ABSTRACT

We consider the cryptographic problem of constructing an invertible random permutation from a public random function (i.e., which can be accessed by the adversary). This goal is formalized by the notion of indifferentiability of Maurer et al. (TCC 2004). This is the natural extension to the public setting of the well-studied problem of building random permutations from random functions, which was first solved by Luby and Rackoff (Siam J. Comput., ’88) using the so-called Feistel construction.

The most important implication of such a construction is the equivalence of the random oracle model (Bellare and Rogaway, CCS ’93) and the ideal cipher model, which is typically used in the analysis of several constructions in symmetric cryptography.

Coron et al. (CRYPTO 2008) gave a rather involved proof that the six-round Feistel construction with independent random round functions is indifferentiable from an invertible random permutation. Also, it is known that fewer than six rounds do not suffice for indifferentiability. The first contribution (and starting point) of our paper is a concrete construction.

To overcome this gap, our main contribution is a proof that the Feistel construction with fourteen rounds is indifferentiable from an invertible random permutation. The approach of our proof relies on assigning to each of the rounds in the construction a unique and specific role needed in the proof. This avoids many of the problems that appear in the six-round case.

Categories and Subject Descriptors
F.0 [General]; E.3 [Data Encryption]

General Terms
Theory

Keywords
Cryptography, random oracle model, ideal cipher model, Feistel construction, indifferentiability

1. INTRODUCTION

1.1 Random Functions and Permutations: The Feistel Construction

Many cryptographic security proofs rely on the assumption that a concrete cryptographic function (e.g. a block cipher or a hash function) behaves as a random primitive, i.e., an ideal object which answers queries “randomly”. A typical example is a random function $F : \{0,1\}^m \rightarrow \{0,1\}^n$, which associates with each $m$-bit input $x$ a uniformly distributed $n$-bit value $F(x)$. We speak of a random oracle if the domain consists of all strings of finite length, rather than all $m$-bit ones. A random permutation $P$ is another example: It behaves as a uniformly-chosen permutation from the set of all permutations on $\{0,1\}^n$, allowing both forward queries $P(x)$ and backward queries $P^{-1}(y)$.

A well known construction in this area is the $r$-round Feistel construction $\Psi_r$: It implements a permutation taking a $2n$-bit input $(L_0, R_0)$ (where $L_0, R_0$ are $n$-bit values) to $L_r, R_r$. The output is computed via $r$ rounds, mapping $L_i, R_i$ to $L_{i+1}, R_{i+1}$ according to $(L_{i+1}, R_{i+1}) := (R_i, L_i \oplus F_{i+1}(R_i))$, where $F_1, \ldots, F_r : \{0,1\}^n \rightarrow \{0,1\}^n$ are so-called round functions. As shown by Luby and Rackoff [2], $\Psi_r$ is indistinguishable from a full-fledged random permutation when independent random functions are used as round functions.

1.2 The Random Oracle and Ideal Cipher Models: Indifferentiability

Random primitives are frequently employed to model an idealized cryptographic function accessible by all parties in the scenario at hand, including the adversary. The most prominent example is the Random Oracle Model $\mathcal{O}$, where a random oracle models an ideal hash function. Although it is known that no concrete hash function can achieve the
functionality of a random oracle \( \mathcal{R} \) (see also [22]), security proofs in the random oracle model provide a common heuristic as to which schemes are expected to remain secure when the random oracle is instantiated with a concrete hash function. In fact, to date, many widely employed practical schemes, such as OAEP [3] and FDH [4], only enjoy security proofs in the random oracle model.

The ideal cipher model is another widespread model in which all parties are granted access to an ideal cipher \( \mathcal{E} : \{0, 1\}^* \times \{0, 1\}^n \rightarrow \{0, 1\}^n \), a random primitive such that the restrictions \( \mathcal{E}(k, \cdot) \) for \( k \in \{0, 1\}^n \) are \( 2^n \) independent random permutations. Application examples of the ideal cipher model range from the analysis of block-cipher based hash function constructions (see, for example [6]) to disproving the existence of generic attacks against constructions such as cascade encryption [9] and to studying generic related-key attacks [10].

**Equivalence of models and indistinguishability.**

This paper adds to the foundational question of determining whether the random oracle model and the ideal cipher model are equivalent, where equivalence is to be understood within a simulation-based security framework such as [7]. In other words, we aim at answering the following two questions:

1. Can we find a construction \( C_1 \), which uses an ideal cipher \( \mathcal{E} \), such that \( C_1^E \) is “as good as” a random oracle \( \mathcal{R} \), meaning that any secure cryptographic scheme using \( \mathcal{R} \) remains secure when using \( C_1^E \) instead?

2. Conversely, is there \( C_2 \) such that \( C_2^R \) is “as good as” an ideal cipher \( \mathcal{E} \)?

Indistinguishability is not sufficient to satisfy the above requirement of being “as good as”, as the adversary can exploit access to the underlying primitive. Instead, the stronger notion of indistinguishability due to Maurer et al. [22] is needed: the system \( C_1^E \) is indistinguishable from \( \mathcal{R} \) if there exists a simulator \( \mathcal{S} \) accessing \( \mathcal{R} \) such that \( \{C_1^E, \mathcal{E}\} \) and \( \{\mathcal{R}, \mathcal{S}\} \) are information-theoretically indistinguishable. This is equivalent to stating that the adversary is able to locally simulate the ideal cipher consistently with \( \mathcal{R} \), given only access to the random oracle and without knowledge of the queries to \( \mathcal{R} \) of the honest users. Of course, indistinguishability generalizes to arbitrary primitives: The definition of \( C_1^E \) being indistinguishable from \( \mathcal{E} \) is analogous.

