old query of type Q1, Q2, Q3, Q4 or Q5.

12 Second Part of Main Theorem: Proof of (3)

With the help of the Type0 to Type4 events described in Sections 11.3, and 11.4, we are equipped to prove (3). First, we first fix a few definitions.

12.1 Definitions: GOOD_i and BAD_i

GOOD_i and BAD_i. BAD_i denotes the event when the variable BAD is set during round i of G1, that is, when Type0, Type 1, Type2, Type3 or Type4 events occur. Let the symbol GOOD_i denote the event $\neg \bigvee_{j=1}^{i} \mathsf{BAD}_i$. The symbol GOOD_0 denotes the event when no queries are submitted. From a high level, the intuition behind the construction of the

rounds, this event is impossible in the current round.

BAD_i event is straightforward: we will show that if BAD_i does not occur, and if GOOD_{i-1} did occur, then the views of G1 and G2 (after i rounds) are identically distributed for *any* attacker $\mathcal{A}$.

$\mathsf{GOOD1}_i$ and $\mathsf{BAD1}_i$. In order to get around a small technical difficulty in establishing the uniform probability distribution of certain random variables, we need to modify the above events GOOD_i and BAD_i slightly. The event $\mathsf{BAD1}_i$ occurs when Type0, Type2, Type3 or Type4 events occur in the i-th round. The event $\mathsf{GOOD1}_i$ is defined as $\mathsf{GOOD}_{i-1} \wedge \neg \mathsf{BAD1}_i$.

12.2 Proof of (3)

To prove (3) we need to show two things:

$$\Big|\Pr\big[\mathcal{A}^{G1} \Rightarrow 1\big] - \Pr\big[\mathcal{A}^{G2} \Rightarrow 1\big]\Big| \le \Pr\big[\neg\mathsf{GOOD1}_\sigma\big], \tag{5}$$

$$\Pr\big[\neg\mathsf{GOOD1}_\sigma\big] \le \Pr\big[\neg\mathsf{GOOD}_\sigma\big] \le \sum_{i=1}^{\sigma} \Pr\big[\mathsf{BAD}_i \mid \mathsf{GOOD}_{i-1}\big]. \tag{6}$$

The proof of (6) is straight-forward. To prove (5), we proceed in the following way. Observe

$$\begin{aligned}
&\Big|\Pr\big[\mathcal{A}^{G1} \Rightarrow 1\big] - \Pr\big[\mathcal{A}^{G2} \Rightarrow 1\big]\Big| \\
&= \Big|\Big(\Pr\big[\mathcal{A}^{G1} \Rightarrow 1 \mid \mathsf{GOOD1}_\sigma\big] - \Pr\big[\mathcal{A}^{G2} \Rightarrow 1 \mid \mathsf{GOOD1}_\sigma\big]\Big) \cdot \Pr\big[\mathsf{GOOD1}_\sigma\big] \\
&+ \Big(\Pr\big[\mathcal{A}^{G1} \Rightarrow 1 \mid \neg\mathsf{GOOD1}_\sigma\big] - \Pr\big[\mathcal{A}^{G2} \Rightarrow 1 \mid \neg\mathsf{GOOD1}_\sigma\big]\Big) \cdot \Pr\big[\neg\mathsf{GOOD1}_\sigma\big]\Big|.
\end{aligned} \tag{7}$$

If we can show that

$$\Pr\big[\mathcal{A}^{G1} \Rightarrow 1 \mid \mathsf{GOOD1}_\sigma\big] = \Pr\big[\mathcal{A}^{G2} \Rightarrow 1 \mid \mathsf{GOOD1}_\sigma\big], \tag{8}$$

then (7) reduces to (5), since

$$\Big|\Pr\big[\mathcal{A}^{G1} \Rightarrow 1 \mid \neg\mathsf{GOOD1}_\sigma\big] - \Pr\big[\mathcal{A}^{G2} \Rightarrow 1 \mid \neg\mathsf{GOOD1}_\sigma\big]\Big| \le 1.$$

As a result, we focus on establishing (8), which is done in Appendix C.

13 Third (or Final) Part of Main Theorem: Proof of (4)

To prove (4), we need individually compute the probabilities Type0, Type1, Type2, Type3 and Type4 events described in Sections 11.3, and 11.4. Since we assume $\sum_{i=1}^{\sigma} \Pr\big[\mathsf{BAD}_i \mid \mathsf{GOOD}_{i-1}\big] \le \varepsilon = 1/2$, (6) implies that $\mathsf{GOOD}_i \ge 1/2$ for all $0 \le i \le \sigma$.

Definition of Type1 event guarantees that T_π has i nodes after $i-1$ rounds, given GOOD_{i-1}. We assume $i \le 2^{n/2}$; this implies $(2^n - i) \ge \frac{1}{2}2^n$.

13.1 Estimating probability of Type0

From Section 11.3.2 we obtain,

$$\Pr\left[\text{Type0}_i \mid \mathsf{GOOD}_{i-1}\right] \leq 3\left(\frac{1}{2^{2n}-i} - \frac{1}{2^{2n}}\right) \leq \frac{1}{2^n}.$$

13.2 Estimating probability of Type1

From Section 11.3.3 we obtain,

$$\begin{aligned}\Pr\Big[\text{Type1}_i \mid \mathsf{GOOD}_{i-1}\Big] &\leq \Pr\Big[\text{Type1-a}_i \mid \mathsf{GOOD}_{i-1}\Big] + \Pr\Big[\text{Type1-b}_i \mid \mathsf{GOOD}_{i-1}\Big] \\ &\quad + \Pr\Big[\text{Type1-c}_i \mid \mathsf{GOOD}_{i-1}\Big] \\ &\leq 3i/(2^n - i) \\ &\leq 6i/2^n.\end{aligned}$$

13.3 Estimating probability of Type2

From Section 11.4.2 we obtain,

$$\begin{aligned}&\Pr\left[\text{Type2}_i \mid \mathsf{GOOD}_{i-1}\right] \leq \frac{\Pr\left[\text{Type2}_i\right]}{\Pr\left[\mathsf{GOOD}_{i-1}\right]} \leq 2 \cdot \Pr\left[\text{Type2}_i\right] \\ &\leq 2 \cdot \Big(\Pr\left[\text{Type2-Q1}_i\right] + \Pr\left[\text{Type2-Q2}_i\right] + \Pr\left[\text{Type2-Q3}_i\right] \\ &+ \Pr\left[\text{Type2-Q4}_i\right] + \Pr\left[\text{Type2-Q5}_i\right]\Big) \\ &\leq 2 \cdot \frac{5i}{2^n - i} \leq \frac{20i}{2^n}.\end{aligned}$$

13.4 Estimating probability of Type3

From Section 11.4.3 we obtain,

$$\begin{aligned}&\Pr\left[\text{Type3}_i \mid \mathsf{GOOD}_{i-1}\right] \leq \frac{\Pr\left[\text{Type3}_i\right]}{\Pr\left[\mathsf{GOOD}_{i-1}\right]} \leq 2 \cdot \Pr\left[\text{Type3}_i\right] \\ &\leq 2 \cdot \Big(\Pr\left[\text{Type3-a}_i\right] + \Pr\left[\text{Type3-b}_i\right] + \Pr\left[\text{Type3-c}_i\right]\Big) \\ &\leq 2 \cdot \left(0 + \frac{2}{2^n - i}\right) \leq \frac{8}{2^n}.\end{aligned}$$

13.5 Estimating probability of Type4

From Section 11.4.4 we obtain,

$$\begin{aligned}
&\Pr\big[\text{Type4}_i \mid \mathsf{GOOD}_{i-1}\big] \le \frac{\Pr\big[\text{Type4}_i\big]}{\Pr\big[\mathsf{GOOD}_{i-1}\big]} \le 2\cdot\Pr\big[\text{Type4}_i\big] \\
&\le 2\cdot\Big(\Pr\big[\text{Type4-Q1}_i\big] + \Pr\big[\text{Type4-Q2}_i\big] + \Pr\big[\text{Type4-Q3}_i\big] \\
&+ \Pr\big[\text{Type4-Q4}_i\big] + \Pr\big[\text{Type4-Q5}_i\big]\Big) \\
&\le 2\cdot\frac{5i}{2^n - i} \le \frac{20i}{2^n}.
\end{aligned}$$

