Early Symmetric Crypto
ESC 2013

Program Chairs

Alex Biryukov University of Luxembourg, Luxembourg
Joan Daemen STMicroelectronics
Stefan Lucks Bauhaus-Universität, Weimar
Serge Vaudenay EPFL, Switzerland
Preface

Early Symmetric Cryptography (ESC) 2013 is a bi-annual five-day seminar taking place in Luxembourg. It is similar in the spirit to the Dagstuhl (Germany) and Luminy (France) seminars. This event focuses on the design and cryptanalysis of symmetric cryptographic primitives such as block ciphers, stream ciphers and hash functions and their modes of use. Additionally, it covers algorithmic challenges in cryptography in general, including any hot topics pursued by the cryptographic community. The goal of the seminar is to provide an opportunity for the top researchers in symmetric and algorithmic cryptography to freely exchange ideas, make new contacts, start new research, and plan new projects.

The 2013 event took place in Mondorf-les-Bains a small spa town in Luxembourg, which provided very warm and quiet atmosphere, amidst a rare snowy week in winter, ideal for doing research. The previous editions of ESC took place in Echternach (2008) and Remich (2010). This edition has gathered 45 cryptographers and IT security experts from academia and industry. About half of the attendees were professors/senior researchers from companies, about half were prominent post-docs and Ph.D. students. There were 35 talks of 40 minutes, 3 informal discussions and a rump session with six brief talks and announcements. A new competition CAESAR (Competition forAuthenticated Encryption: Security, Applicability, and Robustness) has been announced during one of the talks.

Here is the link to all the current and previous editions of the seminar: https://cryptolux.org/ESC/.

The program committee for the ESC 2013 event consisted of:

- A.-Prof. Alex Biryukov, Univ. Luxembourg
- Dr. Joan Daemen, STMicroelectronics, Belgium
- Prof. Stefan Lucks, Univ. Weimar, Germany
- Prof. Serge Vaudenay, EPFL, Switzerland

The ESC 2013 seminar was organized by the research lab LACS (Laboratory of Algorithms, Cryptography and Security) of the Computer Science and Communications research unit of the University of Luxembourg. The organizing committee consisted of:

- Alex Biryukov
- Yann Le Corre
- Fabienne Schmitz
- Vesselin Velichkov

This volume is a book of abstracts/extended abstracts of the talks given during the seminar.

March 2013 Alex Biryukov, Joan Daemen, Stefan Lucks, Serge Vaudenay
# Table of Contents

**Tree Hashing** .......................................................... 5  
*Stefan Lucks*

**COFFE: Ciphertext Output Feedback Faithful Encryption** ............... 6  
*Christian Forler, David McGrew, Stefan Lucks, and Jakob Wenzel*

**Differential Attacks against ARX Designs** .................................. 25  
*Gaëtan Leurent*

**Using Multiple Differentials... On the LLR and $\chi^2$ Statistical Tests** ...... 26  
*Céline Blondeau*

**A Framework for Automated Independent-Biclique Cryptanalysis** ....... 27  
*Farzaneh Abed, Christian Forler, Eik List, Stefan Lucks, and Jakob Wenzel*

**Choosing new authentication and key generation algorithms for mobiles**. 47  
*Steve Babbage*

**RFID Authentication: Security, Privacy and the Real World** ............ 48  
*Jens Hermans and Roel Peeters*

**Strong Privacy for RFID Systems from Plaintext-Aware Encryption** .... 49  
*Serge Vaudenay*

**On the Need for Secure Distance-Bounding** .......................... 52  
*Ioana Boureanu, Aikaterini Mitrokotsa, and Serge Vaudenay*
The Fundamental Theorem and Links Between Attacks on Block Ciphers

Kaisa Nyberg

Practical-Time Attacks Against Reduced Variants of MISTY1

Orr Dunkelman and Nathan Keller

Collision Attacks on Up to 5 Rounds of SHA-3 Using Generalized Internal Differentials

Itai Dinur, Orr Dunkelman, and Adi Shamir

Generalizing/Improving Long Standing Results on RC4 Key Scheduling

Subhamoy Maitra, Goutam Paul, Santanu Sarkar, Michael Lehmann, and Willi Meier

General Methodology for Automatic Search for Differential Trails in ARX Ciphers with Application to TEA and XTEA

Alex Biryukov and Vesselin Velichkov

Power Analysis of Hardware Implementations Protected with Secret Sharing

Guido Bertoni, Joan Daemen, Nicolas Debande, Thanh-Ha Le, Michaël Peeters and Gilles Van Assche

On Group Homomorphic Encryption

Frederik Armknecht, Tommaso Gagliardoni, Stefan Katzenbeisser, and Andreas Peter

Impossibility Results for Indifferentiability with Resets

Atul Luykx, Elena Andreeva, Bart Mennink, and Bart Preneel

BLAKE2: simpler, smaller, fast as MD5

Jean-Philippe Aumasson, Samuel Neves, Zooko Wilcox-O’Hearn, and Christian Winnerlein
Tree Hashing
A Simple Generic Tree Hashing Mode
Designed for SHA-2 and SHA-3,
Applicable to Other Hash Functions
(Extended Abstract)

Stefan Lucks
Bauhaus-Universität Weimar

Tree hashing deals with hash functions who’s data flow from the leaves to the root of a graph-theoretical tree. Tree hashing has been proposed by Merkle and Damgård (1989) and was a topic of discussion during the SHA-3 competition. Several of the SHA-3 candidates came with an optional or integrated tree hashing mode (MD6, SANDstorm, Skein, . . . ), and the Keccak team also presented some theoretical studies on Tree hashing. The reasons to be interested in tree hashing include

1. the availability of parallelism (for multi-core architectures, distributed computing and clouds)
2. the ability to quickly recomputate the hash, after some minor message modification,
3. and the option verify hash without reading all message blocks
   (Merkle/Lamport signatures, timestamping, . . . )

Two simplified forms of tree hashing have been discussed on the SHA-3 mailing list: Interleaving by Shay Gueron (also IACR eprint 2012/371), clustering by Dan Bernstein. As it turns out, both clustering and interleaving allow to optimize a hashing scheme for very specific applications, at the cost of generality. I.e., the performance improvement greatly depends on the parameter choices, namely the size of the clusters or the number of interleaved blocks. A different parameter implies a different tree topology and thus a different hash function – quite the opposite of a decent standard. On the other hand, real tree hashing allows to define a single tree topology (and thus a single hash function), with different ways to traverse the tree to support different applications and architectures. Thus, for a standard, a real tree hashing scheme is preferable over the simplified schemes.

There are still a lot of choices discussed in the talk, the tree arity, the size of the inner vertices and the size of the leaves. Also, a tree hashing mode could be either using the internal compression function, or the hash function itself (with padding, finalization, etc.).
COFFE: Ciphertext Output Feedback Faithful Encryption
Authenticated Encryption Without a Block Cipher

Christian Forler¹, David McGrew², Stefan Lucks¹, and Jakob Wenzel¹

¹ Bauhaus-University Weimar, Germany. ² Cisco Systems, USA
{Christian.Forler, Stefan.Lucks, Jakob.Wenzel}@uni-weimar.de
mcgrew@cisco.com

Abstract. In this paper, we introduce the first authenticated encryption scheme based on a hash function, called COFFE. This research has been motivated by the challenge to fit secure cryptography into constrained devices – some of these devices have to use a hash function, anyway, and the challenge is to avoid the usage of an additional block cipher to provide authenticated encryption. The COFFE scheme satisfies the common security requirements regarding authenticated encryption, i.e., IND-CPA and INT-CTXT security. Beyond that, it provides the following additional security features: resistance against side-channel attacks and misuse-resistance. It also support failure-friendly authentication under reasonable assumptions.

Keywords: Authenticated encryption, provable security, misuse-resistance, side-channel resistance, internet of things.

1 Introduction

An important trend in information technology is the connection of physical objects like smart meters or home automation systems to large scale networks using standard communication protocols, the “Internet of Things” (IoT). Many IoT applications need to be cryptographically secure, to prevent damage, theft, and other malfeasance.

It is a challenge to fit cryptography into a computationally constrained environment, such as a small circuit or an inexpensive microprocessor, of the sort used in many IoT devices. The manufacturers try to reduce the cost for the devices to a minimum, which usually leads to a limitation of the computing resources they can incorporate with. But, secure communication depends on algorithms for authenticated encryption (AE) and key derivation – and often also key establishment and digital signatures. It is desirable to reduce the number of cryptographic primitives needed to implement these algorithms, i.e., to keep the size of the cryptographic footprint small. Next, we introduce some possibilities to realize this approach.

Block Cipher Based AE and Hashing. There has been a lot of research on block cipher based hash functions [4, 20]. More recently, the focus is on double-block-length hash functions [14, 10, 9]. At least in part, this research has been motivated by the idea of eliminating one cryptographic primitive, namely the hash function.

However, there is no official “standard” for, say, an AES-based hash function. Furthermore, the strategy of abandoning a block cipher may be more appealing than that of abandoning a hash function, since the use of a hash function is a strict requirement for digital signatures, and its use is desirable for other operations such as deriving keys from a Diffie-Hellman shared secret. The hash-abandonment strategy has been pursued [18], but it is not clear that the forfeiture of cryptographic hashing and digital signatures will prove to be a good trade-off.

Generic Composition. Generically composing a hash-based MAC and a hash-based encryption scheme would require to maintain two different and independent keys – perhaps pseudorandomly derived from a single master key – and two different internal states, one for authentication
and one for encryption. Since memory is a very precious resource on constrained devices, generic
composition does not fit very well.

**Compression Function Based AE.** All practical hash functions are based on the idea of
iterating an internal compression function. This compression function can be seen as the real
primitive, to be used both for hashing and for authenticated encryption. In fact, we did consider
this approach at the beginning of our research. It would even allow us to design a more efficient
AE scheme than the one we actually propose.

But, while cryptographers know what is meant by the internal compression function, typical
standards, such as the SHA-2 standard [19], do not formally define it. So without an explicit
specification of a “new” cryptographic primitive, engineers (non-cryptographers) would not be
likely to properly implement the authenticated hash function. Also, while on many constrained
devices “jumping” to the address of the internal compression function may be easy, this may be
not the case for all such devices. An example for using a compression function based authenti-
cated encryption scheme is given in [3].

**Replacing the Block Cipher by a Hash Function.** If an AE scheme makes only use of the
block cipher encryption operation \( E_K(X) \) (encrypting some block \( X \) under a key \( K \)), without
ever using the decryption operation, not even for authenticated decryption, one can just replace
\( E_K(X) \) by \( H(K||X) \) (hashing the concatenation of \( K \) and \( X \)) to turn the AE scheme into a
hash function based one. This actually applies to most of the established AE schemes. Under
reasonable assumptions on the security of \( H \), the proof of security for the original scheme should
be adapted to the new situation.

**Our Contribution (Hash Function Based AE Scheme).** In this paper we introduce
**COFFE**, a novel hash function based AE scheme. It can be part of a minimal cryptographic
suite that includes hashing and digital signatures. Because it is an authenticated encryption with
associated data (AEAD) algorithm, it could be used in the AEAD interface of the Datagram TLS
security protocol [21]. That protocol has been identified by the IETF Constrained Application
working group as suitable for IoT applications. We claim that COFFE fits very well to these
requirements.

We could have used existing AE schemes like discussed or presented in [12, 15], and [16],
respectively, by replacing the internal building block by a raw hash function. But, the point
is, that a new design, as ours, can take specific properties of hash functions into account, thus
providing some advanced security properties, mainly related to a “second line of defense” for
authenticity. More precisely, our scheme provides the standard INT-CTXT and IND-CPA
(CCA3) security, one would expect from any good AE scheme, plus the following nonstandard
security features.

1. **Misuse-resistant authenticity:** It is standard for an authenticated encryption scheme to claim
   and prove security against nonce-respecting adversaries. Almost always, security breaks apart
   if nonces are ever reused [8]. While the privacy of COFFE only holds for nonce-respecting
   adversaries (unlike the scheme presented in [8]), we prove that its authenticity does not
   depend on unique nonces.

2. **Failure-friendly authenticity:** The security proofs of most modes of operation assume PRF
   or PRP security for the underlying block cipher. If this fails, the mode is likely to be insecure.
   While the privacy of COFFE requires the hash function under a secret key to behave like
   a random function, authenticity can be established even for failing pseudorandomness. Our
   mode ensures authenticity under a much weaker unpredictability assumption assuming a
   strong key. Due to space limitation, we will only give a brief and informal discussion on this
   (cf. Section 5.1).
3. Resistance against side-channel attacks: Resistance against side-channel attacks is usually a matter of the implementation of a cryptosystem, rather than of the cryptosystem itself. Nevertheless, the design of a cryptosystem can contribute and ease side-channel resistant implementations.

To deal with the noise in a typical side-channel attack scenario, one usually requires many measurements under the same key. On the one hand, in a typical block cipher based implementation, changing the key comes at the cost of re-running the internal key scheduler, so, block cipher based schemes usually avoid changing the key. On the other hand, changing the key in our hash function based setting is free. COFFE benefits from this by using the secret key $K$ only once for processing a message or a ciphertext. A “session key” $S$ is derived from $K$ and a nonce, and then, all message or ciphertext blocks are processed using this key. Thus, $K$ is used at most once in a single hash operation for each message. Therefore, in the nonce-respecting scenario, the number of measurements for $K$ and $S$ an adversary can make are limited.

Outline. In Section 2 we present a practical instance of COFFE using SHA-224 as the underlying primitive. Section 3 contains some technical preliminaries. The generic construction of our scheme is introduced in Section 4 and in Section 5 we analyze the security of our scheme. Section 6 concludes the paper.

2 Practical Instance

In this section we introduce a practical instance of COFFE using SHA-224 as the underlying hash function – called COFFE-SHA-224. The first part of this section contains the justification for the decision in favor of SHA-224 and in the second part we introduce the encryption and decryption process of COFFE-SHA-224.

In the next section we justify our usage of SHA-224 over SHA-256 as the underlying hash function. Note that, to optimize the performance of COFFE, we require to have exactly one compression function invocation per hash function call.

2.1 Choice of the Hash Function

Both SHA-224 and SHA-256 share the same compression function $f : \{0, 1\}^{256} \times \{0, 1\}^{512} \rightarrow \{0, 1\}^{256}$. It compresses a 256-bit chaining value and a 512-bit message block into a 256-bit output value. These two hash function standards differ in two properties 1) they are using different initial values, and 2) SHA-224 truncates the output of the final compression function invocation while SHA-256 does not. Following the Merkle-Damgård paradigm [5, 17], SHA-224 applies the secure 10*-padding followed by a 64-bit value encoding the message length. Thus, the maximum possible input size to fit our requirements is given by $512 - 64 - 1 = 447$ bit. Due to the sake of simplification, we consider only byte-aligned values and we assume all values to be encoded octet-strings. Thus, we can only process message blocks with a size up to 440 bit, i.e., 55 byte.

Since our construction (cf. Figure 2) is inspired by the CFB and the OFB modes of operation [6], it follows the idea of using the chaining value and the ciphertext as a feedback for the next iteration. As our hash function input is limited to 447 bit, we do have to find a trade-off between security, i.e., the size of the chaining value, and performance, i.e., the size of the ciphertext block. Using SHA-256 implies a 256-bit chaining value and thus, only 191 bits are left for the remaining input, including the domain separation byte and the previous ciphertext. Assuming only 32-bit word aligned plaintexts implies that we can process messages blocks up to 160 bits, since the generation of the authentication tag requires two additional input bytes – the length of the last
message block $b$ and the tag length $L_T$. The application of SHA-224 allows us to process message blocks up to a length of 192 bits, which leads to an estimated performance speedup of about 20% in comparison to SHA-256. Furthermore, the 224-bit session key used in SHA-224 is sufficient to make practical attacks infeasible. This makes SHA-224 a logical choice for us.

### 2.2 Encryption and Decryption Process

The encryption and decryption process consists of four steps that are reflected by the domain separation idea, which is also used in the SHA-3 finalist Skein [7]. From this idea, it follows that we can assume independence between the steps of session key generation, processing of the associated data (header), encryption, and the generation of the authentication tag. The header describes non-confidential data, which is usually public known and does not have to be encrypted but authenticated, e.g., the header of a network package. Moreover, we distinguish between headers with different lengths. Hence, we always get disjunct input spaces for each different domain, which is similar to the usage of independent PRF’s for each domain.

**Step 1: Session Key Generation.** The first domain (0) is used to generate the session key $S$ (short term key), which is derived from the secret key $K$ (long term key) and the nonce $V$. More precisely, $S$ is computed as follows:

$$S = \text{SHA-224}(K \ || \ V \ || \ 0^* \ || \ L_K \ || \ L_V \ || \ 0),$$

where “$||$” denotes the concatenation of strings of bytes. Note that $L_K$, $L_V$, and the domain description value are encoded as one-byte values and describe the three least significant bytes of the input. Thus, we have $440 - 24 = 416$ bits left for the sum of $L_K$ and $L_V$. Further, the values $L_K$ and $L_V$ represent the length of $K$ and $V$ in bits, respectively. For the choice of the default parameter, we recommend to use at least 224 bit for the secret key $K$ and 192 bit for the nonce $V$. This parameter choice implies, that we do not need any $0^*$-padding at all.

**Step 2: Processing of Associated Data.** In this step we describe the processing of the associated data $H$, which can be of arbitrary length. Note that the 8-bit domain $x$ and the 224-bit initial chaining value $T[0]$ used in the processing of the first message block depend on the size of the associated data. These two values are computed as follows:

$$(x, T[0]) \leftarrow G(H, \text{SHA-224}) := \begin{cases} 1, H \ || \ 10^* & \text{if } |H| < 224, \\ 2, H & \text{if } |H| = 224, \\ 3, \text{SHA-224}(H) & \text{else}, \end{cases}$$

where $|H|$ denotes the size of $H$ in bits.

**Remark.** We use this separation process for security and performance reasons. Domain ‘1’ is for the case that the authenticated data is less than one input block, (i.e., 224 bit for SHA-224) thus requiring $10^*$-padding to make a full block. Domain ‘2’ is for associated data actually fitting into exactly one block, where no padding is applied. When the data is larger than one block, we must apply the hash function. This case is represented by Domain ‘3’. Not calling the hash function when the associated data is small improves the performance. The domain separation ensures that an adversary cannot replace the header with the message without getting different results.
Step 3: Processing of the Message. We denote $M = M[1], M[2], \ldots, M[L]$ as the message, where $L = \lceil |M|/192 \rceil$ is the number of message blocks processed. Here, all blocks $M[i]$ with $1 \leq i \leq L - 1$ are of size 192 bits, and the last message block $M[L]$ consists of at most 192 bit. $M$ is encrypted as follows:

$$T[1] = \text{SHA-224}(S \oplus T[0], C[0], 0^*, x),$$
$$C[1] = M[1] \oplus_\alpha T[1],$$
For $i = 2, \ldots, L$:

$$T[i] = \text{SHA-224}(S \oplus T[i-1], C[i-1], 0^*, 4),$$
$$C[i] = M[i] \oplus_\alpha T[i],$$

where all values $T[i]$ are 224-bit values for $1 \leq i \leq L$, all values $M[i]$ and $C[i]$ are 192-bit values for $1 \leq i \leq L - 1$, and the length of $C[L]$ equals the length of $M[L]$, which is at most 192 bit. Furthermore, $T[0]$ is defined as in Step 2 and $C[0]$ denotes the first 48 post decimal positions of $\pi$, interpreted as a string of hex characters ($C[0] = 0x1415926$). The operation $\oplus_\alpha$ processes the XOR operation for the $\alpha$ least significant bits, where $\alpha$ always denotes the bit length of $M[i]$ for $1 \leq i \leq L$.

The decryption of a given ciphertext $C = C[1], C[2], \ldots, C[L]$ can be described similarly, where the decryption function is given by:

$$T[1] = \text{SHA-224}(S \oplus T[0], C[0], 0^*, x),$$
$$M[1] = C[1] \oplus T[1],$$
For $i = 2, \ldots, L$:

$$T[i] = \text{SHA-224}(S \oplus T[i-1], C[i-1], 0^*, 4),$$
$$M[i] = C[i] \oplus_\alpha T[i].$$

Thus, we only need a single one-way function for encryption and decryption, respectively.

Remark. The XOR operation $S \oplus T[i-1]$ is a less-than-ideal solution, since we have to consider input collisions for the building block over two distinct sessions keys. In the ideal world, each key determines a certain independent PRF. In our approach, a collision $S \oplus T[i-1] = S' \oplus T[j-1]$ for two distinct keys $S \neq S'$ violates the independency assumption. But, the limited input size of the SHA-224 compression function does not allow to separate the values $S$ and $T[i-1]$. Thus, one has to diminish at least one of these two values, which obviously leads to a negative security impact. Our security analysis in Section 5 shows that our approach still satisfies the birthday bound security. Additionally, our algorithm allows to process message and ciphertext blocks of reasonable size of 192 bit. Using a similar block size without the combination of the two values $S$ and $T[i-1]$, would imply that this scheme does not match any security claims of comparable “state of the art” modes like OCB[22], McOE[8], or GCM[16], since the remaining 232 bit do not provide enough security for handling $S$ and $T[i-1]$ separately.

Step 4: Generation of the Authentication Tag. In the final step we derive the authentication tag from the last chaining value $T[L]$ and the last ciphertext $C[L]$ as follows:

$$T = \text{SHA-224}(S \oplus T[L], C[L], C[L], 0^*, L_T, b + 5),$$

where $b$ is the bit length of the last message block $M[L]$ and $L_T$ denotes the length of the tag encoded as a byte. Note that the length of the tag is constrained by the output size of SHA-224,
EncryptAndAuthenticate(H, V, M)
1. S ← SHA-224(K || V || 0∗ || LK || LV || 0)
2. x, T[0] ← G(H, SHA-224)
3. T[1] ← SHA-224(S ⊕ T[0] || C[0] || 0∗ || x)
5. for i = 2, . . . , L − 1 loop
   I ← S ⊕ T[i − 1]
   T[i] ← SHA-224(I || C[i − 1] || 016 || 4)
   C[i] ← T[i] ⊕192 M[i]
6. b ← |M[L]|
7. I ← S ⊕ T[L − 1]
10. I ← S ⊕ T[L]
11. T ← SHA-224(I || C[L] || 0∗ || LT || b + 5)
12. return (C[1], . . . , C[L], T)

DecryptAndVerify(H, V, C, T)
1. S ← SHA-224(K || V || 0∗ || LK || LV || 0)
2. x, T[0] ← G(H, SHA-224)
3. T[1] ← SHA-224(S ⊕ T[0] || C[0] || 0∗ || x)
5. for i = 2, . . . , L − 1 loop
   I ← S ⊕ T[i − 1]
   T[i] ← SHA-224(I || C[i − 1] || 016 || 4)
   M[i] ← T[i] ⊕192 C[i]
6. b ← |C[L]|
7. I ← S ⊕ T[L − 1]
10. I ← S ⊕ T[L]
11. T∗ ← SHA-224(I || C[L] || 0∗ || LT || b + 5)
12. if T = T∗ then
    return (M[1], . . . , M[L])
else
    return ⊥

Fig. 1. The encryption and decryption (and verify) algorithm for COFFE-SHA-224. The value H denotes the associated data (header), V denotes the nonce, the value T[i] denotes the i-th chaining value, S the session key, and LT the length of the tag. We denote ⊕y as the XOR operation over the least significant y bit and ⊥ as a reject code.

e.g., at most 224 bits. When using SHA-224 as we do in our practical instance, the value b is restricted to the range [0, . . . , 192], since the size of a message/ciphertext block is given by at most 192 bit. The last domain allows a user to authenticate the header without any message to encrypt. Thus, the value b can become zero, but for SHA-224, b + 5 is always in the range [5, . . . , 197].

Wrap Up. Using a compression function where b + 5 exceeds one byte, the domain must be encoded as a two-byte value instead of a one-byte value. The encryption and decryption algorithm for COFFE-SHA-224 is given in Figure 1.

3 Technical Preliminaries

Notions. Let {0, 1}n∗ denote an arbitrary number of blocks of size n and let ⊥ denote a rejection of a message, i.e., that the verification of a message failed. Furthermore, we write 0y for a chain of y zero-bits, and 0∗ for a string of zero or more zero-bits to fill a given input for F up to exactly the required input size.

We define an adversary as a computationally unbounded but always-halting algorithm A. In this paper we assume vlog that the adversary A never asks a query which answer is already known. We denote AO for an adversary A with access to an oracle O.

Keyed Hash Function. A keyed hash function F : {0, 1}k × {0, 1}∗ → {0, 1}n is a family of functions that computes a fixed-size hash value Y ∈ {0, 1}n from a message X ∈ {0, 1}∗ of arbitrary length under a given key K ∈ {0, 1}k. We write Y = FK(X) for Y = F(K, X).

It is easy to build a keyed hash function from an un-keyed hash function by dividing the input space into two parts, the key space and a new message space. In our approach, the key is derived by the session key S and the previous chaining value T[i − 1], which can be seen as a tweak. We denote hash functions of this type by tweakable keyed hash functions.
Fig. 2. Illustration of the encryption and authentication process of COFFE. We denote $H$ as the associated data (header), $S$ as the session key, and $T[i]$ as the $i$-th chaining value. The value $b$ denotes the length in bits of the last message block $M[L]$, the value $L_T$ denotes the length in bits of the authentication tag $T$, and $X \oplus Y$ the XOR of the $g$ least significant bits of $X$ and $Y$. The variable $x \in \{1, 2, 3\}$ is a value depending on the length of the associated data.

Since we combine $S$ and $T[i-1]$ using an XOR operation, the underlying primitive of COFFE must be secure in the related-key model. Thus, we define the related-key PRF (PRF-RK) security of a keyed hash function $F$ by the success probability of an adversary trying to differentiate between the keyed hash function and a random function. In this scenario the adversary can 'partially control' some relations among the key using a function $\varphi$: \( \{0,1\}^k \times \{0,1\}^* \rightarrow \{0,1\}^k \). Note that this is a stronger assumption than the PRF security model as you can easily reduce PRF-RK security to PRF security by fixing the inputs to $\varphi$.

In the following, we define three security notions, which are necessary to proof the security of our scheme.

**Definition 1 (PRF).** Let $F: \{0,1\}^k \times \{0,1\}^* \rightarrow \{0,1\}^n$ be a keyed hash function with a secret key $K \xleftarrow{} K$, and $A$ a fixed adversary. Let $(\cdot,\cdot)$ denote a random bit oracle which returns always $n$-bit random values. The PRF advantage of $A$ in distinguishing $F$ from a random function is defined as

$$\text{Adv}_F^{\text{PRF}}(A) = \left| \Pr_K \left[ A^{F_K(\cdot)} \Rightarrow 1 \right] - \Pr \left[ A^{\mathbb{R}_n(\cdot)} \Rightarrow 1 \right] \right|.$$ 

The PRF advantage among all adversaries that run in time at most $t$ and make at most $q$ queries to the available oracle is given by

$$\text{Adv}_F^{\text{PRF}}(q,t) = \max_A \left\{ \text{Adv}_F^{\text{PRF}}(A) \right\}.$$

**Definition 2 (PRF-RK).** Let $F: \{0,1\}^k \times \{0,1\}^* \rightarrow \{0,1\}^n$ be a keyed hash function with a secret key $K \xleftarrow{} K$, and $A$ a fixed adversary. Let $(\cdot,\cdot)$ denote a random bit oracle which returns always $n$-bit random values, and let $\varphi(K,\cdot)$ be an injective key modifying function as defined above. The PRF-RK advantage of $A$ in distinguishing $F$ from a random function is defined as

$$\text{Adv}_F^{\text{PRF-RK}}(A) = \left| \Pr_K \left[ A^{F_{\varphi(K,\cdot)}(\cdot)} \Rightarrow 1 \right] - \Pr \left[ A^{\mathbb{R}_n(\cdot)} \Rightarrow 1 \right] \right|.$$
The PRF-RK advantage among all adversaries that run in time at most $t$ and make at most $q$ queries to the available oracle is given by

$$\text{Adv}_{F}^{\text{PRF-RK}}(q,t) = \max_{A} \{ \text{Adv}_{F}^{\text{PRF-RK}}(A) \}.$$  

**Definition 3 (Collision Resistance).** Let $F : \{0,1\}^k \times \{0,1\}^* \rightarrow \{0,1\}^n$ be a hash function with and $A$ a fixed adversary. The advantage of $A$ in finding a collision for $F$ is defined as

$$\text{Adv}_{F}^{\text{COLL}}(A) = \Pr[ X,X' \leftarrow A^F : X \neq X' \land F(X) = F(X')] .$$

The collision advantage among all adversaries that run in time at most $t$ and make at most $q$ queries to the available oracle is given by

$$\text{Adv}_{F}^{\text{COLL}}(q,t) = \max_{A} \{ \text{Adv}_{F}^{\text{COLL}}(A) \}.$$  

**Authenticated Encryption (With Associated Data).** An authenticated encryption scheme is a triple $\Pi = (K, E, D)$. It aims to provide both privacy and data integrity. The key generation function $K$ takes no input and returns a randomly chosen key $K$ from the key space $\{0,1\}^k$. The encryption algorithm $E$ and the decryption algorithm $D$ are deterministic algorithms that map values from $\{0,1\}^k \times \{0,1\}^n \times \{0,1\}^e \times \{0,1\}^n$ to a byte string or – if the input is invalid – the value $\perp$. For sake of convenience, we usually write $E_K(H,V,M)$ for $E(K,H,V,M)$ and $D_K(H,V,M)$ for $D(K,H,V,M)$, where the message $M$ and the associated data $H$ are chosen from the set $\{0,1\}^n$, a key $K$ from the key space $\{0,1\}^k$, and a nonce $V$ from the nonce space $\{0,1\}^e$. We require $D_K(H,V,E_K(H,V,M)) = M$ for any possible quadruple $(K,V,H,M)$.

**Definition 4 (CCA3).** Let $\Pi = (K, E, D)$ be an authenticated encryption scheme, $A$ a fixed adversary, and $K \leftarrow K$ be a randomly chosen key. The CCA3 advantage of $A$ in breaking $\Pi$ is defined as

$$\text{Adv}_{\Pi}^{\text{CCA3}}(A) = \left| \Pr_K \left[ A^{E_K(\cdot,\cdot),D_K(\cdot,\cdot)} \Rightarrow 1 \right] - \Pr_K \left[ A^{\parallel(\cdot,\cdot),\perp(\cdot,\cdot)} \Rightarrow 1 \right] \right| .$$

The adversary’s random-bit oracle $\parallel(\cdot,\cdot)$ returns on a plaintext query $(H,V,M) \in \{0,1\}^n \times \{0,1\}^e \times \{0,1\}^n$ a random string of length $|E_K(H,V,M)|$. The $\perp(\cdot,\cdot)$ oracle returns $\perp$ on every input. It is easy to see that we can rewrite the term given in Definition 4 as

$$\text{Adv}_{\Pi}^{\text{CCA3}}(A) = \left| \Pr_K \left[ A^{E_K(\cdot,\cdot),D_K(\cdot,\cdot)} \Rightarrow 1 \right] - \Pr_K \left[ A^{E_K(\cdot,\cdot),\perp(\cdot,\cdot)} \Rightarrow 1 \right] \right| .$$

One can interpret (1) as the advantage that an adversary has on the integrity of the ciphertext (INT-CTXT) as shown in Figure 3. Thus, the advantage of an adversary which runs in time at most $t$, asks a total maximum of $q$ queries to $E$ and $D$, and whose total query length is at most $\ell$ blocks is given by

$$\text{Adv}_{\Pi}^{\text{INT-CTXT}}(A) \leq \Pr[A^{\text{G-INT-CTXT}} \Rightarrow 1] .$$

Furthermore, one can interpret (2) as the advantage that a chosen plaintext adversary has on the privacy (CPA). We are using this decomposition to proof the CCA3-security of our construction.

For convenience, we introduce a notation for a restriction on a set. Let $Q = A \times B \times C$, then we denote $Q|_{B,C} = \{(B,C) \mid \exists A : (A,B,C) \in Q\}$ as the restriction of $Q$ to $B$ and $C$ with $A \in A, B \in B$, and $C \in C$. This generalizes in the obvious way.
Fig. 3. Game $G_{\text{INT-CTX}}$ is the INT-CTX$_H$ game where $H = \langle K,E,D \rangle$. $V$ denotes the initial value/nonce which is used to generate the session key.

4 Generic Construction

In this section we introduce the generic COFFE construction based on a generic PRF-RK secure hash function $F: \{0,1\}^* \rightarrow \{0,1\}^n$, which is used in a tweakable keyed hash mode. For the encryption and authentication domains, the maximum input size of $r$ bits must be chosen wisely due to performance issues as mentioned before.

Processing Associated Data. The initial chaining value $T[0]$ – for the encryption/decryption and authentication process of a message – is derived from the associated data $H$ given as the output of a function $G(H,F)$, which was already introduced in Section 2 in the context of the practical instantiation. From now on, the function $G$ is a generalization of the function $G$ presented in Section 2, where SHA-224 is replaced by a hash function $F$.

In the following, we present the generic definition of our COFFE scheme.

Definition 5 (COFFE). Let $F: \{0,1\}^* \rightarrow \{0,1\}^n$ be a hash function. Let denote $M$ the message with $M = M[1], \ldots, M[L]$, and $M[i] \in \{0,1\}^a$ for $i = 1, \ldots, L - 1$ and $M[L] \in \{0,1\}^b$ with $b \leq a$. Let $H$ denote the associated data. Then,

$$G(H,F) := \begin{cases} 
1, H \ || \ 10^* & \text{if } |H| < n, \\
2, H & \text{if } |H| = n, \\
3, F(H) & \text{else.} 
\end{cases}$$

Furthermore, let $V$ be a nonce and $K \overset{\$}{\leftarrow} K$ the long term key. The one-byte values $L_K$ and $L_V$ denote the length of the key and the nonce in bits, respectively, where

$$L_K + L_V + 16 < r \quad \text{and} \quad L_K \geq n. \quad (3)$$

Then, the COFFE encryption algorithm of the triple $(H,V,M)$ and the decryption algorithm of the quadruple $(H,V,C,T)$ are defined as shown in Figure 4.

5 Security

This section describes the security for our generic COFFE construction considered under the reasonable assumption that the size of the secret key $K$ can be larger or equal to the size of the session key $S$, i.e., $|K| \geq |S|$. The first step is to show the CPA security when considering a nonce-respecting adversary. For the INT-CTX proof we generalize the adversary by allowing it to reuse a nonce, i.e., transforming it to a nonce-ignoring adversary. Note that this assumption provides a better security for COFFE than for the majority of block cipher based authenticated encryption schemes [2, 11–13, 15, 16].
EncryptAndAuthenticate($H, V, M$)
1. $S \leftarrow F(K || V || 0^* || L_K || L_V || 0)$
2. $x, T[0] \leftarrow G(H, F)$
3. $T[1] \leftarrow F(S \oplus |S|, i.e., the domain of the session key generation is always 0 and the domain of the message processing is always > 0).
5. for $i = 2, \ldots, L - 1$ loop
   I. $I \leftarrow S \oplus T[i - 1]$
   II. $T[i] \leftarrow F(I || C[i - 1] || 0^* || 4)$
   III. $C[i] \leftarrow T[i] \oplus_a M[i]$
6. $b \leftarrow |M[L]|$
7. $I \leftarrow S \oplus T[L - 1]$
8. $T[L] \leftarrow F(I || C[L - 1] || 0^* || 4)$
9. $C[L] \leftarrow T[L] \oplus_a M[L]$
10. $I \leftarrow S \oplus T[L]$
11. $T \leftarrow F(I || C[L] || 0^* || L_T || b + 5)$
12. return $(C[1], \ldots, C[L], T)$

DecryptAndVerify($H, V, C, T$)
1. $S \leftarrow F(K || V || 0^* || L_K || L_V || 0)$
2. $x, T[0] \leftarrow G(H, F)$
3. $T[1] \leftarrow F(S \oplus T[0] || C[0] || 0^* || x)$
5. for $i = 2, \ldots, L - 1$ loop
   I. $I \leftarrow S \oplus T[i - 1]$
   II. $T[i] \leftarrow F(I || C[i - 1] || 0^* || 4)$
   III. $M[i] \leftarrow T[i] \oplus_a C[i]$
6. $b \leftarrow |C[L]|$
7. $I \leftarrow S \oplus T[L - 1]$
8. $T[L] \leftarrow F(I || C[L - 1] || 0^* || 4)$
9. $M[L] \leftarrow T[L] \oplus_a C[L]$
10. $I \leftarrow S \oplus T[L]$
11. $T' \leftarrow F(I || C[L] || 0^* || L_T || b + 5)$
12. if $T = T'$ then
   return $(M[1], \ldots, M[L])$
else
   return ⊥

Fig. 4. The encryption and decryption (and verify) algorithm for COFFE. The value $H$ denotes the associated data, $V$ denotes the nonce, the value $T[i]$ denotes the $i$-th chaining value, $S$ is the session key, and $L_T$ is the length of the tag. We denote $\oplus_a$ as the XOR operation over the least significant $y$ bit.