**Prior work and applications.**

Question (1) above is, to date, well understood: Coron et al. [9], and long series of subsequent work, have presented several constructions of random oracles from ideal ciphers based on hash-function constructions such as the Merkle-Damgård construction [23] with block-cipher based compression functions. In particular, indistinguishability has become a de-facto standard security requirement for hash function constructions, generally interpreted as the absence of

\[ \log(n) \]

1.3 Our Contributions

The surprising starting point of our work is a distinguishing attack which shows that the proof of [12] is not correct: For the simulator given in the proof, our attack distinguishes with overwhelming advantage. Despite hopes, at first, that we could fix the proof of Coron et al. by minor modifications, we were unable to do so. In fact, we provide a stronger attack which appears to succeed against a large class of simulators following the natural approach of [12].

Thus, the question of settling the equivalence of the ideal cipher model and the random oracle model remains open. Due to space constraints, we omit a detailed description of the attack in this version. We remark, however, that the full version can be found online [17], including simulations of the attack, and a proof of its correctness. We also sketch the problems in the proof given by [12] below.

In order to resolve the question of the equivalence of the two models, the main contribution of this paper is a proof (sketched in Section 3 and given in full in the full version) that the fourteen round Feistel construction \( \Psi_5 \) is indistinguishable from a random permutation. The round number is motivated by the goal of providing a simple to understand proof, rather than by the goal of minimizing the number of rounds. Our proof relies on techniques which are significantly different from the ones used in [12].

We discuss our results in more detail in the following section.

\[ \text{In fact, standard model instantiations of OAEP for certain classes of trapdoor functions exist [19].} \]

\[ \text{2. Usually required to be efficient, i.e., with running time polynomial in the number of queries it processes.} \]

\[ \text{3. Interestingly, we cannot construct a non-invertible random permutation from a random oracle. This follows from a well-known result by Rudich [24] and Kahn et al. [18].} \]

\[ \text{4. The notion of honest-but-curious indistinguishability is very subtle, as in general it is not even implied by full indistinguishability.} \]

\[ \text{5. Note that this implies a construction of an ideal cipher from a random oracle, as we can construct the independent round functions from a random oracle. Moreover, they can be keyed to obtain an ideal cipher for each value of the key.} \]
1.4 Sketch of the Previous Problems and the New Proof

First, we discuss the basic idea of building a random permutation from a random oracle via the r-round Feistel construction $\Psi_r$. Then we discuss the problems in the previous proofs and finally sketch our new proof.

Simulation via chain-completion.

The core of the proof is the construction of a simulator $S$ that uses a given random permutation $P : \{0,1\}^{2n} \rightarrow \{0,1\}^{2n}$ to consistently simulate $r$ independent random functions $F_1, \ldots, F_r$ from $\{0,1\}^n \rightarrow \{0,1\}^n$. In particular, suppose that a distinguisher queries the round functions to evaluate $\Psi_r$ on input $x \in \{0,1\}^{2n}$. Then, it is required that the result matches the output of $P$ on input $x$. To this end, the simulator needs to somehow recognize queries belonging to such a sequence $x_1, \ldots, x_t$, and to set the values $F_i(x_i)$ to enforce consistency with $P$. In the following, such sequences $x_1, \ldots, x_t$ will be called chains.

The natural idea used by Coron et al. is to isolate so-called partial chains among queries made to the round functions. An example of a partial chain is a triple $(x_1, x_2, x_3)$ such that $x_3 = x_1 \oplus F_2(x_2)$, and each of $x_1$, $x_2$, and $x_3$ has previously been queried to the corresponding round function $F_i$. In particular, upon each query to $F_i$, the simulator checks whether one (or more) partial chains are created. When such a partial chain is detected (and some additional conditions are met), the simulator completes it to a (full) chain $x_1, x_2, \ldots, x_r$ such that $x_{i+1} = F_i(x_i) \oplus x_{i-1}$ for all $i = 2, \ldots, r - 1$, and $P(x_0, x_1) = (x_r, x_{r+1})$, where $x_0 := F_1(x_1) \oplus x_2$ and $x_{i+1} := F_i(x_i) \oplus x_{r-i}$. In particular, the simulator defines two consecutive values $F_i(x_i)$ and $F_{i+1}(x_{i+1})$ adaptively to satisfy all constraints. In our example, the simulator could complete the partial chain by first finding $x_0$, computing $x_r$, and $x_{r+1}$ from $P(x_0, x_1)$, and finally evaluate the Feistel construction backwards, by setting each undefined $F_i(x_i)$ to a fresh uniform random string, until only $F_1(x_4)$ and $F_5(x_5)$ are undefined. These two values are then defined as $F_1(x_4) := x_3 \oplus x_5$, and $F_5(x_5) := x_4 \oplus x_6$. We refer to this step as adapting the output values of $x_4$ and $x_5$.