13.6 Final computation

We conclude by combining the above bounds into the following inequality which holds for $1 \le i \le \sigma$:

$$\begin{aligned}
&\sum_{i=1}^{\sigma} \Pr\big[\mathsf{BAD}_i \mid \mathsf{GOOD}_{i-1}\big] \\
&\le \sum_{i=1}^{\sigma} \Big[\Pr\big[\text{Type0}_i \mid \mathsf{GOOD}_{i-1}\big] + \Pr\big[\text{Type1}_i \mid \mathsf{GOOD}_{i-1}\big] + \Pr\big[\text{Type2}_i \mid \mathsf{GOOD}_{i-1}\big] \\
&+ \Pr\big[\text{Type3}_i \mid \mathsf{GOOD}_{i-1}\big] + \Pr\big[\text{Type4}_i \mid \mathsf{GOOD}_{i-1}\big]\Big] \\
&\le \sum_{i=1}^{\sigma} \frac{55i}{2^n} \le \frac{28\sigma^2}{2^n}.
\end{aligned}$$

14 A New Hash Function Family SAMOSA

Now we design a concrete hash function family SAMOSA based on the FP mode defined in Section 2. In the subsequent sections, we also provide security analysis and hardware implementation results of SAMOSA.

14.1 Description of SAMOSA

SAMOSA hash family is based on the FP mode and P-permutation of the Grøstl hash function family. Letting n denote the length of hash in bits ($n = 256$ and 512 bits), the complete description of the hash function SAMOSA-n is provided in Figure 10. SAMOSA is composed of three components: (1) The FP mode and the padding rule $\mathsf{pad}_n(\cdot)$ (see Section 2), (2) $IVIV' = \langle 0\rangle_n || \langle n\rangle_n$, and (3) the Grøstl permutation P_{2n} (see [17]).

Figure 10: SAMOSA-256 and SAMOSA-512

```
SAMOSA-256($M$)
01. $m_1 m_2 \dots m_{k-1} m_k := \mathsf{pad}_{256}(M)$;
02. $y_0 || y'_0 := \langle 0 \rangle_{256} || \langle 256 \rangle_{256}$;
03. for($i := 1, 2, \dots k$)
        $y_i || y'_i := P_{512}(y_{i-1} || m_i) \oplus (y'_{i-1} || \langle 0 \rangle_{256})$;
04. $r := P_{512}(y_k || y'_k)$;
05. return $r[256, 511]$;
```

```
SAMOSA-512($M$)
11. $m_1 m_2 \dots m_{k-1} m_k := \mathsf{pad}_{512}(M)$;
12. $y_0 || y'_0 := \langle 0 \rangle_{512} || \langle 512 \rangle_{512}$;
13. for($i := 1, 2, \dots k$)
        $y_i || y'_i := P_{1024}(y_{i-1} || m_i) \oplus (y'_{i-1} || \langle 0 \rangle_{512})$;
14. $r := P_{1024}(y_k || y'_k)$;
15. return $r[512, 1023]$;
```

14.2 Security analysis of the SAMOSA family

There are two ways to attack the SAMOSA hash function family: (1) Attacking the FP mode and (2) attacking the underlying permutation P_{512} or P_{1024}. In the next subsections we present the analysis results on the mode and the permutations. Based on that we conjecture that the SAMOSA family cannot be attacked non-trivially with work less than the brute force.

14.2.1 Security of the FP mode.

In Section 4 we have shown that the FP mode is indifferentiable from a random oracle up to approximately $2^{n/2}$ queries (up to a constant factor) where n is the hash size in bits. Our rigorous analysis with the FP mode reveals that it may be possible to improve the bound to nearly 2^n queries. The analysis implies that it is *hard* to attack any concrete hash function based on the FP mode without discovering non-trivial weaknesses in the underlying permutation. In our case, the permutations are P_{512} and P_{1024} of the Grøstl hash family.

14.2.2 Security analysis of Grøstl permutations P_{512} and P_{1024}.

The permutations P_{512} and P_{1024} of the Grøstl hash function have been two most heavily analyzed primitives in the SHA-3 hash function competition [32, 37, 19, 23, 40]. The best analysis on P_{512} so far has been the discovery of differential properties up to 9 (out of 10) rounds with work 2^{368} and memory 2^{64}; for the permutation P_{1024}, the best analysis is the discovery of differential properties up to 10 (out of 14) rounds with work 2^{392} and memory 2^{64}. Given the enormous costs to implement these attacks, and also given the huge third-party cryptanalysis the permutations of Grøstl have resisted so far, it seems fair to say that P_{512} and P_{1024} are secure for all practical purposes.

15 FPGA Implementations of SAMOSA-256 and SAMOSA-512

15.1 Motivation and previous work

In case the security of two competing cryptographic algorithms is the same or comparable, their performance in software and hardware decides which one of them get selected for use by standardization organizations and industry.

In this section, we will analyze how SAMOSA compares to Grøstl, one of the five final SHA-3 candidates, from the point of view of performance in hardware. This comparison makes sense, because both algorithms share a very significant part, permutation P, but differ in terms of the mode of operation. The FP mode requires only a single permutation P, while Grostl mode requires two permutations P and Q, executed in parallel. Our goal is to determine how much savings in terms of hardware area are introduced by replacing the Grøstl construction for hash function with the FP mode. We also would like to know whether these savings come at the expense of any significant throughput drop. Finally, we would like to analyze how significant is the improvement in terms of the throughput to area ratio, a primary metric used to evaluate the efficiency of hardware implementations in terms of a trade-off between speed and cost of the implementation.

Multiple hardware implementations of Grøstl (and its earlier variant, referred to as Grøstl-0) have been reported in the literature and in the on-line databases of results (see [38], [2]). Most of these implementations use two major hardware architecture types: a) parallel architectures, denoted (P+Q), in which Groestl permutations P and Q are implemented using two independent units, working in parallel, and b) quasi-pipeline architectures, denoted (P/Q), in which, the same unit, composed of two pipeline stages, is used to implement both P and Q, and the computations belonging to these two permutations are interleaved [16]. Additional variants of each architecture type are possible, and the two most efficient ones are the basic iterative architecture (denoted as x1), and vertically folded architecture, with the folding factor 2 (denoted as /2(v)) [16].

A summary of implementation results, obtained for various architectures, using Xilinx Virtex 5 FPGAs, is given in Table 2. Although, the implementation by Latif et al. [25] is currently the most efficient on Virtex 5, this implementation relies on the use of low-level Xilinx FPGA primitives, and as a result is not portable to FPGAs of other vendors, such as Altera. Since our implementation of SAMOSA presented in this paper is fully portable, and does not use any low-level primitives, we compare it with the second best design of Grøstl reported earlier in the literature [16], which has the same features. This design is based on the quasi-pipelined basic iterative architecture denoted as x1 (P/Q). This way, we will be also able to provide comparison for an alternative FPGA family, Stratix III from Altera.

Table 2: Implementation results for Grøstl-256, without padding unit, obtained using Xilinx Virtex 5 FPGAs.