**Theorem 1.** Let $\Pi = (K, \mathcal{E}, D)$ be an COFFE scheme as in Definition 5, i.e., $K$ is the key derivation function, $\mathcal{E} = $ EncryptAndAuthenticate and $D = $ DecryptAndVerify. Then,

$$\text{Adv}^\text{CCA3}_H(q, \ell, t) \leq \text{Adv}^\text{CPA}_H(q, \ell, t) + \text{Adv}^\text{INT-CTXT}_H(q, \ell, t) \leq 4\ell^2 + 3q^2 \leq 3\ell^2 + 2q^2 + \frac{q}{2L_T} + 2\text{Adv}^\text{PRF-RK}_F(q + \ell, O(t))$$

$$\leq \frac{7\ell^2 + 5q^2}{2^n} + \frac{q}{2L_T} + 2\text{Adv}^\text{PRF-RK}_F(q + \ell, O(t)),$$

where $L_T$ denotes the length of the tag value in bits.

**Proof (Theorem 1).** The proof follows from Theorem 1 introduced in [8] together with Lemma 1 and Lemma 2. \hfill \Box

**Lemma 1.** Let $\Pi = (K, \mathcal{E}, D)$ be an COFFE scheme as in Definition 5. Let $q$ be the number of total queries an adversary $A$ is allowed to ask and $\ell$ be an integer representing the total length in blocks of the queries to $\mathcal{E}$. Then,

$$\text{Adv}^\text{CPA}_H(q, \ell, t) \leq \frac{4\ell^2 + 2q^2}{2^n} + 2\text{Adv}^\text{PRF}_F(q, \ell, t) + \text{Adv}^\text{COLL}_F(q + \ell, O(t)).$$

**Proof (Lemma 1).** This proof is using common game playing arguments. As stated above, the length of the secret key $K$ can differ from the length of the session key $S$. If this is the case, we can partition $F$ into two independent PRF’s: $F_1 : \{0, 1\}^{|K|} \times \{0, 1\}^* \rightarrow \{0, 1\}^n$ for the generation of the session key and $F_2 : \{0, 1\}^{|S|} \times \{0, 1\}^* \rightarrow \{0, 1\}^n$ for the processing of the message and the generation of the authentication tag. Due to the domain separation, the partitioning of $F$ is still valid if $|K| = |S|$, i.e., the domain of the session key generation is always 0 and the domain of the message processing is always $> 0$. 

Under this assumption, we can replace the functions $F_1$ by a PRF. This can be upper bounded by

$$\text{Adv}_{F_1}^{\text{PRF}}(q, O(t)).$$

Furthermore, we can replace the function $F_2$ by a PRF-RK, since the adversary has partial control over the key $S \oplus T[i]$. This can be upper bounded by

$$\text{Adv}_{F_2}^{\text{PRF-RK}}(q + \ell, O(t)).$$

Due to the sake of simplification, we define

$$\text{Adv}_{F_2}^{\text{PRF}}(q, \ell, t) = \max \{ \text{Adv}_{F_2}^{\text{PRF-RK}}(q + \ell, O(t)), \text{Adv}_{F_1}^{\text{PRF}}(q, O(t)) \}.$$ 

In the following analysis, we always consider the full output length $n$ of the tag generation step, i.e., even if $L_T$ is smaller than $n$, we skip the truncation step for the proof. This is valid, since showing CPA security for the tag generation step without truncation implies CPA security for the tag generation with truncation. Assume an adversary $B$ which is successful by attacking the truncated version, then we can build an adversary $B'$ using $B$ to be successful with the same probability attacking the untruncated version.

Next, we denote $Q$ as the query history of the adversary, where $Q_{i,T_i[\mu],T_i}$ contains the produced keystream $T_i[\mu]$ with $1 \leq \mu \leq \ell$ and the tags $T_i$ with $1 \leq i \leq q$. Note that all samples $T_i[\mu]$ and $T_i$ are output values of the hash function $F_2$. We can say that our COFFE scheme is CPA secure, if the produced keystream and the tag values within the query history are indistinguishable from a sequence $R$ of distinct random values of the same size, where the length of this sequence is limited to $\ell + q$. This event can be upper bounded by using similar arguments as stated in the PRP-PRF-Switching Lemma [1]:

$$\frac{(\ell + q)^2}{2^n}.$$ 

To complete our proof, we have to estimate the probability $\Pr[\text{Dist}]$ that all values within the list $Q_{i,T_i[\mu],T_i}$ are distinct. Therefore, we upper bound the probability $\Pr[\text{Coll}]$ for a collision of at least two of the values within this list, since

$$\Pr[\text{Dist}] = 1 - \Pr[\text{Coll}].$$

To upper bound $\Pr[\text{Coll}]$, we first consider the input parameter of $F_2$ represented by the quadruple $(S_i, T_i[\mu], C_i[\mu], d_i)$. Note that we ignore the 0*-padding, which leads to a higher success probability for an adversary. Let $Z_i = (S_i, T_i[\mu], C_i[\mu], d_i)$ and $Z_j = (S_j, T_j[\nu], C_j[\nu], d_j)$ with $1 \leq i, j \leq q$ and $1 \leq \mu, \nu \leq L^*$, where $L^*$ denotes the number of blocks of the longest message. A collision between two such tuples is given either when we have found a collision for $F$ or we have found an input collision for the values $S_i \oplus T_i[\mu] = S_j \oplus T_j[\nu]$. For our case analysis (cf. Table 5), we encode the difference between two such input tuples $Z_i$ and $Z_j$ using a five-bit value. For example, the value “10110” is defined as follows:

$$10110 := \begin{cases} 
    \text{i} \neq j \\
    T_i[\mu] = T_j[\nu] \\
    S_i \oplus T_i[\mu] \neq S_j \oplus T_j[\nu] \\
    C_i[\mu] \neq C_j[\nu] \\
    d_i = d_j,
\end{cases}$$

where $1 \leq i, j \leq q$ and $1 \leq \mu, \nu \leq L^*$.

**Note that Table 5 contains a complete case analysis, since all possible cases are covered.**

The cases which occur with a zero-probability are obviously seen as impossible and marked by
<table>
<thead>
<tr>
<th>Case</th>
<th>Event</th>
<th>Case</th>
<th>Event</th>
<th>Case</th>
<th>Event</th>
<th>Case</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>trivial</td>
<td>00100</td>
<td>1</td>
<td>11000</td>
<td>1</td>
<td>01000</td>
<td>2,4</td>
</tr>
<tr>
<td>00001</td>
<td>3</td>
<td>00101</td>
<td>1</td>
<td>11001</td>
<td>3</td>
<td>10001</td>
<td>1</td>
</tr>
<tr>
<td>00010</td>
<td>3</td>
<td>00101</td>
<td>1</td>
<td>11010</td>
<td>1</td>
<td>11010</td>
<td>3</td>
</tr>
<tr>
<td>00011</td>
<td>3</td>
<td>00111</td>
<td>1</td>
<td>11011</td>
<td>1</td>
<td>11011</td>
<td>3</td>
</tr>
<tr>
<td>00100</td>
<td>–</td>
<td>01100</td>
<td>3</td>
<td>10100</td>
<td>3</td>
<td>11100</td>
<td>3</td>
</tr>
<tr>
<td>00101</td>
<td>–</td>
<td>01101</td>
<td>3</td>
<td>10110</td>
<td>3</td>
<td>11101</td>
<td>3</td>
</tr>
<tr>
<td>00110</td>
<td>–</td>
<td>01110</td>
<td>3</td>
<td>10110</td>
<td>3</td>
<td>11110</td>
<td>3</td>
</tr>
<tr>
<td>00111</td>
<td>–</td>
<td>01111</td>
<td>3</td>
<td>10111</td>
<td>3</td>
<td>11111</td>
<td>3</td>
</tr>
</tbody>
</table>

Fig. 5. This table illustrates the case analysis for the proof of Lemma 1, where each case with a non-zero probability is covered by at least one event. The case “11000” is covered by two events depending on the considered domain (Event 2 covers the domain 1,2, and 3; Event 4 covers all other domains). The second special case “11100” is covered by Event 1 if $S_i = S_j$ and by Event 3 if $S_i \neq S_j$.

"–". The reason for the occurrence of these cases is a violation of the XOR relation between the values $S_i$ and $T_i[\mu]$ or $S_j$ and $T_j[\nu]$, respectively. For example, $S_i = S_j, T_i[\mu] = T_i[\nu]$, and $S_i \oplus T_i[\mu] \neq S_j \oplus T_j[\nu]$ is an impossible case. The Case “00000” implies that a collision must have happened before in the same query and is already covered by one of the other non-zero cases.

In the following, we analyze four events which cover all remaining cases with a non-zero probability given in Table 5.

After asking at most $q$ queries, we check the adversaries query history $Q$ – which contains all queries and their results – for the occurrence of bad events. We let the adversary win immediately if one of the bad events becomes true. Let denote $A_\alpha$ the $\alpha$-th event and $A_\beta$ the $\beta$-th event. The occurrence of an event $A_\alpha$ implies that no event $A_\beta$ with $\beta \in \{1, \ldots, \alpha - 1\}$ occurred before. Hence, the order of the events matters.

**Event 1: Session Key Collision.** The first case describes the scenario, where an adversary finds two values $S_i$ and $S_j$, generated using the function $F_1$, with $i \neq j$, $S_i = S_j$, and $1 \leq i, j \leq q$. This can be upper bounded by $q^2/2^n$.

Since $F_1$ and $F_2$ are independent, all values $S_i$ are independent from the values $T_i$ (tag values) and $T_i[\mu]$ (chaining values) with $1 \leq \mu \leq L^*$.

If an adversary is able to find two values $S_i$ and $S_j$ with $i \neq j$ and $S_i = S_j$, then it is able to distinguish COFFE from a PRF using the following attack with a complexity of $O(1)$. Consider two queries $(H, V, M)$ and $(H, V', M')$, where $M \neq M'$ are two single-block messages of the same length, i.e., $L_M = L_{M'}$ and $V \neq V'$ with $S = S'$. Then, it is obvious that $T[0] = T'[0]$ and $C[0] = C'[0]$. For the distinguishing attack we test whether $M[1] \oplus C[1] = M'[1] \oplus C'[1]$. Therefore, we let the adversary win if it finds two nonces $V \neq V'$ which lead to the same $S = S'$.

**Event 2: Input Collision – Associated Data.** In this case we consider an adversary which finds two pairs $(T_i[0] \oplus S_i, x_i)$ and $(T_j[0] \oplus S_j, x_j)$ with $T_i[0] \oplus S_i = T_j[0] \oplus S_j$, $x_i = x_j$, and $i \neq j$. This leads to two colliding inputs for $F$ in the first iteration. If no collision occurs, all $T_i[1]$ are independent random values. The success probability for this case can be upper bounded by $\text{Adv}_F^{\text{COLL}}(q + \ell, O(t))$. 
**Event 3: Output Collision.** For this case we consider an adversary which finds two values \( T_i[\mu] \) and \( T_j[\nu] \) with \( T_i[\mu] = T_j[\nu], 1 \leq \mu, \nu \leq L^*, 1 \leq i, j \leq q, \) and \((\mu, i) \neq (\nu, j)\). This can be upper bounded by

\[
\ell^2 / 2^n.
\]

If no collision is found, the values \( T_i[L] \) and \( T_j[L] \) with \( i \neq j \) must differ, where \( L \) denotes the index of the last chaining value. This implies that all authentication tags \( T_i \) can be seen as independent random values.

**Event 4: Input Collision – Message and Tag.** In this case we consider an adversary which finds two values \( T_i[\mu] \oplus S_i \) and \( T_j[\nu] \oplus S_j \) with \( T_i[\mu] \oplus S_i = T_j[\nu] \oplus S_j \) for \( 1 < \mu, \nu \leq L^* \) and \( 1 \leq i, j \leq q \). This leads to two colliding inputs for \( F_2 \). Note that we assume that the adversary did not find an output collision before. The probability for this event can be upper bounded by

\[
\ell^2 / 2^n.
\]

Our claim follows by adding up the individual bounds. \( \square \)

**Lemma 2.** Let \( \Pi = (K, \mathcal{E}, \mathcal{D}) \) be an \textsc{Coffee} scheme as in Definition 5. We assume the adversary to be nonce-ignoring, i.e., it is able to choose two nonces \( V_i = V_j \) with \( i \neq j \). Then,

\[
\text{Adv}_{\Pi}^{\text{INT-CTX}}(q, \ell, t) \leq \frac{3\ell^2 + 2q^2}{2^n} + \frac{q}{2^{L_T}} + \text{Adv}_{F}^{\text{PRF}}(q + \ell, O(t)),
\]

where \( L_T \) denotes the length of the tag value in bits.

The proof of Lemma 2 is given in Appendix A.

### 5.1 Authenticity under Weaker Assumptions (Sketch)

In the following, we analyze the case where the underlying primitive of \textsc{Coffee} is not \textsc{PRF}-\textsc{RK}-secure. For example, assume that \( c \geq 1 \) bits of the hash function output \( T[i] \) are constant. Then, privacy is gone for good, since the adversary can easily learn \( c \) bits of the message block \( M[i] \). But, as it turns out, the core structure of \textsc{Coffee} still preserves message integrity under a reasonable “unpredictability” assumption, since an \textsc{INT-CTX} adversary apparently must predict the output of the hash function, not just distinguish it from random. This observation holds even for nonce-ignoring adversaries. We make the following assumption.

If \( S \) is a random variable, or hard to distinguish from one, we could just assume the outputs \( F_S(\cdot, \cdot) \) to be hard to predict. But if we do not assume \( F \) to be a good \textsc{PRF}, we cannot assume \( S = F_K(\cdot, \cdot, \cdot) \) to be uniformly and randomly distributed. We solve this by defining a function

\[
F_K^2(X, Y, Z) := F(F(K \oplus X, Y) \oplus K, F(K \oplus X, Y) \oplus Z).
\]

It is straightforward to rewrite \textsc{Coffee} as the sequential application of \( F_K^2(\cdot, \cdot, \cdot) \), rather than as the sequential application of \( F_S(X, Y) := F(S \oplus X, Y) \) with \( S = F_K(V, L_K, L_V) \). Note that we do not pay attention to the 0\(^*\)-paddung and the domain. This leads to a higher success probability for the adversary.

**Unpredictability of \( F_K^2 \):** Fix the key length \( L_K \) and the length \( L_T \) of the authentication tag.

Choose a secret \( L_K \)-byte key \( K \). Allow the adversary to make queries to the oracle \( F_K^2 \).

Let denote \( Q \) the query history containing all queries \((X_i, Y_i, Z_i)\), with \( 1 \leq i \leq q \), made by the adversary. We assume that no efficient adversary can find any tuple \((X, Y, Z, T)\) with \((X, Y, Z) \notin Q\) and \( F_K^2(X, Y, Z) = T\).
Under this assumption, forgeries (INT-CTXT attacks) are infeasible. For COFFE, a forgery is a triple (nonce, ciphertext, tag) that is neither the output from the encryption oracle nor rejected by the decryption oracle – the only event, that would allow an INT-CTXT adversary to win its game. Wlog., we assume the (nonce, ciphertext) pair of a forgery has not been used as the output of an encryption query, before. Thus, there has either been a weak collision, i.e., two calls of $F^2$ under the secret key $K$ with different inputs share the same output, or, there has been no such collision, and the INT-CTXT adversary has correctly predicted the tag – directly violating our assumption. But even a weak collision is a violation of the unpredictability assumption, since an adversary finding a weak collision within $q$ queries can easily be turned into an adversary predicting $F^2_K(\cdot,\cdot,\cdot)$ with probability $> 2/q^2$. (For an analysis aiming at good concrete security, we might formally assume weak collision resistance for $F^2$.)

Cryptographic schemes could be used out of the specifications’ range or employing a primitive not quite as strong as demanded. Ideally, such schemes should provide a second line of defense. The brief analysis of COFFE under an unpredictability assumption, as well as our analysis regarding nonce-ignoring adversaries above, show that COFFE actually provides a second line of defense, maintaining authenticity in situations where privacy could not be defended any more.

6 Conclusion

In this paper we present COFFE, the first provably secure authenticated encryption scheme that has been natively designed for the usage of a hash function, rather than a block cipher as the underlying primitive. COFFE provides the security one would expect from a traditional scheme, plus three additional properties:

1. It provides reasonable resistance against side-channel attacks based on statistical information, since for each encryption process a new short term key is derived from a nonce and the long term key.
2. It provides ciphertext integrity in a nonce misuse scenario.
3. It provides ciphertext integrity even if the underlying primitive leaks information about its input. This can be formally proven, based on some unpredictability and strong key assumptions.

Considering these properties, COFFE is designed to be well-suited for critical security applications in resource-restricted, embedded devices, which require strong security requirements by providing only a small cryptographic suite.

References


A Proof of Lemma 2

A.1 Proof Preliminaries

Length of Longest Common Prefix (LLCP). The bit length of a string $x \in \{0,1\}^n$ is denoted by $|x| := n$. For integers $n, \ell, d \geq 1$, set $D_n^d = \{(0,1)^{n}\}^d$, $D_n^* := \bigcup_{d \geq 0} D_n^d$, and $D_{\ell,n} = \bigcup_{0 \leq d \leq \ell} D_n^d$. Note that $D_0^d$ only contains the empty string. For $M \in D_n^*$, we write $M = M_1, \ldots, M_d$ with $M_i \in D_n$ for $1 \leq i \leq d$. For $P, R \in D_n^*$, say, $P \in D_n^p$ and $R \in D_n^q$, we define the length of the longest common $n$-prefix of $P$ and $R$ as

$$\text{LLCP}_n(P,R) = \max_i \{P_i = R_{i}, \ldots, P_i = R_{i}\}.$$ 

For a non-empty set $Q$ of strings in $D_n^*$, we define $\text{LLCP}_n(Q,P)$ as $\max_{q \in Q} \{\text{LLCP}_n(q,P)\}$. For example, if $P \in Q$, then $\text{LLCP}_n(Q,P) = |P|/n$.

```python
1  LLCP'(R, (H, V, M))
2  p ← 0;
3  for each (H', V', M') ∈ R do
4      if (H' = H and V' = V) then
5          p = max(p, LLCP_n(M', M));
6  return p;
```

Fig. 6. $\text{LLCP}'$ computes the LLCP$_n$ of two inputs, a triple $(H, V, M)$ and a set of tuples $R$, where $a$ is the size of a message block.
A.2 Proof

1 **Initialize** ()
2 $K \leftarrow K()$;
3 $B_0, B_1, B_2, B_3, B_4, B_5 \leftarrow 0$;
4 win $\leftarrow$ false;
5 **Finalize** ()
6 return win;

100 **Encrypt**($H, V, M$) Game $G_1$
101 $L \leftarrow \|M|/a |$
102 $S \leftarrow F_1(K \parallel V \parallel 0^* \parallel L_K \parallel L_V \parallel 0)$;
103 $x, T[0] \leftarrow G(H, F)$;
104 $C[0] \leftarrow \kappa(x)$;
105 $I \leftarrow S \oplus T[0]$;
106 $T[1] \leftarrow F_2(I \parallel C[0] \parallel 0^* \parallel x)$;
108 for $i = 2, ..., L - 1$ do
109 $I \leftarrow S \oplus T[i - 1]$;
110 $T[i] \leftarrow F_2(I \parallel C[i - 1] \parallel 0^* \parallel |i|)$;
111 $C[i] \leftarrow T[i] \oplus_a M[i]$;
112 $b \leftarrow |M[L]|$
113 $I \leftarrow S \oplus T[L - 1]$;
114 $T[L] \leftarrow F_2(I \parallel C[L - 1] \parallel 0^* \parallel 4)$;
115 $C[L] \leftarrow T[L] \oplus_a M[L]$;
116 $I \leftarrow S \oplus T[L]$;
117 $T \leftarrow F_2(I \parallel C[L] \parallel 0^* \parallel L_T \parallel b + 5)$;
118 $\mathbb{Q} \leftarrow (H, V, C, T)$;
119 return $(C[1], ..., C[L], T)$;

119 **DecryptAndVerify**($H, V, C, T$) Game $G_1$
120 $L \leftarrow \|C|/a |$
121 $S \leftarrow F_1(K \parallel V \parallel 0^* \parallel L_K \parallel L_V \parallel 0)$;
122 $x, T[0] \leftarrow G(H, F)$;
123 $C[0] \leftarrow \kappa(x)$;
124 $I \leftarrow S \oplus T[0]$;
125 $T[1] \leftarrow F_2(I \parallel C[0] \parallel 0^* \parallel x)$;
127 for $i = 2, ..., L - 1$ do
128 $I \leftarrow S \oplus T[i - 1]$;
129 $T[i] \leftarrow F_2(I \parallel C[i - 1] \parallel 0^* \parallel |i|)$;
130 $M[i] \leftarrow T[i] \oplus_a C[i]$;
131 $b \leftarrow |C[L]|$
132 $I \leftarrow S \oplus T[L - 1]$;
133 $T[L] \leftarrow F_2(I \parallel C[L - 1] \parallel 0^* \parallel 4)$;
134 $M[L] \leftarrow T[L] \oplus_a C[L]$;
135 $I \leftarrow S \oplus T[L]$;
136 $T' \leftarrow F_2(I \parallel C[L] \parallel 0^* \parallel L_T \parallel b + 5)$;
137 if $(T = T')$ then
138 win $\leftarrow$ true;
139 return win;

Fig. 7. Games $G_1$ for the proof of Lemma 2. The variable $a$ denotes the block size of the message and ciphertext blocks with $a \leq n$, where $n$ is the output size of $F$, $F_1$, and $F_2$, each. The function $\kappa(x)$ returns the first $a/4$ post decimal positions of $x$ interpreted as a string of hex characters, and $L_T$ denotes the bit-length of the tag.

This proof borrows ideas from the INT-CTXT proof presented by Fleischmann et al. [8].

Our bound is derived by game playing arguments. Consider games $G_1$-$G_3$ of Figure 7 and Figure 8, and a fixed adversary $A$ asking at most $q$ queries with a total length of at most $l$ blocks. We assume that the adversary never asks for a query for which the answer is already known. The functions **Initialize** and **Finalize** are identical for all games in this proof.

Lets denote $G_0$ as the INT-CTXT game defined in Figure 3. Therefore, we have

\[
\text{Adv}_{\text{INT-CTXT}}(A) \leq \text{Pr}[A^{G_0} \Rightarrow 1].
\]

In $G_1$, the encryption and verify placeholders are replaced by their generic COFFE counterparts as of Definition 5 and, using similar arguments as in the proof for Lemma 1, we can partition $F$ into two independent PRF’s $F_1$ and $F_2$. Thus,

\[
\text{Pr}[A^{G_0} \Rightarrow 1] \leq \text{Pr}[A^{G_1} \Rightarrow 1] + \text{Adv}_F^{\text{PRF}}(q + l, O(t)).
\]

We now discuss the differences between $G_1$ and $G_2$. The sets $B_0, ..., B_5$ are initialized as empty sets (cf. line 3 of Figure 7) and collect fresh values as follows:

- $B_0$ collects all fresh values $T[0]$, where $|H| > n$ in lines 208 and 245.
- $B_1$ collects all fresh pairs $(T[0], S)$ in lines 213 and 250.
- $B_2$ collects all fresh values $I = T[0] \oplus S$ in lines 214 and 251.
- $B_3$ collects all fresh pairs $(I = T[0] \oplus S, C[I])$ with $1 \leq \mu \leq L - 1$, where $L$ is the message length in blocks. This is done in lines 222 and 259.
- $B_4$ collects all fresh values $T[I]$ with $1 \leq \mu \leq L$ in lines 225 and 263.
- $B_5$ collects all fresh pairs $(I = T[L] \oplus S, C[L])$. This is done in lines 233 and 270.
Games $G_2$ and $G_3$ for the proof of Lemma 2. Game $G_3$ contains the code in the box while $G_2$ does not. The variable $a$ denotes the block size of the message and ciphertext blocks with $a \leq n$, where $n$ is the output size of $F$, $F_1$, and $F_2$, each. The function $\kappa(\pi)$ returns the first $a/4$ post decimal positions of $\pi$ interpreted as a string of hex characters, and $L_T$ denotes the bit-length of the tag.
In lines 201 and 238, the LLCP’ oracle is inquired as defined in Figure 6. Finally, the variable \texttt{bad} is set to \texttt{true} if one of the if-conditions in lines 206, 211, 220, 224, 231, 243, 248, 257, 261, or 268 is \texttt{true}. None of these modifications affect the values returned to the adversary and therefore,

$$\Pr[A^{G_1} \Rightarrow 1] = \Pr[A^{G_2} \Rightarrow 1].$$

It follows that

$$\Pr[A^{G_2} \Rightarrow 1] = \Pr[A^{G_3} \Rightarrow 1] + |\Pr[A^{G_2} \Rightarrow 1] - \Pr[A^{G_3} \Rightarrow 1]|$$

$$\leq \Pr[A^{G_3} \Rightarrow 1] + \Pr[A^{G_3 \text{ sets bad}}].$$

(4)

We now proceed to upper bound the two terms contained in (4) – in right to left order.

The success probability of Game $G_3$ does not differ from the success probability of Game $G_2$ unless one of the following cases occur, where each case causes a bad event, i.e., the variable \texttt{bad} is set to \texttt{true}. In the following $i$ denotes the $i$-th query and $j$ denotes the $j$-th query with $1 \leq i, j \leq q$.

\textbf{Case 1 (Collision – Initial Chaining Value):} In lines 207 and 244 the initial chaining value $T[0]$ is set to a new random value if the function $G$ returns the same $T[0]$ twice for two distinct values $H_i \neq H_j$ with $i \neq j$ and $|H_i|, |H_j| > n$, i.e., in the case when $x = 3$. The probability for such a collision can be upper bounded by

$$q^2/2^n.$$

\textbf{Case 2 (Input Collision – Domain 1,\ldots,3):} In lines 212 and 249 the chaining value $I$ is set to a new random value if there is a non-trivial input collision between the two input values $I_i = S_i \oplus T_i[0]$ and $I_j = S_j \oplus T_j[0]$ with $x_i = x_j$, so that $I_i = I_j$ with $i \neq j$. We can upper bound the success probability for this case by

$$\ell^2/2^n.$$

\textbf{Case 3 (Input Collision – Domain 4):} In lines 220 and 257 we test for a non-trivial input collision for the pairs $\rho_i = (S_i \oplus T_i[\mu], C_i[\mu])$ and $\rho_j = (S_j \oplus T_j[\nu], C_j[\nu])$ with $i \neq j$, $\rho_i = \rho_j$, and $1 \leq \mu, \nu \leq L - 1$. The success probability for this case can be upper bounded by

$$\ell^2/2^n.$$

\textbf{Case 4 (Output Collision – Domain 4):} In lines 224 and 261 we test, if the adversary has found a non-trivial collision of the form $T_i[\mu] = T_j[\nu]$ with $2 \leq \mu, \nu \leq L - 1$ and $(i, \mu) \neq (j, \nu)$. The success probability is then given by

$$\ell^2/2^n.$$

\textbf{Case 5 (Input Collision – Domain 5):} In lines 231 and 268 we test for a non-trivial input collision for the pairs $\rho_i = (S_i \oplus T_i[L], C_i[L])$ and $\rho_j = (S_j \oplus T_j[L], C_j[L])$ with $i \neq j$ and $\rho_i = \rho_j$. We can upper bound the success probability for this case by

$$q^2/2^n.$$

By adding up the individual bounds, it follows that

$$\Pr[A^{G_3 \text{ sets bad}}] \leq \frac{3\ell^2 + 2q^2}{2^n}.$$
The adversary wins Game $G_3$ iff the variable $\text{win}$ is set to $\text{true}$, i.e., the if-condition in Line 272 holds. This implies that the adversary can only win with a fresh query to the $\text{DecryptAndVerify}$ oracle, which leads to $T =_{LT} T'$, where $T'$ is computed as shown in Line 271 and $=_{LT}$ denotes the comparison over the $LT$ least significant bits. Lines 268 and 269 ensure that the input for the function $F$ in Line 271 is always a fresh value, i.e., it was never asked before. Since $F$ is modelled as a PRF, the probability for $T =_{LT} T'$ can be upper bounded by

$$1/2^{LT}.$$

As we allow the adversary to ask at most $q$ queries, the success probability for Game $G_3$ can be upper bounded by

$$\Pr[A^{G_3} \Rightarrow 1] \leq q/2^{LT}.$$

Our claim follows by adding up the individual bounds. $\square$
Differential Attacks against ARX Designs

Gaëtan Leurent
Université Catholique de Louvain

Abstract. In this talk, we study differential attacks against ARX schemes. We build upon the generalized characteristics of de Cannière and Rechberger; we introduce new multi-bit constraints to describe differential characteristics in ARX designs more accurately, and quartet constraints to analyze boomerang attacks. We describe an efficient way to propagate multi-bit constraints, that allows us to use the complete set of $2^{32}$ 2.5-bit constraints.

We have developed a set of tools for this analysis of ARX primitives based on this set of constraints. We show that the new constraints are more precise than what was used in previous works, and can detect several cases of incompatibility. In particular, we show that several published attacks are in fact fact invalid because the differential characteristics cannot be satisfied. This highlights the importance of verifying differential attacks more thoroughly.

Moreover, we are able to build automatically complex non-linear differential characteristics for reduced versions of the hash function Skein. We describe several characteristics for use in various attack scenarios; this results in attacks with a relatively low complexity, in relatively strong settings. In particular, we show practical free-start and semi-free-start collision attacks for 20 rounds and 12 rounds of Skein-256, respectively. To the best of our knowledge, these are the first example of complex differential trails built for pure ARX designs.
Using Multiple Differentials... On the LLR and $\chi^2$ Statistical Tests

Céline Blondeau
Aalto University School of Science,
Department of Information and Computer Science
celine.blondeau@aalto.fi

Abstract. In parallel to similar extensions in linear cryptanalysis, a few papers about multiple differential cryptanalysis of block ciphers have been recently published. In the recent works published in SAC 2012 [1] and SCN 2012 [2] different and complementary approaches on how to use information from multiple differentials were presented. In [2], we explain how the LLR and the $\chi^2$ statistical tests can be used in the differential cryptanalysis context to handle information from a number of differentials or from truncated differentials. We present and compare different techniques, classify them, and discuss their relevance for cryptanalysis in light of available experimental results. This method have been applied on the PRESENT block cipher. In this recent work [3] using the link between differential and linear cryptanalysis provided by Chabaud and Vaudenay, we are able to compute good estimate of the differential probabilities from the correlation of the cipher. Using this estimates, we show that the LLR statistical test can be used in practice to perform a multiple differential cryptanalysis.

This is based on a joint work done with Benoît Gérard and Kaisa Nyberg and published in SCN 2012 [2] and EUROCRYPT 2013 [3].

References
A Framework for Automated Independent-Biclique Cryptanalysis

Farzaneh Abed, Christian Forler, Eik List, Stefan Lucks, and Jakob Wenzel

Bauhaus-Universität Weimar, Germany

Abstract. In this paper we introduce Janus, a software framework – written in Java – which is built to provide assistance in finding independent-biclique attacks for a user-chosen set of parameters, e.g., number of rounds and dimension of the biclique. Given a certain cipher, Janus does not only find an optimal bipartite graph (biclique), but also provides an all-round carefree package including an optimal matching-with-precomputation step, rendering of the found biclique, the matching steps and computation of the attack complexity.

We have used the Janus framework to verify existing results on the AES and ARIA. Additionally, by using this framework, we could find the first full-round biclique attacks on all versions of the AES-like cipher BKSQ.

Keywords: automated cryptanalysis, biclique, BKSQ

1 Introduction

Overview. Biclique cryptanalysis was first introduced by Khovratovich et al. in 2011 [14] and presented at the FSE 2012 [15]. The authors used the approach to find preimages for reduced-round versions of the block-cipher-based hash functions Skein [10] and SHA-2 [18]. Bicliques represent an improvement of the splice-and-cut approach [1,19,20] which is on the other hand based on common meet-in-the-middle techniques. More detailed, biclique cryptanalysis uses a complete bipartite graph (biclique) which can be constructed over a part of a primitive to extend an existing meet-in-the-middle or similar attack. From block-cipher-based compression functions, Bogdanov et al. adapted the approach in [2], and used bicliques for key-recovery attacks on block ciphers. Their work obtained a high level of attention, since they demonstrated the first attacks which could break all full versions of the AES in the single-key model. Since then, biclique attacks have become a well-known technique, and attacks on several further ciphers have been published in [4,5,11,12,17,22].

Finding good (independent) bicliques over a certain number of rounds for a given cipher is a time-consuming task, which requires in-depth knowledge of the investigated ciphers to find optimal differentials. Thus, it is adequate to think about using a computer to find such bicliques. Usually, implementations of common block cipher APIs are not designed to provide a sufficiently fine granularity, e.g., access to single steps and the basic operations of the cipher is not supported but required to find good bicliques.

Our Contribution. A unified API is needed to reduce the effort of modifying a block-cipher implementation for the biclique search. In addition, such an API would allow to apply one single biclique-searching framework that fits all. In this paper, we present such a framework, called Janus, which is free to use and open source1. The main feature of Janus is to find a complete and independent bipartite graph for a certain number or range of given rounds. In addition, it computes the corresponding step of matching with precomputations, and the overall complexity. Finally, it also supports a graphical illustration of the found biclique and the matching part.

1 https://github.com/eiklist/bicliquefinder
Janus provides a highly modular and flexible API, i.e., it allows the user to determine parameters like the used cryptographic primitive, the starting/ending round, the dimension of the biclique, the starting difference, and others.

First, we used our framework to verify and validate published attacks on variants of the AES and ARIA (see Section 4). Thereby, we detected a flaw in the complexity computation of the attack on the AES-192. So firstly, we were not able to verify the claim made for this attack. But secondly, further analysis revealed that the authors just forgot to include one round during the matching with precomputation phase. This example points out the importance of an automated framework to validate claims for existing attacks.

Additionally, we used Janus to find the first full-round attacks on variants of the AES-like cipher BKSQ [7]. Results of our work can be found in Section 4 in Table 1.

Related Work. There are several published tools and frameworks which support certain cryptanalytic techniques. Thus, these frameworks are limited to a mostly specific area of application. For example, the work of Daemen and Van Assche\(^2\) concentrates only on analyzing their SHA-3 winner Keccak. They provide, among other things, a computation of linear and differential trails. The results are published in [6]. Another framework was introduced by Leurant [16] to analyze ARX-based hash functions (like Skein or Blake) with the goal to assist in finding good differential trails. Further, Stankovski implemented an automated algebraic cryptanalysis framework [21]. His framework uses the Maximum-Degree-Monomial (MDM) test to launch algebraic attacks against stream and block ciphers. Currently, it supports more than 20 stream and block ciphers, and provides a possibility to produce tex-code for graphs.