At this point, one faces (at least) two possible problems:

(i) The simulator defines new values at chain completion, and may keep producing new partial chains while it completes chains, hence potentially running forever. Coron et al. solve this problem very elegantly by a smart decision of which partial chains are completed, and when this occurs. Then, they are able to show that the recursion stops after at most $\text{poly}(q)$ steps, where $q$ is the number of queries the distinguisher makes to the permutation. We use their strategy in our proof, even though in the simplified version of [25], as we detect fewer chains.

(ii) The simulator may try to adapt $F_i(x_i)$ to some value, even though $F_i(x_i)$ has been fixed to a different value before. In this case the simulator by Coron et al. aborts, and it hence becomes necessary to show that no distinguisher can make the simulator abort except with negligible probability.

Breaking previous simulators.

Unfortunately, the proof given in [12] does not solve (ii) above. In fact, it is possible to find a sequence of queries such that the simulator, with high probability, needs to change the value of a previously fixed $F_i(x_i)$. Thus, the simulator given in [12] can be fully broken. A proof of this fact is contained in the full version of this paper [17].

We formally prove that our attack distinguishes with overwhelming advantage. However, in view of the complexity of the considered random experiments, we have also decided to gain extra confidence in the correctness of our proof by simulating the setting of the attack. We therefore implemented the simulator from [12] in Python, and then used our distinguisher on it. The results confirm our analysis. The code is included as an ancillary file in the full version of this paper [17], and is available for download.

We also point out that the proceedings version of [12], as well as an earlier version available on the eprint archive [11], presented a significantly simpler simulator. However, it suffered from the same problem, and a simpler attack was possible. We assume that the authors were aware of this problem when they modified the simulator, as some modifications appear to specifically rule out some of the attacks we found against the simpler simulator. However, this is speculation: no explanation is given in [12].

Moreover, in the 10-round case, Seurin gives a much simpler simulator in his PhD thesis [25]. At present, we do not know whether this simulator can be attacked.

Problems with the previous proofs.

Given our attack, it is natural to ask where the proof given in [12] fails. We explore this in the full version of the paper, but we can give a short explanation here. Consider the example above. When the simulator attempts to define the value of $F_3(x_5)$, the proof assumes that it can do so, because earlier on, $F_3(x_5)$ was chosen uniformly at random, and $x_5$ was set to be $x_5 := x_7 \oplus F_6(x_6)$. The hope is that this implies that in the meanwhile $F_3(x_5)$ has not been defined, except with very small probability. Unfortunately, between the moment where $F_6(x_5)$ was chosen uniformly, and the moment where $F_3(x_7)$ needs to be defined, the simulator may have completed a large number of other partial chains. This can destroy our expectation completely, and indeed, our attack does exploit this fact. We cannot hope to complete each detected chain immediately when it is detected. For example, the definition of a single function value may cause that we detect many new chains at the same time. These chains have to be completed in some order, and thus there exist chains such that between their detection and their completion many other function values are defined. This means that it is not obvious how to solve this problem.

Furthermore, while we do not know whether the simulator in [25] can be attacked, the problems with the proof we describe here are present in [25] as well.

Further problems with previous proofs.

There are, in fact, further problems with the previous proofs [12] [25]. All previous proofs reduced the task of proving indistinguishability to the task of upper bounding the abort probability of the given simulator. Yet, it turns out that this reduction is quite delicate as well. In fact, both proofs of [12] and [25] have several gaps in this part, which
we were not able to fill directly\footnote{In very broad terms, both proofs present a step where an ideal permutation is replaced by the Feistel construction, and values of the round functions are set by the evaluation of the construction: While each of the proofs presents a different approach how this is done, neither of them presents a convincing argument of why this modification does not affect the input-output behavior.} Thus, we give a completely new proof for this part as well.

**Ideas we use from the previous proofs.**

Since evidence points towards the fact that simulating a 6-round Feistel construction is difficult, we consider the simulator for the 10-round construction used in \cite{25}, which is significantly simpler and much more elegant. Even though our simulator is for 14 instead of 10 rounds, it is similar to the one in \cite{25}: the zones where we detect and adapt chains are analogous. This allows us to reuse the elegant idea of \cite{25} for bounding the simulator’s running time.

**Modifications and contributions in this work.**

We proceed to explain the main new ideas we use in order to solve the problems we described.

Our first idea is that we add \textit{buffer rounds} between the zones where chains are detected, and the zones where values are adapted. The function values in the buffer rounds are always defined by setting them to uniform random values. It is possible to show that at the moment a chain is detected, the corresponding function values in the buffer rounds around the relevant adapt zone have not been defined yet.

The hope is that (unless an unlikely event happens) this protects the adapt zones in the following sense: The simulator does not define the values in the buffer rounds while completing other chains. Intuitively this should hold at least if the function values next to the buffer rounds are set at random: usually the simulator should only run into trouble if some kind of collision happens. Note that it is also possible that a value next to the buffer round is defined by adapting a function value, in which case it seems unclear why there should be no collision. (Such collisions are non-trivial to prevent even if we introduce large buffer zones, because it is not easy to understand how chains can be set up in general.)

Furthermore, once the function values next to the buffer rounds are defined, it is not a priori clear why no other chain can be set up so it collides with a value in the buffer round. For example, it could be that once the input value to $F_i$ in the buffer round is known, other chains are set up so that they match this input value. However, it turns out that if we use a \textit{queue} to enqueue the chains to be completed, this cannot happen.