Source	**Architecture Variant**	**Throughput [Mbit/s]**	**Area [CLB slices]**	**Thr/Area [(Mbit/s)/CLB_slices]**
Latif et al. [25]	x1 (P/Q)	6200	1419	4.37
Gaj et al. [16]	x1 (P/Q)	6117	1795	3.41
Homsirikamol et al. [18]	x1 (P/Q)	6072	1912	3.18
Gaj et al. [16]	/2(v) (P/Q)	3721	1195	3.11
Homsirikamol et al. [18]	/2(v) (P+Q)	4014	1598	2.51
Gaj et al. [16]	x1 (P+Q)	7213	2906	2.48
Baldwin et al. [2]	x1 (P+Q)	7709	3137	2.46
Guo et al. [2]	x1 (P+Q)	5027	3798	1.32

15.2 High-speed architectures of SAMOSA and Grøstl

In case of SAMOSA the best high speed architecture is the basic iterative architecture shown in Figure 11. In this architecture, a single round of the permutation P is implemented as a combinational logic, and executed in a single clock cycle. As a result, r clock cycles are required to process each h-bit message block (where r is the number of SAMOSA rounds; $r = 10$ for $h = 256$, and $r = 14$ for $h = 512$), and the throughput becomes equal to $h/(r \cdot T_{CLK}) = f_{CLK} \cdot h/r$.

In case of Grøstl, the best high-speed architecture, based on Table 2, is a quasi-pipelined architecture, denoted as x1 (P/Q). This architecture is shown in Figure 12. The most important difference compared to the architecture of SAMOSA is that the central part of this architecture can be used to implement either a round of P or a round of Q, depending on a value of a control signal. We denote this logic as the P/Q round. Additionally, in order to speed up processing, we introduce a pipeline register that divides this logic into two independent pipeline stages. As a result, at the same time, one of these stages can process a part of permutation P, and the other can process a part of permutation Q. A total of $2r + 1$ clock cycles are required to finish r rounds of both P and Q, and the clock frequency increases compared to the non-pipelined version. The throughput of this architecture is given by $b/((2r + 1) \cdot T_{CLK}) = f_{CLK} \cdot b/(2r + 1) = f_{CLK} \cdot 2h/(2r + 1)$, where $b = 2h$ is a message block size, and the datapath width in the Grøstl architecture.

For fairness, both designs use the same circuit interface, proposed in [12], the same design methodology, and the same coding style. In particular, both designs use 64-bit input and output data buses, and the standard I/O units known as SIPO (Serial-In Parallel-Out) and PISO (Parallel-In Serial-Out).

The padding units of SAMOSA and Grøstl are illustrated in Figure 13. The major difference between these two padding units is the existence of Block Counter in the padding unit of Grøstl. This counter and the following multiplexer have a small affect on the area of the Grøstl implementation with padding unit, but are not likely to affect the critical

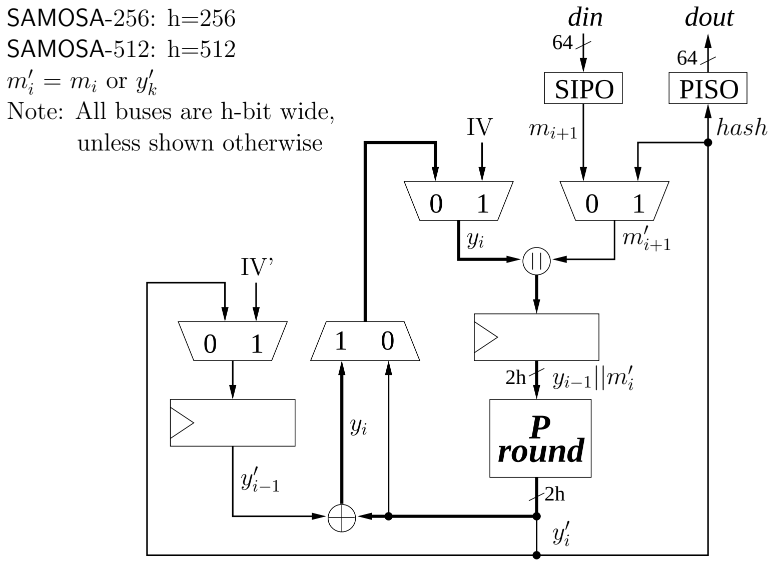


Figure 11: Basic iterative architecture of SAMOSA.

path, and thus throughput of the entire circuit.

15.3 Comparison of SAMOSA and Grøstl in terms of the hardware performance

Below, we compare SAMOSA and Grøstl in terms of three major hardware performance metrics: Area, Throughput, and Throughput to Area Ratio. The exact results of the comparison are shown in Tables 3 and 4. All results were generated using Xilinx ISE v13.1 and Altera Quartus II v11.1. Automated Tool for Hardware EvaluatioN (ATHENa) [2] was used to automate the optimization and result extraction process. No low-level primitives and no embedded resources (such as Block Memories or DSP units) were used in our implementations, which makes them fully portable among multiple FPGA families from various vendors. Each design has been implemented in two different versions: with and without padding unit. The designs with padding unit are more complete, while the designs without padding units are more suitable for comparison with hardware implementations presented in earlier academic papers on Grøstl and other SHA-3 candidates (as these implementations typically did not contain padding units).

15.3.1 Comparison in terms of Area

As shown in Tables 3 and 4, for comparable hardware architectures, SAMOSA has significantly lower area requirements than Grøstl. For Xilinx FPGAs, the area reduction is

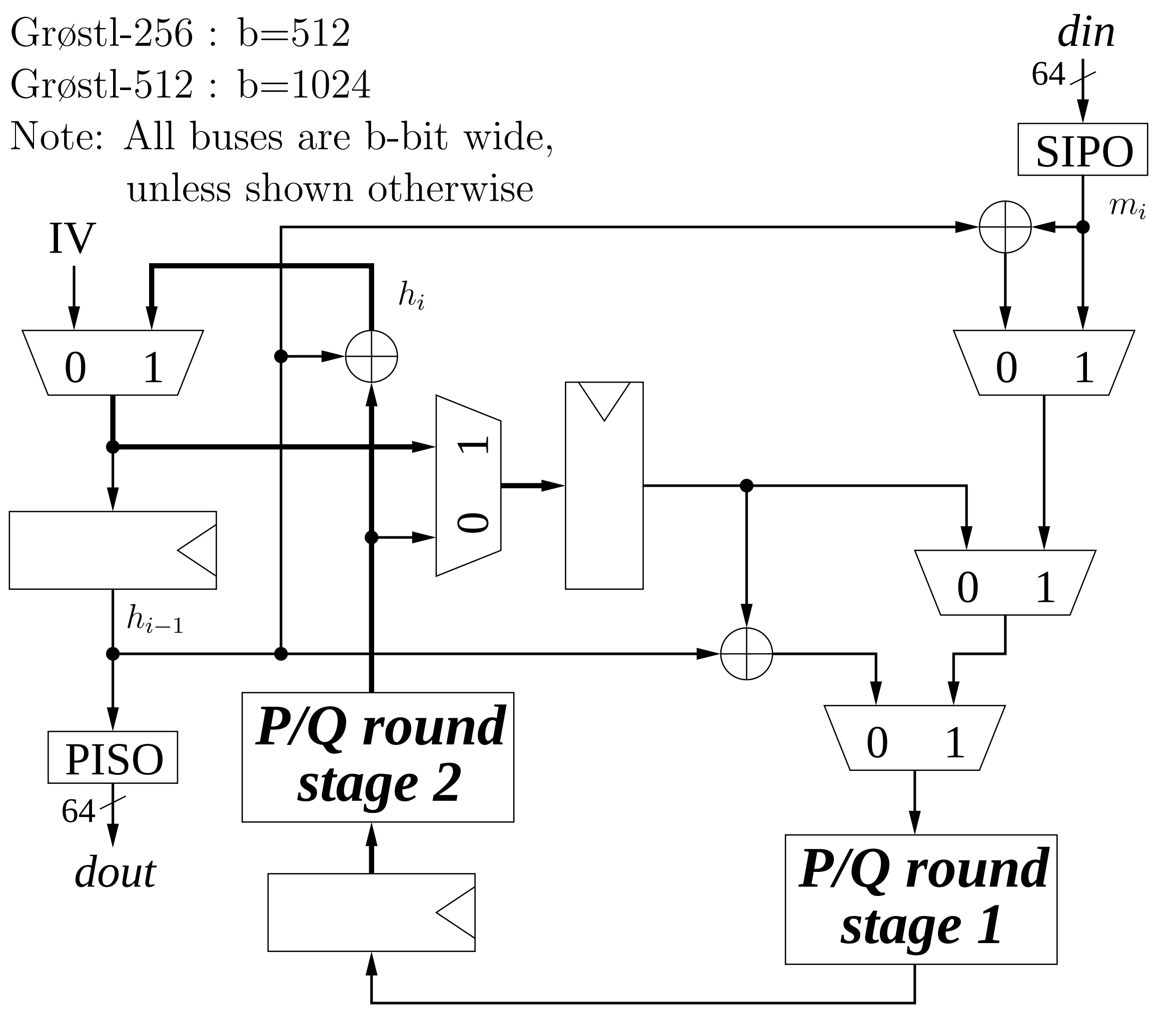


Figure 12: Basic iterative quasi-pipelined architecture of Grøstl, denoted as x1 (P/Q).