Outline. In Section 2, we will provide a brief introduction of biclique cryptanalysis. In Section 3, we introduce Janus—containing the search for bicliques, the matching phase, and the rendering option. We used our framework for the verification of existing attacks on the ciphers AES, ARIA, and new attacks on different versions of BKSQ are shown in Section 4. Section 5 concludes the paper.

2 Independent-Biclique Cryptanalysis

In this section, we review the basics of independent-biclique cryptanalysis following the work of [14]. A biclique is a complete bipartite graph which covers some steps of a cipher. It connects every element in a set of starting states \(S\) with every element in a set of ending states \(C\). We enumerate the elements in \(C\) by \(C_i\) and the elements in \(S\) by \(S_j\), where a path from \(S_j\) to \(C_i\) represents the encryption under a key \(K[i, j]\). Figure 1 shows the schematic view of bicliques. More formally, the 3-tuple of sets \([\{S_j\}, \{C_i\}, \{K[i, j]\}]\) is called a \(d\)-dimensional biclique, if

\[
\forall i, j \in \{0, \ldots, 2^d - 1\} : \quad S_j \xrightarrow[K[i,j]]{K} C_i.
\]

The basic idea is to divide the key space into \(2^{k-2d}\) groups of \(2^{2d}\) keys, where \(k\) denotes the length of the secret key and \(d\) is the dimension of the biclique. A biclique can then be defined for one such group of keys \(K[i, j]\). The individual keys can be represented relative to a so-called base key of the group, \(K[0, 0]\), and two differences \(\Delta^K_i\) and \(\nabla^K_j\):

\[
K[i, j] = K[0, 0] \oplus \Delta_i \oplus \nabla_j.
\]

\(^2\)http://keccak.noekeon.org/KeccakTools-doc/ [October 2012]
An adversary can construct a biclique over one part of the cipher and apply a meet-in-the-middle attack over the remaining parts. Assume we are given some cipher $E$ which can be split into three parts $E = B \circ E_2 \circ E_1$, where $E_1$ is the subcipher that maps a plaintext $P$ to an internal state $V$, $E_2$ is the subcipher which maps $V$ to another internal state $S$ and $B$ is the subcipher which maps the state $S$ to the ciphertext $C$.

$$P \xrightarrow{E_1} V \xrightarrow{E_2} S \xrightarrow{B} C.$$  

A biclique can be constructed on an arbitrary portion of the cipher; in the following, we construct the biclique over the subcipher $B$.

### 2.1 Independent Bicliques

In [14, 13, 2], Khovratovich et al. proposed two different paradigms for biclique attacks: bicliques from independent differential trails (or independent bicliques) and bicliques from interleaving differential trails (or long bicliques). Independent bicliques allow the construction of bicliques over some subcipher $B$ from two sets of differentials. The steps are then as follows:

1. First step is base computation, i.e., a 3-tupel $\{S_0, C_0, K[0,0]\}$, where the key $K[0,0]$ maps the internal state $S_0$ to the ciphertext $C_0$ over $B$:

   $$S_0 \xrightarrow{K[0,0]} B C_0.$$  

2. Then, choosing $2^d$ keys $K[i,0] = K[0,0] \oplus \Delta^K_i$. Then, computing $\Delta_i$-trails from the state $S_0$ in forward direction for each key and obtains $2^d$ different ciphertexts $C_i$:

   $$S_0 \xrightarrow{K[0,0] \oplus \Delta^K_i} B C_0 \oplus \Delta_i = C_i \quad i \in \{0, \ldots, 2^d - 1\}.$$  

3. Similarly, choosing $2^d$ further keys $K[0,j] = K[0,0] \oplus \nabla^K_j$ but starting point is from the ciphertext $C_0$ and computation $2^d \nabla_j$-trails are in backward direction, each under a different key $K[0,0]$. Thus, one obtains $2^d$ different states $S_j$:

   $$S_j = S_0 \oplus \nabla_j \xleftarrow{K[0,0] \oplus \nabla^K_j} B C_0 \quad j \in \{0, \ldots, 2^d - 1\}.$$  

If the trails of the $\Delta_i$-differentials do not share any active non-linear operations with the $\nabla_j$-differentials, then every state $S_j$ can be connected with every state $C_i$ by encryption of $S_j$ under the key $K[i,j] = K[0,0] \oplus \Delta^K_i \oplus \nabla^K_j$. So, one obtains a set of $2^{2d}$ independent $(\Delta_i, \nabla_j)$-differential trails:

$$S_0 \oplus \nabla_j \xrightarrow{K[0,0] \oplus \Delta^K_i \oplus \nabla^K_j} B C_0 \oplus \Delta_i \quad \forall i, j \in \{0, \ldots, 2^d - 1\}.$$
The length of independent bicliques is limited by the diffusion properties of the cipher. An adversary can potentially create bicliques over more rounds by using the long-biclique approach. Though, the construction of long bicliques is a quite sophisticated and requires significantly more computational effort compared to the independent bicliques. The required independency of the differentials which is very important to us, is very clear and well-understood criteria and allows us to test it by using an automated approach. Therefore, we focus on the independent-biclique approach in this work.

2.2 Matching with Precomputations

If a constructed independent biclique is quite short and the matching part needs to cover too many rounds, then a meet-in-the-middle attack may not be applicable. In such cases, \cite{2} proposed to use a procedure called *matching with precomputations* as an alternative. For this approach, an adversary first chooses the internal state \( V \) between \( E_1 \) and \( E_2 \). Secondly, it computes and stores \( 2^d \) values \( \overrightarrow{V_{i,0}} \) in forward direction from the plaintexts to \( V \):

\[
P_i \xrightarrow{K[i,0]} E_1 \overrightarrow{V_{i,0}},
\]

and \( 2^d \) values \( \overleftarrow{V_{0,j}} \) in backward direction from each of the starting states \( S_j \):

\[
\overleftarrow{V_{0,j}} \xleftarrow{K[0,j]} E_2^{-1} S_j.
\]

Then, the adversary re-uses the stored values for the remaining \( 2^{2d} - 2^d \) computations

\[
P_i \xrightarrow{K[i,j]} E_1 \overrightarrow{V_{i,j}}, \quad \text{and} \quad \overleftarrow{V_{i,j}} \xleftarrow{K[i,j]} E_2^{-1} S_j,
\]

where it needs to recompute only those parts of the key schedule and the round transformation that differ from the stored values. By using this method, one can reduce the computational effort significantly even if no attacks are known which cover the remaining parts of the cipher. The costs for the recomputations can be further reduced by only computing and matching in a part of \( v \), which is called *partial matching*.

2.3 Complexity Calculation

For every biclique, the adversary tests \( 2^{2d} \) keys with \( 2 \cdot 2^d \) computations. Hence, the effort for one such set of keys is upper bounded by \( 2^d \) computations of \( E \). Therefore, the adversary needs to construct \( 2^{n-2d} \) bicliques to cover the full key space. For the time complexity, \cite{2} proposed the equation:

\[
C_{full} = 2^{n-2d} (C_{biclique} + C_{decrypt} + C_{precompute} + C_{recompute} + C_{falsepos}),
\]

where

- \( C_{biclique} \) denotes the costs for constructing a biclique.
- \( C_{decrypt} \) is the complexity of the oracle to decrypt \( 2^d \) ciphertexts.
- \( C_{precompute} \) denotes the costs for the computation of \( v \) for \( 2^d \) computations of \( E_2 \circ E_1 \).
- \( C_{recompute} \) is the complexity of recomputing \( 2^{2d} \) values \( v_{i,j} \) in both directions.
- \( C_{falsepos} \) is the complexity to eliminate false positives.

The complexity is dominated by the recomputations costs, \( C_{recompute} \). The memory requirements are upper bounded by storing \( 2^d \) values of the intermediate states \( v_{i,j} \).
3 Framework Design

Our initial motivation to automate the search for bicliques was first to automate testing differentials for independency but during our progress, we developed our work and created a framework which helps the cryptanalyst in related aspects of biclique attacks such as the matching part. The current implementation consists of four subsystems which are illustrated in Figure 2:

Fig. 2. Components of the framework.

1. The **biclique search** subsystem is responsible for searching for independent differential trails over some subcipher $B$ of a given primitive $E$.
2. Given a found biclique, the **matching** subsystem analyzes the remaining parts of the cipher to find a matching with minimum computational complexity.
3. The **rendering** subsystem can visualize bicliques as well as matching phase differentials.
4. In addition, the framework contains some common classes such as cipher implementations, helper and utility classes and differential building classes.

In the following, we discuss the first three components in detail since the common components contain only ciphers and helper classes to generate and compare differentials.

### 3.1 Biclique Search

The problem of finding independent bicliques can be transformed into the problem of finding pairs of independent differentials $(\Delta^a, \nabla^b)$, *i.e.*, differentials which share no active components in non-linear operations over some subcipher $B$. In advance, the user needs to specify

- a target cipher,
- the round range of the subcipher $B$,
– the dimension of bicliques $d$,
– a strategy to test the independency of differentials,
– and a strategy to generate key differences.

The latter two will be explained later in this Section. The general biclique search follows the steps from Section 2.1 and are listed in Algorithm 1.

\begin{algorithm}[h]
\begin{algorithmic}[1]
\State $K[0,0] \leftarrow 0^n$
\State $S_0 \leftarrow 0^n$
\For{$f = 1 \to N_d$}
\State $\Delta^f$-differential $\leftarrow \bigvee_{i=1}^{2^d-1} (S_0 \xrightarrow{K[0,0]} C_0) \oplus (S_0 \xrightarrow{K[0,0] \oplus \Delta^K f} C^f_i)$
\State Store $\Delta^f$-differential
\EndFor
\For{$b = 1 \to N_d$}
\State $\nabla^b$-differential $\leftarrow \bigvee_{i=1}^{2^d-1} (S_0 \xrightarrow{K[0,0]} C_0) \oplus (S_b \xrightarrow{K[0,0] \oplus \nabla^K b} C_0)$
\For{$f = 1 \to N_d$}
\If{shareNoActiveComponents($\Delta^f$-differential, $\nabla^b$-differential)}
\State biclique $\leftarrow (\Delta^f, \nabla^b)$
\If{stopAfterFoundFirstBiclique}
\State break;
\EndIf
\EndIf
\EndFor
\EndFor
\end{algorithmic}
\caption{Algorithm for Biclique Search.}
\end{algorithm}

First, a base computation is chosen by fixing $K[0,0]$ and $S_0$ to zeros and then deriving $C_0$. This trail $S_0 \rightarrow C_0$ computed only once. Then, we construct $N_d$ forward differentials $\Delta^f \forall f \in \{1, \ldots, N_d\}$ and for each forward differential $\Delta^f$ a unique part of $d$ bits is chosen in the key. We compute $2^{d-1}$ trails for all values of these $d$ bits
\[
S_0 \xrightarrow{K[0,0] \oplus \Delta^K f} C^f_i, \quad \forall i \in \{1, \ldots, 2^d - 1\}
\]
and derive the differential trails $\Delta^f_i = (S_0 \rightarrow C_0) \oplus (S_0 \rightarrow C^f_i)$, where $\oplus$ denotes the bit-wise XOR between the corresponding states after every round $i$ in $B$ and the XOR of all corresponding round keys in $B$. Bytes which are active in any of the $2^{d-1}$ differential trails $\Delta^f_i$ should keep active in the accumulated differential $\Delta^f$. So, the $\Delta^f$-trails are accumulated to $\Delta^f$ by applying the logical OR pair-wise to all corresponding states and round keys:
\[
\Delta^f \leftarrow \bigvee_{i=0}^{2^d-1} \Delta^f_i.
\]
The forward trails are stored in a list and the backward trails $\nabla^b$ are then created equivalently. We need to test independency of every pair of differentials $(\Delta^f, \nabla^b)$ in each round in $B$ by comparing the corresponding state differences before or after the non-linear operations. Any identified biclique can be used to mount a successful attack; therefore we provide an option for an early abort as soon as the first such pair has been recognised. In general, the time complexity of the biclique search is given by
\[
C_{time} = C_{forward} + C_{backward} + C_{testing},
\]
where
- \( C_{\text{forward}} \) is given by constructing \( 2^{N_d} \Delta f \)-differentials,
- \( C_{\text{backward}} \) is given by constructing \( 2^{N_d} \nabla b \)-differentials,
- \( C_{\text{testing}} \) is given by testing with \( 2^{2N_d} \) pairs of differentials \((\Delta f, \nabla b)\).

The complexity is dominated by the effort for testing \( 2^{2N_d} \) pairs of differentials. Considering the memory complexity, one needs to store all \( 2^{N_d} \Delta f \)-differentials. In general, every differential holds \( N_r + 1 \) state differences (where \( N_r \) denotes the number of rounds covered by \( \mathbb{B} \)) and a cipher-dependent amount of \( N_k \) round key differences. Thus, the memory complexity is given by

\[
C_{\text{memory}} = 2^{N_d} \cdot (N_r + 1) \cdot n + N_k \cdot k,
\]

where \( n \) and \( k \) denote state and key lengths in bits respectively. If less memory is available, the iteration biclique search can be performed where in each iteration, only a fraction of all forward differentials is tested.

Considering the implementation, the class BicliqueFinder is the central point of the biclique search. This class is responsible for accepting the required parameters, starting and coordinating the biclique search. The client can pass the listed parameters in the beginning of this section to the finder encapsulated into an instance of the class BicliqueFinderContext. To speed up the search, the BicliqueFinder then delegates the computation and testing of differentials to parallel-working instances of two internal classes DeltaThread and NablaThread.

Since the non-linear operations can be placed at different steps of the round function of ciphers, the finder needs a cipher-dependent strategy to test the independence of differential pairs. Therefore, we have created an interface DifferentialComparator and a package with helper classes (e.g., AESHelper or ARIAHelper) that implement this interface for every cipher. The correct class can be set as a parameter in the class BicliqueFinderContext by the client.

Ciphers. Throughout the framework, we employ a unified interface for cipher implementations. Standard cipher implementations usually allow the user to specify the plaintext for the state and the secret key. To obtain better bicliques, one should consider round keys with a low difference at the starting point of the differentials and invert the key schedule. Then, one can derive the secret keys from the chosen round keys. Thus, our implementations provide functions to invert the key schedule. Most ciphers we are interested in – AES-like ciphers and modern lightweight ciphers – use a key register with iterative update and the round key is extracted after every update. So, for such ciphers, the secret key can be reconstructed from a given state of the key-register. We have included cipher implementations in our framework which offers access to the round keys and the key registers to specify if their key schedule is invertible and the secret key can be constructed from register states.

Key Differences. Until here, it remains open to clarify the strategy to generate the key differences. The number of the differentials to test, \( N_d \), depends on the dimension of the biclique and the key length of the tested cipher. If one is given a cipher with a key length of \( k \) bits and a chosen dimension \( d \), then, one could potentially test \( N_d = \binom{k}{d} \) differentials. Though, this effort can be significantly reduced if one regards byte-wise- or nibble-wise-operating ciphers. Here, we consider three different strategies to generate key differences which are illustrated in Figure 3.

1. Firstly, one can activate only the minimum \( d \) active bits in the key difference and keep all other bits constant. Considering our example with byte-wise operating primitives, there are only \( N_b = k/8 \) bytes which can be active in the round key at the first or last round of the biclique if \( d \leq 8 \). As a consequence, for byte-wise and nibble-wise operating primitives,
Fig. 3. Approaches to iterate over key differences for byte-wise/nibble-wise operating ciphers: iterate over a minimum number of active bytes/nibbles (left), iterate over multiple bytes/nibbles with equal value (middle), iterate over bytes/nibbles in other parts of the key in the hope to cancel out results of the round transformation (right).

the effort reduces to test \( N_d = \binom{k/8}{d/8} \) and \( N_d = \binom{k/4}{d/4} \) differentials, respectively. For bit-wise operating primitives, one can limit the tests to a user-definable number of unique random start key differences.

2. As a second approach, one can use the same value in several active bytes. At the first sight, these will produce additional active bytes in the state after a key injection, making it harder for the differential to be independent from others. At second sight, they may cancel out byte differences in the key schedule and/or the round transformation of AES-like ciphers, as we can from the attack on SQUARE by Mala \([17]\). So, while testing all such differences increases the number of tested keys to \( N_d = 2^{N_b} \), one may obtain longer bicliques for SQUARE using this approach.

3. Alternatively, one can employ custom rules to generate round-key differences. In their attack on AES-192, Bogdanov \textit{et al.} employed the inverse result of a MixColumns operation as a part of the round key difference \([2]\). And in their attack on ARIA-256, \([5]\) uses a dedicated difference where the right part of the 256-bit key cancels the difference injected by the left part. One can learn from those examples that cipher-specific key differences can bring longer bicliques for AES-like ciphers. Since the testing of all custom differences in the key space is infeasible, the task of choosing “good” key differentials can be left as task to the user.

We provide an interface \texttt{DifferenceBuilder} for classes to generate the round-key differences for the biclique. Thus, the client can provide a custom implementation as a parameter in the \texttt{BicliqueFinderContext} according to one of the above strategies.

3.2 Matching

The matching phase is executed after a successful biclique search. The framework applies the matching-with-precomputations \([14]\) procedure to identify the number of recomputed parts. We denote the state before \( E_1 \) by \( P \) and the state after \( E_2 \) by \( S \). For every round \( i \) which can be used to split the remaining parts into \( E_2 \circ E_1 \), the matching part is performed in four steps:

1. First, one computes differentials from the start and the end of the matching part to the middle:

\[
P \xrightarrow{\begin{array}{c} K[0,0] \oplus \nabla K \\ E_1 \end{array}} v_i \oplus \nabla v_i, \quad \text{and} \quad v_i \oplus \Delta v_i \xleftarrow{\begin{array}{c} K[0,0] \oplus \Delta K \\ E_2^{-1} \end{array}} S.
\]

2. For partial matching, a difference \( \delta v \) is created in those bits that are used for matching and are set to ’1’, while all other bits in \( \delta v \) are set to zero. Then, we compute the differentials from \( v \) to the start and to the end:

\[
P \oplus \delta P \xrightarrow{\begin{array}{c} K[0,0] \\ E_1^{-1} \end{array}} v_i \oplus \delta v_i \quad \text{and} \quad v_i \oplus \delta v_i \xrightarrow{\begin{array}{c} K[0,0] \\ E_2 \end{array}} S \oplus \delta S,
\]
3. Only those parts of the states and round keys which are active in both differential trails, \( P \rightarrow \delta v \) and \( \delta P \leftarrow v \), need to be considered. Therefore, we use the logical AND to compute the differential
\[
\Delta_p^i = (P \rightarrow \delta v_i) \land (\delta P \leftarrow v_i),
\]
where \( \land \) means for all rounds \( i \) in \( E_2 \circ E_1 \), only those bytes which are active in the corresponding states of both differentials are set active in \( \Delta_p^i \) and only those bytes which are active in the corresponding round keys of both differentials are set active in \( \Delta_p^i \). Similarly, we compute
\[
\nabla_s^i = (\delta v_i \rightarrow S) \land (v_i \leftarrow \delta S).
\]

For nibble-wise operating ciphers, \( \land \) denotes the nibble-wise AND for bit-wise ciphers, the bit-wise AND.

4. At the end, the active bits/nibbles/bytes in non-linear operations are counted in both \( \Delta_p^i \) and \( \nabla_s^i \).

Steps (1) to (4) are repeated to find a matching with a number of active components in non-linear operations. The recomputation costs are the major contributor to the total attack complexity.

In our implementation, the class `MatchingDifferentialBuilder` is mainly responsible to perform the steps for the matching-with-precomputations part. Similar to the biclique search, required parameters are bundled in a context class `MatchingContext`. As a parameter, the context expects the found biclique, the cipher and again a cipher-specific helper class, which allows to determine the number of active bits, nibbles or bytes in non-linear operations. After the optimal matching has been found and returned by the `MatchingDifferentialBuilder`, the class `ComplexityCalculator` computes the total complexity of the attack according to the Equation (1) (cf. Section 2.3).

3.3 Rendering

Our framework allows to render identified bicliques and matching phases in PDF format. We have used this to render our attacks on BKSQ. For PDF creation, we employ the community version 5.3.0 of the open-source library iText [3], which is licensed under the AGPL.

In our framework, the classes `BicliqueRenderer` and `MatchingPhaseRenderer` are responsible to carry out the rendering process. In the following, we break the process down and provide the interfaces `DifferentialRenderer` and `StateRenderer` in order to allow extending rendering classes which can be simply exchanged by the client according to the type of differential (biclique, matching) and the type of the cipher (bit-wise, byte-wise, nibble-wise operating).

4 Applications

We have used our implementation to validate existing biclique attacks on the AES and ARIA from [2,5]. In this section, we first describe our studies of these ciphers, and point out our observations in the previous works. In the further parts of this section, we propose three new attacks on the different versions of the cipher BKSQ. All attacks we could construct with the help of our framework, also on the AES, are summarized in Table 1 and compared with the previous attacks.
### Table 1. Independent-biclique attacks constructed by automated search in comparison with previously published attacks. All attacks consider full-round attacks. In all cases, the dimension of constructed bicliques is eight.

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Biclique rounds</th>
<th>Comp. complexity</th>
<th>Data complexity</th>
<th>Memory complexity</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AES-128</td>
<td>3</td>
<td>$2^{126.18}$</td>
<td>$2^{88}$</td>
<td>$2^8$</td>
<td>[2]</td>
</tr>
<tr>
<td>AES-128</td>
<td>3</td>
<td>$2^{126.72}$</td>
<td>$2^{72}$</td>
<td>$2^8$</td>
<td>This work</td>
</tr>
<tr>
<td>AES-192</td>
<td>4</td>
<td>$2^{189.74}$</td>
<td>$2^{80}$</td>
<td>$2^8$</td>
<td>[2]</td>
</tr>
<tr>
<td>AES-192</td>
<td>4</td>
<td>$2^{190.28}$</td>
<td>$2^{48}$</td>
<td>$2^8$</td>
<td>This work</td>
</tr>
<tr>
<td>AES-256</td>
<td>4</td>
<td>$2^{254.42}$</td>
<td>$2^{40}$</td>
<td>$2^8$</td>
<td>[2]</td>
</tr>
<tr>
<td>AES-256</td>
<td>4</td>
<td>$2^{254.53}$</td>
<td>$2^{64}$</td>
<td>$2^8$</td>
<td>This work</td>
</tr>
<tr>
<td>BKSQ</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BKSQ-96</td>
<td>3</td>
<td>$2^{24.47}$</td>
<td>$2^{60}$</td>
<td>$2^8$</td>
<td>This work</td>
</tr>
<tr>
<td>BKSQ-144</td>
<td>4</td>
<td>$2^{142.63}$</td>
<td>$2^{96}$</td>
<td>$2^8$</td>
<td>This work</td>
</tr>
<tr>
<td>BKSQ-192</td>
<td>5</td>
<td>$2^{190.78}$</td>
<td>$2^{96}$</td>
<td>$2^8$</td>
<td>This work</td>
</tr>
<tr>
<td>ARIA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ARIA-256</td>
<td>2</td>
<td>$2^{255.20}$</td>
<td>$2^{80}$</td>
<td>$2^8$</td>
<td>[5]</td>
</tr>
</tbody>
</table>

#### 4.1 Verifications

**AES.** In our experiments on the AES, we could construct bicliques on up to three rounds for the 128-bit, and on up to four rounds for the 192-bit and 256-bit versions. So, our results confirm to the findings of Bogdanov et al., regarding the maximal biclique lengths. Further, we can confirm their statement that bicliques for the AES have to be placed ideally at the end of the cipher to avoid the initial key addition at the beginning and to profit from the fact that the last round omits the MixColumn operation (cf. [9]). In their independent-biclique attacks, Bogdanov et al. pointed out that the round key differences are a linear function of the indices $i$ and $j$. Thus, the authors could neglect the effort for recomputing the S-boxes in the key schedule. We did not employ this optimization, since we searched for a more general approach in our implementation. Further, we detected a minor flaw in the complexity calculation for the independent-biclique attack on the 192-bit version. There, the authors forgot to consider either the round 6 or 7 with 16 active S-boxes, which increases the number of SubByte operations from $2^{189.74}$ to $2^{190.16}$. Nevertheless, the described attacks are correct in the remaining parts.

**ARIA.** ARIA is a Korean variant of the AES. This cipher employs a high diffusion in its round transformation, where every input byte is involved in the computation of seven output bytes. In its key schedule, the round function is used four times to create four words $W_0, W_1, W_2, W_3$ in a Feistel structure. All round keys are then extracted from these words using rotations and XORs. Chen and Xu exploited a property of the key schedule: the leftmost 128 bits of the secret key is used as the first input to the Feistel network and the output is XORed with the rightmost 128 bits. The authors injected one-byte differences for the $\Delta_i$- and $\nabla_j$-differentials in the leftmost, and canceled the resulting seven-byte difference in the round-function output with the right half of the secret key. We have implemented and verified the attack on ARIA-256. Yet, the key schedule of ARIA refused more efficient attacks. So, we could construct bicliques of a single round only for ARIA-128 and ARIA-192 from our approach, which did not deliver attacks with significant advantage.
4.2 New Independent-Biclique Attack on the Full AES-128 and AES-192.

While the time complexities of the previous works on the AES are better than our results for it, we could decrease the data complexity for the 128-bit and 192-bit versions. The biclique and matching illustrations of our results are included in the Appendix. In the biclique for the 128-bit version, the ciphertexts $C_i$ differ in 11 out of 16 bytes only. The biclique is illustrated in Figure 14 (see Appendix D). Due to the key schedule of the AES, the bytes 0, 8, 12 (from left: the first, third and fourth byte in the uppermost row) in the round key of the final round always have an equal difference. As a consequence, the ciphertexts can only take on $(2^8)^9$ values and the data complexity is upper bounded by $2^{72}$.

Similarly, in the biclique for the 192-bit version, the ciphertexts $C_i$ differ in only five out of 16 bytes before the final key addition. It is depicted in Figure 16 (see Appendix E). Due to the key schedule, the bytes 1, 5, 9 (from left: the first, third and fourth byte in the second row) in the round key for the final round always have an equal difference. So, the ciphertexts for this biclique can take on only $(2^8)^6$ values and the data complexity of an attack using this biclique is upper bounded by $2^{48}$.

4.3 Specification of BKSQ

In the following, we give a brief introduction of the block cipher BKSQ and introduce attacks on the three different versions.

BKSQ is a block cipher, which was proposed by Joan Daemen and Vincent Rijmen in [7]. The cipher represents a generalization of SQUARE and Rijndael, since the state has now a rectangular $m \times n$-structure (cf. [8]). There are three different versions which all have a state size of 96 bits, but use individual key lengths of 96, 144, or 192 bits. The internal state is represented by a $3 \times 4$-byte matrix with three rows; the key is represented as a $3 \times 4$-, $6 \times 4$- or $9 \times 4$-byte matrix. the 96-bit version uses 10, the 144-bit version 14, and the 192-bit version 18 rounds. The round structure consists of four operations similar to those of the AES:

- **MixColumns/$\theta$:** The internal state is multiplied column-wise with a circulant MDS-matrix in the Galois-Field $GF(2^8)$.
- **SubBytes/$\gamma$:** Each byte in the internal state is replaced using an $8 \times 8$-bit S-box.
- **ShiftRows/$\pi$:** The $i$-th row of the internal state for $i \in \{0..2\}$ is rotated by $i$ bytes to the left.
- **AddRoundKey/$\sigma[k_i]$:** The internal state is XORed byte-wise with the subkey $k_i$ for round $i$.

Figure 4 visualizes the initial key whitening and the first round of the cipher. The round function $\rho$ and the full cipher $BKSQ$ are defined by:

$$\rho[k_i] = \sigma[k_i] \circ \pi \circ \gamma \circ \theta$$

$$BKSQ[k] = \rho[k_r] \circ \rho[k_{r-1}] \circ \ldots \circ \rho[k_1] \circ \sigma[k_0] \circ \theta^{-1}$$

Fig. 4. Round structure of the BKSQ, showing the initial whitening and the first round.
4.4 New Independent-Biclique Attack on Full BKSQ-96.

This subsection includes an explanation of the independent-biclique attack on a full-round version of BKSQ-96. The attack includes three steps: partitioning the key space, constructing a biclique, and matching over the remaining parts of the cipher. The complexity of the attack comes at the end.

**Key Space Partitioning.** We partition the key space with respect to the round key for Round 8, \( k_8 \). The base keys \( K[0, 0] \) are 2\(^{80} \) 12-byte values with two bytes fixed to 0 and where the ten other bytes take on all other possible values, which is visualized in Figure 5.

\[
K[0, 0] = \begin{bmatrix} 0 & 0 \end{bmatrix} \quad \Delta^K (k_8) = \begin{bmatrix} 1 \end{bmatrix} \quad \nabla^K (k_8) = \begin{bmatrix} 2 \end{bmatrix}
\]

*Fig. 5. Key differences used in the biclique for BKSQ-96.*

The key schedule of BKSQ is a bijective mapping where every value for the secret key is mapped uniquely to one value of every round key. Thus, when we have divided the key space of the round key for Round 8, we have also divided the secret key space. Here, we obtain an appropriate splitting of the key space into 2\(^{80} \) groups of 2\(^{16} \) keys each.

**3-Round Biclique of Dimension 8.** We construct a biclique of dimension eight over the rounds eight to ten. Figure 8 shows the base computation as well as the \( \Delta_i \) - and \( \nabla_j \) -differentials. Due to space limitations, the biclique and matching illustrations can be found in the Appendix A. It can be seen that all \( \Delta_i \) - and \( \nabla_j \) -differentials fulfill the independence criterion. The resulting ciphertexts \( C_i \) have to be decrypted by an oracle before the adversary can apply a matching on the rounds which are not covered by the biclique. At the end of the \( \Delta_i \) -differentials, the key differences have affected only 10 bytes in the ciphertexts \( C_i \). This means that by using the same value for \( S_0 \) in all 2\(^{80} \) bicliques, the resulting ciphertexts will only differ in 10 bytes, and the data complexity is upper bounded by 2\(^{80} \) ciphertexts.

**Matching over 7 Rounds.** The matching part covers the first seven rounds of the cipher, which are depicted in Figure 9 (see Appendix A). We choose the first byte of the state after Round 3 for the partial matching. In the matching part, one is interested how the recomputations of \( \overline{v}_{i,j} \) differ from the stored values \( \overline{v}_{i,0} \), due to the the differences. These bytes which have to be recomputed are darkened in 9.

Similar to the attacks on the AES in [2], one has to be accurate about the number of recomputed \( \theta \) -, \( \gamma \) - and \( \sigma \) -operations. In all our attacks on BKSQ, we follow the argumentation of [2] and use the number of S-boxes to recompute in both the round transformation and the key schedule to have a good approximation of the effort, since the number of S-boxes is the dominant summand compared to the number of \( \theta \) - and \( \sigma \) - operations.

As we can see from in Figure 9, we need to recompute 9 S-boxes in the first round, 3 S-boxes in the second and one S-box in Round 3, which sum up to 9 + 3 + 1 = 13 S-boxes in the round transformation of the forward part. In backward direction (covering rounds 4 to 7) we need to consider 3 + 9 + 7 + 3 = 22 S-boxes in the round transformation. Further, we include the effort to recompute the S-boxes in the key schedule. BKSQ uses the S-box for the rightmost column of its round keys. In the matching phase, there are in 3 + 3 + 3 + 3 + 1 + 1 + 0 + 1 = 15 active S-boxes in the last column of the round keys. In total, we have to recompute 13 + 22 + 15 = 50 S-boxes.
Complexity of the Attack. In BKSQ-96, there are $10 \cdot 12 = 120$ S-boxes in the $\gamma$-operations of the full cipher and 30 bytes in the key schedule. Thus, for $2^{16}$ keys in one key group, $C_{\text{recomp}}$ is equivalent to $2^{16} \cdot \frac{50}{195} = 2^{14.42}$ full encryptions. In all of our attacks on BKSQ, we use bicliques of dimension 8. Therefore, the decryption oracle needs $2^{8}$ full-cipher decryptions per biclique. Since we match in eight bits in the state $v$ in all of our attacks, we can expect to have $2^{8}$ false positive key candidates per biclique in average which need to be tested each with a few operations. So, we can approximate the effort for $C_{\text{falsepos}}$ by $2^{8}$.

For BKSQ-96, the biclique construction costs $C_{\text{biclique}}$ are given by computing $2 \cdot 2^{8} \times 4$ out of 14 rounds, which is equal to $2^{7.19}$ full encryptions. The precomputations costs $C_{\text{precomp}}$ are given by computing $2^{8} \times 3$ rounds in forward direction $P$ to $v$ and $2^{8} \times 4$ rounds in backward direction from $S$ to $v$. So, one has to consider $2^{8} \times 7$ out of 10 rounds, which is equal to $2^{7.49}$ encryptions. The full computational complexity is given by

$$2^{80} \cdot (2^{7.26} + 2^{7.49} + 2^{14.42} + 2^{8} + 2^{8}) = 2^{94.48}.$$ 

The data complexity is $2^{80}$, and we need to store $2^{8}$ ciphertexts per group.

4.5 New Independent-Biclique Attack on Full BKSQ-144.

Key Space Partitioning. We partition the key space with respect to the block $(k_{12}||k_{L13})$, which contains the full round key $k_{12}$ and the leftmost two columns of $k_{13}$. The base keys $K[0,0]$ are the $2^{144}$ 18-byte values with two bytes fixed to 0 and the remaining 16 bytes can take all other possible values. Figure 6 visualizes the key differences $\Delta^K_i, \nabla^K_j$ and the base keys.

![Fig. 6. Key differences used in the biclique for BKSQ-144.](image)

4-Round Biclique of Dimension 8. We construct a 4-round biclique which covers the rounds 11 to 14, as shown in Figure 10. Due to space limitations, the biclique and matching illustrations can be found in the Appendix B. This time, the ciphertexts $C_i$ are affected in all bytes. Thus, if we fix $C_0$ for all key groups, the data complexity may include all $2^{96}$ texts.

Matching over 9 Rounds. We match in the first byte of the state after Round 3. Figure 11 (see Appendix B) shows the active bytes in the matching phase. We consider those bytes which are active in the $\gamma$-operations in the middle of each round. So, we consider $9 + 3 + 1 = 13$ S-boxes in forward direction and $3 + 9 + 12 + 12 + 12 + 6 + 2 = 56$ active S-boxes in backward direction. In the key schedule, we have to recompute one active S-box in the round key $k_1$, one in $k_4$ and one in $k_7$ and one in $k_{10}$. So, there are in total $13 + 56 + 4 = 73$ active S-boxes in the matching phase.

Complexity of the Attack. In the full cipher, there are $14 \cdot 12 = 168$ S-boxes in the $\gamma$-operations and 27 S-boxes in the key schedule. Thus, for $2^{16}$ keys in one key group, $C_{\text{recomp}}$ is equivalent to $2^{16} \cdot \frac{73}{195} = 2^{14.58}$ full encryptions. The biclique construction costs $C_{\text{biclique}}$ are given by computing $2 \cdot 2^{8}$ times 4 out of 14 rounds, which is equivalent to $2^{7.49}$ full encryptions. The
precomputations costs $C_{\text{precomp}}$ are given by computing $2^8$ times 10 out of 14 rounds or $2^{7.51}$ encryptions. The full time complexity is given by

$$2^{128} \cdot (2^{7.19} + 2^{7.51} + 2^{14.58} + 2^8 + 2^8) = 2^{142.63}$$

full encryptions. The data complexity is $2^{96}$, and we need to store $2^8$ ciphertexts per group.