Showing that the values $F_i(x_i)$ for the buffer rounds really are undefined when a chain is dequeued (in order to adapt the values so that it matches the permutation) does in fact constitute a large part of our proof. Furthermore, none of the previous proofs convincingly explains why this property should hold for any round at the moment the chain is completed, even if it holds at the moment it is detected (in fact, this is exactly the property our attack on \cite{12} exploits). Once we have given this statement, it is not so hard to show that our simulator does not abort\footnote{Recall that the previous simulator aborted if it needed to overwrite a value when ensuring consistency with the permutation. Our simulator simply overwrites the value in this case, but only because the notation becomes slightly simpler. For the introduction we assume that the simulator aborts instead.} ensuring point (ii) on the previous page.

Furthermore, we still have to solve the problem of showing that if it is unlikely that the simulator aborts, then the simulated random function is indistinguishable from an actual random function. For this, we take the following, very explicit approach: we consider the two experiments which we want to show to behave almost the same, and define a map associating randomness in one experiment to randomness in the other experiment. We then study this map. This leads to a more fine-grained understanding and a much more formal treatment of the indistinguishability proof\footnote{For reference: this step can be found in Section 2.3.4}.

2. THE FEISTEL CONSTRUCTION IS INDIFFERENTIABLE FROM A RANDOM PERMUTATION

We prove that the 14-round Feistel construction is indifferentiable from a random permutation.

**Theorem 1.** The 14-round Feistel construction using 14 independent random functions is indifferentiable from a random permutation.

Using known results \cite{22,9}, this implies that the random oracle model and the ideal cipher model are equivalent.

Due to space constraints, we omit most of the proof of Theorem\footnote{For reference: this step can be found in Section 2.3.4} in this extended abstract. Indeed, we do not even define indifferentiability formally. Instead we concentrate on a lemma below, which we will explain.

2.1 Simulator Definition

As explained in the introduction, the main task is to give a simulator which uses a given random permutation and simulates the round functions of a Feistel construction in a coherent way. We call this simulator $S$. Here, we give a somewhat informal, but detailed description of a related simulator that we call $T(f)$, and which is used in the proof. In the full paper, we show as a first step that using $T(f)$ instead of $S$ does not change much. We omit this part of the proof in this extended abstract. A description of both $S$ and $T(f)$ using pseudocode can be found in Appendix A.

Additionally, in the same first step, we replace the random permutation with a two-sided random function $R(p)$, which provides an interface $P$ and $P^{-1}$. Whenever a query is made, the system checks if a previous query “observed” this input on the respective side (before as query or as answer). If so, it answers consistently. If not, it picks a \textit{uniform} random answer (from a given table $p$), and outputs it. In the same first step as explained above, we also show that instead of a permutation $P$ one can use $R(p)$.

2.1.1 Informal description

The simulator provides an interface $T.F(k, x)$ to query the simulated random function $F_k$ on input $x$. For each $k$, the simulator internally maintains a table that has entries which
are pairs \((x, y)\). They denote pairs of inputs and outputs of \(T.F(k, x)\). We denote these tables by \(T, G_k\) or just \(G_k\) when the context is clear. We write \(x \in G_k\) to denote that \(x\) is a preimage in this table, often identifying \(G_k\) with the set of preimages stored. When \(x \in G_k, G_k(x)\) denotes the corresponding image.

Upon a query \(T.F(k, x)\), the simulator first checks whether \(x \in G_k\). If so, it answers with \(G_k(x)\). Otherwise the simulator picks a random value \(y\) and inserts \((x, y)\) into \(G_k\).

In order to allow transition later, the randomness used to pick \(y\) is made explicit; it comes from a table \(f(i, x)\). After this, the simulator takes steps to ensure that in the future it answers consistently with \(R(p)\).

There are two cases in which the simulator performs a specific action for this. First, if \(k \in \{2, 13\}\), the simulator considers all newly generated tuples \((x_1, x_2, x_{13}, x_{14}) \in G_1 \times G_2 \times G_{13} \times G_{14}\), and computes \(x_0 := x_2 \oplus G_1(x_1)\) and \(x_{15} := x_{13} \oplus G_{14}(x_{14})\). It then checks whether \(R(x_0, x_1) = (x_{14}, x_{15})\) for any previous query of the distinguisher or the simulator.\(^{10}\)

Whenever the answer to such a check is positive, the simulator enqueues the detected values in a queue. More precisely, it enqueues a four-tuple \((x_1, x_2, 1, \ell)\). The value \(1\) ensures that later the simulator knows that the first value \(x_1\) corresponds to \(G_1\). The value \(\ell\) describes where to adapt values of \(G_k\) to ensure consistency with the given permutation. If \(k = 2\), then \(\ell = 4\) and if \(k = 13\) then \(\ell = 10\).