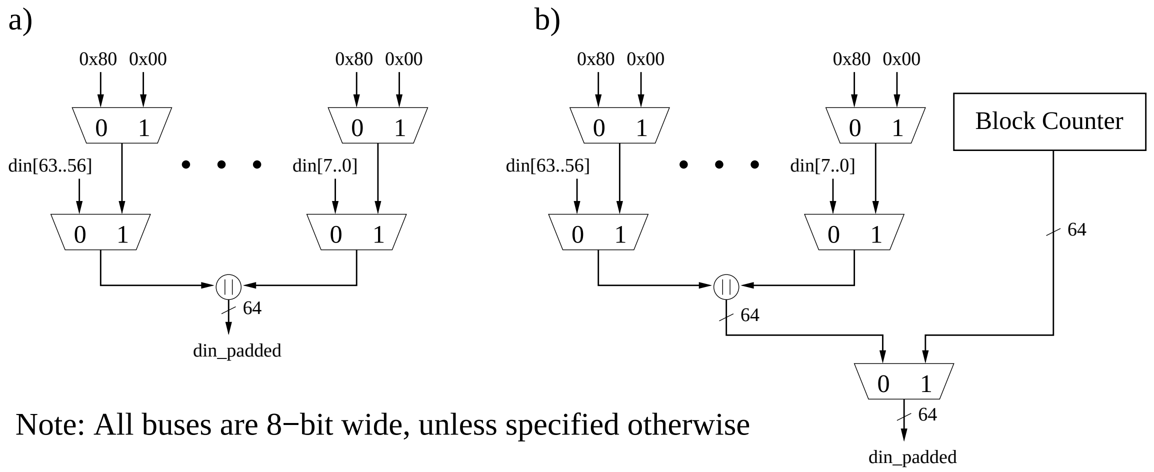


Figure 13: a) SAMOSA padding unit b) Grøstl padding unit.

between 27 and 35%; for Altera FPGAs, it is between 31 and 34%. This reduction is explained as follows. First, P round is simpler than P/Q round, as the relevant logic does not need to be switched from implementing P permutation to implementing Q permutation of Grøstl. Although both permutations are quite similar, they still differ in two out of four major operations: AddRoundConstant and ShiftBytes. Additional area requirements may result from inserting a pipeline register between two stages of the P/Q round, as shown in Figure 12 (in some FPGA families, these registers may be combined with the preceding logic and no increase in the number of configurable logic units will be observed). Secondly, SAMOSA requires less surrounding logic than Grøstl. The total width of registers outside of the P round in the basic iterative architecture of SAMOSA is $3h$. In Grøstl, the registers outside of the P/Q round have the total width of $2b = 4h$. The total width of the multiplexers, outside of the P round in SAMOSA is $4h$. The width of similar multiplexers outside of the P/Q round in Grøstl is $5b = 10h$. Finally, the number of the 2-input XOR gates in SAMOSA is h, while in Grøstl it is $3b = 6h$. Additionally, in the designs with padding unit, SAMOSA benefits from eliminating Block Counter from the padding logic, as shown in Figure 13. All these differences amount to a significant advantage of SAMOSA over Grøstl in terms of the circuit area. This advantage is particularly important taking into account that one of the major weakness of Grøstl is its inherently large area in any high-speed hardware implementations.

15.3.2 Comparison in terms of Throughput

In terms of Throughput, SAMOSA and Grøstl have very similar equations for Throughput. For SAMOSA the Throughput is given by $f_{CLK} \cdot h/r$, while for Grøstl it is $f_{CLK} \cdot 2h/(2r+1)$. Since r is relatively large (10 for the 256-bit hash function variants, and 14 for the 512-bit

Table 3: Implementation results of Grøstl and SAMOSA for Xilinx Virtex 5. CLB stands for Configurable Logic Block.

	Grøstl	**Samosa**	**Percentage Difference [%]**	**Grøstl**	**Samosa**	**Percentage Difference [%]**
	Without Padding Unit			*With Padding Unit*		
256-bit						
Frequency (MHz)	250.9	215.5	-14.1	269.5	217.0	-19.5
Throughput (Mbit/s)	6117	5516	-9.8	6572	5556	-15.5
Area (CLB slices)	1795	1305	-27.3	2020	1318	-34.8
Throughput/Area ((Mbit/s)/CLB slices)	3.41	4.23	24.0	3.25	4.22	29.6
512-bit						
Frequency (MHz)	217.7	195.0	-10.4	211.3	199.0	-5.9
Throughput (Mbit/s)	7686	7133	-7.2	7462	7276	-2.5
Area (CLB slices)	3853	2559	-33.6	3895	2732	-29.9
Throughput/Area ((Mbit/s)/CLB slices)	1.99	2.79	39.7	1.92	2.66	39.0

Table 4: Implementation results of Grøstl and SAMOSA for Altera Stratix III. ALUT stands for Adaptive Look-Up Table.

	Grøstl	**Samosa**	**Percentage Difference [%]**	**Grøstl**	**Samosa**	**Percentage Difference [%]**
	Without Padding Unit			*With Padding Unit*		
256-bit						
Frequency (MHz)	246.4	233.2	-5.4	251.8	238.7	-5.2
Throughput(Mbit/s)	6008	5969	-0.6	6140	6111	-0.5
Area (ALUTs)	7386	4851	-34.3	7564	5082	-32.8
Throughput/Area ((Mbit/s)/ALUTs)	0.81	1.23	51.3	0.81	1.20	48.1
512-bit						
Frequency (MHz)	232.6	226.9	-2.5	235.4	223.1	-5.2
Throughput (Mbit/s)	8214	8298	1.0	8310	8157	-1.8
Area (ALUTs)	14291	9810	-31.4	14578	9833	-32.5
Throughput/Area ((Mbit/s)/ALUTs)	0.57	0.85	47.2	0.57	0.83	45.5

hash function variants), $2h/(2r+1) \approx h/r$, and thus the primary difference comes from different clock frequencies.

As shown in Tables 3 and 4, the quasi-pipelined implementation of Grøstl has higher clock frequency than the basic iterative architecture of SAMOSA. However, this difference is relatively small. It does not exceed 20% for Xilinx Virtex 5 implementations, and 6% in case of Altera Stratix III implementations.

The critical paths of both architectures are marked with bold lines in Figures 11 and 12. In case of SAMOSA the critical path includes P round, one XOR gate, and two multiplexers. In case of Grøstl, it covers P/Q round stage 2, one XOR gate, and two multiplexers. In theory, one could expect a larger difference in frequency due to pipelining. However, in practice, the effect of pipelining is limited due to difficulties of dividing critical path into two equal halves. Additionally, the frequency of Grøstl before pipelining is already quite high (and similar to the frequency of SAMOSA), and its increase is limited also by the delays of other signal paths in the circuit.