4.6 New Independent-Biclique Attack on Full BKSQ-192.

Key Space Partitioning. For the attack on the 192-bit version of BKSQ, we divide the key space with respect to the block $(k_{16}||k_{17})$, which contains the full round keys for Round 16 and 17, to build groups of $2^{176}$ base keys. The base keys $K[0,0]$ are the $2^{176}$ 24-byte values with two bytes fixed to 0 where all other bytes take all other possible values. Figure 7 visualizes the key differences.

\[
K[0,0] = \begin{bmatrix}
\ldots & 0 & 0 & \ldots \\
\ldots & 1 & 1 & \ldots \\
\end{bmatrix}
\quad \Delta K_{i}(k_{16}||k_{17}) = \begin{bmatrix}
\ldots & 1 & 1 & \ldots \\
\ldots & 0 & 0 & \ldots \\
\end{bmatrix}
\quad \nabla K_{j}(k_{16}||k_{17}) = \begin{bmatrix}
\ldots & 0 & 0 & \ldots \\
\ldots & 1 & 1 & \ldots \\
\end{bmatrix}
\]

Fig. 7. Key differences used in the biclique for BKSQ-192.

5-Round Biclique of Dimension 8. We construct a 5-round biclique, which covers the rounds 14 to 18. As we can see from Figure 12 (see Appendix C), all bytes of the ciphertexts $C_i$ are active.

Matching over 13 Rounds. We match in the first byte of the state after Round 5, as shown in Figure 13 (see Appendix C). There, an adversary should recompute $12 + 12 + 9 + 3 + 1 = 37$ S-boxes in the forward direction, $3 + 9 + 4 \cdot 12 + 6 + 2 = 68$ S-boxes in backward direction and 6 S-boxes in the key schedule. So, for this attack, $37 + 68 + 6 = 111$ S-boxes need to be recomputed in total.

Complexity of the Attack. In BKSQ-192, there are $18 \cdot 12 = 216$ S-boxes in the $\gamma$-operations of the full cipher and 51 bytes in the key schedule. Thus, for $2^{16}$ keys in one key group, $C_{\text{recomp}}$ results in $2^{16} \cdot \frac{111}{2^{51}} = 2^{14.73}$ full encryptions. $C_{\text{biclique}}$ are given by computing $2 \cdot 2^8$ times 5 out of 18 rounds, which is equivalent to $2^{7.15}$ full encryptions. $C_{\text{precomp}}$ results from computing $2^8$ times 13 out of 18 rounds or $2^{7.53}$ encryptions. The full computational complexity is given by

$$2^{176} \cdot (2^{7.15} + 2^{7.53} + 2^{14.73} + 2^8 + 2^8) = 2^{190.78}.$$ Again, the data complexity is $2^{96}$, and the memory complexity is $2^8$.

5 Conclusion and Outlook

With Janus, we have introduced user-friendly, highly flexible and expandable framework for cryptanalysts which supports automated biclique cryptanalysis of a user-specified cryptographic algorithm. With this framework, we found the first full-round attacks on BKSQ-96, BKSQ-144 and BKSQ-192. It is planned to increase the number of supported primitives, e.g., the AES and SHA-3 finalists to analyze the resistance against biclique attacks.
References

A Independent-Biclique Attack on Full BKSQ-96

Fig. 8. Biclique for BKSQ-96 in rounds 8 - 10 with $\Delta_i$- and $\nabla_j$-differentials.

Fig. 9. Recomputations for BKSQ-96 in forward and backward direction.
B Independent-Biclique Attack on Full BKSQ-144

Fig. 10. Biclique for BKSQ-144 in rounds 11 - 14 with $\Delta_i$ and $\nabla_j$-differentials.

Fig. 11. Recomputations for BKSQ-144 in forward and backward direction.
C Independent-Biclique Attack on Full BKSQ-192

Fig. 12. Biclique for BKSQ-192 in rounds 14 - 18 with $\Delta_i$- and $\nabla_j$-differentials.

Fig. 13. Recomputations for BKSQ-192 in forward and backward direction.
D Independent-Biclique Attack on Full AES-128

Fig. 14. Biclique for the AES-128 in rounds 8 - 10 with \( \Delta_i \) and \( \nabla_j \)-differentials.

Fig. 15. Recomputations for the AES-128 in forward and backward direction.
E Independent-Biclique Attack on Full AES-192

Fig. 16. Biclique for the AES-192 in rounds 9 - 12 with $\Delta_i$- and $\nabla_j$-differentials.

Fig. 17. Recomputations for the AES-192 in forward and backward direction.
Choosing new authentication and key generation algorithms for mobiles

Steve Babbage
Vodafone
steve.babbage@vodafone.com

Abstract. Steve chairs the group that specifies crypto algorithms for the GSM/GPRS/UMTS/LTE family of mobile phone standards. This is a talk about some practical considerations when choosing a new set of authentication and key generation algorithms for the mobile devices. This is work in progress: I will be asking for advice from the audience, and your feedback can help to influence what we choose.

The authentication and session key generation algorithms used in all these mobile standards exist in two places: the Authentication Centre of your home network operator, and the SIM / USIM that is provided to you by your home network operator. Operators can therefore choose their own algorithms for this purpose - there is no need for interoperability between different operators. Nevertheless, it is good to have a strong default choice of algorithm that any operator can use if it wants, rather than having each operator invent its own proprietary algorithm. For UMTS (3G) and LTE (4G), an algorithm called Milenage exists for this purpose, and lots of operators use it. Milenage is a construction based on AES.

We now want to create a second such algorithm. We want to base it on an existing, well trusted cryptographic component, not to design something from scratch. We want something fundamentally different from AES, so that any advance in cryptanalysis that threatens Milenage is unlikely also to threaten the new algorithm.

This talk was about the building blocks we are considering, and the design choices we are facing. It was designed to lead into an interactive discussion, encouraging audience members to express their opinions and influence the final design. Some very useful input was received, both in the Q&A session after the talk and in private discussions during the remaining days of the workshop. The new algorithm should be published later in 2013 - watch this space!
Abstract. We give an overview of recent work on RFID privacy models and RFID protocols originating from both research and industry. For several applications we derive appropriate security and privacy levels and we discuss why wide-forward-insider privacy is sufficient for all described applications. We present the first wide-forward-insider and wide-strong RFID identification protocols that are based on zero-knowledge. Until now this notion has only been achieved by schemes based on IND-CCA2 encryption. Rigorous proofs in the standard model are provided for the security and privacy properties of our protocols. Furthermore our protocols are the most efficient solution presented in the literature. Using only Elliptic Curve Cryptography (ECC), the required circuit area can be minimized such that our protocol even fits on small RFID tags. Concerning computation on the tag, we only require two scalar-EC point multiplications for the wide-forward-insider protocol.
Strong Privacy for RFID Systems from Plaintext-Aware Encryption

Serge Vaudenay

EPFL
CH-1015 Lausanne, Switzerland
http://lasec.epfl.ch

Abstract. Modeling privacy for RFID protocols went through different milestones. One pretty complete model was proposed by Vaudenay at ASIACRYPT 2007. It provides a hierarchy of privacy levels, depending on whether corruption is addressed by the protocol and on whether the return channel from the reader is available. The strongest notion of privacy was proven to be impossible to achieve, but the counterexample which was given was not convincing. Somehow, it showed that the requirements for strong privacy were unnecessarily too high. Several amendments were considered until a slight change in the definition which was proposed at CANS 2012. There, the simulator (blinder) was given access to the adversary’s random tape, making him able to read his mind. Thanks to plaintext-aware encryption, we can now prove that strong privacy is possible.

1 RFID Protocols and Privacy Issues

RFID protocols must face to contradictory requirements. On the one hand, RFID tags should be able to securely authenticate to a reader (or a network of readers). On the other hand, nobody else should be able to identify or trace tags as they move and interact with the environment. Since communication is wireless, modeling these requirements, then providing provably secure protocols, are challenging tasks for cryptographers.

We assume that tags are simple devices which can only respond to requests and do simple operations and data storage. They interact with readers. Readers are connected to a central server which maintains a database. Typically, we focus on the tag-reader protocol and assume that the reader network system is perfectly secure.

Ideally, an RFID protocol would be a two-path protocol in which the reader sends a challenge, then the tag answers. The tag may remain stateless. Otherwise, writing in memory would induce an overhead.

Clearly, if the response function is deterministic, one can easily trace a tag by replaying a challenge: the answer to the same challenge by the same tag will always be the same. So, the response function must be randomized.

Now, if the adversary can corrupt all tags and read their memory at the end of the attack, it may be the case that what leaks enables to identify which tag answered, when we use symmetric cryptography. To avoid that, [5] considered stateful protocols in which the key which was used to respond is updated in a one-way manner. This is the notion of forward privacy.

In [1], the authors consider that corruption may occur at earlier stages. I.e., not only at the end of the attack.

Finally, [3] notices that an adversary could try do desynchronize a stateful tag from its database so that it will no longer be identified. Then, if the adversary gets whether readers identify tags through a return channel (e.g., by looking at whether a door opens or not, if the reader is used for access control to buildings), then the protocol by [5] offers no longer privacy.

2 The 2007 Privacy Model(s)

We review here the formalism from [7].

Adversaries are assumed to have full control on the communication with tags and readers. They can activate the reader to start a protocol. They can draw anonymous tags with a chosen distribution. They
can communicate with drawn tags. They can initiate the creation of tags which belong to the system or not. They can free drawn tags so that they may be drawn again. They may corrupt drawn tags to retrieve their memory. They may read the return channel from the reader to see whether the protocol succeeded.

We distinguish $2 \times 3$ classes of adversaries. On one dimension, we distinguish whether they use the return channel or not. Adversaries not using it are called narrow adversaries. Others are wide adversaries. On the other dimension, we distinguish the type of corruption. Adversaries using no corruption are called weak. Adversaries making all corruptions at the end of their attack are called forward. Others are called strong.

The (strongest) security notion implies that for any wide-strong adversary, the probability that there is a reader protocol which succeeds to identify a tag but that tag has no matching conversation is negligible.

In the privacy game, we consider an adversary running the attack, then getting the table of ID’s of all drawn tags, then producing a bit. Privacy implies that the bit they produce would be the same if all communication and return channels were simulated. Concretely, privacy holds if for all adversaries, there exists a simulator (called a blinder) who sees the interaction between the adversary and the system but simulates all messages from tags, the reader, and the return channel; which blinder would be such that the blinded experiment produces indistinguishable outcomes form the un-blinded one.

We can achieve security and wide-weak privacy with a simple protocol based on a PRF. Protocols like [5] may achieve security and narrow-forward privacy in the random oracle model. Finally, we can achieve security, narrow-strong privacy, and wide-forward privacy at the same time using a public-key cryptosystem which is IND-CCA secure.

This protocol, called the PKC protocol, works as follows. A public/secret key pair is generated. The reader receives the secret key $K_S$ while the public one $K_P$ is stored in all tags. Each tag receives an ID and a secret $K$ which is also stored in the database. To authenticate, the tag receives a random challenge $a$ and encrypts, with $K_P$, the ID, $K$, and $a$ together. The ciphertext is the response. The reader can decrypt it with $K_S$, check that the challenge is correct, then check the entry in the database.

$$
\text{Tag} \quad \text{System}
$$

\begin{align*}
\text{state: } & K_P, \text{ID}, K \\
\text{secret key: } & K_S \\
\{ \ldots, (\text{ID}, K), \ldots \} \\
pick a & \\
c = \text{Enc}_{K_P}(\text{ID} \parallel K \parallel a) & \text{Dec}_{K_S}(c) = \text{ID} \parallel K \parallel a \\
\text{check } a, (\text{ID}, K) & \\
\text{output: } & \text{ID}
\end{align*}

It was shown that security and wide-strong privacy were impossible to achieve at the same time. To prove that, we first consider a wide-strong adversary who creates a legitimate tag, then corrupt it. Then, he simulates the creation of an illegitimate tag. He flips a coin and, based on the outcome, decides to simulate one tag or the other to the reader. Then, the result channel gives a bit and the adversary compares it with the coin flip to produce the result.

If the protocol was wide-strong private, there would be a blinder to simulate the reader and yield whether or not the simulated tag was the legitimate one or not. This blinder would work based on the state of the legitimate tag (obtained from corruption). Then, this blinder could be used by a new adversary.

The new adversary creates two legitimate tags, corrupt them both, then draw one or the other at random, and interact with the drawn tag. Simulating the previous blinder would make it possible to identify the tag. Then, the adversary would check from the table of ID’s if this was correct. Clearly, this new adversary cannot be blinded.

One crucial point in this proof is that the first adversary is querying the result channel for a bit that he already knows but that a blinder has troubles to simulate. This was observed by [4] who suggested that adversary should not ask questions to the environment for which they know the answer. This is the notion of wise adversary. However, it is pretty complicated to formalize it.
3 The 2012 Amendment

An alternate amendment was proposed in [6]. There, the definition of the blinder was updated so that the blinder would have access to the adversary’s random tape. Thus, the blinder could compute the same information that the adversary knows, and he would therefore be able to simulate the answers from the environment that the adversary knows. Somehow, the blinder would read the adversary’s mind.

With this new formalism, we can prove that the above PKC protocol provides wide-strong privacy (in the updated formalism), when the cryptosystem is IND-CPA secure and PA2-secure. PA2-security stands for plaintext awareness. The idea is that by reading the adversary’s random tape and the ciphertexts that they produce, a blinder could deduce which plaintext was encrypted by the adversary.

We could further show that some IND-CCA-secure cryptosystems which are not PA2-secure do not make the PKC protocol wide-strong private.

4 Conclusion

In [2], another privacy model was presented. This model is much simpler as it is not based on simulation. However, it was shown that IND-CCA-secure cryptosystems make the PKC protocol wide-strong private in this model. This suggests that there is a gap between the two models. So far, no separating protocol has shown to leak any private information in a real-life setting. Providing such a protocol is an open problem.

References

On the Need for Secure Distance-Bounding

Ioana Boureanu¹, Aikaterini Mitrokotsa², and Serge Vaudenay¹

² University of Applied Sciences of Western Switzerland (HES-SO)

Abstract. Distance-bounding is a practical solution to be used in security-sensitive contexts, mainly to prevent relay attacks. But subtle security shortcomings related to the PRF (pseudorandom function) assumption and ingenious attack techniques based on observing verifiers’ outputs have recently been put forward. In this extended abstract, we survey some of these security concerns and attempt to incorporate the lessons taught by these new developments in ideas of distance-bounding protocol design.

1 Introduction

In [4], Brands and Chaum introduced the notion of distance-bounding (DB) protocols. The aim is to have a prover demonstrate his proximity to a verifier, and authenticate himself to this verifier. The proof of proximity can be an efficient deterrent against relay attacks [9]. DB protocols [11,12,14,15] generally consist of an initialisation phase (where the parties establish some short-term secret) and a distance-bounding (DB) phase. This latter phase is time-critical. It imposes very fast computation, typically of less than a single clock cycle per round, and the verifier measures the time-of-flight of the messages exchanged. This is how the verifier ascertains a distance-bound between him and the prover.

Most distance-bounding protocols follow the following design pattern, where $f$ is a PRF and $F$ is a response-function for the DB phase.

![Fig. 1. The General Structure of Numerous DB Protocols (see [15,14])](image)

In the literature covering such protocols, the threat-model comprises three well-established types of attacks. The first is distance-fraud (DF), in which a prover tries to convince the verifier that he is closer than he really is. In the second type, mafia-fraud (MF), an adversary communicates with both a prover and a verifier which are far apart, and the adversary tries to convince the verifier that the prover would be close enough to be granted privileges. Finally, in a terrorist-fraud (TF), an adversary is getting the necessary and sufficient help from a coerced, far-away prover in order to pass the protocol only during
In [8], Cremers et al. describe distance-hijacking as a mixture between distance-fraud and terrorist-fraud: one dishonest, far-away prover exploits several honest provers to gain privileges. Impersonation-fraud is presented in [10]; as its name suggests, one dishonest prover tries to impersonate an honest one.

In this extended abstract, we first survey the most recent such attacks that have been proposed. Then, we attempt to incorporate the lessons taught by these new developments in ideas of secure distance-bounding protocol design.

2 DB: Instability of Security Results

2.1 PRF-based Unfortunate Arguments

Some security models [10] contain incorrect proofs/arguments: i.e., they replace a PRF by a random function at a place where the adversary has access to the PRF key or at a place where the PRF key is simultaneously used at other places in the protocol. If we observe Fig. 1, we can see that a dishonest prover holds the key $x$ of the PRF instance, which is also used inside the call to the response-function $F$. So, in a distance-fraud resistance proof, we cannot simply call the PRF assumption when speaking about the “leakage” produced via using $f$.

\begin{figure}
\centering
\begin{tikzpicture}
\tikzstyle{place} = [draw, fill=white, rectangle, minimum height=2.5em, minimum width=5em]
\tikzstyle{edge} = [draw, -latex, line width=1.5pt]

\node (P) [place] at (0,0) {Verifier $V$} ;
\node (Q) [place] at (3,0) {Prover $P$} ;
\node (R) [place] at (6,0) {Verifier $V$} ;
\node (S) [place] at (9,0) {Prover $P$} ;

\draw [edge] (P) -- node [above] {$\text{shared key } s \in G^m$} (Q) ;
\draw [edge] (Q) -- node [above] {$\text{shared key } s \in G^m$} (S) ;
\draw [edge] (P) -- node [above] {$N_V \leftarrow \{0,1\}^m$} (R) ;
\draw [edge] (R) -- node [above] {$N_V \leftarrow \{0,1\}^m$} (S) ;
\draw [edge] (P) -- node [above] {$N_P \leftarrow \{0,1\}^m$} (Q) ;
\draw [edge] (Q) -- node [above] {$N_P \leftarrow \{0,1\}^m$} (S) ;
\draw [edge] (P) -- node [above] {$\text{For } j = 1, \ldots, m$} (Q) ;
\draw [edge] (Q) -- node [above] {$\text{For } j = 1, \ldots, m$} (S) ;
\draw [edge] (P) -- node [above] {$\text{For } i = 1, \ldots, n$} (Q) ;
\draw [edge] (Q) -- node [above] {$\text{For } j = 1, \ldots, m$} (S) ;
\draw [edge] (P) -- node [above] {$\text{compute } r_{i,j} \text{ based on } f_s(N_P, N_V)$} (Q) ;
\draw [edge] (Q) -- node [above] {$\text{compute } r_{i,j} \text{ based on } f_s(N_P, N_V)$} (S) ;
\draw [edge] (P) -- node [above] {$\text{Distance-bounding phase}$} (Q) ;
\draw [edge] (Q) -- node [above] {$\text{Distance-bounding phase}$} (S) ;
\draw [edge] (P) -- node [above] {$\text{for } i = 1 \text{ to } m$} (Q) ;
\draw [edge] (Q) -- node [above] {$\text{for } i = 1 \text{ to } m$} (S) ;
\draw [edge] (P) -- node [above] {$\text{Pick } c_i \in [1, n]$} (Q) ;
\draw [edge] (Q) -- node [above] {$\text{for } i = 1 \text{ to } m$} (S) ;
\draw [edge] (P) -- node [above] {$\text{Start Clock}$} (Q) ;
\draw [edge] (Q) -- node [above] {$\text{Start Clock}$} (S) ;
\draw [edge] (P) -- node [above] {$c_i$} (Q) ;
\draw [edge] (Q) -- node [above] {$c_i$} (S) ;
\draw [edge] (P) -- node [above] {$\text{Stop Clock}$} (Q) ;
\draw [edge] (Q) -- node [above] {$\text{Stop Clock}$} (S) ;
\draw [edge] (P) -- node [above] {$r_{c_i}$} (Q) ;
\draw [edge] (Q) -- node [above] {$r_{c_i}$} (S) ;
\draw [edge] (P) -- node [above] {$\text{verify the responses and}$} (Q) ;
\draw [edge] (Q) -- node [above] {$\text{verify the responses and}$} (S) ;
\draw [edge] (P) -- node [above] {$\text{that } \Delta t_i \leq 2\Delta t_{\text{max}}$} (Q) ;
\draw [edge] (Q) -- node [above] {$\text{that } \Delta t_i \leq 2\Delta t_{\text{max}}$} (S) ;
\draw [edge] (P) -- node [above] {$\text{Out}_V(...)$} (Q) ;
\draw [edge] (Q) -- node [above] {$\text{Out}_V(...)$} (S) ;
\end{tikzpicture}
\caption{The TDB [1] Protocol}
\end{figure}

Indeed, in [3], the authors have recently shown that if pseudorandom functions existed, then trapdoor PRFs can be constructed such that if used in instances of numerous DB protocols, they would lead to DF attacks and generalised MF attacks. We leave the reader to consult [3] for the technicalities of the general construction of such programmed PRFs. In here, we only recall how one such PRF can bring DF-insecurity if used inside an instance of the TDB [1] protocol.

In Figure 2, we first recall the TDB protocol:
Now, consider one suggested instance of the TDB protocol, i.e., one where \( n = k = 3 \), \( G = F_2 \) and the \( 3 \times m \) response-matrix is of the form: \[
\mathcal{R}_i = \begin{pmatrix}
    r_{1,1} & \cdots & r_{1,m} \\
    r_{2,1} & \cdots & r_{2,m} \\
    s_1 \oplus r_{1,1} \oplus r_{2,1} \cdots s_m \oplus r_{1,m} \oplus r_{2,m}
\end{pmatrix}
\]

Then, assuming that PRFs exist, [3] shows that the PRF assumption on \( f \) is not enough to protect against DF in the above TDB instance. Namely, if PRFs exist, then let \( g \) be a PRF from \( \{0,1\}^{2m} \) to itself. Take the PRF \( f \) as follows:

\[
f_s(N_P, N_V) = \begin{cases} 
    s || s, & \text{if } N_P = s \\
    g_s(N_P, N_V), & \text{otherwise.}
\end{cases}
\]

According to the general construction in [3], if \( g \) is a PRF, then this programmed \( f \) is a PRF as well. Then, let \( P^* \) be a dishonest prover who chooses \( N_P = s \). In this case, the TDB instance above implies that the \( \mathcal{R}_i \) response-matrix has all its rows equal to \( s \). Then, for all challenges \( c_i \), the response will be the \( i \)-th bit of the secret key \( s \), hence \( P^* \) can win in a DF by sending the responses before the challenges even arrive at him.

With similar constructions (all based on their general design of trapdoor PRFs), [3] shows a series of attacks on numerous DB protocols and implies that possibly other similar DB protocols could be defeated similarly. A summary of the results in [3] is stated in Table 1.

<table>
<thead>
<tr>
<th>protocol</th>
<th>distance fraud</th>
<th>man-in-the-middle attack</th>
</tr>
</thead>
<tbody>
<tr>
<td>TDB</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>Dürholz-Fischlin-Kasper-Onete [ISC 2011]</td>
<td>√</td>
<td>–</td>
</tr>
<tr>
<td>Hancke-Kuhn [Securecomm 2005]</td>
<td>√</td>
<td>–</td>
</tr>
<tr>
<td>Avoine-Tchamkerten [ISC 2009]</td>
<td>√</td>
<td>–</td>
</tr>
<tr>
<td>Reid-Nieto-Tang-Senadji [ASIACCS 2007]</td>
<td>√</td>
<td>√</td>
</tr>
</tbody>
</table>

Table 1. A Summary of the PRF-based DB Attacks in [3]

2.2 PKC is not the solution for DB!

In some sense, in [2], the authors have recently shown that techniques from public key cryptography (PKC), i.e., commitments, proofs of knowledge, etc, are not the key to strengthen DB. More concretely, the authors have shown that the Bussard-Bagga protocols [6,7] suffer from TF attacks, even if their PKC techniques were supposed to prevent that.

In Figure 3, we first recall the generic Bussard-Bagga protocol, denoted DBPK. In their protocol, the prover \( P \) has a secret key \( x \) and a published certificate on its public key \( y = \Gamma(x) \). In the initialization phase, the prover generates a random secret session key \( k \in_R \{0,1\}^m \) and uses this session key in order to encrypt his private key \( x \). The encryption of \( x \) is done under a symmetric key encryption scheme \( E : \{0,1\}^m \times \{0,1\}^m \rightarrow \{0,1\}^m \). After encrypting \( x \) and computing \( e \), the prover \( P \) uses a bit commitment scheme to commit to each bit of \( k \) and \( e \) using randommesses \( v \) and \( v' \) respectively.

In each distance-bounding round, the verifier \( V \) selects a random bit as the challenge \( c_i \) and the prover responds with a response \( r_i \) such that

\[
r_i := \begin{cases} 
    k_i, & \text{if } c_i = 0, \\
    e_i, & \text{if } c_i = 1.
\end{cases}
\]
### Verifier $V$

- public key $y = \Gamma(x)$

### Prover $P$

- private key $x$

#### Initialization phase

session key $k \leftarrow \{0, 1\}^m$

$e = \mathcal{E}_k(x) \in \{0, 1\}^*$

$v, v' \leftarrow (\{0, 1\}^*)^m$

for $i = 1$ to $m$

$z_{k,i} = \text{commit}(k_i, v_i)$

$z_{e,i} = \text{commit}(e_i, v'_i)$

endfor

$z_k := (z_{k,i})_{i \in \{1, \ldots, m\}}$

$z_e := (z_{e,i})_{i \in \{1, \ldots, m\}}$

#### Distance-bounding phase

for $i = 1$ to $m$

Pick $c_i \in U \{0, 1\}$

**Start Clock**

$c_i \rightarrow r_i := \begin{cases} k_i, & \text{if } c_i = 0 \\ e_i, & \text{if } c_i = 1 \end{cases}$

**Stop Clock**

Commitment opening phase

for $i = 1$ to $m$

$y_i := \begin{cases} v_i, & \text{if } c_i = 0 \\ v'_i, & \text{if } c_i = 1 \end{cases}$

verify the responses

$z_{k,i} \overset{?}{=} \text{commit}(k_i, y_i)$, if $c_i = 0$

$z_{e,i} \overset{?}{=} \text{commit}(e_i, y_i)$, if $c_i = 1$

#### Proof of knowledge phase

$\mathcal{PK}[(\alpha, \beta) : z = \Omega(\alpha, \beta) \land y = \Gamma(\alpha)]$

---

**Fig. 3.** The DBPK Protocol proposed by Bussard and Bagga [6,7,5]
In the commitments’ opening phase, the prover $P$ opens some commitments on the bits of $k$ and $e$ corresponding to his answers in the distance-bounding phase. This is denoted in Figure 3 through sending the value $\gamma_i$ which stands for the respective randomness used at the commitment phase. In case that the openings of $z_{k,i}$ and $z_{e,i}$ do not pass, the verifier $V$ sends an error notification message to the prover $P$.

In the Proof of knowledge phase, the prover $P$ convinces the verifier $V$ with a zero-knowledge interaction that he has generated the commitments which correspond to a unique private key $x$ and this private key corresponds to the public key $y$ that is used by the verifier to authenticate the prover. The proof of knowledge is denoted as

$$PK[(\alpha, \beta) : z = \Omega(\alpha, \beta) \land y = \Gamma(\alpha)],$$

where the knowledge of $\alpha, \beta$ is being proven, while $z, y$ are as per the protocol, known to the verifier. In the protocol, we have $y = \Gamma(x)$ and $z = \Omega(x, (v, v'))$. The value of $z$ can be computed from the $z_{k,i}$ and $z_{e,i}$.

The number $m$ of DB rounds and the size $m$ of the key is dictated by a security parameter. Typically, $m$ varies between 128 and 1024.

Commitments and the Proof of Knowledge in DBPK-Log. The only instances of DBPK providing concrete commitments and proofs of knowledge are based on the discrete logarithm in $\mathbb{Z}_p^*$ and are called DBPK-Log. We now describe these commitments and proofs of knowledge.

We use a strong prime $p$, two generators $g, h$ of $\mathbb{Z}_p^*$, an element $x$ of $\mathbb{Z}_{p-1}$, and $y = g^x \bmod p$.\(^3\)

We have $\text{commit}(b, v) = g^{bh^v} \bmod p$. The main property of this commitment is that given all $z_{k,i}, z_{e,i}, v_i, v_i'$, we can form

$$z = \prod_i(z_{k,i}z_{e,i})^{2^{-i}}, v = \sum_i(v_i + v_i')2^{i-1},$$

and obtain that

$$z = \text{commit}((k + e) \bmod (p - 1), v).$$

The proposed encryption methods use $e = (ux - k) \bmod (p - 1)$ with either $u = 1$ [7] or $u$ random and publicly revealed [7,6]. So, the proof of knowledge consists of proving knowledge of $x$ and $v$ such that $y = g^x$ and $z = g^{ux}h^v$ in $\mathbb{Z}_p$.

The proof of knowledge [6] is repeating $t$ times what follows: the prover sends $w_1 = g^{u\rho_1}h^{\rho_1} \bmod p$ for some random $\rho_1, \rho_2 \in \mathbb{Z}_{p-1}$; the verifier sends some challenge $c \in \{0, 1\}$; the prover responds by $s_1 = \rho_1 - cx \bmod (p - 1)$ and $s_2 = \rho_2 - cv \bmod (p - 1)$; the verifier checks $w_1 = c'g^{u\rho_1}h^{s_1} \bmod p$ and $w_2 = y^c g^{s_1} \bmod p$.

Terrorist Fraud and Distance Fraud against DBPK-Log We now summarise how, in [2], the authors show that the above public-key techniques are ineffective in defeating terrorist-fraud. For this, we consider a malicious prover who is far away from an honest verifier. There is an adversary close to the verifier who will get some help from the prover to pass the protocol without getting any advantage to further impersonate the prover. The attack is sketched in Fig. 4.

First, the malicious prover selects $u$ and $v \in \mathbb{Z}_{p-1}$ (with either $u = 1$ or a random $u$, as specified in DBPK-Log), then computes $z = g^{ux}h^v \bmod p$ and sends $z$ to the adversary. The adversary selects some random $k_i, e_i, v_i, v_i'$, $i = 1, \ldots, m$, and a random bit $c_1$. Then, he computes $z_{k,i} = \text{commit}(k_i, v_i)$ and $z_{e,i} = \text{commit}(e_i, v_i')$ for $i = 2, \ldots, m$. If $c_1 = 0$, he sets $z_{k,1} = \text{commit}(k_1, v_1)$ and $z_{e,1}$ remains free. If $c_1 = 1$, he sets $z_{e,1} = \text{commit}(e_1, v_1')$ and $z_{k,1}$ remains free. Then, he can solve the equation $z = \prod_i(z_{k,i}z_{e,i})^{2^{-i}} \bmod p$ in the remaining free variable. Next, the adversary runs the DBPK-Log initialization phase, distance-bounding phase, and opening phase using these values. Note that if the value of the challenge $c_1$ received from the verifier differs from the value of the bit $e_1$ which were selected, the attack aborts.\(^4\) Otherwise, it is straightforward to see that the adversary can answer all challenges and open all commitments. Then,

\(^3\) In [6,7,5], $h$ is not necessarily a generator and $x \in \mathbb{Z}_{p-1} \setminus \{q\}$ with $q = \frac{p-1}{2}$.

\(^4\) The attack could also go on with the adversary taking $c_1$ as the value he selected, and counting on that the verifier will accept this error as due to noise.
<table>
<thead>
<tr>
<th>Verifier V</th>
<th>Adversary A</th>
<th>Prover P</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Initialization phase</strong></td>
<td>[ z \leftarrow z = g^{ux}h^v \mod p ]</td>
<td>[ z = g^{ux}h^v \mod p ]</td>
</tr>
<tr>
<td>A guesses the future value ( c_1 ) and compute the commitments:</td>
<td>[ \text{set } z_{k,1} := \text{commit}(k_i, v_i), \ z_{e,1} := \text{commit}(e_i, v_i') \text{ for all } i &gt; 1 ]</td>
<td>[ \text{set } z_{k,1} := \text{commit}(k_1, v_1), \ z_{e,1} := \text{commit}(e_1, v_1') \text{ if } c_1 = 0 \text{ ( } z_{e,1} \text{ is a free variable) } ]</td>
</tr>
<tr>
<td>[ \text{set } z_{e,1} := \text{commit}(e_1, v_1') \text{ if } c_1 = 1 \text{ ( } z_{k,1} \text{ is a free variable) } ]</td>
<td>[ \text{solve } z = \prod_i (z_{k,i}z_{e,i})^{2^{i-1}} \text{ in the remaining free variable for } i = 1 \text{ to } m ]</td>
<td>[ z := \prod_i (z_{k,i}z_{e,i})^{2^{i-1}} \leftarrow \prod_i (z_{k,i}z_{e,i}) ]</td>
</tr>
<tr>
<td><strong>Distance-bounding phase</strong></td>
<td>[ c_i \rightarrow \text{if } i = 1 \text{ and } c_1 \text{ incorrect, abort} ]</td>
<td>[ r_i := \begin{cases} k_i &amp; \text{if } c_i = 0 \ e_i &amp; \text{if } c_i = 1 \end{cases} ]</td>
</tr>
<tr>
<td><strong>Commitment opening phase</strong></td>
<td>[ r_i \leftarrow r_i ]</td>
<td>[ \gamma_i := \begin{cases} v_i &amp; \text{if } c_i = 0 \ v_i' &amp; \text{if } c_i = 1 \end{cases} ]</td>
</tr>
<tr>
<td><strong>Proof of knowledge phase</strong></td>
<td>[ \text{verify } z \leftarrow \text{prove } x, v \text{ for } z ]</td>
<td>[ \text{prove } x, v \text{ for } z ]</td>
</tr>
</tbody>
</table>

Fig. 4. Terrorist Fraud against DBPK-Log
the verifier will compute \( z \) which matches the one selected by the malicious prover. Finally, the adversary relays the proof of knowledge for \( x \) and \( v \) (such that \( z = g^{uv} r^v \mod p \)).

Clearly, this attack succeeds with probability \( \frac{1}{2} \) (or even 1 if the verifier allows an error in the first round). It is also clear that since the proof-of-knowledge is zero-knowledge and that \( x \) is not used anywhere else, that the adversary learns no information about \( x \). So, it is a valid terrorist-fraud.

In fact, it is easy to see that the attack above can be transformed into a DF as well. This is also exhibited in [2].

In fact, [2] further proposes a series of attacks on follow-ups of the Bussard-Bagga protocols, this time protocols based only on symmetric key techniques. The authors of [2] notice that the information leaked from the combination of the protocol with the answers on the verifier’s return channel can be exploited by a man-in-the-middle attacker to mount generalised mafia-fraud attacks.

We conclude this section by saying that recently alarming shortcomings of the security claims on DB have been published. Thus, we call for provable security of DB. In the following section, we are going to give some directive lines towards that, drawn from the lessons taught by the attacks above.

3 Towards Provably Secure DB

3.1 PRF Masking

First recall the general schema of DB in Fig.1. Also, remember that in the DF attack presented in Section 2.1, a dishonest prover was adaptively choosing a nonce \( N_p \) to influence the distribution of the output of the PRF instance \( f_x(\ldots,N_P) \) and eventually mount a DF attack. To avoid such situations, we propose enhancing the DB protocols with a technique that we call PRF masking:

— instead of \( f_x(\ldots) \) being identically calculated on messages\( _p \), messages\( _V \) on \( P \)'s and \( V \)'s sides, a bit-string \( a \) is chosen by the verifier and sent encrypted using the PRF instance \( f_x \), i.e.,

\[
M = a \oplus f_x(\text{messages}_p, \text{messages}_V)
\]

is sent by \( V \) to \( P \).

3.2 Circular Keying Security

The MiM attacks presented in [3], attacks that follow the idea in Section 2.1, are possible due to the use of the key \( x \) both in the PRF instance and in the response-function \( F \). This is not how PRF instances are normally used, when invoked for their security properties, i.e., for their random-like outputs. Therefore, we would like to amend the shortfalls presented in [3] by requiring that the PRF instances used inside DB protocols are such that their underlying key can be re-used outside of the function in some linear fashion, without this causing a key leakage. More precisely, we would require that a PRF \( f \) is circular keying secure:

— if \( \mathcal{A} \) makes a query \((y_i,a_i,b_i)\), the oracle answers \((a_i \cdot x') + (b_i \cdot f_x(y_i))\) and \( \mathcal{A} \) cannot distinguish if \( x = x' \) or \( x \) and \( x' \) are independent.