The second case is when \(k \in \{7, 8\}\). Then, the simulator enqueues all newly generated pairs \((x_7, x_8) \in G_7 \times G_8\). It enqueues all these pairs into the queue as \((x_7, x_8, 7, \ell)\), where \(\ell = 4\) if \(k = 7\) and \(\ell = 10\) if \(k = 8\) (this is illustrated in Figure 1). After enqueuing this information, the simulator immediately takes the partial chain out of the queue again, and starts completing it. For this, it evaluates the Feistel chain forward and backward (invoking \(R.P\) or \(R.P^{-1}\) at one point in order to wrap around), until \(x_\ell\) and \(x_{\ell+1}\) are computed, and only the two values \(G_\ell(x_\ell)\) and \(G_{\ell+1}(x_{\ell+1})\) are possibly undefined. The simulator defines the remaining two values in such a way that consistency is ensured, i.e., \(G_\ell(x_\ell) := x_{\ell-1} \oplus x_{\ell+1}\) and \(G_{\ell+1}(x_{\ell+1}) := x_\ell \oplus x_{\ell+2}\). If a value for either of these is defined from a previous action of the simulator, the simulator overwrites the value (possibly making earlier chains inconsistent).\(^{10}\)

During the evaluation of the Feistel chain, the simulator usually defines new values for the tables \(G\). Whenever a value \(G_k(x_k)\) for \(k \in \{2, 13\}\) is defined, the exact same checks as above are performed on the newly generated tuples \((x_1, x_2, x_{13}, x_{14})\). Whenever a value \(G_k(x_k)\) for \(k \in \{7, 8\}\) is defined, the simulator similarly enqueues all new pairs \((x_7, x_8)\).

When the simulator has finished completing a chain, it checks whether the queue is now empty. While it is not empty, it keeps dequeuing entries and completing chains, otherwise, it returns the answer to the initial query to the caller.

In order to make sure the simulator does not complete the same chains twice, the simulator additionally keeps a set \(\text{CompletedChains}\) that contains all triples \((x_k, x_{k+1}, k)\) which have been completed previously. Whenever the sim-ulator dequeues a chain, it only completes the chain if it is not in the set \(\text{CompletedChains}\).

\[\text{Lemma 1. The probability that a fixed distinguisher answers 1 in } S_2(f, p) \text{ for uniform random } (f, p) \text{ differs at most by } \frac{1}{10^{12}} \text{ from the probability that it answers 1 in } S_3(h) \text{ for uniform random } h.\]

In the full version we show how to use this lemma to prove Theorem.\(^4\) However, these steps are rather standard, so we omit them here.

\[\text{2.3 Equivalence of the Second and the Third Experiment}\]
\[\text{2.3.1 Partial chains}\]

A partial chain is a triple \((x_k, x_{k+1}, k) \in \{0, 1\}^n \times \{0, 1\}^n \times \{0, \ldots, 14\}\). Given such a partial chain \(C\), and a set of tables \(T, G\) and \(R, P\), it can be that we can move “forward” or “backward” one step in the Feistel construction (w.r.t. the values that are in \(T, G\) and \(R, P\)). This is captured in an intuitive way by the functions next and prev (i.e., next gives the next partial chain, if possible, and \(\perp\) otherwise, prev is defined analogously). Additionally, the functions \(\text{val}^+\) and \(\text{val}^-\) allow us to access additional values of the chain indexed by \(C\); \(\text{val}^+\) by invoking next, and \(\text{val}^-\) by invoking prev (i.e., \(\text{val}^+_i(C)\) gives us the value \(x_3\) of the extension of the chain \(C\), or \(\perp\), if it is not reachable by next). The function \(\text{val}_i(C)\) finally gives us the same information in case we do not want to bother about the direction. For formal definitions, we refer to the full version.

We use the convention that \(\perp \notin G_i\) for any \(i \in \{1, \ldots, 14\}\).

Thus, the expression \(\text{val}_i(C) \notin \perp\) or \(\text{val}_i(C) \neq \perp\) and \(\text{val}_i(C) \notin G_i\).

\[\text{Definition 1. For a given set of tables } G\text{ and } P\text{, two partial chains } C\text{ and } D\text{ are equivalent (denoted } C \equiv D\text{) if they are in the reflexive transitive closure of the relations given by next and prev.}\]
2.3.2 Bad events and good executions

As usual in indistinguishability proofs, for some pairs \((f,p)\) the system \(S_2(f,p)\) does not behave as “it should”. In this section we collect events which we show later to occur with low probability. We later study \(S_2(f,p)\) for pairs \((f,p)\) for which these events do not occur.

Definition 2. The set of table-defined partial chains contains all chains \(C\) for which next\((C)\) \(\neq \perp\) and prev\((C)\) \(\neq \perp\).

Definition 3. The event BadP occurs in an execution of \(S_2(f,p)\) if after a call \((x_1, x_{15}) := p(\downarrow, x_0, x_1)\) we have one of \((\uparrow, x_1, x_{15}) \in P\) or \(x_{14} \in G_{14}\). Also, it occurs if after a call \((x_0, x_1) := p(\uparrow, x_1, x_{15})\) we have one of \((\downarrow, x_0, x_1) \in P\) or \(x_1 \in G_1\).

Definition 4. The event BadlyCollide occurs if one the following happens in an execution of \(S_2(f,p)\): After an assignment \(G_1(x_k) := f(k, x_k)\) there is a table-defined partial chain \((x_k, x_{k+1}, k)\) such that prev\((x_k, x_{k+1}, k)\) \(\neq \perp\). Also, it happens if the analogous condition for next holds.