15.3.3 Effect of padding in low-area architectures

SAMOSA has a simpler padding unit. The difference is shown in Figure 13 for the case of byte padding, *i.e.,* padding of messages that end on a boundary of a byte. The elimination of Block Counter reduces the complexity of the control unit as well as the area associated with the padding logic. This reduction, although relatively minor for high-speed implementations, may prove to be quite significant for low area implementations.

15.4 Comparison of SAMOSA with the SHA-3 finalists

Tables 5 and 6 present the comparison between SAMOSA and the SHA-3 finalists using the best single-message architecture, *i.e.,* architecture capable of processing only one message at a time. All algorithms have been implemented without padding units, in two variants, with 256-bit and 512-bit output, in Xilinx Virtex 5 and Altera Stratix III FPGAs. The primary metric used for comparison is throughput to area ratio. All results, other than the results for SAMOSA, are based on [16].

In terms of the throughput to area ratio, SAMOSA performs consistently better than BLAKE, Grøstl and Skein, and loses only to Keccak and JH in both 256-bit and 512-bit variants. Additionally, it reduces the gap in performance to Keccak and JH as compared to Grøstl. It also outperforms Skein in the 512-bit variant on Xilinx Virtex 5, where Grøstl loses to Skein. Furthermore, due to its similarity to Grøstl, SAMOSA has an additional advantage compared to other SHA-3 candidates when resource sharing with the Advanced Encryption Standard (AES) is possible, as demonstrated in [36].

16 Conclusion and Open Problems

This paper gives proposal for a novel permutation based hash mode of operation named FP. Our indifferentiability security analysis establishes that the new mode is secure against all generic attacks up to approximately $2^{n/2}$ queries; more interestingly, our experimental results, based on randomly generated *reconstruction* graphs using C programs, suggest that the security bound can be improved to nearly 2^n queries (n is the hash size in bits). We leave the proof of this improved result as an open problem.

Table 5: SAMOSA and the best single message architectures of the SHA-3 finalists for the 256-bit variants of hash functions

Xilinx Virtex 5					*Altera Stratix III*				
Ranking	Architecture	Throughput (Mbits/s)	Area (CLB slices)	TP/Area	Ranking	Architecture	Throughput (Mbits/s)	Area (ALUTs)	TP/Area
Keccak	x1	13337	1369	9.74	Keccak	x1	15493	3531	4.39
JH	x1	4955	982	5.05	JH	x1	5276	3221	1.64
SAMOSA	x1	5516	1305	4.23	SAMOSA	x1	5969	4851	1.23
Grøstl	x1 (P/Q)	6117	1795	3.41	Grøstl	/2(v) (P/Q)	3818	3914	0.98
Skein	x4	3023	1218	2.48	Skein	x4	2475	3943	0.63
BLAKE	/4(v)/4(h)	389	231	1.68	BLAKE	/2(h)	2158	3553	0.61

Table 6: SAMOSA and the best single message architectures of the SHA-3 finalists for the 512-bit variants of hash functions

Xilinx Virtex 5					*Altera Stratix III*				
Ranking	Architecture	Throughput (Mbits/s)	Area (CLB slices)	TP/Area	Ranking	Architecture	Throughput (Mbits/s)	Area (ALUTs)	TP/Area
Keccak	x1	7612	1320	5.77	Keccak	x1	8526	3471	2.46
JH	x1	4686	992	4.72	JH	x1	5011	3288	1.52
SAMOSA	x1	7133	2559	2.79	SAMOSA	x1	8298	9810	0.85
Skein	x4	3084	1418	2.17	Grøstl	/2(v) (P/Q)	5262	7763	0.68
Grøstl	/2(v) (P/Q)	4816	2336	2.06	Skein	x4	2438	4006	0.61
BLAKE	/4(v)/4(h)	560	386	1.45	BLAKE	/2(h)	2928	6977	0.42

We also design a concrete hash function family SAMOSA based on the FP mode and the P permutations of the SHA-3 finalist Grøstl; we claim it is hard to attack SAMOSA with complexities significantly less than the brute force. Our FPGA hardware implementations of SAMOSA show remarkable improvement in the throughput to area ratio compared to the SHA-3 finalists Grøstl, BLAKE and Skein. It is still not known how efficient SAMOSA is in software. We leave the software implementations of SAMOSA as future work.

Acknowledgments

The authors like to thank Dustin Moody and Daniel Smith-Tone for numerous helpful discussions.

References

[1] Elena Andreeva, Bart Mennink, and Bart Preneel. The parazoa family: generalizing the sponge hash functions. Int. J. Inf. Sec., vol. 11, Number 3, pp. 149–165, 2012. (Cited on page 6.)

[2] ATHENa Project Website, `http://cryptography.gmu.edu/athena` (Cited on pages 33, 34 and 35.)

[3] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Sponge Functions. ECRYPT 2007, 2007. `http://sponge.noekeon.org/SpongeFunctions.pdf`. Accessed March 2012. (Cited on page 6.)

[4] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. On the Indifferentiability of the Sponge Construction. In Nigel P. Smart, editor, *EUROCRYPT*, volume 4965 of *Lecture Notes in Computer Science*, pages 181–197. Springer, 2008. (Cited on page 6.)

[5] Rishiraj Bhattacharyya and Avradip Mandal. On the Indifferentiability of Fugue and Luffa. In Javier Lopez and Gene Tsudik, editors, *ACNS*, volume 6715 of *Lecture Notes in Computer Science*, pages 479–497, 2011. (Cited on page 6.)

[6] Rishiraj Bhattacharyya, Avradip Mandal, and Mridul Nandi. Security Analysis of the Mode of JH Hash Function. In Seokhie Hong and Tetsu Iwata, editors, *FSE*, volume 6147 of *Lecture Notes in Computer Science*, pages 168–191. Springer, 2010. (Cited on page 6.)

[7] Eli Biham and Orr Dunkelman. A framework for iterative hash functions – HAIFA. Second NIST Cryptographic Hash Workshop, 2006, 2006. (Cited on page 5.)

[8] Alex Biryukov, Orr Dunkelman, Nathan Keller, Dmitry Khovratovich, and Adi Shamir. Key Recovery Attacks of Practical Complexity on AES-256 Variants with up to 10 Rounds. In Henri Gilbert, editor, *EUROCRYPT*, volume 6110 of *Lecture Notes in Computer Science*, pages 299–319. Springer, 2010. (Cited on page 5.)

[9] John Black, Phillip Rogaway, and Thomas Shrimpton. Black-Box Analysis of the Block-Cipher-Based Hash-Function Constructions from PGV. In Moti Yung, editor, *CRYPTO*, volume 2442 of *Lecture Notes in Computer Science*, pages 320–335. Springer, 2002. (Cited on page 5.)

[10] Simon R. Blackburn, Douglas R. Stinson, and Jalaj Upadhyay. On the complexity of the herding attack and some related attacks on hash functions. *Des. Codes Cryptography*, 64(1-2):171–193, 2012. (Cited on page 5.)

[11] Christophe De Cannière, Hisayoshi Sato, and Dai Watanabe. The Luffa Hash Function. The 1st SHA-3 Candidate Conference. (Cited on page 6.)

[12] CERG Group, George Mason University: Hardware Interface of a Secure Hash Algorithm (SHA), available on-line at `http://cryptography.gmu.edu/athena/index.php?id=interfaces` (Cited on page 34.)

[13] Jean-Sébastien Coron, Yevgeniy Dodis, Cécile Malinaud, and Prashant Puniya. Merkle-Damgård Revisited: How to Construct a Hash Function. In Victor Shoup,

editor, *CRYPTO 2005*, volume 3621 of *Lecture Notes in Computer Science*, pages 430–448. Springer, 2005. (Cited on pages 5 and 10.)

[14] 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, *FSE*, volume 5665 of *Lecture Notes in Computer Science*, pages 104–121. Springer, 2009. (Cited on page 6.)