There is small caveat to this requirement, namely that for all \( c_1, \ldots, c_q \) such that \( c_1 b_1 + \cdots + c_q b_q = 0 \), we must have \( c_1 a_1 + \cdots + c_q a_q = 0 \).

3.3 A Secure DB Attempt

Summing up the above requirements, we conjecture that a protocol of the following kind would resist the new attacks in [3].

4 Conclusions

With the above protocol in mind and with the recent threats summarised herein, we conclude that best-known attacks or security claims on DB are not enough. DB protocols will soon be implemented by
Verifier: secret: x

Prover: secret: x

**initialization phase**

\[ \text{pick } N_P, \text{NP} \quad \text{pick } M,N_V \quad M,N_V \quad \rightarrow \quad a_1 \parallel a_2 = M \oplus f_x(N_P,N_V) \]

**distance bounding phase**

for \( i = 1 \) to \( n \)

\[ \text{pick } c_i \in \{1,2,3\} \quad \text{start clock } c_i \quad \text{stop clock } r_i \quad r_i = \begin{cases} a_{1,i} & \text{if } c_i = 1 \\ a_{2,i} & \text{if } c_i = 2 \\ x_i \oplus a_{1,i} \oplus a_{2,i} & \text{if } c_i = 3 \end{cases} \]

check \( \tau \) responses

check timers

\[ \text{Out}_V \quad \rightarrow \]

**NOTE:** \( f \) is a circular-keying secure PRF and it has many possible variants

Fig. 5. A Protocol Directing Towards Provably Secure DB

car manufacturers or bank payment companies in their products, as platforms for such deployments arise [13]. Thus, what we would need now is solid communication/threat models, clear design and more than anything else reliable security proofs in these models. We launch therefore a call for this.

**References**


The Fundamental Theorem and Links Between Attacks on Block Ciphers

Kaisa Nyberg
Aalto University School of Science
Department of Information and Computer Science
kaisa.nyberg@aalto.fi

Abstract. Chabaud and Vaudenay showed a mathematical relation between linear and differential cryptanalysis in 1994. Since then various new cryptanalysis methods have been established. Not until in 2011, when Leander proved the connection between the statistical saturation attack and multidimensional linear cryptanalysis, the relationships between different attacks have received much attention. In this brief note, we take another look at the principles behind Leander’s proof. We also show a new link between multidimensional linear approximations and truncated differentials. The results given in this short note are based on joint work with Céline Blondeau to appear at Eurocrypt 2013.

1 Fundamental Theorem

Computations of averages of squared correlations of Boolean functions over partially fixed input is a result that has taken different appearances in literature. To describe some of them let us first introduce some notation for a Boolean function $f : \mathbb{F}_2^n \rightarrow \mathbb{F}_2$. These are

$$\hat{f}(u) = \sum_{x \in \mathbb{F}_2^n} (-1)^{\langle u, x \rangle + f(x)}$$

$$\Pr(\langle u, x \rangle + f(x) = 0) = \frac{c + 1}{2}$$

$$c = \text{cor}(\langle u, x \rangle + f(x)) = \text{cor}_f(u) = \frac{\hat{f}(u)}{2^n}$$

where $\langle u, x \rangle$ is the inner product of two vectors $u$ and $x$ in $\mathbb{F}_2^n$, $\hat{f}$ is the Walsh-Hadamard transform of $f$ and $c$ is the correlation between the Boolean function $\langle u, x \rangle + f(x)$ and the all-zero function.

We focus on a situation where the input space of $f$ is divided into two parts and is presented as a Cartesian product of two subspaces. Without loss of generality we can describe such a situation as follows:

$$f : \mathbb{F}_2^r \times \mathbb{F}_2^s \rightarrow \mathbb{F}_2, \quad \hat{f}(u, v) = \sum_{x \in \mathbb{F}_2^r, z \in \mathbb{F}_2^s} (-1)^{\langle u, x \rangle + \langle v, z \rangle + f(x, z)}$$

$$f_x(z) = f(x, z), \quad f_x : \mathbb{F}_2^s \rightarrow \mathbb{F}_2, \quad x \in \mathbb{F}_2^r$$

In this situation the following theorem holds.

Theorem 1. For all $v \in \mathbb{F}_2^s$

$$2^r \sum_{x \in \mathbb{F}_2^r} \hat{f}_x(v)^2 = \sum_{u \in \mathbb{F}_2^r} \hat{f}(u, v)^2, \quad \text{or equivalently,}$$

$$2^{-r} \sum_{x \in \mathbb{F}_2^r} \text{cor}_{f_x}(v)^2 = \sum_{u \in \mathbb{F}_2^r} \text{cor}_f(u, v)^2.$$
Theorem 2. Let $X$, $Z$ and $Y$ be random variables in $\mathbb{F}_m^2$, $\mathbb{F}_2^\ell$, and $\mathbb{F}_n^2$, resp. where $Y = F(X,Z)$ and $X$ and $Z$ are independent. If $Z$ is uniformly distributed, then for all $a \in \mathbb{F}_2^m$ and $b \in \mathbb{F}_2^n$,

$$\text{Exp}_Z \text{cor}((a,X) + (b,Y))^2 = \sum_{c\in\mathbb{F}_2^n} \text{cor}((a,X) + (b,Y) + (c,Z))^2.$$ 

It was subsequently applied to the case $F(X,Z) = E_Z(X)$ of an encryption function of a (reduced-round) block cipher with key $Z$ and plaintext $X$. The result is known as the Linear Hull theorem.

1.1 Partially Fixed Plaintext

In this section, we discuss the applications of the Fundamental theorem to the case of partially fixed plaintext.

In 2009 Collard and Standaert presented an attack on block cipher PRESENT and called it statistical saturation attack [5]. They identified two subsets of the state bits, one with 8 bits and a second one with 27 bits, that are diffused poorly in the round operation. They studied using simulations how the distribution of the values in the 8-bit set evolve when passing from round to round. The initial distribution was taken as the most non-uniform as possible, that is, at the initial round all bits in the subset under consideration were fixed. In lack of theoretical model the strength of the attack remained open.

One year later Cho presented a multidimensional cryptanalysis attack where he constructed a distinguisher on 23 rounds of PRESENT using multidimensional linear approximations over the same subset of 27 bits as previously identified in [5] as a possible target for a statistical saturation attack.

Leander showed in [8] that these attacks are mathematically equivalent in the sense that the capacity of the multidimensional linear approximation in Cho’s attack is equal to the average capacity of the output distribution in the statistical saturation attack taken over the fixed parts of the plaintext. To this end he proved the following result.

Theorem 3. Let $F: \mathbb{F}_2^r \times \mathbb{F}_2^n \rightarrow \mathbb{F}_2^k$ be a vectorial Boolean function. Then

$$2^{-r} \sum_{x \in \mathbb{F}_2^r} \sum_{w \in \mathbb{F}_2^n \setminus \{0\}} \text{cor}((w,F(x,z)))^2 = \sum_{u \in \mathbb{F}_2^r} \sum_{w \in \mathbb{F}_2^n \setminus \{0\}} \text{cor}((u,x) + (w,F(x,z)))^2$$

Note that the proof of this result can be obtained by a direct application of the Fundamental theorem and removing the all-zero linear approximation from the summations on both sides of the equation. The expression on the right-hand side of the equation is the capacity of the multidimensional linear approximation $\langle u,x \rangle + \langle w,F(x,z) \rangle$, $u \in \mathbb{F}_2^r$, $w \in \mathbb{F}_2^n$.

The expression on the left-hand side is the average capacity of the output distribution. While the attacks are the same in theory, they differ significantly in practice. Running the known plaintext multidimensional linear attack takes $2^{r+k}$ memory. On the other hand, sampling for evaluation of the expression on the left-hand side can be done with $2^k$ memory using chosen plaintext. Moreover, it may be possible to restrict to a subset of fixed plaintexts $x$, and in this manner, reduce time complexity of the statistical saturation attack. It remains to be studied, how much the behaviour of the output distribution for a fixed $x$ differs from the average behaviour.

1.2 Integral Attacks and Zero-Correlations

As a straightforward consequence of Theorem 3 we get another equivalence.

Theorem 4. Let $F: \mathbb{F}_2^r \times \mathbb{F}_2^n \rightarrow \mathbb{F}_2^k$ be a vectorial Boolean function. Then the following conditions are equivalent

(i) $\text{cor}((w,F(x,z))) = 0$, for all $x \in \mathbb{F}_2^r$ and $w \in \mathbb{F}_2^n \setminus \{0\}$

(ii) $\text{cor}((u,x) + \langle w,F(x,z) \rangle = 0$, $u \in \mathbb{F}_2^r$, $w \in \mathbb{F}_2^n$, $(u,w) \neq (0,0)$

Condition (i) means that we have an integral distinguisher, and condition (ii) means that we have a zero-correlation distinguisher. Indeed, this theorem was only of the main results of [2], where also applications were presented.
2 Links Between Differential and Linear Cryptanalysis

For a differential $F(x + \delta) + F(x) = \Delta$ we use notation $\delta \xrightarrow{F} \Delta$. The seminal link given by Chabaud and Vaudenay can be stated as follows [4].

**Theorem 5.** Let $F : \mathbb{F}_2^n \rightarrow \mathbb{F}_2^m$ be a vectorial Boolean function. Then

$$
\Pr(\delta \xrightarrow{F} \Delta) = 2^{-m} \sum_{a \in \mathbb{F}_2^n, b \in \mathbb{F}_2^m} (-1)^{\langle a, \delta \rangle + \langle b, \Delta \rangle} \text{cor}(\langle a, x \rangle + \langle b, F(x) \rangle)^2
$$

for all $\delta \in \mathbb{F}_2^n$ and $\Delta \in \mathbb{F}_2^m$.

Assume now that the input space and output space is split into two subspaces so that $F : \mathbb{F}_2^s \times \mathbb{F}_2^t \rightarrow \mathbb{F}_2^r \times \mathbb{F}_2^s$. For each element in such a Cartesian product we denote the components by subscripts indicating the dimension of the space they belong to. For example, $x = (x_s, x_t) \in \mathbb{F}_2^s \times \mathbb{F}_2^t$. We consider the truncated differential $(\delta_s, \ast) \xrightarrow{F} (\Delta_q, \ast)$ and define its probability as

$$
\Pr((\delta_s, \ast) \xrightarrow{F} (\Delta_q, \ast)) = 2^{-q} \sum_{\delta_t \in \Delta_r} \Pr((\delta_s, \delta_t) \xrightarrow{F} (\Delta_q, \Delta_r)).
$$

Then by summing up on both sides of (1) over $\delta_t$ and $\Delta_r$ we obtain the following result [1].

**Theorem 6.**

$$
\Pr((\delta_s, \ast) \xrightarrow{F} (\Delta_q, \ast)) = 2^{-q} \sum_{a_s, b_q} (-1)^{\langle a_s, \delta_s \rangle + \langle b_q, \Delta_q \rangle} \text{cor}(\langle (a_s, 0), x \rangle + \langle (b_q, 0), F(x) \rangle)^2
$$

for all $\delta_s$ and $\Delta_q$.

The consequences of this theorem, in case where all nontrivial correlations are zero, are discussed in detail in [1]. Here we focus on the following result.

**Corollary 1.** Let $F : \mathbb{F}_2^s \times \mathbb{F}_2^t \rightarrow \mathbb{F}_2^r \times \mathbb{F}_2^s$. Then the following are equivalent.

(i) $\text{cor}(\langle (b_q, 0), F(x_s, x_t) \rangle) = 0$ for all $x_s$ and $b_q \neq 0$

(ii) $\Pr((\delta_s, \ast) \xrightarrow{F} (\Delta_q, \ast)) = 2^{-q}$ for all $\delta_s$ and $\Delta_q$

The first condition means that the distribution of the first $q$ bits of the output is uniform when taken over a fixed component $x_s$ and variable component $x_t$ in the input. The latter condition, on the other hand, means that all truncated differentials $(\delta_s, \ast) \xrightarrow{F} (\Delta_q, \ast)$ have equal probability.

A concrete example of a function $F$ where the former condition is known to hold is three rounds backwards or four rounds forward of the AES encryption function, see [7] or the presentation by Gilbert at ESC2013 [6]. In this example, the fixed part of the input consists of nine subbytes and the output part, with uniform distribution, is one (any) subbyte. Hence in this case $F : \mathbb{F}_2^9 \rightarrow \mathbb{F}_2^9 \rightarrow \mathbb{F}_2^9 \rightarrow \mathbb{F}_2^9 \rightarrow \mathbb{F}_2^9 \rightarrow \mathbb{F}_2^9 \rightarrow \mathbb{F}_2^9 \rightarrow \mathbb{F}_2^9 \rightarrow \mathbb{F}_2^9$.

Then by the latter condition of Corollary 1 all truncated differentials $(\alpha, \ast) \xrightarrow{F} (\beta, \ast)$, where $\alpha$ is active only in certain nine subbytes and $\beta$ is active only in one subbyte, are equally likely. Such a property can be developed to a chosen state distinguisher on the known-key AES [7]. Moreover, by the equivalence of zero-correlation linear attacks and integral attacks, we also obtain a known state distinguisher based on the property that all nontrivial linear approximations involving input bits from the nine subbytes and output bits from one subbyte have correlation zero.

3 Conclusion

We discussed the Fundamental theorem and some of its previous appearances in the literature on Boolean functions. In particular we show how some recently established results on equivalences between different cryptanalysis methods can be proved by a straightforward application of it. Then we revisited the classical link by Chabaud and Vaudenay between differential and linear cryptanalysis and showed how to establish some new links between multidimensional linear cryptanalysis, integral attacks and truncated differential attacks.
Acknowledgments. I wish to thank Céline Blondeau, Andrey Bogdanov and Gregor Leander for inspiring discussions.

References

Practical-Time Attacks Against Reduced Variants of MISTY1

Orr Dunkelman\(^1\) and Nathan Keller\(^2\)

\(^1\) University of Haifa, Israel
orrd@cs.haifa.ac.il
\(^2\) Bar-Ilan University, Israel
nkeller@math.biu.ac.il

Abstract. MISTY1 is a block cipher designed by Matsui in 1997. It is widely deployed in Japan where it is an e-government standard, and is recognized internationally as a NESSIE-recommended cipher as well as an ISO standard and an RFC. Moreover, MISTY1 was selected to be the blueprint on top of which KASUMI, the GSM/3G block cipher, was based. Since its introduction, and especially in recent years, MISTY1 was subjected to extensive cryptanalytic efforts, which resulted in numerous attacks on its reduced variants. Most of these attacks aimed at maximizing the number of attacked rounds, and as a result, their complexities are highly impractical.

In this paper we pursue another direction, by focusing on attacks with a practical time complexity. The best previously-known attacks with practical complexity against MISTY1 could break either 4 rounds (out of 8), or 5 rounds in a modified variant in which some of the FL functions are removed. We present an attack on 5-round MISTY1 with all the FL functions present whose time complexity is \(2^{38}\) encryptions. When the FL functions are removed, we present a devastating (and experimentally verified) related-key attack on the full 8-round variant, requiring only \(2^{18}\) data and time.

While our attacks clearly do not compromise the security of the full MISTY1, they expose several weaknesses in MISTY1’s components, and improve our understanding of its security. Moreover, future designs which rely on MISTY1 as their base, should take these issues into close consideration.
Collision Attacks on Up to 5 Rounds of SHA-3 Using Generalized Internal Differentials *

Itai Dinur¹, Orr Dunkelman², and Adi Shamir³

¹ The Weizmann Institute
itai.dinur@weizmann.ac.il
² University of Haifa and The Weizmann Institute
orr.dunkelman@weizmann.ac.il
³ The Weizmann Institute
adi.shamir@weizmann.ac.il

Abstract. On October 2nd 2012 NIST announced its selection of the Keccak scheme as the new SHA-3 hash standard. In this paper we present the first published collision finding attacks on reduced-round versions of Keccak-384 and Keccak-512, providing actual collisions for 3-round versions, and describing an attack which is $2^{45}$ times faster than birthday attacks for 4-round Keccak-384. For Keccak-256, we increase the number of rounds which can be attacked to 5. All these results are based on a generalized internal differential attack (introduced by Peyrin at Crypto 2010), and use it to map a large number of Keccak inputs into a relatively small subset of possible outputs with a surprisingly large probability. In such a squeeze attack it is easier to find random collisions in the reduced target subset by a standard birthday argument.

* Full version of this paper is available from http://eprint.iacr.org/2012/672.
Generalizing/Improving Long Standing Results on RC4 Key Scheduling

Subhamoy Maitra¹, Goutam Paul², Santanu Sarkar³, Michael Lehmann⁴, and Willi Meier⁴

¹ Applied Statistics Unit, Indian Statistical Institute, Kolkata 700 108, India
subho@isical.ac.in
² Department of Computer Science and Engineering, Jadavpur University, Kolkata 700 032, India
goutam.paul@ieee.org
³ Computer Security Division, NIST, MD 20899-8930, U.S.A.
sarkar.santanu.bir@gmail.com
⁴ FHNW, 5210 Windisch, Switzerland
michael.lehmann@students.fhnw.ch, willi.meier@fhnw.ch


Andrew Roos pointed out in 1995 that the most likely value of the y-th element of the permutation after the KSA for the first few values of y is given by $S_N[y] = f_y$, some linear combinations of the secret keys. Surprisingly, the association $S_N[y] = f_y \pm t$ for small positive integers $t$ (e.g., $t \leq 4$), though appears to be an immediate natural extension, had never been studied before. In this paper, we investigate this in detail and establish that though the event $S_N[y] = f_y + t$ occurs with random association, the event $S_N[y] = f_y - t$ occurs with a significantly high probability. We also show that the nested bias of the form $S_N[S_N[y]] = f_y$ (Maitra-Paul, FSE 2008) also generalizes to $S_N[S_N[y]] = f_y - t$.

Finding full or partial key collisions in RC4 is an important research problem. In this paper, we investigate near-colliding key-pairs that lead to related states after key scheduling and then related output bytes in the keystream. We find that near-colliding states do not necessarily lead to near-colliding keystreams. From this motivation, we present a heuristic to find a related key-pair with differences in two bytes, that lead to significant collision in the initial segment of the keystream. This results in a class of related key distinguishers for RC4. The best distinguisher in this class corresponds to a related key-pair, such that the last two bytes in the second key is increased and decreased by 1 respectively from that of the first key (the other bytes in the two keys are the same), and the corresponding pair of first keystream bytes collide with very high probability (e.g., approximately 0.011 for 16-byte keys to 0.044 for 30-byte keys).
A Method for Automatic Search for Differential Trails in ARX Ciphers

Alex Biryukov and Vesselin Velichkov
Laboratory of Algorithmics, Cryptology and Security (LACS)
University of Luxembourg
{Alex.Biryukov,Vesselin.Velichkov}@uni.lu

Abstract. We propose a method for automatic search for differential trails in ARX ciphers, based on Matsui's branch-and-bound technique. By introducing the concept of a partial difference distribution table (pDDT) we extend Matsui's algorithm, originally proposed for DES-like ciphers, to the class of ARX ciphers. To the best of our knowledge this is the first application of Matsui's algorithm to ciphers that do not have S-boxes. We have implemented a practical tool for automatic search for differential trails and we use the block ciphers TEA and XTEA as a testbed for its demonstration. For TEA we find multiple differential trails with probability ≈ 2⁻⁶⁰ covering between 15 and 18 rounds depending on the value of the key. The 18 round trail has probability ≈ 2⁻⁶³ for approx. 2¹¹⁶ (≈ 0.1%) of all keys. It covers one round more than the best differential attack on TEA which is on 17 rounds and significantly improves the best previously known truncated trail which is on 8 rounds. As far as we know these are the first reported full (i.e. not truncated) differential trails on TEA. For XTEA, the best XOR trail found by our tool is based on an iterative pattern for 3 rounds, covers 14 rounds in total and confirms the previous best known trail reported by Hong et al. Finally, we comment on general problems and limitations arising when studying differential trails in ARX ciphers. In particular, we discuss difficulties arising from the large word size, the dependence of the probabilities of trails on the value of the key, the dependence between the round keys as well as the influence of the additive round constants. The source code of the described tool will be made publicly available as part of a larger toolkit for the analysis of ARX.

Keywords: symmetric-key, differential trail, tools for cryptanalysis, automatic search, ARX, TEA, XTEA

1 Introduction

A broad class of symmetric-key cryptographic algorithms are designed by combining a small set of simple operations such as modular addition, bit rotation, bit shift and XOR. Although such designs have been proposed as early as the 1980s, only recently the term ARX (from Addition, Rotation, XOR) was informally adopted [40, 19] in reference to them.


By combining linear (XOR, bit shift, bit rotation) and non-linear (modular addition) operations, and iterating them over multiple rounds, ARX algorithms achieve strong resistance against standard cryptanalysis techniques such as linear [25] and differential [5] cryptanalysis. Additionally, due to the simplicity of the underlying operations, they are typically very fast in software.

Although ARX designs have many advantages and have been widely used for many years now, the methods for their rigorous security analysis are lagging behind. This is especially true when compared to algorithms such as AES [10] and DES [30]. The latter were designed using fundamentally different
principles, based on the combination of linear transformations and non-linear substitution tables (S-boxes). For such algorithms, it is possible to compute quantities such as the maximum differential probability and linear correlation of the S-box, the differential branch number of a transformation and the minimum number of active S-boxes over a given number of rounds. These allow to quantitatively estimate the security level of the cipher by proving lower bounds on the maximum probability and correlation of differential trails.

One of the reasons why the security analysis techniques for ciphers such as AES and DES are not applicable to ARX designs is the fact that the latter do not have S-boxes. Since a typical S-box operates on 8 or 4-bit words, it is easy to efficiently evaluate its differential (resp. linear) properties by simply enumerating all possible input and output differences (resp. masks) and computing the differential probability (resp. the correlation) for each one of them. This can be done by means of a difference distribution table (DDT) (resp. linear approximation table (LAT)), which for an 8-bit S-box is of size $2^8 \times 2^8$ Bytes. In contrast, ARX algorithms use modular addition as a source of non-linearity, rather than S-boxes. Constructing a DDT or a LAT for this operation for $n$-bit words would require $2^n \times 2^n \times 2^n \times 4$ Bytes of memory and would clearly be infeasible for a typical word size of 32 bits.

In this paper we demonstrate that although the computation of a full DDT for ARX is infeasible, it is still possible to efficiently compute a partial DDT containing (a fraction of) all differentials that have probability above a fixed threshold. This is possible due to the fact that the probabilities of XOR (resp. ADD) differentials through the modular addition (resp. XOR) operation are monotonously decreasing with the bit size of the word.

Based on the concept of partial DDT-s we develop a method for automatic search for differential trails in ARX ciphers. It is based on Matsui’s branch-and-bound algorithm [24], originally proposed for S-box based ciphers, and used to find the best differential trails and linear approximations for up to 16 rounds of DES. While other methods for automatic search for differential trails in ARX designs exist in literature [13, 26, 21] they have been exclusively applied to the analysis of hash functions where the key (the message) is known and can be freely chosen. With the proposed algorithm we address the more general setting of searching for trails in block ciphers, where the key is fixed and unknown to the attacker.

Beside the idea of using partial DDT-s another fundamental concept at the heart of the proposed algorithm is what we refer to as the highways and country roads analogy. If we liken the problem of finding high probability differential trails in a cipher to the problem of finding fast routes between two cities on a road map, then differentials that have high probability (w.r.t. a fixed threshold) can be thought of as highways and conversely differentials with low probability can be viewed as slow roads or country roads. To further extend the analogy, a differential trail for $n$ rounds represents a route between points 1 and $n$ composed of some number of highways and country roads. A search for high probability trails is analogous to searching for a route in which the number of highways is maximized while the number of country roads is minimized.

The differentials from the pDDT are the highways on the road map from the above analogy. Beside those highways, the proposed search algorithm explores also a certain number of country roads (low probability differentials). While the list of highways is computed offline prior to the start of the search, the list of country roads is computed on demand for each input difference to an intermediate round that is encountered during the search. Of all possible country roads that can be taken at a given point (note that there may be a huge number of them), the algorithm considers only the ones that lead back on a highway. This prevents the number of explored routes from exploding and at the same time keeps the total probability of the resulting trail high.
Table 1. Maximum number of rounds covered by single (truncated) differential trails used in existing differential attacks on TEA and XTEA compared to the best trails reported in this paper.

<table>
<thead>
<tr>
<th>Cipher</th>
<th>Type of trail</th>
<th>#Rounds</th>
<th>Ref.</th>
<th>Cipher</th>
<th>Type of trail</th>
<th>#Rounds</th>
<th>Ref.</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEA</td>
<td>Trunc.</td>
<td>5</td>
<td>[27]</td>
<td>XTEA</td>
<td>Trunc.</td>
<td>6</td>
<td>[27]</td>
</tr>
<tr>
<td></td>
<td>Trunc.</td>
<td>7</td>
<td>[9]</td>
<td></td>
<td>Trunc.</td>
<td>7</td>
<td>[9]</td>
</tr>
<tr>
<td></td>
<td>Trunc.</td>
<td>8</td>
<td>[17, 7]</td>
<td></td>
<td>Trunc.</td>
<td>8</td>
<td>[17, 7]</td>
</tr>
<tr>
<td></td>
<td>Full</td>
<td>18</td>
<td>this paper</td>
<td></td>
<td>Full</td>
<td>14</td>
<td>this paper</td>
</tr>
</tbody>
</table>

Due to the fact that it uses a partial, rather than the full DDT, our algorithm is not guaranteed to find the best differential trail. However, experiments on small word sizes of 11, 14 and 16 bits show that the probabilities of the found trails are within a factor of at most $2^{-3}$ from the probability of the best one.

We have developed a tool based on our technique and tested it by searching for additive differential trails in block cipher TEA and for XOR differential trails in its extended variant – XTEA. The reason to choose those two particular ciphers as our testbed is their very simple ARX structure. At the same time, the two are good representatives of the ARX class of algorithms, since they concentrate in one place multiple characteristics that are often encountered in the differential analysis of ARX. Such are, for example, the dependence of the probability of trails on the value of the round keys and round constants, the dependence between the round keys as well as the notorious difficulty of computing the exact differential probabilities through larger ARX components.

In addition to the above, finding differential trails in TEA represents a challenge in itself. Due to the mentioned problems there are virtually no published results that report full (i.e. not truncated) differential trails in TEA. Indeed, in [17, Sect. 1] Hong et al. acknowledge that it is difficult to find a good differential characteristic for TEA.

By applying our tool, we are able to find multiple differential characteristics for TEA. They cover between 15 and 18 rounds, depending on the value of the key and have probabilities $\approx 2^{-60}$. The 18 round trail, in particular, has probability $\approx 2^{-63}$ for approx. $2^{116}$ ($\approx 0.1\%$) of all keys. To put those results in perspective, we note that the best differential attack on TEA covers 17 rounds and is based on an impossible differential [9] while the best attack overall applies zero-correlation cryptanalysis and is on 23 rounds but requires the full codebook [7]. For XTEA, we confirm the best previously known full differential trail based on XOR differences [17], but this time it was found in a fully automatic way.

In Table 1 is provided a comparison between the number of rounds covered by single (truncated) differential trails used in existing attacks on TEA and XTEA to the number of rounds covered by the trails found with our method. As can be seen the reported 18 round trail on TEA significantly improves the previously best truncated differential trail which is for 8 rounds.

A minor secondary contribution of the paper is that it is the first to report closed formulas for computing the exact additive differential probabilities of the left and right shift operations. These formulas are derived in a similar way as the ones for computing the DP of left and right rotation reported by Daum [12, Sect. 4.1.3]. Note that Fouque et al. [15] have previously analyzed the propagation of additive differences through the shift operations, but not the corresponding differential probabilities.

The outline of the paper is as follows. In Sect. 2 we recall the definitions of the differential probabilities of the operations XOR and modular addition resp. $\text{xdp}^+$ and $\text{adp}^+$ and in Sect. 3 is analyzed the additive differential probabilities of left and right shift. In Sect. 4 we define partial difference distribu-
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n )</td>
<td>Number of bits in one word</td>
</tr>
<tr>
<td>( x(2) )</td>
<td>The binary representation of ( x )</td>
</tr>
<tr>
<td>( x[i] )</td>
<td>The ( i )-th bit of ( x )</td>
</tr>
<tr>
<td>( x[i : j] )</td>
<td>The sequence of bits ( x[j], x[j + 1], \ldots, x[i] : j \leq i ); ( x[0] ) is the LSB</td>
</tr>
<tr>
<td>( x_n )</td>
<td>( n )-bit word ( x ) (equivalent to ( x[n - 1 : 0] ), but more concise)</td>
</tr>
<tr>
<td>#A</td>
<td>Number of elements in the set ( A )</td>
</tr>
<tr>
<td>( x</td>
<td>y )</td>
</tr>
</tbody>
</table>

### 2 The Differential Probabilities of ADD and XOR

In this section we recall the definitions of the differential probabilities of the operations XOR and modular addition. Before we begin – a brief remark on notation: in the same way as XOR is used to denote both the XOR operation and an XOR difference, we use ADD to denote both the modular addition operation and an additive difference.

**Definition 1.** Let \( \alpha, \beta \) and \( \gamma \) be fixed \( n \)-bit XOR differences. The XOR differential probability (DP) of addition modulo \( 2^n \) (\( \text{xdp}^+ \)) is the probability with which \( \alpha \) and \( \beta \) propagate to \( \gamma \) through the ADD operation, computed over all pairs of \( n \)-bit inputs \( (x, y) \):

\[
\text{xdp}^+(\alpha, \beta \rightarrow \gamma) = 2^{-2n} \cdot \# \{(x, y) : ((x \oplus \alpha) + (y \oplus \beta) \oplus (x + y)) = \gamma \}.
\]  

---

1 Some of the appendix sections are provided only for the review process and can be removed later in order to fit the text within the 20 page limit.
The dual of $\text{xdp}^+$ is the probability $\text{adp}^\oplus$ and is defined analogously:

**Definition 2.** Let $\alpha$, $\beta$ and $\gamma$ be fixed $n$-bit ADD differences. The additive DP of XOR ($\text{adp}^\oplus$) is the probability with which $\alpha$ and $\beta$ propagate to $\gamma$ through the XOR operation, computed over all pairs of $n$-bit inputs $(x, y)$:

$$\text{adp}^\oplus(\alpha \rightarrow \gamma) = 2^{-2n} \cdot \# \{ (x, y) : ((x + \alpha) \oplus (y + \beta)) - (x + y) = \gamma \} \ .$$  \hspace{1cm} (2)

The probabilities $\text{xdp}^+$ and $\text{adp}^\oplus$ have been studied in [22] and [23] respectively, where methods for their efficient computation have been proposed. In [22] is also described an efficient algorithm for the computation of $\text{xdp}^+$ maximized over all output differences: $\max_\gamma \text{xdp}^+(\alpha, \beta \rightarrow \gamma)$. In [28] the methods for the computation of $\text{xdp}^+$ and $\text{adp}^\oplus$ are further generalized using the concept of S-functions.

Finally, in [39, Appendix C, Algorithm 1] a general algorithm for computing the maximum probability output difference for certain types of differences and operations is described. It is applicable to both $\max_\gamma \text{xdp}^+(\alpha, \beta \rightarrow \gamma)$ and $\max_\gamma \text{adp}^\oplus(\alpha, \beta \rightarrow \gamma)$.

### 3 The Additive DP of Left and Right Shift

**Definition 3.** For fixed input and output ADD differences resp. $\alpha$ and $\beta$, the additive differential probability of the operation **right bit shift** (RSH) by $r$ positions is defined over all $n$-bit ($n \geq r$) inputs $x$ as:

$$\text{adp}^{\gg r}(\alpha \rightarrow \beta) = 2^{-n} \cdot \# \{ x : ((x + \alpha) \gg r) - (x \gg r) = \beta \} \ .$$  \hspace{1cm} (3)

Analogously, the additive differential probability of the operation **left bit shift** (LSH) by $r$ positions is defined as in (3) after replacing $\gg r$ with $\ll r$.

**Theorem 1.** The LSH operation is linear with respect to ADD differences i.e. $(x + \alpha) \ll r) - (x \ll r) = (\alpha \ll r)$, where $x, \alpha$ and $r$ are as in Definition 3. It follows that

$$\text{adp}^{\ll r}(\alpha \rightarrow \beta) = \begin{cases} 1 , & \text{if } (\beta = \alpha \ll r) , \\ 0 , & \text{otherwise} . \end{cases}$$  \hspace{1cm} (4)

**Proof.** Appendix E.2.

In contrast to LSH, the RSH operation is not linear w.r.t. ADD differences. The following theorem provides expressions for the computation of $\text{adp}^{\gg r}$.

**Theorem 2.** Let $\alpha$ be a fixed $n$-bit input ADD difference to an RSH operation with shift constant $r \leq n$. Then there are exactly four possibilities for the output difference $\beta$. The four differences together with their corresponding probabilities computed over all $n$-bit inputs are:

$$\text{adp}^{\gg r}(\alpha \rightarrow \beta) = \begin{cases} 2^{-n}(2^{n-r} - \alpha_L)(2^r - \alpha_R) , & \beta = (\alpha \gg r) , \\ 2^{-n}\alpha_L(2^r - \alpha_R) , & \beta = (\alpha \gg r) - 2^{n-r} , \\ 2^{-n}\alpha_R(2^{n-r} - \alpha_L - 1) , & \beta = (\alpha \gg r) + 1 , \\ 2^{-n}(\alpha_L + 1)\alpha_R , & \beta = (\alpha \gg r) - 2^{n-r} + 1 . \end{cases}$$  \hspace{1cm} (5)

where $\alpha_L$ and $\alpha_R$ denote respectively the $(n - r)$ most-significant (MS) bits and the $r$ least-significant (LS) bits of $\alpha$ so that: $\alpha = \alpha_L 2^r + \alpha_R$ and additions and subtractions are performed modulo $2^n$. If $\alpha : \beta = \beta_i : \beta_j$ for some $0 \leq i \neq j < 4$ then $\text{adp}^{\gg r}(\alpha \rightarrow \beta) = \text{adp}^{\gg r}(\alpha \rightarrow \beta_i) + \text{adp}^{\gg r}(\alpha \rightarrow \beta_j)$.

**Proof.** Appendix E.3.
4 Partial Difference Distribution Tables

Definition 4. A partial difference distribution table (pDDT) \( D \) for the ADD (resp. XOR) operation is a DDT that contains all XOR (resp. ADD) differentials \((\alpha, \beta \rightarrow \gamma)\) whose probabilities are larger than or equal to a pre-defined threshold \( p_{\text{thres}} \):

\[
(\alpha, \beta, \gamma) \in D \iff \text{DP}(\alpha, \beta \rightarrow \gamma) \geq p_{\text{thres}}. 
\]  

(6)

If a DDT contains only a fraction of all differentials that have probability above a pre-defined threshold, it is an incomplete pDDT.

The following proposition is crucial for the efficient computation of a pDDT:

**Proposition 1.** The DP of ADD and XOR (resp. \( x_{\text{dp}}^+ \) and \( \text{adp}^\oplus \)) are monotonously decreasing with the word size \( n \) of the differences \( \alpha, \beta, \gamma \):

\[
p_n \leq \ldots \leq p_k \leq p_{k-1} \leq \ldots \leq p_1 \leq p_0, 
\]

where \( p_k = \text{DP}(\alpha_k, \beta_k \rightarrow \gamma_k), n \leq k \leq 1, p_0 = 1, \) and \( x_k \) denotes the \( k \) LSB-s of the difference \( x \) i.e. \( x_k = x[k - 1:0] \).