Definition 5. Let \(G\) and \(P\) be a set of tables, let \(x_k \notin G_k\), and consider two partial chains \(C\) and \(D\). An assignment \(G_k(x_k) := y\) badly collides \(C\) and \(D\) if for some \(\ell \in \{0, 15\}\) and \(\sigma, \rho \in \{\uparrow, \downarrow\}\) all of the following happen: (1) Before the assignment, \(C\) and \(D\) are not equivalent. (2) Before the assignment, \(val\sigma(C) = \perp\) or \(val\rho(D) = \perp\). (3) After the assignment, \(val\sigma(C) = val\rho(D) = \perp\).

We say that the event BadlyHit occurs in an execution \(S_2(f,p)\), if an assignment of the form \(G_i(x_i) := f(i, x_i)\) makes two partial chains badly collide, and the two chains are table-defined after the assignment.

Finally, we say that a pair \((f,p)\) is good if none of the above three events happen in an execution of \(S_2(f,p)\). The proof of the following lemma is in the full version.

Lemma 2. Suppose that \(S_2(f,p)\) is such that for any \((f,p)\) the tables satisfy \(|G_i| \leq T\) for all \(i\) and \(|P| \leq T\) at any point in the execution. Then, the probability that a uniform randomly chosen \((f,p)\) is not good is at most \(16000 \cdot \frac{r_5^k}{2^{2n}}\).

2.3.3 Properties of good executions

We now analyze some properties of executions in case \((f,p)\) is good. Most proofs in this section are omitted, and can be found in the full version of the paper [17].

Most assignments preserve equivalence among chains:

Lemma 3. Consider an execution of \(S_2(f,p)\) with a good pair \((f,p)\). Let \(C\) and \(D\) be two table-defined partial chains at some point in the execution. Suppose that after this point, there is an assignment \(G_i(x_i) := f(i, x_i)\) or a pair of assignments to \(P\) in \(R\). Then \(C \equiv D\) before the assignment(s) if and only if \(C \equiv D\) after the assignment(s).

Under some conditions, the calls to \(\text{FORCEVAL}\) also preserve equivalence.

Lemma 4. Consider an execution of \(S_2(f,p)\) with a good pair \((f,p)\). Let \(\ell \in \{4, 10\}\) and suppose that for a call \(\text{ADAPT}(x_{1-2}, x_{13-4}, x_{10-2}, x_{13+3}, \ell)\) it holds that \(x_{1-1} \notin G_{1-1}\) and \(x_{13+2} \notin G_{13+2}\) before the call.

Then, the following properties hold:

(a) For both calls \(\text{FORCEVAL}(x_, j)\) we have \(x \notin G_3\) before the call.

(b) Let \(C\) be a table-defined chain before the call to \(\text{ADAPT}\), \(i \in \{1, \ldots, 14\}\). Then, \(\text{val}_\ell(C)\) stays constant during both calls to \(\text{FORCEVAL}\).

(c) If the chains \(C\) and \(D\) are table-defined before the call to \(\text{ADAPT}\), then \(C \equiv D\) before the calls to \(\text{FORCEVAL}\) if and only if \(C \equiv D\) after the calls to \(\text{FORCEVAL}\).

Lemma 5. Suppose that \((f,p)\) is good. Fix a point in the execution of \(S_2(f,p)\), and suppose that until this point, for no call to \(\text{FORCEVAL}\) of the form \(\text{FORCEVAL}(x_, \ell)\) we had \(x \in G_\ell\) before the call. Suppose that at this point \(C = (x_k, x_{k+1}, k)\) with \(k \in \{1, 7\}\) and \(D = (y_m, y_{m+1}, m)\) with \(m \in \{1, 7\}\) are equivalent. Then, \(C \in \text{CompletedChains}\) if and only if \(D \in \text{CompletedChains}\).

Lemma 6. Consider an execution of \(S_2(f,p)\) with a good pair \((f,p)\). Suppose that at some point in the execution, a chain \(C = (x_\ell, x_\ell, 7)\) is enqueued for which \(val\sigma(C) \in G_2\) or \(val\sigma(C) \in G_{15}(C)\). Then, there is a chain equivalent to \(C\) which was previously enqueued.

If a chain \(C\) is enqueued, and previously no equivalent chain has been enqueued, then the assumptions of Lemma 6 actually do hold in good executions. We first show that they hold at the moment when the chain is enqueued (Lemma 7), and then that they still hold when the chains are dequeued (Lemma 5).

Lemma 7. Consider an execution of \(S_2(f,p)\) with a good pair \((f,p)\). Let \(C\) be a partial chain which is enqueued in the execution at some time and to be adapted at position \(\ell\). Suppose that at the moment the chain is enqueued, no equivalent chain has been previously enqueued.

Then, before the assignment \(G_\ell(x_k) := f(k, x_k)\) happens which just precedes \(C\) being enqueued, \(val\sigma(C) = \perp\) and \(val\ell(C) = \perp\).

Proof. We have \(\ell \in \{4, 10\}\). We will assume \(\ell = 4\), and due to symmetry of the construction, this also implies the lemma in case \(\ell = 10\) for the corresponding rounds. The assignment sets either the value of \(G_7(x_7)\) or \(G_7(x_7)\) uniformly (otherwise, \text{ENQUEUENewChains} is not called in the simulator). Consider first the case that \(G_7(x_7)\) was just set. Then, before this happened, \(val\ell(C) = \perp\), since otherwise \(val\ell(C) \in G_7\), thus \(\text{val}(C)\), \(val\ell(C)\) would be an equivalent, previously enqueued chain. This implies the statement in case \(G_7(x_7)\) is just set. The second case is if \(G_7(x_7)\) was just set. Then, before the assignment, \(val\ell(C) = \perp\), since otherwise \(val\ell(C) \in G_7\) and so an equivalent chain would have been previously enqueued, according to Lemma 5.