[15] Ewan Fleischmann, Michael Gorski, and Stefan Lucks. Some Observations on Indifferentiability. In Ron Steinfeld and Philip Hawkes, editors, *ACISP*, volume 6168 of *Lecture Notes in Computer Science*, pages 117–134. Springer, 2010. (Cited on page 10.)

[16] Kris Gaj, Ekawat Homsirikamol, Marcin Rogawski, Rabia Shahid, and Malik Umar Sharif. Comprehensive Evaluation of High-Speed and Medium-Speed Implementations of Five SHA-3 Finalists Using Xilinx and Altera FPGAs. Cryptology ePrint Archive, Report 2012/368, 2012, available online at `http://eprint.iacr.org/2012/368.pdf` (Cited on pages 8, 33, 34 and 39.)

[17] Praveen Gauravaram, Lars R. Knudsen, Krystian Matusiewicz, Florian Mendel, Christian Rechberger, Martin Schläffer, and Søren S. Thomsen. Grøstl - a SHA-3 candidate. The 1st SHA-3 Candidate Conference. (Cited on pages 6 and 31.)

[18] Ekawat Homsirikamol, Marcin Rogawski, and Kris Gaj. Throughput vs. Area Trade-offs in High-Speed Architectures of Five Round 3 SHA-3 Candidates Implemented Using Xilinx and Altera FPGAs. LNCS 6917, Cryptographic Hardware and Embedded Systems workshop, CHES 2011, Nara, Japan, Sep. 28-Oct. 1, pp. 491-506. (Cited on page 34.)

[19] Jérémy Jean, María Naya-Plasencia, and Thomas Peyrin. Improved Rebound Attack on the Finalist Grstl. FSE 2012, Washington DC, March 19-21, 2012, `http://www.di.ens.fr/~jean/pub/fse2012.pdf` (Cited on page 32.)

[20] Antoine Joux. Multicollisions in Iterated Hash Functions: Application to Cascaded Constructions. In Matthew K. Franklin, editor, *CRYPTO 2004*, volume 3152 of *Lecture Notes in Computer Science*, pages 306–316. Springer, 2004. (Cited on page 5.)

[21] John Kelsey and Tadayoshi Kohno. Herding Hash Functions and the Nostradamus Attack. In Serge Vaudenay, editor, *EUROCRYPT*, volume 4004 of *Lecture Notes in Computer Science*, pages 183–200. Springer, 2006. (Cited on page 5.)

[22] John Kelsey and Bruce Schneier. Second Preimages on n-Bit Hash Functions for Much Less than 2^{n} Work. In Ronald Cramer, editor, *EUROCRYPT 2005*, volume 3494 of *Lecture Notes in Computer Science*, pages 474–490. Springer, 2005. (Cited on page 5.)

[23] Dmitry Khovratovich. Bicliques for permutations: collision and preimage attacks in stronger settings. Cryptology ePrint Archive, Report 2012/141, `http://eprint.iacr.org/2012/141` (Cited on page 32.)

[24] Özgül Küçük. *Design and Analysis of Cryptographic Hash Functions*. PhD thesis, KU Leuven, 2012. `http://www.iacr.org/phds/?p=detail&entry=777`. (Cited on page 6.)

[25] Kashif Latif, M Muzaffar Rao, Arshad Aziz, and Athar Mahboob, Efficient Hardware Implementations and Hardware Performance Evaluation of SHA-3 Finalists, The Third SHA-3 Candidate Conference, Washington, D.C., March 22-23, 2012. (Cited on pages 33 and 34.)

[26] Stefan Lucks. A failure-friendly design principle for hash functions. In Bimal K. Roy, editor, *ASIACRYPT*, volume 3788 of *Lecture Notes in Computer Science*, pages 474–494. Springer, 2005. (Cited on page 5.)

[27] Ueli M. Maurer, Renato Renner, and Clemens Holenstein. Indifferentiability, impossibility results on reductions, and applications to the random oracle methodology. In *TCC*, pages 21–39, 2004. (Cited on page 10.)

[28] Dustin Moody, Souradyuti Paul and Daniel Smith-Tone. Indifferentiability Security of the Fast Widepipe Hash: Breaking the Birthday Barrier. Cryptology ePrint Archive, Report 2011/630. (Cited on page 11.)

[29] Dustin Moody, Souradyuti Paul and Daniel Smith-Tone. Improved Indifferentiability Security Bound for the JH Mode. 3rd SHA-3 Candidate Conference, 2012. (Cited on pages 6 and 11.)

[30] NIST. Secure hash standard. In *Federal Information Processing Standard, FIPS 180-2*, April 1995. (Cited on page 5.)

[31] Mridul Nandi and Souradyuti Paul. Speeding up the wide-pipe: Secure and fast hashing. In Guang Gong and Kishan Chand Gupta, editors, *INDOCRYPT*, volume 6498 of *Lecture Notes in Computer Science*, pages 144–162. Springer, 2010. (Cited on page 6.)

[32] Christian Rechberger. Grøstl Update. 3rd SHA-3 Candidate Conference, 2012, Washington DC, USA. (Cited on page 32.)

[33] Thomas Ristenpart, Hovav Shacham, and Thomas Shrimpton. Careful with Composition: Limitations of the Indifferentiability Framework. In Kenneth G. Paterson, editor, *EUROCRYPT*, volume 6632 of *Lecture Notes in Computer Science*, pages 487–506. Springer, 2011. (Cited on page 10.)

[34] Ron Rivest. The MD5 message-digest algorithm. In *IETF RFC 1321*, 1992. (Cited on page 5.)

[35] Ron Rivest. The MD6 Hash Function. (Cited on page 6.)

[36] Marcin Rogawski and Kris Gaj, A High-Speed Hardware Architecture for AES and the SHA-3 Candidates Grøstl, 15th EUROMICRO Conference on Digital System Design Architectures, DSD 2012. (Cited on page 39.)

[37] Martin Schläffer. Updated Differential Analysis of Grøstl. January 2011, `http://www.groestl.info/groestl-analysis.pdf` (Cited on page 32.)

[38] SHA-3 Zoo Hardware Implementations, `http://ehash.iaik.tugraz.at/wiki/SHA-3_Hardware_Implementations` (Cited on page 33.)

[39] Hongjun Wu. The JH Hash Function. The 1st SHA-3 Candidate Conference (2009). (Cited on page 6.)

[40] Shuang Wu, Dengguo Feng, Wenling Wu, Jian Guo, Le Dong, and Jian Zou. (Pseudo) Preimage Attack on Round-Reduced Grøstl Hash Function and Others (Extended Version). Cryptology ePrint Archive, Report 2012/206, `http://eprint.iacr.org/2012/206`

(Cited on page 32.)

A Definitions

Definition A.1 (Random oracle) *A random oracle is a function* $\mathsf{RO} : X \to Y$ *chosen uniformly at random from the set of all* $|Y|^{|X|}$ *functions that map* $X \to Y$*. In other words, a function* $\mathsf{RO} : X \to Y$ *is a random oracle if and only if, for each* $x \in X$*, the value of* $\mathsf{RO}(x)$ *is chosen uniformly at random from* Y*.*

B Time costs of FullGraph and the simulator-pair (S, S^{-1})

Since there are i queries after i rounds, the maximum number of nodes in T_s after i round is i^2. Therefore, to construct T_s in the i-th round, the amount of time required by FullGraph is $\mathcal{O}(i^4)$. Now, if the adversary submits σ queries, then the time complexity of FullGraph is $\mathcal{O}(\sigma^5)$. Since the time of FullGraph dominates over the other costs such as MessageRecon, the worst-case simulator time complexity of (S, S^{-1}) is also $\mathcal{O}(\sigma^5)$.