**Proof.** Appendix E.1.

For \( x_{\text{dp}}^+ \), the proposition follows from the following result by Lipmaa et al. [22]:

\[
x_{\text{dp}}^+(\alpha, \beta \rightarrow \gamma) = 2^{-\sum_{i=0}^{n-2} \text{eq}(\alpha[i], \beta[i], \gamma[i])}, \quad \text{where } \text{eq}(\alpha[i], \beta[i], \gamma[i]) = 1 \iff \alpha[i] = \beta[i] = \gamma[i].
\]

Proposition 1 is also true for \( \text{adp}^\oplus \).

Due to Proposition 1 a recursive procedure for computing a pDDT for a given probability threshold \( p_{\text{thres}} \) can be defined as follows. Starting at the least-significant (LS) bit position \( k = 0 \) recursively assign values to bits \( \alpha[k], \beta[k] \) and \( \gamma[k] \). At every bit position \( k : n > k \geq 0 \) check if the probability of the partially constructed \((k+1)\)-bit differential is still bigger than the threshold i.e. check if \( p_k = \text{DP}(\alpha_k, \beta_k \rightarrow \gamma_k) \geq p_{\text{thres}} \) holds. If yes, then proceed to the next bit position, otherwise backtrack and assign other values to \((\alpha[k], \beta[k], \gamma[k])\). This process is repeated recursively until \( k = n \), at which point the differential \((\alpha_n, \beta_n \rightarrow \gamma_n)\) is added to the pDDT together with its probability \( p_n \). A pseudo-code of the described procedure is listed in Algorithm 1. The initial values are: \( k = 0, \ p_0 = 1 \) and \( \alpha_0 = \beta_0 = \gamma_0 = 0 \).

**Algorithm 1** Computation of a pDDT for ADD and XOR.

**Input:** \( n, p_{\text{thres}}, k, p_k, \alpha_k, \beta_k, \gamma_k \).

**Output:** pDDT \( D \): \((\alpha, \beta, \gamma) \in D : \text{DP}(\alpha, \beta \rightarrow \gamma) \geq p_{\text{thres}}\).

1: **procedure compute\_pddt**\((n, p_{\text{thres}}, k, p_k, \alpha_k, \beta_k, \gamma_k)\) **do**
2: 2: **if** \( n = k \) **then**
3: 3: **Add** \((\alpha, \beta, \gamma) \leftarrow (\alpha_k, \beta_k, \gamma_k)\) **to** \( D \)
4: 4: **return**
5: 5: **for** \( x, y, z \in \{0, 1\} \) **do**
6: 6: \( \alpha_{k+1} \leftarrow x|\alpha_k, \beta_{k+1} \leftarrow y|\beta_k, \gamma_{k+1} \leftarrow z|\gamma_k \)
7: 7: \( p_{k+1} = \text{DP}(\alpha_{k+1}, \beta_{k+1} \rightarrow \gamma_{k+1}) \)
8: 8: **if** \( p_{k+1} \geq p_{\text{thres}} \) **then**
9: 9: **compute\_pddt**\((n, p_{\text{thres}}, k + 1, p_{k+1}, \alpha_{k+1}, \beta_{k+1}, \gamma_{k+1})\)
The correctness of Algorithm 1 follows directly from Proposition 1. After successful termination the computed pDDT contains all differentials with probability equal to or larger than the threshold. The complexity of Algorithm 1 depends on the value of the threshold \( p_{\text{thres}} \). Some timings for both ADD and XOR differences for different thresholds are provided in Table 3. As can be seen from the data in the table it is infeasible to compute pDDT-s for XOR differences for values of the threshold \( p_{\text{thres}} \leq 0.01 = 2^{-6.64} \), while for ADD differences this is still possible, but requires significant time (more than 17 hours).

### 5 Threshold Search

In his paper from 1994 [24] Matsui proposed a practical algorithm for searching for the best differential trail (and linear approximation) for the DES block cipher. The algorithm performs a recursive search for differential trails over a given number of rounds \( n \geq 1 \). From knowledge of the best probabilities \( B_1, B_2, \ldots, B_{n-1} \) for the first \( (n-1) \) rounds and an initial estimate \( \mathcal{B}_n \) for the probability for \( n \) rounds it derives the best probability \( B_n \) for \( n \) rounds. For the estimate the following must hold: \( \mathcal{B}_n \leq B_n \). As already noted, Matsui’s algorithm is applicable to block ciphers that have S-boxes. In this section we extend it to the case of ciphers without S-boxes such as ARX by applying the concept of pDDT. We describe the extended algorithm next. Its description in pseudo-code is listed in Algorithm 2.

In addition to Matsui’s notation for the probability of the best \( n \)-round trail \( B_n \) and of its estimate \( \mathcal{B}_n \) we introduce \( \hat{B}_n \) to denote the probability of the best found trail for \( n \) rounds: \( \mathcal{B}_n \leq \hat{B}_n \leq B_n \). Given a pDDT \( H \) of size \( m \), an estimation for the best \( n \)-round probability \( \mathcal{B}_n \) with its corresponding \( n \)-round differential trail \( T \) and the probabilities \( \hat{B}_1, \hat{B}_2, \ldots, \hat{B}_{n-1} \) of the best found trails for the first \( n-1 \) rounds, Algorithm 2 outputs an \( n \)-round trail \( \hat{T} \) that has probability \( \hat{B}_n \geq \mathcal{B}_n \).

Similarly to Matsui’s algorithm, Algorithm 2 operates by recursively extending a trail for \( i \) rounds to \( (i+1) \) rounds, beginning with \( i = 1 \) and terminating at \( i = n \). The recursion at level \( i \) continues to level \( (i+1) \) only if the probability of the constructed \( i \)-round trail multiplied by the probability of the best found trail for \( (n-i) \) rounds is at least \( \mathcal{B}_n \) i.e. if \( p_1 p_2 \ldots p_i \hat{B}_{n-i} \geq \mathcal{B}_n \). For \( i = n \) the last equation is equivalent to: \( p_1 p_2 \ldots p_n = \hat{B}_n \geq \mathcal{B}_n \). If the latter holds, the initial estimate is updated with the new: \( \mathcal{B}_n \leftarrow \hat{B}_n \) and the corresponding trail is also updated accordingly: \( T_n \leftarrow \hat{T}_n \).

During the search process Algorithm 2 explores multiple differential trails. It is important to stress that the differentials that compose those trails are not restricted to the entries from the initial pDDT \( H \). The latter represent only the starting point of the first two rounds of the search, as in those rounds both the input and the output differences of the round transformation can be freely chosen (due to the specifics of the Feistel structure). From the third round onwards, excluding the last round, beside the

<table>
<thead>
<tr>
<th>( p_{\text{thres}} )</th>
<th>#elements in pDDT</th>
<th>Time</th>
<th>#elements in pDDT</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>252 940</td>
<td>36 sec.</td>
<td>3 951 388</td>
<td>1.23 min.</td>
</tr>
<tr>
<td>0.07</td>
<td>361 420</td>
<td>37 sec.</td>
<td>3 951 388</td>
<td>2.29 min.</td>
</tr>
<tr>
<td>0.05</td>
<td>3 038 668</td>
<td>5.35 min.</td>
<td>167 065 948</td>
<td>44.36 min.</td>
</tr>
<tr>
<td>0.01</td>
<td>2 715 532 204</td>
<td>17.46 hours</td>
<td>( \geq 72 589 325 174 )</td>
<td>( \geq 29 ) days</td>
</tr>
</tbody>
</table>

Table 3. Timings on the computation of pDDT for ADD and XOR on 32-bit words using Algorithm 1. Target machine: Intel® Core™ i7-2600, 3.40GHz CPU, 8GB RAM.
entries in $H$ the algorithm explores also an additional set of low-probability differentials stored in a temporary pDDT $C$ and sharing the same input difference.

**Algorithm 2** Matsui Search for Differential Trails Using pDDT (Threshold Search).

**Input:** $n$: number of rounds; $r$: current round; $H$: pDDT; $B = (B_1, B_2, \ldots, B_{n-1})$: probs. of best found trails for the first $(n-1)$ rounds; $\hat{B}_n \leq B_n$: initial estimate; $\hat{T} = (\hat{T}_1, \ldots, \hat{T}_n)$: trail for $n$ rounds with prob. $\hat{B}_n$; $p_{\text{thres}}$: probability threshold.

**Output:** $\hat{B}_n, \hat{T} = (\hat{T}_1, \ldots, \hat{T}_n)$: trail for $n$ rounds with prob. $\hat{B}_n \leq \hat{B}_n \leq B_n$.

1. **procedure threshold_search**($n, r, H, \hat{B}, \hat{B}_n, \hat{T}$) do
2. // Process rounds 1 and 2
3. if $((r = 1) \lor (r = 2)) \land (r \neq n)$ then
4. for all $(\alpha, \beta, p)$ in $H$ do
5. $p_r \leftarrow p$, $\hat{B}_n \leftarrow p_1 \cdots p_n \hat{B}_n$–$r$
6. if $\hat{B}_n \geq \hat{B}_n$ then
7. $\alpha_r \leftarrow \alpha$, $\beta_r \leftarrow \beta$, add $\hat{T}_r \leftarrow (\alpha_r, \beta_r, p_r)$ to $\hat{T}$
8. call **threshold_search**($n, r + 1, H, \hat{B}, \hat{B}_n, \hat{T}$)
9. // Process intermediate rounds
10. if $(r > 2) \land (r \neq n)$ then
11. $\alpha_r = \alpha_{r-2} + \beta_{r-1}$
12. $p_{r, \text{min}} = \hat{B}_n/(p_1 p_2 \cdots p_{r-1} \hat{B}_n)$
13. $C \leftarrow \emptyset$ // Initialize the country roads table
14. for all $\beta_r : (p_r(\alpha_r \rightarrow \beta_r) \geq p_{r, \text{min}}) \land ((\alpha_{r-1} + \beta_r) = \gamma \in H)$ do
15. add $(\alpha_r, \beta_r, p_r)$ to $C$ // Update country roads table
16. for all $(\alpha, \beta, p): \delta = \alpha_r \in H$ and all $(\alpha, \beta, p) \in C$ do
17. $p_r \leftarrow p$, $\hat{B}_n \leftarrow p_1 p_2 \cdots p_n \hat{B}_n$–$r$
18. if $\hat{B}_n \geq \hat{B}_n$ then
19. $\beta_r \leftarrow \beta$, add $\hat{T}_r \leftarrow (\alpha_r, \beta_r, p_r)$ to $\hat{T}$
20. call **threshold_search**($n, r + 1, H, \hat{B}, \hat{B}_n, \hat{T}$)
21. // Process last round
22. if $(r = n)$ then
23. $\alpha_r = \alpha_{r-2} + \beta_{r-1}$
24. if $(\alpha_r \in H)$ then
25. $(\beta_r, p_r) \leftarrow p_r = \max_{\beta \in H} p(\alpha_r \rightarrow \beta)$ // Select the max. from the highway table
26. else
27. $(\beta_r, p_r) \leftarrow p_r = \max_{\beta} p(\alpha_r \rightarrow \beta)$ // Compute the max.
28. if $p_r \geq p_{\text{thres}}$ then
29. add $(\alpha_r, \beta_r, p_r)$ to $H$
30. $p_n \leftarrow p_r$, $\hat{B}_n \leftarrow p_1 p_2 \cdots p_n$
31. if $\hat{B}_n \geq \hat{B}_n$ then
32. $\alpha_n \leftarrow \alpha_r$, $\beta_n \leftarrow \beta_r$, add $\hat{T}_n \leftarrow (\alpha_n, \beta_n, p_n)$ to $\hat{T}$
33. $\hat{B}_n \leftarrow \hat{B}_n$, $\hat{T} \leftarrow \hat{T}$
34. $\hat{B}_n \leftarrow \hat{B}_n$, $\hat{T} \leftarrow \hat{T}$ // Update the target bound and the best found trail
35. return $\hat{B}_n, \hat{T}$

The table $C$ is computed on demand for each input difference to an intermediate round (any round other than the first two and the last) encountered during the search. All entries in $C$ additionally satisfy the following two conditions: (1) Their probabilities are such that they can still improve the probability of the best found trail for the given number of rounds i.e. if $(\alpha_r, \beta_r, p_r)$ is an entry in $C$ for round $r$, then $p_r \geq \hat{B}_n/(p_1 p_2 \cdots p_{r-1} \hat{B}_n)$; (2) Their structure is such that they guarantee that the input difference for the next round $\alpha_{r+1} = \alpha_{r-1} + \beta_r$ will have a matching entry in $H$. While the need for
condition (1) is self-evident, condition (2) is necessary in order to prevent the exploding of the size of
C while at the same time keeping the probability of the resulting trail high. The meaning of the tables
\( H \) and \( C \) is further clarified with the following analogy.

**Example 1 (The Highways and Country Roads Analogy).** The two tables \( H \) and \( C \) employed in the
search performed by Algorithm 2 can be thought of as lists of highways and country roads on a road
map. The differentials contained in \( H \) have high probabilities w.r.t. to the fixed probability threshold
and correspond therefore to fast roads such as *highways*. Analogously, the differentials in \( C \) have low
probabilities and can be seen as slow roads or *country roads*. To continue this analogy, the problem of
finding a high probability differential trail for \( n \) rounds can be seen as a problem of finding a fast route
between points 1 and \( n \) on the map. Clearly such a route must be composed of as many highways as
possible. Condition (2), mentioned above, essentially guarantees that any country road that we may
take in our search for a fast route will bring us back on a highway. Note that it is possible that the
fastest route contains two or more country roads in sequence. While such a case will be missed by
Algorithm 2, it may be accounted for by lowering the initial probability threshold.

Algorithm 2 terminates when the initial estimate \( B_n \) can not be further improved. The complexity
of Algorithm 2 depends on the following factors: (1) the closeness of the best found probabilities
\( B_1, B_2, \ldots, B_{n-1} \) for the first \((n-1)\) rounds to the actual best probabilities, (2) the tightness of the
initial estimate \( B_n \) and (3) the number of elements \( m \) in \( H \). The latter is determined by the probability
threshold used to compute \( H \).

6 General Methodology for Automatic Search for Differential Trails in ARX

We describe a general methodology for the automatic search for differential trails in ARX algorithms.
In our analysis we restrict ourselves to Feistel ciphers, although the proposed method is applicable to
other ARX designs as well.

Let \( F \) be the round function (the F-function) of a Feistel cipher \( E \), designed by combining a
number of ARX operations, such as \( \text{XOR} \), \( \text{ADD} \), bit shift and bit rotation. To search for differential trails
for multiple rounds of \( E \) perform the following steps:

1. Derive an expression for computing the differential probability (DP) of \( F \) for given input and output
difference. The computation may be an approximation obtained as the multiplication of the DP of
the components of \( F \).
2. Compute a pDDT for \( F \). It can be an incomplete pDDT obtained e.g. by merging the separate
pDDT-s of the different components of \( F \).
3. Execute the threshold search algorithm described in Sect.5 with the (incomplete) pDDT computed
in Step. 2 as input.

In the following sections we apply the proposed methodology to automatically search for differential
trails in block ciphers TEA and XTEA.

7 Description of TEA and XTEA

The Tiny Encryption Algorithm (TEA) is a block cipher designed by Wheeler and Needham and
presented at FSE 1994 [41]. It has a Feistel structure composed of 64 rounds. Each round operates on
64-bit blocks divided into two 32-bit words \( L_i, R_i \): \( 0 \leq i \leq 64 \), so that \( P = L_0|R_0 \) is the plaintext and \( C = L_{64}|R_{64} \) is the ciphertext. TEA has 128-bit key \( K \) composed of four 32-bit words: \( K = K_3|K_2|K_1|K_0 \). The key schedule is such that the same two key words are used at every second round i.e. \( K_0, K_1 \) are used in all odd rounds and \( K_2, K_3 \) are used in all even rounds. Additionally, thirty-two 32-bit constants \( \delta_r \): \( 1 \leq r < 32 \) (the \( \delta \) constants) are defined. A different \( \delta \) constant is used at every second round. The round function \( F \) of TEA takes as input a 32-bit value \( x \), two 32-bit key words \( k_0, k_1 \) and a round constant \( \delta \) and produces a 32-bit output \( F(x) \). For fixed \( \delta, k_0 \) and \( k_1 \), \( F \) is defined as:

\[
(\delta, k_0, k_1) : F(x) = ((x \ll 4) + k_0) \oplus (x + \delta) \oplus ((x \gg 5) + k_1).
\] (8)

For fixed round keys \( K_j, K_{j+1} : j \in \{0, 2\} \) and round constant \( \delta_r \), round \( i \) of TEA \((1 \leq i < 64)\) is described as: \( L_{i+1} = R_i, R_{i+1} = L_i + F(R_i) \).

XTEA is an extended version of TEA proposed in [31] by the same designers. It was designed in order to address two weaknesses of TEA pointed by Kelsey et al. [18]: (1) a related-key attack on the full TEA and (2) the fact that the effective key size of TEA is 126, rather than 128 bits. The structure of XTEA is very similar to the one of TEA: 64-round Feistel network operating on 64-bit blocks using a 128-bit key. The main difference is in the key schedule: at every round XTEA uses one rather than two 32-bit key words from the original key according to a new non-periodic key schedule. Additionally, the number of \( \delta \) constants is increased from 32 to 64 and thus a different constant is used at every round. The \( F \)-function of XTEA is also slightly modified and for a fixed round key \( k \) and round constant \( \delta \) is defined as:

\[
(\delta, k) : F(x) = (\delta + k) \oplus (x + ((x \ll 4) \oplus (x \gg 5))).
\] (9)

The \( F \)-functions of TEA and XTEA are depicted in Fig. 1.

In Table 1 are listed the published differential attacks on TEA and XTEA ordered by the maximum number of rounds covered by the differential trail/s on which they are based. These results are compared to the maximum number of rounds covered by the best found trails using our method.

8 Automatic Search for Differential Trails in TEA and XTEA

We apply the steps described in Sect. 6 to search for differential trails for multiple rounds of block ciphers TEA and XTEA. Due to space limitations the detailed description of this process is provided in Appendix B (TEA) and Appendix C (XTEA).
We analyze TEA w.r.t. \( \text{ADD} \) differences and XTEA w.r.t. \( \text{XOR} \) differences. Additive differences are more appropriate for the differential analysis of TEA (as opposed to \( \text{XOR} \) differences) due to two reasons. First, the round keys and round constants are \( \text{ADD} \)-ed. Second, there are 4 \( \text{ADD} \) vs. 1 \( \text{XOR} \) operations in one round of TEA and so more components are linear w.r.t. \( \text{ADD} \) than to \( \text{XOR} \). Similarly, XTEA is more suitably analyzed with \( \text{XOR} \) differences since the round keys are \( \text{XOR} \)-ed.

In Table 4 (left) is shown the best found \( \text{ADD} \) differential trail for 18 rounds of TEA with probability \( 2^{-62.6} \) and on the right side is shown the best found \( \text{XOR} \) trail for 14 rounds of XTEA with probability \( 2^{-60.76} \) confirming a previous result by Hong et al. [17]. The top line of the table shows the fixed values of the keys for which the two trails were found and for which their probabilities were experimentally verified (the leftmost key word is \( K_0 \), the next one is \( K_1 \), etc.).

<table>
<thead>
<tr>
<th>key</th>
<th>11CAD84E 96168E6B 704A8B1C 57BB55D3</th>
<th>E15C938 DC8DBE76 B3BB0110 FFB0440</th>
</tr>
</thead>
<tbody>
<tr>
<td>( r )</td>
<td>( \beta )</td>
<td>( \alpha )</td>
</tr>
<tr>
<td>1</td>
<td>F</td>
<td>FFFFFFFF</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>F</td>
<td>FFFFFFFF</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>F</td>
</tr>
<tr>
<td>5</td>
<td>FFFFFFFF1</td>
<td>FFFFFFFF</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>FFFFFFFF1</td>
<td>FFFFFFFF</td>
</tr>
<tr>
<td>8</td>
<td>F</td>
<td>FFFFFFFF1</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>FFFFFFFF1</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>FFFFFFFF</td>
<td>FFFFFFFF</td>
</tr>
<tr>
<td>13</td>
<td>F</td>
<td>FFFFFFFF</td>
</tr>
<tr>
<td>14</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>FFFFFFFF</td>
<td>FFFFFFFF</td>
</tr>
<tr>
<td>16</td>
<td>0</td>
<td>11</td>
</tr>
<tr>
<td>17</td>
<td>FFFFFFFE</td>
<td>FFFFFFFF</td>
</tr>
<tr>
<td>18</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

\[ \prod \] Time: 21.36 min. 10 min.

The reason to perform the search for a fixed key rather than averaged over all keys as is usually done is the fact that for TEA the assumption of independent round keys, commonly made in differential cryptanalysis, does not hold. This is a consequence of the simple key schedule of the cipher according to which the same round keys are re-used every second round. Thus a trail that has very good probability when computed as an average over all keys, may in fact have zero probability for many or even all keys. This problem is further discussed in Sect. 9.3.

The mentioned effect is not so strong for XTEA due to the slightly more complex key schedule of the latter. In XTEA, the round keys are re-used according to a non-periodic schedule and, more importantly, a round constant that is different for every round, is added to the key before it is applied to the state (see
In this way the round keys are randomized in every round and thus the traditional differential analysis with probabilities computed as an average over all keys is more appropriate for XTEA.

The upshot of the above discussion is that while the 14 round trail for XTEA from Table 4 can directly be used in a key-recovery attack, as has indeed been already done in [17], it is not straightforward to do so for the 18 round trail for TEA. The reason is that this trail is valid only for a fraction of all keys. The size of this fraction is approx. 0.098% ≈ 0.1%, which is equal to $2^{116}$ weak keys (note that the effective key size of TEA is 126 bits [18]). The size of the weak key class was computed by observing that only the 9 LS bits of $K_2$ and the 3 LS bits of $K_3$ influence the probability of the trail. By fixing those 12 bits to the corresponding bits of the key values in Table 4 (resp. 11C and 3), we have experimentally verified that for any assignment of the remaining 116 bits of the key the 18 round trail has probability $\approx 2^{-63}$. Note that other assignments of the relevant 12 bits may also be possible and therefore the size of the weak key class may actually be bigger.

While the fixed-key trails for TEA found by the threshold search algorithm may have limited use for an attacker due to the reasons discussed above, they already provide very useful information for a designer. By running Algorithm 2 for many fixed keys we saw that the best found trails typically cover between 15 and 17 rounds and in more rare cases 18 rounds (see Appendix A.1). If this information has been available to the designers of TEA at the time of the design, they may have considered reducing the total number of rounds from 64 to 32 or less. Similarly, the threshold search algorithm can be used in order to estimate the security of new ARX designs and to help to select the appropriate number of rounds accordingly.

More differential trails for TEA covering between 15 and 17 rounds for six different keys chosen at random are shown in Appendix A.1. Comparisons between the trails found with the threshold search algorithm and the actual best trails on TEA with reduced word size of 11, 14 and 16 bits are shown in Appendix A.2.

9 Difficulties, Limitations and Common Problems

In this section we discuss the common problems and difficulties encountered when studying differential trails in ARX ciphers. This discussion is also naturally related to the limitations of the methodology proposed in Sect. 6. Although below we often use the TEA block cipher as an example, our observations are general and are therefore applicable to a broader class of ARX algorithms, such as for example the block ciphers Threefish [14] and HIGHT [16].

9.1 Accuracy of the Approximation of the DP of F

The first step in the methodology presented in Sect. 6 is to derive an expression for computing the DP of the F-function of the target cipher. Since it is often difficult to efficiently compute the exact probability (see Appendix B.1 for more details), this expression would usually be an approximation obtained as the multiplication of the DP of the separate components of F. The probability computed in this way will often deviate from the actual value. A common reason for the deviation is the dependency between the inputs to the components of F. Indeed, this phenomenon is well-known and has been studied before e.g. in [38].

To illustrate the mentioned problem, consider the three inputs to the XOR operation in the TEA F-function. These are $(k_0 + (x \ll 4)), (\delta + x)$, and $(k_1 + (x \gg 5))$ (see Fig. 1) and since they are obtained from the same initial input $x$, they are clearly not independent. Yet, when we compute the
DP of the XOR with three inputs \((\text{adp}^{3\oplus})\) we implicitly assume independence of the inputs. Consequently the resulting approximation will not be an accurate estimation of the DP of \(F\) \((\text{eadp}^F)\) for all input and output differences \(\alpha, \beta\). This problem can be addressed to an extent by additionally adjusting the approximate probability to the value of a (set of) fixed key/s. More details on this as well as on the computation of \(\text{adp}^{3\oplus}\) and \(\text{eadp}^F\) are provided in Appendix B.

9.2 Dependency of the DP of \(F\) on the Round Keys

Another difficulty arises from the fact that in some cases the DP of the \(F\)-function is dependent on the value of the round key(s). Ciphers for which this is the case are not key-alternating ciphers (cf. [11, Definition 2]) and are typically harder to analyze. As noted in [10, § 5.7] an important advantage of key-alternating ciphers is that the study of their differential and linear trails can be conducted independently of the value of the round key. In contrast, for a non-key-alternating cipher, a trail that has high probability for one key may have a significantly lower (even zero) probability for another. Such ciphers violate the hypothesis of stochastic equivalence, according to which: for virtually all values of the cipher key, the probability of a differential trail can be approximated by the expected value of the probability of the differential trail, averaged over all possible values of the cipher key [10, § 8.7.2].

The block cipher TEA is an example of a non-key-alternating cipher. The DP of its \(F\)-function is key-dependent w.r.t. both XOR and ADD differences. This behavior is particularly counter-intuitive in the case of ADD differences, in view of the fact that the round keys are ADD-ed (rather than XOR-ed) to the state and hence one wouldn’t expect them to influence the additive DP of \(F\). In practice it does and the explanation is provided below.

In a differential cryptanalysis setting, the LSH (resp. RSH) operation in TEA reduces the number of possible input pairs to the modular addition with key \(k_0\) (resp. key \(k_1\)) from \(2^n\) to \(2^n-4\) (resp. \(2^n-5\)). When computing the DP of the subsequent XOR with three inputs \((\text{adp}^{3\oplus})\), the number of right pairs for two of the inputs are counted over the reduced sets while the inputs still have size \(n\) bits due to the addition with the \(n\)-bit key word. As a result the reduced sets contain different elements for different values of the keys, which causes the DP of \(F\) to be key-dependent. Note that this is not the case when the two sets have full size \(2^n\) (e.g. if the shift constants of LSH and RSH are set to 0 or if the shifts are replaced by bit rotations). In that case the right pairs for the XOR would always be counted over the same set of \(2^n\) pairs (possibly re-ordered) irrespective of the actual value of the key. More details on this effect illustrated by an example are provided in Appendix D.2.

A solution to the problem of key-dependency of the DP of the \(F\)-function is to search for differential trails with probabilities computed for several fixed keys rather than for trails with probabilities averaged over all keys. As discussed in Sect. 8, this is the approach that we took in the analysis of TEA (see also Appendix B.3).

9.3 Dependency Between the Round Keys

In differential cryptanalysis of keyed primitives it is common practice to assume that the round keys are independent [20]. This is known as making the hypothesis of independent round keys [11]. Citing from [10, § 8.7.2]: the hypothesis states that the expected probability of a differential trail, averaged over all possible values of the cipher key, can be approximated by the expected probability of the differential trail, averaged over all independently specified round key values.

In ciphers with weak key schedule, the hypothesis of independent round keys does not hold. As a consequence, obtaining an accurate estimation of the expected probabilities of differential trails in such
ciphers is difficult. As discussed in Sect. 8, TEA is an extreme example of a block cipher in which the round keys are strongly dependent. Indeed, the keys to every second round are, in fact, identical.

A possible solution to the dependent round keys problem is to analyze the cipher with respect to a set of randomly chosen fixed keys. The analysis in this case will be based on the minimum probability, among all keys within the set rather than on the expected probabilities averaged over all keys. The reason to select the minimum rather than the average probability is in order to guarantee that the resulting differential trail is possible (i.e. has non-zero probability) for every key in the set. More details on this are provided in Appendix B.3.

9.4 Influence of the Round Constants

We make a final remark concerning the influence of fixed round constants in ARX algorithms on the structure and probability of differential trails. Fixed constants are commonly used in the design of symmetric-key primitives in order to destroy similarities between the rounds. Since they are typically added to the state by applying the same operation as for the round keys, in differential cryptanalysis it is generally assumed that constants influence neither the probabilities nor the structure of differential trails and hence can be safely ignored. Surprisingly, this assumption does not hold for TEA and possibly for other ARX constructions as well.

The fixed round constants of TEA (the $\delta$ constants) influence both the probabilities and the structure of differential trails. They influence the probabilities as an indirect consequence of the key-dependency effect discussed in Sect. 9.3. On the one hand, as noted, the DP of $F$ depends on the value of the two round keys added resp. to two of the three inputs of the $\oplus$ (cf. Fig. 1 (left)). On the other, the third input, to which a $\delta$ constant is added, is dependent on the other two inputs (since all three are produced from the same initial input $x$) and hence indirectly also influences the DP of $F$.

As to the influence of round constants on the structure of differential trails, after modifying TEA to use the same $\delta$ constant at every round, for many keys the best found trail after several rounds eventually becomes iterative with period 2 and of the form $(\alpha \to 0), (0 \to 0), (\alpha \to 0), \ldots$. The difference that maximizes the probability of the differential $(\alpha \to 0)$ is $\alpha = 0xF$ and has probability $2^{-8}$ for exactly $6 \cdot 2^{59} \approx 2^{61.6}$ keys (approx. 10% of all keys). We use the two-round iterative trail $(0xF \to 0), (0 \to 0)$ to construct a trail over 15 rounds with probability $2^{-56}$. We also find a 4-round iterative pattern with probability $< 2^{-15}$ which holds for a smaller number of keys. It is used to construct a trail with probability $2^{-61.36}$ on 18 rounds of the modified TEA. For more detailed discussion of these results ref. Appendix D.3.

10 Conclusions and Future Work

In this paper we proposed the first extension of Matsui’s algorithm for automatic search for differential trails, originally proposed for S-box based ciphers, to the class of ARX ciphers. We used the block ciphers TEA and XTEA as a testbed for demonstrating the practical application of this method. Using the proposed algorithm we were able to find the first full differential trails on TEA. The best one covers 18 rounds which is one round more than the best differential attack on TEA (17 rounds) and significantly improves the best previously known truncated trail which is on 8 rounds. The reported trail holds for a fraction of 0.1% of all keys with prob. $\approx 2^{-63}$. For XTEA, our algorithm confirmed the best previously known differential trail on 14 rounds. The threshold search algorithm can be used by both cryptanalysts for attacking ARX-based ciphers and designers for estimating the strength of their designs and for choosing the appropriate number of rounds for their ciphers accordingly.
A natural problem for future work would be to investigate whether the reported 18 round trail on TEA can be used in an attack scenario. Attacks in a similar setting have previously been reported on the block ciphers Lucifer [3] and IDEA [6]. The applicability of these techniques to TEA has to be further examined.

Acknowledgments. We thank our colleagues from LACS for the useful discussions, and especially Yann Le Corre for his help with visualizing the experimental data. Some of the experiments presented in this paper were carried out using the HPC facility of the University of Luxembourg.

References
A More Experimental results

A.1 More Differential Trails for TEA

More differential trails for TEA for six different keys chosen at random are shown in Table 5.

A.2 Threshold Search on TEA with Reduced Word Size

In Fig. 2, Fig. 3 and Fig. 4 are compared the probabilities of the best trails found by the threshold search algorithm using pDDT to the actual best trails found by applying Matsui’s search using full DDT on TEA with word size reduced to 11, 14 and 16 bits respectively. For 11 and 14 bits 50 experiments are performed and in each experiment a new fixed key is chosen uniformly at random. For 16 bits, the number of experiments is 20. In the experiments on 14 and 16 bits the same $\delta$ constant (equal to the initial value) was used in every round. The reason is that if different constants are used, then a separate
Table 5. TEA: ADD trails for six different keys (shown between bold lines) chose n at random; \( p_{\text{thres}} = 0.05 = 2^{-4.32} \).

<table>
<thead>
<tr>
<th>( r )</th>
<th>( \beta )</th>
<th>( \log_{2} p )</th>
<th>( \beta )</th>
<th>( \log_{2} p )</th>
<th>( \beta )</th>
<th>( \log_{2} p )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>FFFFFF1</td>
<td>1</td>
<td>2^{-2.94}</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>2^{-0.00}</td>
<td>2</td>
<td>FFFFFE</td>
<td>FFFFFF</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>2^{-7.89}</td>
<td>3</td>
<td>0</td>
<td>FFFFFF</td>
</tr>
<tr>
<td>4</td>
<td>FFFFFF</td>
<td>1</td>
<td>2^{-8.06}</td>
<td>4</td>
<td>1</td>
<td>FFFFFF</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>2^{-0.00}</td>
<td>5</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>F</td>
<td>1</td>
<td>2^{-2.88}</td>
<td>6</td>
<td>F</td>
<td>FFFFFF</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>F</td>
<td>2^{-9.07}</td>
<td>7</td>
<td>2</td>
<td>F</td>
</tr>
<tr>
<td>8</td>
<td>FFFFFF1</td>
<td>1</td>
<td>2^{-3.64}</td>
<td>8</td>
<td>FFFFFF1</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>0</td>
<td>2^{-0.00}</td>
<td>9</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>F</td>
<td>1</td>
<td>2^{-2.95}</td>
<td>10</td>
<td>F</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>F</td>
<td>2^{-7.88}</td>
<td>11</td>
<td>FFFFFF</td>
<td>F</td>
</tr>
<tr>
<td>12</td>
<td>FFFFFF</td>
<td>1</td>
<td>2^{-3.64}</td>
<td>12</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td>0</td>
<td>0</td>
<td>2^{-0.00}</td>
<td>13</td>
<td>FFFFFF</td>
<td>F</td>
</tr>
<tr>
<td>14</td>
<td>F</td>
<td>1</td>
<td>2^{-2.88}</td>
<td>14</td>
<td>FFFFFF1</td>
<td>FFFFFF</td>
</tr>
<tr>
<td>15</td>
<td>FFFFFF</td>
<td>F</td>
<td>2^{-8.03}</td>
<td>15</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>0</td>
<td>0</td>
<td>2^{-0.00}</td>
<td>16</td>
<td>FFFFFF1</td>
<td>FFFFFF</td>
</tr>
</tbody>
</table>

\( \Pi_1 \) 2^{-59.96} 2^{-59.6} 2^{-57.21} 2^{-59.6} 2^{-8.94} 2^{-8.94} 2^{-57.21} 8.42 min. 8 min. 6.54 min.

BC3FDB33 D7A0E8 473657E C29B8AF9 | E028DF9A 8819R4C3 3AB116AF | 3CS0723 | 37527D25 EBB25BDA CD7CADB9 1768A2BB

Time: 17.9 min. 6.44 min. 5.33 min.
DDT has to be computed for every round, which for 10 rounds and 14-bit words is infeasible. Also note that for 16 bits it takes longer to compute the full DDT-s due to their larger size (compared to the 14 bit case). The memory consumption is also much bigger – 320 GB of RAM are required to store all DDT-s. Due to the mentioned limitations, less number of experiments on 16 bits were performed.

![Graph](image)

**Fig. 2.** Threshold Search vs. DDT Search: word size $n = 11$ bits.

## B Automatic Search for Differential Trails in TEA

In this section are provided more details on the application of the general methodology outlined in Sect. 6 to the automatic search for high probability differential trails in TEA.

### B.1 An Approximation for the DP of $F$

Computing the exact additive DP of $F$ for fixed round keys and round constant (step 1 in the general methodology from Sect. 6) is very inefficient (ref. Appendix D.1 for details). Therefore, as an approximation, we use the expected DP averaged over all round keys and $\delta$ constants. It is defined as:

$$\text{eadp}^F(\alpha \rightarrow \beta) \triangleq 2^{-4n} \cdot \#\{(k_0, k_1, \delta, x) : F(x + \alpha) - F(x) = \gamma\}.$$  

The probability $\text{eadp}^F$ (10) can be efficiently computed as follows. Note that the only components of $F$ that are non-linear w.r.t. ADD differences are the operations $\text{RSH}$ and $\text{XOR}$ (see Fig. 1 (left)). Therefore, for fixed input and output differences $\alpha$ and $\beta$ resp., the expected DP of $F$ can be expressed as the
Fig. 3. Threshold Search vs. DDT Search: word size $n = 14$ bits; same $\delta$ is used in every round.