Lemma 8. Consider an execution of \(S_2(f,p)\) with a good pair \((f,p)\). Let \(C\) be a partial chain which is enqueued in the execution at some time and to be adapted at position \(\ell\). Then, at the moment \(C\) is dequeued, it holds that \(C \in \text{CompletedChains}\), or that \((val\ell-1(C) \notin G_{\ell-1}) \land (val\ell+2(C) \notin G_{\ell+2})\).

Proof. Suppose that the lemma is wrong, and let \(C\) be the first chain for which it fails. Because this is the first
chain for which it fails, Lemma 4(a) implies that until the moment
C is dequeued, no call to FORCEVAL overwrote a value.
Now, consider the set ℂ of table-defined chains at some
point in the execution that is not in an ADAPT call, and
before C is dequeued. Because of Lemmas 3 and 4(c),
the equivalence relation among chains in ℂ stays constant
from this point until the moment C is dequeued.

We distinguish two cases to prove the lemma. Consider
first the case that at the moment C is enqueued, an equiva-

alent chain D was previously enqueued. This point in the
execution is clearly not in an ADAPT call, and both C and
D are table-defined. Then, at the moment C is dequeued,
clearly D ∈ CompletedChains. Thus, because of Lemma 5
and the remark about equivalence classes of ℂ above, this
implies that C ∈ CompletedChains when it is dequeued.

The second case is when C has no equivalent chain which
was previously enqueued. To simplify notation we assume
ℓ = 4 and show val3(C) ̸∈ G3, but the argument is com-
pletely generic. From Lemma 4 we get that before the as-

signment which led to C being enqueued, val3(C) = ⊥. If
val3(C) ∈ G3 at the time C is dequeued, it must be that
G3(val3(C)) was set during completion of a chain D. This
chain D was enqueued before C was enqueued, and dequeued
after C was enqueued. Also, at the moment C is enqueued,
val3(C) = val3(D). From the point C is enqueued, at any
point until C is dequeued, it is not possible that C = D.
We assumed that there is no chain in the queue that is equiva-

lent to C when C is enqueued, and at the point C is enqueued
both C and D are table-defined. Furthermore, this point in
the execution is not during an ADAPT call. Therefore,
by our initial remark, the equivalence relation between C and
D stays constant until the moment C is dequeued.

Consider the last assignment to a table before val3(C) = val3(D) ̸= ⊥ was true. We first argue that this assignment
was of the form G3(x3) := f(i, x3), as otherwise the event
BadlyCollide would have happened to see this.
We check the conditions for BadlyCollide for C and D. The
chain D is table-defined even before the assignment, since
it is in the queue. The assignment happens earliest right
before C is enqueued, in which case C is table-defined after
the assignment. If the assignment happens later, C is table-

defined even before the assignment. Furthermore, we have
already seen that C ∈ D is not possible. Clearly, val3(C) = ⊥ or val3(D) = ⊥ before the assignment, and val3(C) = val3(D) ̸⊥ after the assignment.

The assignment cannot have been of the form P(⊥, x0, x1) =
(x1, x2), or P(⊥, x1, x2) = (x0, x1), since val can be eval-
uated at most one step further (since otherwise BadP or
BadlyHit would occur). Finally, the assignment cannot have
been in a call to FORCEVAL, because of Lemma 4(b).

Thus, val3(C) ̸∈ G3 when C is dequeued, and the same
argument holds for the other cases as well.

The following lemma is an important intermediate goal.
It states that the simulator never overwrites a value in G in
case (f, p) is good.

**Lemma 9.** Suppose that (f, p) is good. Then, for any call
to FORCEVAL of the form FORCEVAL(x, ·, ℓ) we have x ̸∈ Gℓ.

**Proof.** Assume otherwise, and let C be the first chain
during completion of which the lemma fails. Since the lemma
fails for C, C ̸∈ CompletedChains when it is dequeued.
Thus, Lemma 8 implies that both valℓ−1(C) ̸∈ Gℓ−1 and
valℓ+2(C) ̸∈ Gℓ+2 when C is dequeued, and so by Lemma 4(a)
we get the result.

### 2.3.4 Mapping randomness of S2 to randomness of S3

We next define a map τ which maps a pair of tables (f, p)
to a partial table h, where a partial table h : {1, . . . , 14} ×
{0, 1}n → {0, 1}n ∪ {⊥} either has an actual entry for a
pair (i, x), or a symbol ⊥ which signals that the entry is
unused. This map will be such that S2(f, p) and S3(τ(f, p))
have "exactly the same behaviour".

**Definition 6.** The function h = τ(f, p) is defined as fol-

ows: Run a simulation of S2(f, p) in which the distinguisher
completes all chains. If f(i, x) is read at some point, then
h(i, x) := f(i, x). If f(i, x) is never read, but for some y
a call FORCEVAL(i, x, y) occurs, then h(i, x) := y for
the first such call. If f(i, x) is never read and no such call to
FORCEVAL occurs, then h(i, x) := ⊥.