C Proof of (8)

(8) is as follows: $\Pr\big[\mathcal{A}^{G1} \Rightarrow 1 \mid \mathsf{GOOD1}_\sigma\big] = \Pr\big[\mathcal{A}^{G2} \Rightarrow 1 \mid \mathsf{GOOD1}_\sigma\big]$.

Let V_i^1 and V_i^2 denote the views of the systems G1, and G2 respectively, after i queries have been processed. To prove (8), it suffices to show that given $\mathsf{GOOD1}_\sigma$, the views V_σ^1 and V_σ^2 are identically distributed. We do this by induction on the number of queries $i = \sigma$.

Induction Hypothesis: Given $\mathsf{GOOD1}_i$, V_i^1 and V_i^2 are identically distributed.

Base: When $i = 0$, then no query has been made; therefore the hypothesis is true.

Induction Step: Now assume the induction hypothesis holds. We have to show that if $\mathsf{GOOD1}_{i+1}$ occurred, then V_{i+1}^1 and V_{i+1}^2 are identically distributed.

Let (I_{i+1}^1, O_{i+1}^1) and (I_{i+1}^2, O_{i+1}^2) denote the input-output pairs for the systems G1 and G2 respectively in the $i+1$st round. Note that the induction hypothesis implies that V_i^1 and V_i^2 are identically distributed given GOOD_i occurred. Also note that $V_{i+1}^1 = V_i^1 || I_{i+1}^1 || O_{i+1}^1$, and $V_{i+1}^2 = V_i^2 || I_{i+1}^2 || O_{i+1}^2$.

A little reflection shows that proving the induction step is equivalent to proving the following proposition.

Proposition C.1 (Proof of Induction Step) *Given GOOD1$_{i+1}$ and $V_i^1 = V_i^2$*

1. *the input-views I_{i+1}^1 and I_{i+1}^2 are identically distributed;*
2. *if $I_{i+1}^1 = I_{i+1}^2$ then the output-views O_{i+1}^1 and O_{i+1}^2 are identically distributed.*

Proof.
1. This result is easy since $V_i^1 = V_i^2$.

2. To prove this, we first establish the following lemma which is the main ingredient in our proof.

Lemma C.2 *The reconstruction graphs T_s of the systems G1 and G2 are isomorphic after i rounds, given GOOD$_i$ and $V_i^1 = V_i^2$.*

Proof. For a fresh π- or π^{-1}-query, the graph T_π of system G1 is augmented in one phase (see the subroutine PartialGraph of Figure 5). In that phase, all possible nodes generated from a fresh π-query are added to the graph T_π. A straightforward analysis of the Type1 events shows that if these events do not occur then no nodes can be added beyond this phase. In other words, if Type1 events do not occur in i rounds then the T_π is a *full*

reconstruction graph for D_π. Consequently, T_s – which is a connected subgraph of T_π rooted at IV, IV' – is a *full reconstruction graph* for D_s.
We note that the graph T_s for G2 is also a *full reconstruction graph* for D_s.
Since $V_1^i = V_2^i$, the graphs T_s for G_1 and G2 are isomorphic after i rounds. $\square$

Let I^{i+1} denote the shared query input $I_1^{i+1} = I_2^{i+1}$. We continue by considering all possible cases based on a set of conditions for the system G1 in the $i+1$st round. Our decision tree produced 17 cases, which have been derived from a sequence of questions (see Figure 14): Cases 1 through 9 consider when I_{i+1} is an s-query, cases 10 through 11 consider when I_{i+1} is an s^{-1}-query, while cases 12 through 17 consider when I_{i+1} is the round input of a long query.
Case 1: s-query, Fresh, $|\mathcal{M}| = 0$.
Implication. The condition directly implies that O_{i+1}^1 follows the uniform distribution $\mathcal{U}[0, 2^{2n} - 1]$, since a Type0 event did not occur in the $i+1$st round. Since the graphs T_s are isomorphic in both systems G1 and G2 by Lemma C.2, $|\mathcal{M}| = 0$ for G2. This implies that O_{i+1}^2 follows the uniform distribution $\mathcal{U}[0, 2^{2n} - 1]$.

Case 2: s-query, Old, Type Q1, Q2, Q4, or Q5-2, $|\mathcal{M}| = 0$.
Implication. This case is impossible since $\mathsf{GOOD1}_{i+1}$ implies that Type2 event did not occur for G1 in the current $i+1$st round.

Case 3: s-query, Old, Type Q5-1, $|\mathcal{M}| = 0$.
Implication. Note that Type Q5 query must be the final π-query of some long query M. The M can be present in T_π as two different branches. In the present case, all the intermediate queries of the branch – that represents M – are Q0. Now, note that, if $\mathcal{M} = 0$, then this case is not possible. The other scenario has been considered in Case 2.

Case 4: s-query, Old, Type Q3, $|\mathcal{M}| = 0$.
Implication. The event $\mathsf{GOOD1}_{i+1}$ implies that Type2 event did not occur for G1 in the current $i+1$st round; therefore, since $|\mathcal{M}| = 0$, O_{i+1}^1 follows the uniform distribution $\mathcal{U}[0, 2^{2n} - 1]$. As the graphs T_s are isomorphic in both systems G1 and G2 by Lemma C.2, $|\mathcal{M}| = 0$ for G2. This implies that O_{i+1}^2 follows the uniform distribution $\mathcal{U}[0, 2^{2n} - 1]$, since the s-query is fresh for G2.

Case 5: s-query, $|\mathcal{M}| > 1$.
Implication. $|\mathcal{M}| > 1$ implies node collision in T_s which is impossible since $\mathsf{GOOD1}_{i+1}$ ensures that Type1 event did not occur for G1 in the previous i rounds forbidding the occurrence of node collision in T_s.

Case 6: s-query, Fresh, $|\mathcal{M}| = 1$.
Implication. Since I_{i+1} is fresh, O_{i+1}^1 follows the uniform distribution $\mathcal{U}[0, 2^{2n} - 1]$. Now,