Fig. 4. Threshold Search vs. DDT Search: word size $n = 16$ bits; same $\delta$ is used in every round.
multiplication of the DP of RSH and XOR:

\[
\text{eadp}^F(\alpha \to \beta) = \sum_{i=0}^{3} \left( \text{adp}^{\gg 5}(\alpha \to \gamma_i) \cdot \text{adp}^{\oplus 3}((\alpha \ll 4), \alpha, \gamma_i \to \beta) \right),
\]  

(11)

where \(\gamma_0, \gamma_1, \gamma_2\) and \(\gamma_3\) are the four output differences after the RSH operation (cf. Theorem 2 (5)) and \(\text{adp}^{\oplus 3}\) denotes the additive DP of XOR with three inputs, defined analogously to (2). Equation (11) can be efficiently computed using (5) to compute \(\text{adp}^{\gg 5}\) and the methods described in [28] to compute \(\text{adp}^{\oplus 3}\).

B.2 Computing a pDDT for F

In this section we describe the process of constructing a pDDT based on the expected differential probability of the F-function of TEA (cf. (10)). This is step 2 in the general methodology from Sect. 6. We begin by observing that to construct a pDDT for F, it is sufficient to construct a pDDT for the operation XOR with three inputs.

**Theorem 3.** Let \(\alpha\) and \(\beta\) be respectively input and output ADD differences to F and let \(\gamma_0, \gamma_1, \gamma_2\) and \(\gamma_3\) be the four output differences after the RSH operation (cf. (11)). Let \(p_{\text{thres}} \geq 0\) be a fixed probability threshold. Then

\[
\forall i : 0 \leq i < 4 : \text{adp}^{\oplus 3}((\alpha \ll 4), \alpha, \gamma_i \to \beta) \geq p_{\text{thres}} \implies \text{eadp}^F(\alpha \to \beta) \geq p_{\text{thres}},
\]  

(12)

and

\[
\text{eadp}^F(\alpha \to \beta) \geq p_{\text{thres}} \implies \exists i : 0 \leq i < 4 : \text{adp}^{\oplus 3}((\alpha \ll 4), \alpha, \gamma_i \to \beta) \geq p_{\text{thres}}.
\]  

(13)

**Proof.** Appendix E.4.

Theorem 3 implies that for a fixed threshold \(p_{\text{thres}}\), every entry in the pDDT of XOR with three inputs can be transformed into an entry in the pDDT of F (by multiplying its probability by the probabilities of RSH (11)). Therefore Algorithm 1 can be used to construct a pDDT for F. Note that applying the algorithm directly is very inefficient because for XOR with three inputs the recursion proceeds over four, rather than three, \(n\)-bit words. Thus for \(n = 32\) and \(p_{\text{thres}} \leq 0.01\) the computation becomes quickly infeasible (cf. Table 3). However, the fact that the three inputs to the XOR operation in F are strongly dependent can be used to improve the efficiency. The improvement is at the expense of a small fraction of differentials that are missing from the pDDT, which results in an incomplete pDDT. More details on this process are provided in Appendix D.4.

B.3 Search for Differential Trails

After constructing a pDDT for F as described in Sect. B.2, it is straightforward to apply Algorithm 2 to search for differential trails in TEA. Before doing so, however, one final detail has to be addressed. Note that since in TEA the same key is used every second round, in our analysis we can not assume that the round keys are independent. Therefore the expected probability of a differential trail can not be accurately estimated by multiplying the expected differential probabilities over all keys at every round computed using (11). Indeed, if we do so, we risk ending up with a differential trail that has high probability according to our estimation, but which is in fact impossible for most or all keys.
Unfortunately this is exactly what will happen if Algorithm 2 (cf. Sect. 5) is directly applied with the pDDT of F.

To address the problem mentioned above, we modify Algorithm 2 to perform the search for a (set of) fixed key/s, by adding one additional input – the value of the key/s. Whenever a differential is pulled from the pDDT H, it’s probability is experimentally verified against the specific value of the (set of) round key/s by performing one-round encryptions over $2^\epsilon$ chosen plaintexts satisfying the specific difference, where $\epsilon = c + \log_2(1/p_{\text{thres}})$ for some constant $c \geq 2$. For example, for $p_{\text{thres}} = 0.01 > 2^{-7}$, a suitable choice would be $c = 3$ so that $\epsilon = 3 + 7 = 10$ and $2^\epsilon = 2^{10}$ chosen plaintexts. For a set of more than one key, the minimum of the probabilities among all keys is computed. The reason to take the minimum rather than e.g. the average probability is to guarantee that the resulting trail is possible (i.e. has non-zero probability) for every key in the set.

C Automatic Search for Differential Trails in XTEA

In this section we apply the methodology described in Sect. 6 to search for XOR and ADD differential trails in XTEA.

C.1 XOR Differences

Let $x_{-1}$ be the input to $F$ from the previous round. Then define $F'$ as:

$$(\delta, k) : F'(x_{-1}, x) = x_{-1} + F(x).$$

(14)

An Approximation for the DP of $F'$. The DP of $F'$ is approximated as:

$$\text{xdp}^F(\alpha \rightarrow \beta) \approx \text{xdp}^+(\langle \alpha \ll 4 \rangle \oplus \langle \alpha \gg 5 \rangle, \alpha \rightarrow \tau) \cdot \text{xdp}^+(\tau, \alpha_{-1} \rightarrow \beta).$$

(15)

Computing a pDDT for $F'$.

1. Initialize an empty pDDT for $F'$: $H' \leftarrow \emptyset$.
2. For a fixed threshold $p_{\text{thres}}$ construct an incomplete pDDT $H$ for the first ADD operation in $F'$.
3. Denote $p' = \text{xdp}^+(\langle \alpha \ll 4 \rangle \oplus \langle \alpha \gg 5 \rangle, \alpha \rightarrow \tau)$. For each entry $(\alpha, (\langle \alpha \ll 4 \rangle \oplus \langle \alpha \gg 5 \rangle), \tau, p')$ in $H$ compute $p_{F'} = p' \cdot \max_{\beta} \text{xdp}^+(\tau, \alpha_{-1} \rightarrow \beta)$ (cf. (15)).
4. If $p_{F'} \geq p_{\text{thres}}$ add $(\alpha, \beta, p_{F'})$ to $H'$.
5. Return $H'$.

Search for Differential Trails. Apply Algorithm 2 with the incomplete pDDT $H'$ to search for XOR differential trails in XTEA.
C.2 ADD Differences

An Approximation for the DP of $F$. Define $f(x) = (x \ll 4) \oplus (x \gg 5)$. An approximation of the DP of $f$ is:

$$\text{adp}^f(\alpha \rightarrow \tau) \approx \sum_{i=0}^{3} (\text{adp}^{\gg 5}(\alpha \rightarrow \gamma_i) \cdot \text{adp}^{\oplus}(\gamma_i, (\alpha \ll 4) \rightarrow \tau)) \quad , \quad (16)$$

and we get the following approximation of the DP of $F$:

$$\text{adp}^F(\alpha \rightarrow \beta) \approx \text{adp}^f(\alpha \rightarrow \tau) \cdot \text{adp}^{\oplus}((\tau + \alpha), 0 \rightarrow \beta) \quad , \quad (17)$$

Computing a pDDT for $F$.

1. Initialize an empty pDDT for $F$: $H \leftarrow \emptyset$.
2. For a fixed threshold $p_{\text{thres}}$ construct an incomplete pDDT $H_f$ for $f$. This process is conceptually the same as the one described in Sect. B.2 for the F-function of TEA.
3. Denote $p_f = \text{adp}^f(\alpha \rightarrow \tau)$. For each entry $(\alpha, \tau, p_f)$ in $H_f$ compute $p_F = p_f \cdot \max_{\beta} \text{adp}^{\oplus}((\tau + \alpha), 0 \rightarrow \beta)$ (cf. (17)).
4. If $p_F \geq p_{\text{thres}}$ add $(\alpha, \beta, p_F)$ to $H$.
5. Return $H$.

Search for Differential Trails. Apply Algorithm 2 with the incomplete pDDT $H$ to search for ADD differential trails in XTEA.

D More Details on the Differential Properties of the TEA F-function

D.1 Computation of the Fixed-key DP of $F$

In this section, an algorithm for the computation of the fixed-key DP of the TEA F-function is presented. For fixed keys $k_0, k_1$, round constant $\delta$ and input and output ADD differences resp. $\alpha$ and $\beta$, the additive DP of the TEA F-function is defined over all $n$-bit inputs $x$ as follows:

$$\text{adp}_{(k_0,k_1,\delta)}^F(k_0, k_1, \delta : \alpha \rightarrow \beta) \triangleq 2^{-n} \cdot \# \{ x : F(x + \alpha) - F(x) = \gamma \} \quad , \quad (18)$$

where $F$ is defined as in (8). Note that expression (18) is related to the expected DP of $F$ (10) as:

$$\text{eadp}^F = 2^{-3n} \sum_{k_0,k_1,\delta} \text{adp}_{(k_0,k_1,\delta)}^F \quad . \quad (19)$$

From (18) it follows that computing the fixed-key probability $\text{adp}_{(k_0,k_1,\delta)}^F$ is equivalent to counting the number of values $x$ for which $y' - y = \beta$, where $y = F(x)$ and $y' = F(x + \alpha)$. We have designed an algorithm that performs this count in a bitwise manner. At every bit position $i$, a value for bit $x[i]$ is assigned and next it is checked whether $y'[i : 0] - y[i : 0] = \beta[i : 0]$ holds modulo $2^{i+1}$. If it does then the algorithm recursively proceeds to bit position $(i + 1)$, otherwise it backtracks and assigns another value to $x[i]$. When all bits of $x$ have been successfully assigned, a counter $c$ is incremented. This process is described in more detail next.
For $0 \leq i < n$, the $(i + 1)$-bit words $y[i : 0]$ and $y'[i : 0]$ are computed as follows:

$$y[i : 0] = ((x[i - 4 : 0]) + k_0[i : 0]) \oplus$$

$$(x[i : 0] + \delta[i : 0]) \oplus$$

$$(x[i + 5 : 5] + k_1[i : 0]),$$

$$y'[i : 0] = (((x + \alpha)[i - 4 : 0] + k_0[i : 0]) \oplus$$

$$((x + \alpha)[i : 0] + \delta[i : 0]) \oplus$$

$$((x + \alpha)[i + 5 : 5] + k_1[i : 0]),$$

where $x[i - 4] = 0 : i < 4$ and $x[i + 5] = 0 : i > ((n - 1) - 5))$. Notice that for $i = 3$, the bits of $x$ that participate in (20)–(25), namely $x[3 : 0]$ and $x[8 : 5]$, are non-overlapping and hence can be assigned independently. At bit position $i = 4$, the first dependency occurs: bit $x[0]$ which has already been assigned and used in the computation of (21) and (24) for $i < 4$, at position $i = 4$ is again used in the computation of (20) and (23). To take this dependency into account the algorithm initially assigns the first 10 bits of $x$ i.e. $x[9 : 0]$. Then it checks if $(y'[4 : 0] - y[4 : 0] = \beta[4 : 0]) \mod 2^5$. Indeed this check is possible for $i = 4$, because bit $x[0]$ needed to compute (20) and (23) is known; bits $x[4 : 0]$ needed to compute (21) and (24) are known; and bits $x[9 : 5]$ needed to compute (22) and (25) are also known. If the check succeeds, the algorithm proceeds by recursively assigning the remaining bits of $x$ bit by bit.

The recursion starts at level $i = 10$ where bit $x[10]$ is assigned and it is checked if $(y'[5 : 0] - y[5 : 0] = \beta[5 : 0]) \mod 2^6$ is consistent. If yes, bit $x[11]$ is assigned and it is checked if $(y'[6 : 0] - y[6 : 0] = \beta[6 : 0]) \mod 2^7$, etc. In general, when $x[i]$ is assigned, equation $(y'[i - 5 : 0] - y[i - 5 : 0] = \beta[i - 5 : 0]) \mod 2^{(i-5)+1}$ is evaluated. This process is repeated until the recursion reaches level $i = n - 1$. At this level, bits $x[n - 1]$ and $x[n - 1]$ are assigned and equation $(y'[n - 6 : 0] - y[n - 6 : 0] = \beta[(n - 6 : 0)] \mod 2^{n-5}$ is checked for consistency. Note that at this point all bits of $x$ have been assigned (x is an $n$-bit word), but only the $(n - 5)$ LS bits have been checked for consistency. Therefore from level $i = n - 1$ up to $i = (n - 2) + 5$ the recursion proceeds without assigning new values at the $i$-th bit positions of $x$ and only checking the consistency of equation $(y'[i - 5 : 0] - y[i - 5 : 0] = \beta[i - 5 : 0]) \mod 2^{i-4}$. If level $i = n + 4$ has been successfully reached then it means that the constructed $x$ is such that $(y' - y = \beta) \mod 2^n$ and a counter $c$ is incremented. A pseudo-code of the described procedure is listed in Algorithm 3.

Clearly Algorithm 3 is worst-case exponential since for a differential that has probability 1 all branches of the recursion will be explored. In general, the efficiency of the algorithm is inversely proportional to the probability of the differential ($\alpha \rightarrow \beta$). The reason is that low-probability differentials correspond to small number of values $x$ that satisfy them, which in turn means that for many assignments of $x[i]$ equation $(y'[i : 0] - y[i : 0] = \beta[i : 0]) \mod 2^{i+1}$ will be inconsistent. As a result the total number of recursive calls will be smaller.

### D.2 Key-dependency of the DP of F

As noted in Sect. 9, the DP of the F-function of TEA is key-dependent w.r.t. both XOR and ADD. As the round keys are added to the state by means of the ADD operation, for XOR differences the mentioned effect is not surprising. This is not so for ADD differences though. The effect for ADD differences is illustrated in Fig. 5, where for a small-scale version of TEA with 7-bit word size for every fixed input difference
**Algorithm 3** Computation of the Fixed-key DP of the TEA F-function.

**Input:** \( n, \alpha, \beta, k_0, k_1, \delta, c, x \)

**Output:** \( \text{adp}_F^c(k_0, k_1, \delta : \alpha \rightarrow \beta) \)

1: procedure assign\_bit\( (n, i, c, x) \) do
2: if \( i = n + 4 \) then
3: \( c = c + 1 \)
4: return
5: Compute \( y[i-5:0] \) and \( y'[i-5:0] \) according to eq. (20)–(25)
6: if \( (y'[i-5:0] - y[i-5:0]) = \beta[i-5:0] \mod 2^{i-4} \) then
7: if \( i < (n-1) \) then
8: for \( q \in \{0, 1\} \) do
9: \( x[i+1:0] \leftarrow q \cdot x[i:0] \)
10: assign\_bit\( (n, i+1, c, x) \)
11: else
12: assign\_bit\( (n, i+1, c, x) \)
13: return \( c \)
14: procedure adp\_tea\_f\( (n, k_0, k_1, \delta, \alpha, \beta) \) do
15: \( c \leftarrow 0 \)
16: for \( x[9:0] \in \{0, \ldots, 2^{10} - 1\} \) do
17: \( i \leftarrow 10 \)
18: \( c = c + \text{assign\_bit}\( (n, i, c, x) \) \)
19: return \( p \leftarrow c \cdot 2^{-n} \)

![TEA F Key Dependence](image)

**Fig. 5.** TEA, \( n = 7 \) bits: dependency of the maximum probability output ADD difference on the value of the round key.
(the X axis) is plotted the variation of the maximum (over all output differences) probabilities for all round keys (the Y axis). We analyze this key-dependency effect in more detail below.

The first thing to observe is that, although ADD differences propagate with probability 1 through the three modular additions in the TEA F-function (see Fig. 1) independently of the actual value of the round keys and the round constant, the probability of ADD differentials through F is actually highly dependent on the values of the keys and the constant. The reason for this counter-intuitive behavior is the following.

In a differential cryptanalysis setting, consider an input pair \((x, x + \alpha)\) to \(F\). For a fixed \(n\)-bit difference \(\alpha\), there are \(2^n\) such pairs that satisfy the difference. Observe that the LSH operation reduces the set of possible input pairs to the modular addition with key \(k_0\) (cf. Fig. 1 (left)) from \(2^n\) to \(2^{n-4}\). Similarly the RSH operation reduces the set of possible input pairs to the modular addition with key \(k_1\) from \(2^n\) to \(2^{n-5}\). Although the key additions will not affect the differences that the pairs in each set satisfy, they will affect the actual values of those pairs which on its part influences the DP of \(F\). The latter is due to the way the DP of the subsequent XOR with three inputs \(\text{adp}^{3\oplus}\) is computed. To compute \(\text{adp}^{3\oplus}\), the number of the right pairs for two of the inputs are counted over the reduced sets of \(2^{n-4}\) and \(2^{n-5}\) elements, while still the elements of those sets are of size \(n\) bits, because of the addition with the \(n\)-bit key word. As a result the reduced sets contain different elements for different values of the keys and so the probability \(\text{adp}^{3\oplus}\) and, by implication, the DP of \(F\) will also be different for different keys. Note that this is not the case when the two sets have full size \(2^n\) (e.g. if the shift constants of LSH and RSH are set to 0). In that case the right pairs for the XOR will always be counted over the same set of \(2^n\) pairs (possibly re-ordered) irrespective of the actual value of the key. This key dependency effect is illustrated in detail by means of the following simple example.

**Example 2.** In order to demonstrate the dependency of the DP of the TEA F-function on the value of the round key, we isolate a small sub-component \(f\) of \(F\) composed of an RSH, key addition and XOR. Let \(x\) and \(y\) be \(n\)-bit inputs, \(k\) be an \(n\)-bit key and \(r\) be a shift constant. Then define

\[
f(k, x, y) = (k + (x \gg r)) \oplus y .
\]

Note that in contrast to \(F\) (8), in \(f\) the inputs \(x\) and \(y\) are independent. We define \(f\) in this way on purpose, so that in the following analysis we can focus exclusively on the influence of the key \(k\), and not be distracted with side effects arising form dependencies between the inputs.

For \(n = 3\) and \(r = 1\), let \(\alpha = 2\), \(\beta = 2\) be fixed input differences, and let \(\gamma = 1\) be fixed output difference. We shall compute the DP of \(f\): \(\text{adp}^{f}(k : \alpha, \beta \rightarrow \gamma)\) for two values of the key: \(k = 0\) and \(k = 1\). We denote the two probabilities as \(p_0 = \text{adp}^{f}(k = 0 : 2, 2 \rightarrow 1)\) and \(p_1 = \text{adp}^{f}(k = 1 : 2, 2 \rightarrow 1)\). Note that by setting \(k = 0\) we effectively discard the key addition operation. Intuitively this should be the same as the case \(k \neq 0\) since ADD differences propagate with probability 1 through addition. With the following example we demonstrate that this intuition is wrong.

Let \(A\) and \(B\) be the sets of pairs that satisfy the inputs differences \(\alpha\) and \(\beta\) respectively. Denote with \(\mathcal{C}\) the set of output pairs after the RSH operation, and with \(\mathcal{D}_k\) the set of output pairs after the key addition with key \(k\). Note that \(\mathcal{D}_k = k + \mathcal{C}\). Finally, let \(\mathcal{E}_k\) be the set of pairs in \(\mathcal{D}_k\) such that when combined element-wise with a pair from \(B\) through XOR, the result is a pair that satisfies the output difference \(\gamma\):

\[
\mathcal{E}_k = \{D_k \times B : (d, d') \in D_k, (b, b') \in B : ((d \oplus b) - (d' \oplus b')) \mod 2^n = \gamma\} .
\]
The probability \(\text{adp}^f\) can then be expressed as:

\[
p_k = 2^{-n} \cdot \#E_k .
\]  

(28)

Since \(\alpha = \beta = 2\),

\[
A = B = \{(2, 0), (3, 1), (4, 2), (5, 3), (6, 4), (7, 5), (0, 6), (1, 7)\} .
\]  

(29)

The output set after the RSH by 1 is

\[
C = \{(1, 0), (1, 0), (2, 1), (2, 1), (3, 2), (3, 2), (0, 3), (0, 3)\} .
\]  

(30)

As expected, the number of unique elements in \(C\) compared to \(A\) is reduced by 2 due to the shift by 1 position. Note that all analysis up to this point is independent of the value of the key \(k\). We shall now demonstrate how the choice of the key influences the probability.

1. Let \(k = 0\). Then \(D_0 = C\):

\[
D_0 = \{(1, 0), (1, 0), (2, 1), (2, 1), (3, 2), (3, 2), (0, 3), (0, 3)\} ,
\]  

(31)

and \(E_0\) is

\[
E_0 = \{(1, 0), (1, 0)\} \times \{(3, 1), (5, 3), (7, 5), (1, 7)\},
\{
(2, 1), (2, 1)\} \times \{(3, 1), (7, 5)\},
\{(0, 3), (0, 3)\} \times \{(3, 1), (7, 5)\} .
\]  

(32)

Since there are 16 elements in \(E_0\), according to (28) \(p_0 = \frac{16}{64} = 0.25\).

2. Let \(k = 1\). In this case \(D_1 = 1 + C\):

\[
D_1 = \{(2, 1), (2, 1), (3, 2), (3, 2), (4, 3), (4, 3), (1, 4), (1, 4)\} .
\]  

(33)

Note that due to the key addition, two of the pairs in \(D_1\) do not belong to \(D_0\): \((1, 4), (4, 3)\), while the remaining two: \((2, 1), (3, 2)\) belong to both sets. As a result the set \(E_1\) also changes with respect to \(E_0\):

\[
E_1 = \{(2, 1), (2, 1)\} \times \{(3, 1), (7, 5)\},
\{(4, 3), (4, 3)\} \times \{(5, 3), (1, 7)\} .
\]  

(34)

There are 8 elements in \(E_1\) and so \(p_1 = \frac{8}{64} = 0.125 < p_0\).

From the above example we can see that the value of the key \(k\) determines the exact way in which the set \(C\) will be mapped to \(D_k\). This in turn influences the size of the set \(D_k \times B\) that ultimately determines the final probability \(\text{adp}^f\).

Note however that although, as a result of the key addition, the sets \(D_0\) and \(D_1\) differ with respect to their elements, they are still the same with respect to the differences between the values within each pair. Let \(\Delta D_k\) be the set of \(\text{ADD}\) differences modulo \(2^n\), satisfied by each pair in \(D_k\). Then:

\[
\Delta D_0 = \{(1 - 0), (1 - 0), (2 - 1), (2 - 1), (3 - 2), (3 - 2), (0 - 3), (0 - 3)\} = \Delta D_1 = \{(2 - 1), (2 - 1), (3 - 2), (3 - 2), (4 - 3), (4 - 3), (1 - 4), (1 - 4)\} = \Delta D = \{1, 1, 1, 1, 1, 5, 5\} .
\]

This example demonstrates that in certain cases (the additive DP of the TEA F-function in particular), the value of the key influences the differential probability, while still keeping the differences unaffected.
D.3 Influence of the \( \delta \) Constants

As discussed in Sect. 9.4, we modified TEA to use the same \( \delta \) constant, equal to the initial value \( 0x9e3779b9 \), at every round. Then we applied the algorithm presented in Sect. B to search for differential trails in this modified version of the cipher. We noticed that for many keys, after some rounds the best found trail would eventually become iterative with period 2 and of the form: \((\alpha \rightarrow 0),(0 \rightarrow 0),(\alpha \rightarrow 0),(0 \rightarrow 0),\ldots\) etc.

We further investigated for what value of \( \alpha \) the probability of the differential \((\alpha \rightarrow 0)\) will be maximal for a fixed key. To do this we modified Algorithm 3 (more specifically the \texttt{assign\_bit()} procedure) to assign the bits not only of the value \( x \) but also of the (unknown) difference \( \alpha \). Note that the modified algorithm computes the probability of the differential \((\alpha \rightarrow 0)\) not only for the difference that maximizes it but also for all other differences. Therefore it requires also more memory: at least \( 2^{32} \times 4 \) Bytes to store all values of \( \alpha \). For many keys we find that the difference that maximizes the probability of the differential is \( \alpha = 0xF \) and the resulting probability is \( \leq 2^{-8} \). One round key for which the probability is exactly \( 2^{-8} \) is for example \( 48585485,3FF378B3 \). We have computed that the exact number of such keys is \( 6 \cdot 2^{59} \approx 2^{61.6} \) i.e. around 10% of all keys. For 32-bit words, we were not able to find any keys for which the differential \((\alpha \rightarrow 0)\) has probability bigger than \( 2^{-8} \) for some \( \alpha \).

Finally, we report the best found differential trail on the modified version of TEA using the same \( \delta \) constant. It is based on a 4-round iterative pattern and results in a differential trail on 18 rounds with probability \( 2^{-61.36} \). The trail is shown in Table 6.

### Table 6. TEA, single \( \delta \): best found ADD differential trail for 18 rounds.

<table>
<thead>
<tr>
<th>key</th>
<th>E028DF9A</th>
<th>8819B4C3</th>
<th>3AB116AF</th>
<th>3C50723</th>
</tr>
</thead>
<tbody>
<tr>
<td>( r )</td>
<td>( \beta )</td>
<td>( \alpha )</td>
<td>( p )</td>
<td>( p, \log_2 )</td>
</tr>
<tr>
<td>1</td>
<td>FFFFFFFF1</td>
<td>1</td>
<td>0.137390</td>
<td>( 2^{-2.86} )</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>( 2^{-0.00} )</td>
</tr>
<tr>
<td>3</td>
<td>F</td>
<td>1</td>
<td>0.135712</td>
<td>( 2^{-2.88} )</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>F</td>
<td>0.001984</td>
<td>( 2^{-8.98} )</td>
</tr>
<tr>
<td>5</td>
<td>FFFFFFFF1</td>
<td>1</td>
<td>0.133148</td>
<td>( 2^{-2.91} )</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>( 2^{-0.00} )</td>
</tr>
<tr>
<td>7</td>
<td>F</td>
<td>1</td>
<td>0.138214</td>
<td>( 2^{-2.86} )</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>F</td>
<td>0.002533</td>
<td>( 2^{-8.62} )</td>
</tr>
<tr>
<td>9</td>
<td>FFFFFFFF1</td>
<td>1</td>
<td>0.137360</td>
<td>( 2^{-2.86} )</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>( 2^{-0.00} )</td>
</tr>
<tr>
<td>11</td>
<td>F</td>
<td>1</td>
<td>0.130371</td>
<td>( 2^{-2.94} )</td>
</tr>
<tr>
<td>12</td>
<td>0</td>
<td>F</td>
<td>0.001984</td>
<td>( 2^{-8.98} )</td>
</tr>
<tr>
<td>13</td>
<td>FFFFFFFF1</td>
<td>1</td>
<td>0.131958</td>
<td>( 2^{-2.92} )</td>
</tr>
<tr>
<td>14</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>( 2^{-0.00} )</td>
</tr>
<tr>
<td>15</td>
<td>F</td>
<td>1</td>
<td>0.137543</td>
<td>( 2^{-2.86} )</td>
</tr>
<tr>
<td>16</td>
<td>0</td>
<td>F</td>
<td>0.002228</td>
<td>( 2^{-8.81} )</td>
</tr>
<tr>
<td>17</td>
<td>FFFFFFFF1</td>
<td>1</td>
<td>0.136597</td>
<td>( 2^{-2.87} )</td>
</tr>
<tr>
<td>18</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>( 2^{-0.00} )</td>
</tr>
</tbody>
</table>

\[ \Pi. \quad 2^{-61.36} \]
D.4 Improving the efficiency of Algorithm 1

In this section we describe in more detail the improvement of the efficiency of Algorithm 1 when used to construct a pDDT for F (cf. Sect. B.2). We exploit the fact that the three inputs to the XOR operation in F are strongly dependent. As a result many differentials that are valid when the XOR is considered in isolation will in fact not be possible when XOR is viewed as a component of F.

Let $\alpha$, $\beta$ and $\gamma$ be input differences to the XOR in F, where $\alpha$ is the initial input difference to F, and $\beta$ and $\gamma$ are the differences after the LSH and the RSH operations respectively. Then by Theorem 1 and Theorem 2, $\alpha$, $\beta$ and $\gamma$ are subject to the following constraints:

\[
(\alpha, \beta, \gamma) : (\beta = (\alpha \ll 4)) \land \\
(\gamma \in \{(\alpha \gg 5), (\alpha \gg 5) + 1, (\alpha \gg 5) - 2^{n-5}, (\alpha \gg 5) - 2^{n-5} + 1\}) .
\] (35)

Let $\alpha[i : 0], \beta[i : 0]$ and $\gamma[i : 0]$ be partially constructed $(i + 1)$-bit differences. Then expressing (35) bitwise for the first $(i + 1)$ bits we get:

\[
(\alpha[i : 0], \beta[i : 0], \gamma[i : 0]) : \\
((\beta[i : 0] = 0 : i < 4) \lor (\beta[i : 0] = \alpha[i - 4 : 0]|0000(2) : i \geq 4)) \land \\
((\gamma[i : 0] \in \{(\alpha[i + 5 : 5])[i : 0], (\alpha[i + 5 : 5] + 1)[i : 0], \\
(\alpha[i + 5 : 5] - 2^{n-5}[i : 0], (\alpha[i + 5 : 5] - 2^{n-5} + 1)[i : 0]) : \\
5 \leq i < 27)) .
\] (36)

Equation (36) is a necessary, but not a sufficient condition for (35). The reason is the following. Consider one of the partially constructed differences after RSH: $(\alpha[i + 5 : 5] - 2^{n-5}[i : 0])$. Note that $-2^{n-5} = 2^n - 2^{n-5}$ mod $2^n$, which is a value that has only its five MS bits set (e.g. for $n = 32 : -2^{n-5} = 2^{32} - 2^{27} = 0x8000000$). Thus the term $-2^{n-5}$ affects the sum $\sum (\alpha[i + 5 : 5] - 2^{n-5}[i : 0])$ only for $i \geq 27$. Therefore, although a given difference may be discarded as invalid when evaluated at position $i < 27$ according to (36), for $i \geq 27$ it may still turn out to be a valid difference after subtraction by $2^{n-5}$ or $(2^{n-5} - 1)$.

Condition (36) in combination with $p_{i+1} \geq p_{\text{thres}}$ (cf. Algorithm 1) improves the efficiency of Algorithm 1 when applied to construct a pDDT for F. Still, due to the fact that it is not a sufficient condition, as discussed above, not all valid differentials are present, and therefore the constructed pDDT is incomplete.

E Proofs

E.1 Proof of Proposition 1

Proof. We shall prove the proposition for $\text{adx}^\oplus$. In this case $\alpha$, $\beta$ and $\gamma$ are ADD differences propagating through the XOR operation. The proof for $\text{adx}^+$ is analogous.

We induct over the word size $n$. The proposition is trivially true for the base case $n = 1$: $p_1 \leq p_0 = 1$. Let $n = k > 1$. We have to prove that $p_k \leq p_{k-1}$.

Let $x$ and $y$ be $n$-bit integers. Define $L_i$ to be the set of $i$-bit pairs $(x_i, y_i)$ that satisfy the differential $(\alpha_i, \beta_i \rightarrow \gamma_i)$ for the operation addition modulo $2^i$:

\[
L_i = \{(x_i, y_i) : ((x_i + \alpha_i) \oplus (y_i + \beta_i)) - (x_i + y_i) = \gamma_i\}, \quad n \geq i \geq 1 .
\] (37)
Let \( l_i = \#L_i \). By definition \( p_k = l_k/2^{2k} \) and \( p_{k-1} = l_{k-1}/2^{2(k-1)} \) (cf. (2)). Note that every element of \( L_k \) can be obtained from an element \((x_{k-1}, y_{k-1})\) of \( L_{k-1} \) by appending bits \( x[k-1] \) and \( y[k-1] \) to \( x_{k-1} \) and \( y_{k-1} \) respectively. Assume that this is not true i.e. assume:

\[
\exists x_k, y_k : \quad (x_k = x[k-1] | x_{k-1}, \ y_k = y[k-1] | y_{k-1}, \ (x_k, y_k) \in L_k) \land \\
(x_{k-1}, y_{k-1}) \not\in L_{k-1} .
\] (38)

If (38) is true then we can construct a new set \( L'_{k-1} = (x_{k-1}, y_{k-1}) \cup L_{k-1} \). Its size is \( l'_{k-1} = l_{k-1} + 1 \) and so \( p_{k-1} = l'_{k-1}/2^{2(k-1)} \). The latter differs from the actual value of the probability \( p_{k-1} = l_{k-1}/2^{2(k-1)} \) and therefore the assumption (38) is false. Thus \( \forall (x_k, y_k) \in L_k : \ (x_{k-1}, y_{k-1}) \not\in L_{k-1} \). Because \#\{(x[k], y[k])\} = 2^2, the size of \( L_k \) can be at most 2^2 times bigger than the size of \( L_{k-1} \):

\[
l_k \leq 2^2 l_{k-1} \Rightarrow \frac{l_k}{2^4} \leq \frac{l_{k-1}}{2^{2(k-1)}} \Rightarrow p_k \leq p_{k-1} .
\] (39)

\[
\square
\]

E.2 Proof of Theorem 1

Proof. Let \( x \) be an \( n \)-bit input to \( \text{LSH} \) with shift constant \( r \leq n \). Let \( x_L, x_R : x = x_L 2^{n-r} + x_R \). Then \( (x \ll r) = x_R 2^r \). Similarly, for the input \( \text{ADD} \) difference \( \alpha \) let \( \alpha_L, \alpha_R : \alpha = \alpha_L 2^{n-r} + \alpha_R \) and thus \( (\alpha \ll r) = \alpha_R 2^r \). The sum \((x + \alpha)\) can then be represented as:

\[
(x + \alpha) = (x_L + \alpha_L) 2^{n-r} + (x_R + \alpha_R) \\
= ((x_L + \alpha_L + c_R) \mod 2^r) 2^{n-r} + ((x_R + \alpha_R) \mod 2^{n-r}) ,
\] (40)

where \( c_R \) is the carry generated from the addition \((x_R + \alpha_R) \mod 2^{n-r} \). From (40) follows that \((x + \alpha) \ll r = (x_R + \alpha_R) 2^r \). Thus for the output difference \( \beta \) we get:

\[
\beta = ((x + \alpha) \ll r) - (x \ll r) = (x_R + \alpha_R) 2^r - x_R 2^r = \alpha 2^r = (\alpha \ll r) .
\] (41)

Note that (41) is independent of the input \( x \) and therefore holds with probability 1 over all values of \( x \). From this the expression (4) for the probability \( \text{adp} \ll r \) immediately follows. \( \square \)

E.3 Proof of Theorem 2

Proof. Let \( x \) be an \( n \)-bit input to \( \text{RSH} \) with shift constant \( r \leq n \). Let \( x_L, x_R : x = x_L 2^r + x_R \). Then \( (x \gg r) = x_L \). Similarly, for the input \( \text{ADD} \) difference \( \alpha \) let \( \alpha_L, \alpha_R : \alpha = \alpha_L 2^r + \alpha_R \) and thus \((\alpha \gg r) = \alpha_L \). Denote by \( c_R \) the carry generated from the addition \((\alpha_R + \alpha_R) \mod 2^r \):

\[
c_R = \begin{cases} 
0 , & \text{if } (x_R + \alpha_R) < 2^r \\
1 , & \text{otherwise}
\end{cases} .
\] (42)

The sum \((x + \alpha)\) can then be represented as:

\[
(x + \alpha) = (x_L + \alpha_L) 2^r + (x_R + \alpha_R) \\
= ((x_L + \alpha_L + c_R) \mod 2^{n-r}) 2^r + ((x_R + \alpha_R) \mod 2^r) .
\] (43)
Therefore \((x + \alpha) \gg r = (xL + \alphaL + cR) \mod 2^{n-r}\) and for the output difference \(\beta\) we derive:

\[
\beta = ((x + \alpha) \gg r) - (x \gg r) = ((xL + \alphaL + cR) \mod 2^{n-r}) - xL
\]

\[= \alphaL - cL 2^{n-r} + cR ,\]

where

\[
cL = \begin{cases} 0 , & \text{if } (xL + \alphaL + cR) < 2^{n-r} , \\ 1 , & \text{otherwise} . \end{cases}
\]

The term \(-cL 2^{n-r}\) in (44) is introduced in order to cancel the carry \(2^{n-r}\) that is generated in the cases in which the sum \((xL + \alphaL + cR)\) is bigger than \((2^{n-r} - 1)\). In such a case \(cL = 1\) and \(-cL 2^{n-r} + (xL + \alphaL + cR) = -2^{n-r} + 2^{n-r} + (xL + \alphaL + cR) \mod 2^{n-r} = (xL + \alphaL + cR) \mod 2^{n-r}\).