It is without loss of generality to assume that the dis-

tinguisher emulates a call to EvalFwd(x0, x0, 14) for all
queries to P(x0, x1) or to (x0, x1) = P−1(x14, x15) which
it made during the execution. The next lemmas assume
this. They follow mainly from Lemma 10. Again we omit the
proofs here.

**Lemma 10.** Suppose h has a good preimage. Consider
any execution of S1(h) and suppose the distinguisher com-
pletes all chains. Then, S3(h) never queries h on an index
(i, x) for which h(i, x) = ⊥. Furthermore, the following two
conditions on (f, p) are equivalent: (1) The pair (f, p) is
good and τ(f, p) = h. (2) The queries and answers to the
two-sided random function in S2(f, p) are exactly the same
as the queries and answers to the Feistel construction in
S1(h); and h(i, x) = f(i, x) for any query (i, x) issued to
f or h by the simulator.

The interesting part of the proof is that queries to P are
answered the same way in S(f, p) and S3(τ(f, p)). This
can be shown using the previous lemmas on good execu-
tions. The next lemma implies that h is close to uniform, which
then allows to prove Lemma 4. (see the full version).

**Lemma 11.** Suppose h has a good preimage and let (f, p)
be uniform. Then, Pr[(f, p) is good ∧ τ(f, p) = h] = 2−|h|,
where |h| is the number of pairs (i, x) for which h(i, x) ̸= ⊥.

### 3. ACKNOWLEDGEMENTS

We would like to thank Ueli Maurer and Yannick Seurin
for insightful feedback.

### 4. REFERENCES

[1] M. Bellare and T. Kohno. A theoretical treatment of
related-key attacks: RKA-PRPs, RKA-PRFs, and
applications. In Advances in Cryptology —
EUROCRYPT 2003, volume 2656 of Lecture Notes in
Computer Science, pages 491–506, 2003.

[2] M. Bellare and P. Rogaway. Random oracles are
practical: a paradigm for designing efficient protocols.
In CCS ’93: Proceedings of the 1st ACM conference
on Computer and communications security, pages
62–73, New York, NY, USA, 1993. ACM.
Variables:

1 - Queue Q
2 - Hashtables $G_1, \ldots, G_{14}$
3 - Set CompletedChains := $\emptyset$

The procedure $F(i, x)$ provides the interface to a distinguisher. It first calls the corresponding internal procedure $F_{\text{inner}}^i$, which defines the value and fills the queue if necessary. Then, the procedure $F(i, x)$ completes the chains in the queue that were not completed previously, until the queue is empty.
Whenever a function value is detected, it is completed; the function values in the red dashed zones are adapted in order to ensure consistency of the chain.

The procedure Adapt adapts the values. It first sets the values marked green in Figure 1 uniformly at random, and also the next ones. It then adapts the values of $G_i(x)$ and $G_{i+1}(x_{i+1})$ such that the chain matches the permutation.

It would be possible to simplify the code by removing lines 22 to 29 below, and changing the parameters in lines 15 and 14 above. The current notation simplifies notation in the proof.

The procedure FInner provides the internal interface for evaluations of the simulated function. It only fills the queue, but does not empty it.

The procedure EnqueueNewChains detects newly created chains and enqueues them. Sometimes, chains may be detected which have been completed before, but they are ignored when they are dequeued.
private procedure ENQUEUENEWCHAINS(i, x):
  if i = 2 then
    forall (x1, x2, x13, x14) ∈ G1 × {x} × G13 × G14 do
      if Check(x2 ⊕ G1(x1), x1, x14, x13 ⊕ G14(x14))
        then Q.QUEUE(x1, x2, 1, 4)
  else if i = 13 then
    forall (x1, x2, x13, x14) ∈ G1 × G2 × {x} × G14 do
      if Check(x2 ⊕ G1(x1), x1, x14, x13 ⊕ G14(x14))
        then Q.QUEUE(x1, x2, 1, 10)
  else if i = 7 then
    forall (x7, x8) ∈ {x} × G8 do
      Q.QUEUE(x7, x8, 7, 4)
  else if i = 8 then
    forall (x7, x8) ∈ G7 × {x} do
      Q.QUEUE(x7, x8, 7, 10)
  return P.Check(x0, x1, x14, x15)

The helper procedures EvalFwd and EvalBwd take indices k and ℓ and a pair (x_k, x_{k+1}) of input values for G_k and G_{k+1}, and either evaluate forward or backward in the Feistel to obtain the pair (x_ℓ, x_{ℓ+1}) of input values for G_ℓ and G_{ℓ+1}.

private procedure EvalFwd(x_k, x_{k+1}, k, ℓ):
  while k ≠ ℓ do
    if k = 14 then
      (x0, x1) := P^{-1}(x14, x15)
      (x0, x1) := R.P^{-1}(x14, x15)
      k := 0
      else
        x_{k+2} := x_k ⊕ FINNER(k + 1, x_{k+1})
        k := k + 1
      return (x_ℓ, x_{ℓ+1})
  while k ≠ ℓ do
    if k = 0 then
      (x_{14}, x_{15}) := P(x_0, x_1)
      (x_{14}, x_{15}) := R.P(x_0, x_1)
      k := 14
      else
        x_{k-1} := x_{k+1} ⊕ FINNER(k, x_k)
        k := k - 1
      return (x_ℓ, x_{ℓ+1})