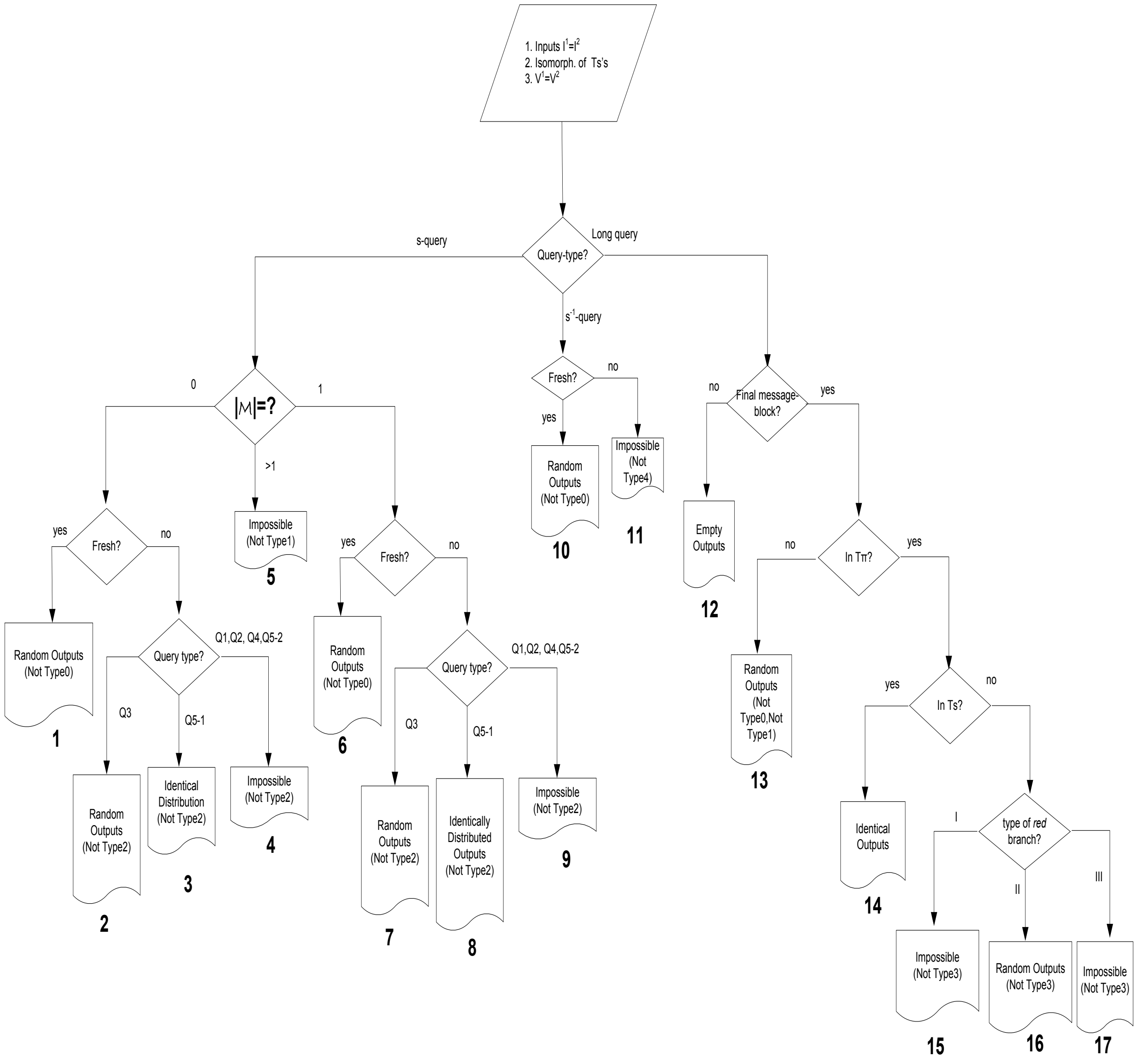


Figure 14: The decision tree for the proof of Proposition C.1. The conditions for the system G1 are shown inside the diamonds of the decision tree. The leaf-node shows the implications of the conditions to the outputs of systems G1 and G2.

for G1, $M \in \mathcal{M}$ implies that $M \notin Dom(D_l)$ in the first i rounds, since the current s-query I_{i+1} is fresh. Also note that $V_i^1 = V_i^2$ and the isomorphism of T_s's together imply that D_l in both systems are identical. Therefore, for G2 too, $M \notin Dom(D_l)$ in the first i rounds. This implies that O_{i+1}^2 follows the uniform distribution $\mathcal{U}[0, 2^{2n} - 1]$.

Case 7: s-query, Old, Type Q1, Q2, Q4, or Q5-2, $|\mathcal{M}| = 1$.
Implication. This case is impossible since $\mathsf{GOOD1}_{i+1}$ implies that Type2 event did not occur for G1 in the current $i+1$st round.

Case 8: s-query, Old, Type Q5-1, $|\mathcal{M}| = 1$.
*Implication.*The event $\mathsf{GOOD1}_{i+1}$ implies that Type2 event did not occur in the $i+1$st round of G1; therefore, $O_{i+1}^1[0, n-1]$ follows the uniform distribution $\mathcal{U}[0, 2^n - 1]$, and $O_{i+1}^1[n, 2n-1]$ is a fixed value. Now, for G1, $M \in \mathcal{M}$ implies that $M \in Dom(D_l)$ after the first i rounds, since the current s-query I_{i+1} is of type Q5-1; also note that $O_{i+1}^1[n, 2n-1] = D_l[M]$. As in the previous case, $V_i^1 = V_i^2$ and the isomorphism of T_s's together imply that D_l in both systems are identical. Therefore, $O_{i+1}^2[n, 2n-1] = D_l[M]$; also note that $O_{i+1}^2[0, n-1]$ follows the uniform distribution $\mathcal{U}[0, 2^n - 1]$. In conclusion, O_{i+1}^1 and O_{i+1}^2 are identically distributed.

Case 9: s-query, Old, Type Q3, $|\mathcal{M}| = 1$.
Implication. The event $\mathsf{GOOD1}_{i+1}$ implies that Type2 event did not occur for G1 in the current $i+1$st round; therefore, since $|\mathcal{M}| = 1$, O_{i+1}^1 follows the uniform distribution $\mathcal{U}[0, 2^{2n} - 1]$. Since the graphs T_s are isomorphic in both systems G1 and G2 by Lemma C.2, this implies that O_{i+1}^2 is the output of a fresh s-query, and therefore it follows the uniform distribution $\mathcal{U}[0, 2^{2n} - 1]$.

Case 10: s^{-1}-query and Fresh.
Implication. The condition implies that O_1^{i+1} follows the uniform distribution $\mathcal{U}[0, 2^{2n} - 1]$, since Type0 event did not occur in the current $i+1$st round. Because $V_1^i = V_2^{i+1}$, we have that the s^{-1}-query is also a fresh query for G2. Therefore, O_2^{i+1} follows the uniform distribution $\mathcal{U}[0, 2^{2n} - 1]$.

Case 11: s^{-1}-query and not Fresh.
Implication. Because of Type4 event, this case is impossible.

Case 12: π-query of long query, Non-final Block.
Implication. Since $V_{i+1}^1 = V_{i+1}^2$, it is easy to verify that $O_{i+1}^1 = O_{i+1}^2 = \lambda$, where, λ is the empty string.

Case 13: π-query of long query not in T_π, Final Block.
Implication. Let M be the long query in question. Since the event $\mathsf{GOOD1}_{i+1}$ implies that

Type1 did not occur in the previous i rounds of G1, there are no node collisions in the graph T_π. Therefore, the final π-query is fresh, implying O^1_{i+1} follows the uniform distribution $\mathcal{U}[0, 2^n - 1]$. As before, the table D_l in both systems were identical when the long query M was submitted; therefore, at that time of submission, $M \notin Dom(D_l)$ for both the systems. This ensures that $O^2_{i+1} = \mathsf{RO}(M)$ follows the uniform distribution $\mathcal{U}[0, 2^n - 1]$.

Case 14: π-query of long query in T_s, Final Block.
Implication. Since the graph T_s in both systems are isomorphic by Lemma C.2, $O^1_{i+1} = O^2_{i+1}$.

Case 15, 16 and 17: π-query of long query in T_π, not in T_s, Final Block. I_{i+1} is the final message block of a long query (denoted by M) which forms a *red* branch.

Case 15: Final π-query is Type Q1, Q2 or Q5.
Implication. Note that this case implies a node collision in T_π. By definition, the π-query – denote by $y_k y'_k$ – is already the final π-query of a previous long query, since it is of type Q1, Q2 and Q5; now, $y_k y'_k$ is also the final π-query of the current long query M. Hence the collision in T_π. This case is impossible since $\mathsf{GOOD1}_{i+1}$ implies that Type1 event did not occur in the first i rounds; therefore, T_π cannot have a node collision.

Case 16: Final π-query is Type Q3 or Q5.
Implication. Since the event Type3 did not occur in the $i + 1$st round, O^1_{i+1} follows the uniform distribution $\mathcal{U}[0, 2^n - 1]$. Now, for G1, the long query $M \notin Dom(D_l)$ when M was submitted since the final π-query of any long query cannot be of type Q3 or Q4. As the table D_l of both systems are identical, for G2, $M \notin Dom(D_l)$ when M was submitted. Therefore, $O^2_{i+1} = \mathsf{RO}(M)$ follows the uniform distribution $\mathcal{U}[0, 2^n - 1]$.

Case 17: Final π-query is Type Q0, and an intermediate query is Type Q1, Q2, Q3, Q4 or Q5.
Implication. This case is impossible since Type3 in the $i + 1$st round did not occur.

To conclude, in all 17 cases above we have shown that the outputs O^1_{i+1} and O^2_{i+1} are identically distributed if the variable BAD is not set. This completes the proof. □