In the expression for \(\beta\) (44), for each distinct value of the tuple \((cL, cR)\) we get one of the four possibilities for \(\beta\):

\[
\beta = \begin{cases} (\alpha \gg r) , & cL = 0, cR = 0 , \\ (\alpha \gg r) - 2^{n-r} , & cL = 1, cR = 0 , \\ (\alpha \gg r) + 1 , & cL = 0, cR = 1 , \\ (\alpha \gg r) - 2^{n-r} + 1 , & cL = 1, cR = 1 . \end{cases}
\]

In order to compute the corresponding probabilities, we have to count the number of inputs \(x\), that result in a given value for \((cL, cR)\). Note that \(cL\) and \(cR\) depend on \(x\) and \(\alpha\), of which \(\alpha\) is fixed and \(x\) can take on all values from 0 to \(2^n - 1\). From (42) it is easy to compute that \(cR = 0\) for exactly \((2^r - \alphaR)\) values of \(xR\) and therefore \(cR = 1\) for the remaining \(2^r - (2^r - \alphaR) = \alphaR\) values. Note that \(xR\) is an \(r\)-bit word. Similarly, if \(cR = 0\) then \(cL = 0\) for \((2^{n-r} - \alphaL)\) values of \(xL\) and \(cL = 1\) for the remaining \(\alphaL\) values. If \(cR = 1\) then \(cL = 0\) for \((2^{n-r} - \alphaL - 1)\) values and \(cL = 1\) for the remaining \(\alphaL + 1\) values. Therefore \((cL, cR) = (0, 0)\) for \((2^{n-r} - \alphaL)(2^r - \alphaR)\) values of \(x\). Since the total number of values is \(2^n\) we obtain the probability:

\[
adp^{>\gamma}(\alpha \rightarrow \beta = (\alpha \gg r)) = 2^{-n}(2^{n-r} - \alphaL)(2^r - \alphaR) .
\]

The expressions for the remaining three probabilities are derived analogously.

\(\square\)

### E.4 Proof of Theorem 3

**Proof.** Denote \(p_i = \adp^{\leq}(\alpha \ll i, \alpha, \gamma_i \rightarrow \beta)\) and \(q_i = \adp^{>\gamma}(\alpha \rightarrow \gamma_i)\). Let \(\forall i : 0 \leq i < 4 : p_i \geq p_{\text{thres}}\). After multiplying both sides by \(q_i\) and summing over \(i\) we obtain:

\[
\sum_{i=0}^{3} (p_i q_i) = \eadp^{F}(\alpha \rightarrow \beta) \geq \sum_{i=0}^{3} (p_{\text{thres}} q_i) = p_{\text{thres}} \sum_{i=0}^{3} q_i = p_{\text{thres}} ,
\]

since \(\sum_{i=0}^{3} q_i = 1\) (cf. Theorem 2), which proves (12). Similarly, to prove (13) assume that \(\exists i : \adp^{\leq}(\alpha \ll i, \alpha, \gamma_i \rightarrow \beta) < p_{\text{thres}}\). Due to (48) this assumption contradicts the fact that \(\eadp^{F}(\alpha \rightarrow \beta) \geq p_{\text{thres}}\) and is therefore false. This completes the proof. \(\square\)
Power Analysis of Hardware Implementations
Protected with Secret Sharing

Guido Bertoni∗, Joan Daemen∗, Nicolas Debande†‡, Thanh-Ha Le†, Michaël Peeters§ and Gilles Van Assche∗,
∗STMicroelectronics, †Morpho, ‡TELECOM ParisTech and §NXP Semiconductors

Abstract—We analyze the security of three-share hardware implementations against differential power analysis and advanced variants such as mutual information analysis. We present dedicated distinguishers that allow to recover secret key bits from any cryptographic primitive that is implemented as a sequence of quadratic functions. Starting from the analytical treatment of such distinguishers and information-theoretic arguments, we derive the success probability and required number of traces in the presence of algorithmic noise. We show that attacks on three-share hardware implementation require a number of traces that scales in the third power of the algorithmic noise variance. Finally, we apply and test our model on Keccak in a keyed mode.

Index Terms—power analysis; quadratic functions; secret sharing schemes; mutual information analysis; Keccak

I. INTRODUCTION

Protection against side channel attacks is essential for any implementation of a cryptographic primitive that needs to process secret keys and to which an adversary has access. Failing to put in place sufficient countermeasures can lead to the recovery of a secret key or otherwise secret information that allows, e.g., to forge an authentication code or to recover decrypted data. An important class of side channel attacks is differential power analysis (DPA) and its variants, which exploits dependencies between the data being processed and the power consumption or electromagnetic radiation [1], [2].

In a hardware circuit, the implementer cannot always control the exact sequencing of gate switching and a proper countermeasure must take glitches into account. To achieve protection under realistic assumptions, including the presence of glitches, Nikova et al. propose techniques based on secret sharing [3], [4]. In their paper, the authors analyze a three-share implementation of the Noekeon block cipher [5], which provides security against first-order DPA. However, the authors noticed that an advanced variant of DPA, called mutual information analysis (MIA) [6], can reveal the correct key bits if the power consumption is not affected by noise.

In this paper, we analyze the security of three-share hardware implementations against DPA (including MIA) under the presence of noise induced by all the bits being processed. The analysis can be applied to any cryptographic primitive whose round function is a Boolean vector function of algebraic degree two, called quadratic in the sequel or is implemented as sequence of quadratic operations. We use the Keccak function as an example, for which the designers defined protected and unprotected hardware architectures that we can explicitly compare using the techniques presented in this paper [7]. Nevertheless, the treatment actually covers a variety of primitives. First, there are those whose round function is explicitly described in terms of quadratic functions. For instance, the stream cipher Trivium is built as three cyclically-connected shift registers with quadratic feedback [8]. Then, there are primitives making use of small s-boxes. For instance, the s-boxes of Present [9] or Noekeon [5] have algebraic degree 3 (cubic) and decomposable into two quadratic functions. Applying secret sharing techniques on each quadratic function requires less shares than on the cubic function [3], [4]. The AES can also be protected using secret sharing, as presented by Moradi et al. [10].

The techniques of this paper share some similarity with those in [11]. Their Zero-Offset 2DPA distinguisher can be applied in the case of hardware implementations using two shares. We adapt it to three shares in Section IV-C, show that three shares concretely provide a masking scheme that provably offers protection for functions implemented with quadratic operations, analytically model the attack success probability, and compare with simulations. In our analysis and simulations we have adopted the Hamming-Distance model [2], applied to the registers. A conclusion of the analysis of this paper is that a three-share implementation does provide security against DPA and MIA. We analytically show that in our model the number of traces needed to distinguish the correct key bits grows with the third power of the noise variance induced by the bits being computed. This suggests that masking with three shares on quadratic functions efficiently provides security that scales similarly as third-order DPA [12], [13], [14]. An analysis taking into account the power consumption of the combinatorial logic, with its occurrence of glitches, is future work.

The paper is organized as follows. Section II gives a model of the power consumption in hardware both for an unprotected and for a protected hardware implementation. Section III proposes a general form of selection function suitable for any quadratic function and different leakage models. Distinguishing against unprotected and protected implementations are presented in Section IV. Finally, Section V applies the selection function and distinguishers on the Keccak hash function and shows simulations.

II. FROM BITS TO POWER CONSUMPTION

In power (or electromagnetic) analysis, an attacker attempts to retrieve key information by analyzing the power consump-
tion (or electromagnetic radiation) of the device performing the cryptographic primitive. The general set-up is that the attacker has one or more traces of the measured power consumption. DPA exploits the dependence, however weak, of the power consumption on the processed variables by taking the power traces of many executions of the cryptographic primitive with different input values and processing them with statistical methods to retrieve the values of the processed variables [1].

During the last decade, researchers have improved the treatment of traces and one of these enhancement is the so-called correlation power analysis (CPA) [15]. CPA exploits the correlation between the power consumption and processed variables. Later, more advanced ways to measure the distance between distributions were introduced. In particular, MIA does not require a non-zero correlation, but can in principle exploit any dependence of the distribution of the power consumption on the value of variables [6].

There are different types of countermeasure for protecting the cryptographic computation. They range from the transistor level (such as dual rail logic) up to the protocol level (via a careful use of different keys). In this paper we study the effectiveness of a countermeasure at algorithmic level, i.e., that of a secret sharing scheme.

A. Modeling leakage

A hardware circuit consumes power through the activity of its registers, combinatorial logic, wires and auxiliary logic such as the clock. In synchronous CMOS circuits, the data-dependent part of the power consumption is dominated by the dynamic power consumption, particularly the switching activity of the registers and the combinatorial logic [2, Chapter 3]. In this paper we adopt the Hamming distance model limited to registers, with the convention that a register contributes −1 if it flips and +1 otherwise. We make abstraction of the dynamic power consumption of the combinatorial logic, and more in particular the effect of glitches. Investigation of the latter requires a model that is closer to the implementation than the Hamming distance model but the analysis of the resulting power consumption functions is of similar nature. The same goes for models in which the switching activity of different elements does not have the same weight.

For a given primitive, let $K$ be the fixed secret key under attack and $M$ the input message controlled by the attacker. The activity in each register bit is given by some binary functions $d_i(M, K) \in \text{GF}(2)$, where $d_i(M, K) = 1$ if the bit flips and 0 otherwise. Then, the contribution of all the registers to the power consumption is expressed as:

$$P(M, K) = \sum_i (-1)^{d_i(M, K)}.$$  

(1)

Adopting a more complex model will result in a similar expression than Eq. (1), with terms $a_i(-1)^{d_i(M, K)}$, where $i$ ranges over all elements (gates, registers, ...) contributing to the power consumption and the factors $a_i$ are weighing factors.

B. Protection with secret sharing

The secret sharing scheme technique proposed in [3], [4] can be seen as an evolved version of the duplication method that was introduced in [16]. These two methods consist in representing the internal variables $x$ of a primitive (i.e., its state) in a number of shares $a, b, c, \ldots$. The sum (typically in GF(2)) of these shares gives the native value of the state: $x = a \oplus b \oplus c \ldots$. Performing computations or storing a single share has a power consumption that is independent of the native value as each share is independent of the native value.

The linear steps of the primitive can be computed on the shares in separate computations as for a linear function $f$ it yields $f(x) = f(a \oplus b \oplus c \ldots) = f(a) \oplus f(b) \oplus f(c) \ldots$. Information-theoretically, the variables processed in each of these calls to $f$ provide no information on the native value of the state. For the non-linear steps, one cannot avoid computations that mix elements from more than one share of the state. However, if one can ensure that no intermediate results are correlated to native state bits, this rules out any form of DPA that requires a correlation between the power consumption and native state bits, such as correlation power analysis (CPA).

In dedicated hardware, however, it is not easy to ensure the absence of such correlations, which can arise from transient states due to glitches in the combinatorial logic [17]. The secret sharing scheme technique proposed in [3], [4] solves this problem essentially by representing the state in $n$ shares such that any computation involves at most $n - 1$ shares. The required number of shares depends on the algebraic degree of the nonlinear step to be computed, and for quadratic functions 3 shares are usually sufficient (see the exact conditions in [3], [4]).

C. Modeling leakage with three shares

The implementation of quadratic functions can be protected using three shares. Each bit of the state is therefore stored as the parity of three registers, a so-called register triplet, i.e., the native value 0 can be represented as 000, 011, 101 or 110, and 1 as 111, 100, 010 or 001. The value of two of the three registers is generated randomly each time the primitive is evaluated, hence for each measurement, and the power consumption becomes a stochastic variable to the adversary.

Considering the power consumed by the changes in registers, we define $d_i(M, K)$ as in Section II-A, where $d_i(M, K) = 1$ if the native value of the bit $i$ changes and 0 otherwise. The impact on the power consumption of the register triplets is as follows.

- $d_i(M, K) = 0$: either no register flips, contributing +3, or 2 of the 3 registers flip, contributing −1. The contribution is a stochastic variable that has value 3 with probability 1/4 and −1 with probability 3/4. We denote this variable by $T_0$ and its distribution by $T_0(t)$.
- $d_i(M, K) = 1$: either all 3 registers flip or 1 of the 3 registers flip. The contribution is a stochastic variable that has value $-3$ with probability 1/4 and 1 with probability 3/4. We denote it by $T_1$ and its distribution by $T_1(t)$.

Both $T_0(t)$ and $T_1(t)$ have mean 0 and variance 3.

The power consumption is the sum of the stochastic variables determined by the bits of $d$:

$$P(M, K) = \sum_i T_{d_i(M, K)}.$$  

(2)
As opposed to an unprotected implementation, the total power consumption is not a deterministic function of \( d \), but a stochastic variable with a probability density function that is determined by \( d \). Its probability distribution is obtained by convoluting the distributions \( T_{d_i(M,K)}(t) \) over all indices \( i \).

Interestingly, the value of a bit of \( d_i(M,K) \) does not affect the mean of the probability density function of \( P(M,K) \), hence precluding CPA. Moreover, as \( T_0 \) and \( T_1 \) have the same variance, it does not even affect the variance of \( P(M,K) \).

III. SELECTION FUNCTION FOR QUADRATIC FUNCTIONS

We describe a general way to build a suitable selection function for the implementation of a quadratic function. We assume that the primitive is implemented as a sequence of quadratic operations, although a slightly more general set of functions is covered. This applies not only to primitives making use of a quadratic round function but also for round functions that can suitably be implemented as the composition of quadratic functions. Let \( K \) be the fixed secret key under attack and \( M \) the input message controlled by the attacker. Let the activity we target be denoted as \( d \), gathering \( b \) binary functions \( d_i(M,K) \), one for each output bit \( i \). The function \( d_i(M,K) \) has the following form in GF(2):

\[
d_i(M,K) = \alpha_i(M) + \beta_i(K) + K^T \Gamma_i M. \tag{3}
\]

Here, \( \alpha_i(M) \) and \( \beta_i(K) \) are any functions of \( M \) and \( K \) only, respectively, while \( \Gamma_i \) is a binary matrix connecting the bits of \( K \) and \( M \) in the bilinear form \( K^T \Gamma_i M \). Note that Eq. (3) encompasses not only all quadratic functions but also some more general functions, as the degree of \( \alpha_i(M) \) and of \( \beta_i(K) \) is not limited.

The function \( d \) can model the switching activity of a register with a quadratic updating function (Hamming distance model) but also consumption imbalance in (quadratic) combinatorial logic (Hamming weight model).

We will consider in this paper attacks based on the consumption of a single register at a time. We do not expect attacks addressing multiple registers to give better success probabilities. It may result in a marginal decrease of algorithmic noise but at the same time it increases the number of hypothesis exponentially.

The bit \( i \) we use to partition the traces at a given time is called the focus point. Our goal is to build a selection function for the focus point that splits measurements into two sets \( M_0 \) and \( M_1 \), whose distributions can be set apart when the hypothesis on key bits is correct. The key point is that the distributions of \( M_0 \) and \( M_1 \) differ if the correct hypothesis is set.

For the activity function in Eq. (3), the selection function is defined as

\[
s_i(M,K) = \alpha_i(M) + K^T \Gamma_i M. \tag{4}
\]

Compared to \( d \), we remove in \( s \) any fixed (unknown) contribution of the key. The influence of the key goes only through \( K^T \Gamma_i M \). The hypothesis on \( K \) need to be set only within a subset of dimension \( \text{rank} \Gamma_i \). For instance, let \( U_i \) be an invertible matrix such that \( U_i^T \Gamma_i \) has only \( \text{rank} \Gamma_i \) non-zero rows, and let \( K = U_i \kappa \). Then, the hypothesis is only on the bits \( \kappa_j \) of the non-zero rows of \( U_i^T \Gamma_i \).

Let the hypothesis on the key \( K^* \) be related to the correct key \( K \) as \( K = K^* + \epsilon \). Then, \( s_i(M,K + \epsilon) = s_i(M,K) + \epsilon^T \Gamma_i M \). When \( \epsilon \in \ker \Gamma_i^T \), then \( s_i(M,K^*) \) differs from \( d_i(M,K) \) only by a constant (unknown) term. Otherwise, if \( \epsilon \notin \ker \Gamma_i^T \) and at the condition that \( \epsilon^T \Gamma_i M \) is balanced over GF(2) when \( M \) is seen as a random variable, then \( s_i(M,K^*) \) behaves as a random variable that is independent from \( d_i(M,K) \). This is the case, for instance, if \( M \) is uniformly distributed over its underlying set.

IV. DISTINGUISHERS

We start with using the Kullback-Leibler divergence to determine a lower bound on the number of measurements needed to distinguish two distributions. Then, we see how to distinguish the distributions in the cases of the unprotected and three-share implementations.

A. Kullback-Leibler divergence as a reference

The number of samples that are needed to distinguish between one distribution \( f \) over another \( g \) with some given success probability is inversely proportional to the Kullback-Leibler divergence \( D(f\|g) \) between the two distributions [18], with

\[
D(f\|g) = \int f(t)(\log f(t)) - \log(g(t)) dt.
\]

This measure allows estimating the required number of samples for successfully distinguishing the correct hypothesis over the wrong ones. We model the distributions that correspond to the different hypotheses as a two-dimensional function of \( t \) and \( s \), with \( t \) the value of the power consumption and \( s \) the value of \( s_i(M,K^*) \). Let us denote them by \( f^*(t,s) \) for the correct hypothesis and \( f^0(t,s) \) for the incorrect hypotheses. We assume the distributions of all incorrect hypotheses are equal. We then have:

\[
D(f^*(t,s)||f^0(t,s)) = \sum_s \int f^*(t,s) \log \left( \frac{f^*(t,s)}{f^0(t,s)} \right) dt. \tag{4}
\]

If we assume that \( \Pr(s_i(M,K) = 0|M) = \Pr(s_i(M,K) = 1|M) = \frac{1}{2} \), we can express \( f^*(t,s) \) as \( f^*(t,s) = \frac{1}{2} f_0^*(t) \) with \( f_0^*(t) \) and \( f_0^*(t) \) the distributions of the power consumption for \( s = 0 \) and \( s = 1 \) respectively. If we further assume for the incorrect hypothesis that the distribution is independent from \( s \), we have \( f^0(t,s) = \frac{1}{2} f_0^0(t) \). This allows us to simplify Eq. (4):

\[
D(f^*||f^0) = \frac{1}{2} \int f_0^0(t) \log \left( \frac{f_0^0(t)}{f_0^0(t)} \right) + \frac{1}{2} \int f_0^0(t) \log \left( \frac{f_0^0(t)}{f_0^0(t)} \right) dt.
\]

Additionally, if we have a symmetry condition such that \( f_0^0(t) = f_0^0(-t) \), \( f^0(t) = f^0(-t) \), we can further simplify this into:

\[
D(f^*||f^0) = \int f_0^0(t) \log \left( \frac{f_0^0(t)}{f_0^0(t)} \right) dt. \tag{5}
\]
B. Distinguishing for unprotected implementations

We show how to distinguish the measurements in the case of a correct hypothesis from those in the case of an incorrect one. For this, we assume that the power consumption follows the model in Section II-A and in particular Eq. (1). Depending on the exact form of the functions \( d_i \), there are ways to select the messages \( M \) such that \( d_i(M, K) \) is perfectly balanced. However, to simplify the discussion, we assume that the messages \( M \) are randomly selected.

For a given focus point \( i \) and hypothesis \( K^* \), we gather in sets \( M_0 \) and \( M_1 \) the power measurements \( P(M, K) \) for which \( s_i(M, K^*) \) is 0 or 1, respectively. In each of these sets, the noise is only algorithmic and its amplitude depends on the number \( b \) of bits computed simultaneously. We study the influence of this noise on the success probability of an attack on an unprotected implementation.

If \( s_i(M, K^*) \) is independent from \( d_i(M, K) \) with \( i' \neq i \), the distribution of the power consumption is the convolution of distributions with peaks at \(-1\) and \(+1\). This leads to a binomial distribution that is close to a normal distribution thanks to the central limit theorem. For the distribution within the sets \( M_0 \) and \( M_1 \), we distinguish two cases:

- For the correct hypothesis, \( P(M, K) \) knowing \( s_i(M, K) \) has mean \( \pm 1 \) and variance \( b - 1 \). So \( f^*_i(t) \approx \mathcal{N}_{i([-1;+1])}(t) \), where \( \delta \) is fixed due to the (unknown) difference between \( s_i \) and \( d_i \).
- For the incorrect hypotheses, \( P(M, K) \) is independent of \( s_i(M, K^*) \) and \( f^{o}_i(t) \approx \mathcal{N}_{i(0;1)}(t) \).

The goal of our attack on the plain core is to build an understanding on the influence of noise on the success probability and to compare later to the three-share core. Therefore, we will consider an attack where a (passive) attacker is provided traces for randomly chosen message blocks. This random process introduces noise that is easy to model. We are aware that in our model of the plain core one can conduct a noiseless attack but this would set the wrong target for comparing with the three-share core, where noise is unavoidable.

If the bits \( q(x', y', z') \) are independent from each other, each one contributes to the distribution by convoluting it with a distribution with a impulse at \(-1\) and one at \(+1\). This leads to a binomial distribution that is very close to a normal distribution with variance \( b - 1 \) thanks to the central limit theorem.

The distributions \( f^* \) and \( f^{o} \) satisfy the assumptions of Section IV-A, so Eq. (5) yields:

\[
\Delta_{\text{DoM}}(K^*) = |E[P(M, K)|s_i(M, K^*) = 0] - E[P(M, K)|s_i(M, K^*) = 1]|.
\]

If we denote the number of traces by \( |M| \) and assume \( M_0 \) and \( M_1 \) have the same size, the average power consumption over \( M_0 \) and over \( M_1 \) have an expected variance of \( 2(b - 1)/|M| \) and \( 2b/|M| \) for the correct and incorrect hypotheses, respectively. The difference of the means is \( 2(-1)^b \), and the variance is the sum of the variances. Taking the absolute value folds the distribution around the \( y \)-axis and discards the unknown sign \( (-1)^b \). This results in the following distributions for \( t \geq 0 \) (and 0 otherwise):

\[
f^\ast_{\text{DoM}}(t) = \mathcal{N}_{(2;4(b - 1))/|M|}(t) + \mathcal{N}_{(-2;4(b - 1))/|M|}(t)
\]

These distributions (assuming \( b - 1 \approx b \)) are illustrated in Figure 1.

![Distributions](image)

**Fig. 1.** Distributions \( f^\ast_{\text{DoM}}(t) \) and \( f^{o}_{\text{DoM}}(t) \) (or \( f^\ast_{\text{DoA}}(t) \) and \( f^{o}_{\text{DoA}}(t) \)).

The probability of success can now readily be computed from these distributions. It is the probability that a variable chosen according to \( f^\ast_{\text{DoM}}(x) \) is larger than \( h - 1 \) values chosen according to \( f^{o}_{\text{DoM}}(x) \), with \( h = 2^{\text{rank} \Gamma} \), the number of hypotheses considered. This is given by:

\[
P_{\text{success}} = \int_0^{\infty} \left( \int_0^t f^\ast_{\text{DoM}}(y) \, dy \right)^{h-1} f^{o}_{\text{DoM}}(t) \, dt.
\]

Let \( G_h \) be the function defined as

\[
G_h(x^2) = \int_0^\infty \left( \text{erf} \left( \frac{t}{\sqrt{2x^2}} \right) \right)^{h-1} \left( \mathcal{N}_{(-1;\sigma^2)}(t) + \mathcal{N}_{(1;\sigma^2)}(t) \right) \, dt.
\]

The expression \( \int_0^t \mathcal{N}_{(0;4b/|M|)}(y) \, dy \) can be expressed in terms of the error function as follows: \( \text{erf}(t/\sqrt{8b/|M|}) \). If we approximate \( b - 1 \approx b \), we see the
success probability is fully determined by the ratio \( b/|M| \), and \( P_{\text{success}} = G_b(b/|M|) \). Using \( D(f^*\|f^0) = 1/2b \) we can express it as a function of the Kullback-Leibler divergence, yielding:

\[
P_{\text{success}} = G_b\left(\frac{1}{2D(f^*\|f^0)|M|}\right)
\]

(6)

### C. Distinguishing for three-share implementations

We assume a power consumption given by Eq. (2) in Section II-C. Hence we have:

\[
f^*_t(t) = T_1 + \delta(t) \oplus N_{0,3,b-1}(t)
\]

\[
= \frac{1}{4} N_{-3(-1)^i,3(1-b)}(t) + \frac{3}{4} N_{(-1)^i,3(1-b)}(t)
\]

\[
f^0_t(t) = N_{0,3b}(t)
\]

with \( \delta \) the (unknown) fixed difference between \( s_i \) and \( d_i \). We can fill this in Eq. (5) to compute the Kullback-Leibler distance. We evaluated this expression numerically and for large values of \( b \) it converges to \( D(f^*\|f^0) = 1/(9b^3) \). So the required number of traces is proportional to the 3rd power of the register size.

\( T_0 \) and \( T_1 \) start to differ only from their third statistical moment, called the coefficient of asymmetry (or skewness) \[19\]. We call the distinguisher based on this the Difference of Asymmetry (DoA):

\[
\Delta_{\text{DoA}}(M^*) = E[P(M, K^3|s_i(M, K^*) = 0] - E[P(M, K^3|s_i(M, K^*) = 1] \]

Note that the DoA can also be seen as the Zero-Offset 2DPA distinguisher of \[11\] adapted to the third order.

For a normal distribution \( N(\mu; \sigma^2) \), its third moment is \( \mu^3 + 3\mu\sigma^2 \) \[20\]. So, for the correct hypothesis, the power measurements in \( M \) have \( E[P^3] = (6(1)^3) + 3 \) and taking the difference gives \( E[\Delta_{\text{DoA}}] = 12(-1)^3 \). For the incorrect hypotheses, the third moment is zero.

The variance on the skewness \( E[P^3] \) for \( |M| \) samples of a normal distribution with variance \( \sigma^2 \) is given by \( 6\sigma^2/|M| \) \[19\]. So assuming \( b \gg 1 \), we have \( \sigma^2(\Delta_{\text{DoA}}) \approx 24(3b^3/|M|) \). Using the central limit theorem and then taking the absolute value, we write for \( t \geq 0 \):

\[
f_{\Delta_{\text{DoA}}}(t) = N_{12;24(3b^3)/|M|}(t) + N_{-12;24(3b^3)/|M|}(t)
\]

\[
f_{\Delta_{\text{DoA}}}(t) = 2N_{0,24(3b^3)/|M|}(t)
\]

Figure 1 illustrates also these distributions. Following the same reasoning as in Section IV-B yields \( P_{\text{success}} = G_b(9b^3/2|M|) \). Using \( D(f^*\|f^0) = 1/(9b^3) \), we can again express the success probability as a function of the Kullback-Leibler divergence. Remarkably, this leads to the same expression as in the case of DoM, i.e., Equation (6).

### D. Information theoretic distinguishers

The mutual information of two discrete variables \( X \) and \( Y \) is \( I(X; Y) = H(X) - H(X|Y) \), with \( H \) denoting the Shannon entropy \[18\]. Informally, the mutual information \( I(X; Y) \) is the amount of information (bits) the variables \( X \) and \( Y \) have in common. When two variables are independent, the mutual information is zero.

We can compute the mutual information between the variables \( P(M, K) \) and \( s_i(M, K^*) \), for a given hypothesis \( K^* \). For the incorrect hypothesis, \( f^*(t, 0) = f^0(t, 1) \), therefore the two variables are independent and \( I(P, s_i) = 0 \). For the correct hypothesis, \( f^*(t, 0) \neq f^*(t, 1) \) and the mutual information is strictly positive.

This makes the mutual information a candidate distinguisher. One needs to estimate or approximate the joint distribution of \( P(M, K) \) and \( s_i(M, K^*) \) using the measurements. This leads to a number of variants (see also, e.g., \[21\]). Most of MIA estimators are in the probability density function (pdf) estimators family. Another family groups methods that use statistics. In the pdf estimators family, the methods are either non-parametric methods or parametric methods. Parametric methods assume particular distributions, e.g., the Gaussian parametric method assumes the distributions are normal. MIA statistics methods use either statistical tests such as computing a distance or statistical tools such as computing statistical moments. An example of the former is the Kolmogorov-Smirnov distance and an example of the latter is cumulant MIA. There are also extensions of MIA from univariate to multivariate distributions \[22\].

The method of cumulant MIA aims at estimating mutual information by using high-order cumulants. This method was described in \[23\], where an Edgeworth expansion \[20\] was used to estimate mutual information. Applied to \( I(P(M, K); Z) \) with \( Z = (-1)^n(M, K^*) \), this becomes

\[
I(P; Z) \approx \frac{1}{4} E[P, Z^2] + \frac{1}{12} (E[P^2, Z^2] + E[P, Z^2])^2
\]

\[
\]

(7)

The attacker can evaluate this expression for the different hypotheses by computing the different terms based on the measurements. This estimation allows to analyze the different dependencies separately, which is not the case with other approaches. Indeed, an attacker can determine which components of the mutual information estimation give the most useful information.

In the case of the three-share implementation, we see that most terms in Eq. (7) disappear. First, \( E[P, Z^2] = 0 \) for any \( i \) as the mean of \( P \) is zero over both \( M_0 \) and \( M_0 \). Then, assuming \( |M_0| = |M_1| \), the term \( E[P^2, Z] \) is also zero, as the variance of \( P \) is the same over both \( M_0 \) and \( M_1 \). Finally, as \( Z^2 = 1 \), the terms \( E[P^2, Z^2] \) and \( E[P^2]E[Z^2] \) are equal and cancel each other out. This results in \( I(P; Z) \approx \frac{1}{12} E[P^3, Z] = \frac{1}{b} \Delta_{\text{DoA}} \).

### V. Keycak

**Keycak** is a function with variable-length input and arbitrary-length output based on the sponge construction \[24\]. In this construction, a permutation \( f \) with \( b \) input/output bits
is iterated. First, the input padded and its blocks are absorbed sequentially into the state, with a simple XOR operation. Then, the output is squeezed from the state block by block. The size of the blocks is denoted by $r$ and called the bitrate. The remaining number of bits $c = b - r$ is called the capacity and determines the security level of the function.

The simplest use case of a sponge function is to use it as a hash function by simply truncating the output. A MAC function can be built by taking as input the concatenation of a secret key and a message. For using it as a stream cipher, it suffices to input the secret key and a nonce and using the output as a key stream. More modes of use are described in [25] and [24].

Seven permutations, denoted Keccak-f[b], are defined with width $b = 256r$ ranging from 25 to 1600 bits, increasing in powers of two. The state of Keccak-f[b] is organized as a set of $5 \times 5 \times w$ bits with $(x, y, z)$ coordinates. The coordinates are always considered modulo 5 for $x$ and $y$ and modulo $w$ for $z$. Coordinates are taken modulo 5 for $x$ and $y$ and modulo $w$ for $z$. A row is a set of 5 bits with given $(y, z)$ coordinates, a column is a set of 5 bits with given $(x, z)$ coordinates and a slice is a set of 25 bits with given $z$ coordinate. A row is a set of 5 bits with given $(y, z)$ coordinates and a column is a set of 5 bits with given $(x, z)$ coordinates.

The round function of Keccak-f[b] consists of the following steps, which are only briefly summarized here. For more details, we refer to the specifications [25] and [24].

- $\theta$ is a linear mixing layer that adds a pattern depending solely on the parity of the columns of the state.
- $\rho$ and $\pi$ displace bits without altering their value. Jointly, their effect is denoted by $(x', y', z') = \rho_{\pi_{\rho}}(x, y, z)$, with $(x', y', z')$ a bit position before $\rho$ and $\pi$ and $(x, y, z)$ its coordinates afterward.
- $\chi$ is a degree-2 non-linear mapping that processes each row independently. It can be seen as the application of a translation-invariant 5-bit quadratic S-box:
  \[
  a_{(x,y,z)} \leftarrow a_{(x,y,z)} + (a_{(x+1,y,z)} + 1)a_{(x+2,y,z)}.
  \]
- $\iota$ adds a round constant.

A. The plain core

The plain core architecture instantiates the Keccak-f round function using combinatorial logic and keeps track of the state in a register [25]. At each clock cycle, the state value in the register is updated by applying the round function to it. Applying the permutation simply consists in performing as many clock cycles as there are rounds in Keccak-f. The absorbing of message blocks is implemented by an XOR stage at the input of the round function logic, which takes its input from a buffer. The round constants are handled by a simple finite state machine.

B. The three-share core

The three-share core architecture implements the secret sharing scheme technique presented in Section II-B to offer protection against CPA. The designers of Keccak show that three shares are sufficient thanks to the low degree of the nonlinear step in the Keccak-f round function $\chi$ [7]. Their architecture is a rather straightforward application of the secret sharing scheme technique to the plain core: it keeps the three shares of the state in separate registers and instantiates the three-share version of the Keccak-f round function in combinatorial logic. The linear layer $\lambda$ is instantiated three times, operating on each share separately. It implements the nonlinear step $\chi$ by three separate logic blocks each implementing a function $a = \chi'(b, c)$ defined by:

\[
\begin{align*}
    a_{(x,y,z)} &\leftarrow b_{(x,y,z)} + (b_{(x+1,y,z)} + 1)b_{(x+2,y,z)} \\
    &\quad + b_{(x+1,y,z)}c_{(x+2,y,z)} + c_{(x+1,y,z)}b_{(x+2,y,z)}.
\end{align*}
\]

Each block computes $\chi'$ for a share taking as input the two other shares. The message blocks are applied to a single share at the input of the round logic and the round constants are applied to a single share at the output of the round logic.

The three-share core is illustrated in Figure 2.

![Fig. 2. The three-share core](image)

Before processing, the three shares are generated from a random source. As the initial state of Keccak is equal to zero, two of the three shares can be generated randomly and the third is computed as the sum (in GF(2)) of the other two. Before processing, the three shares are initialized consistent with the zero value: two of the three shares are generated randomly and the third is computed as their XOR.

C. The attack point

A DPA attack on a Keccak core consists in making it run repeatedly, taking each time the same secret key $K$
and a chosen (or known) input block $M$, and recording the power computation in a so-called trace. The traces can be identified by the corresponding value of $M$ and we can use the measured power consumption $P(M, K)$ to determine the unknown secret value $K$.

In typical cases where a sponge function is used in combination with a secret key, the input is prefixed with the key. The input of the Keccak-$f$ permutation under attack is $K + M$, with $K$ the state of the sponge function after absorbing the key. The knowledge of $K$ is sufficient to, e.g., forge MACs or produce key streams with arbitrary nonces. This setting corresponds with the case that the key consists of (or is padded as) exactly one block and the attacker targets $K = \text{Keccak-}f(\text{Key} || 0^r)$ with $\text{Key} \in \text{GF}(2)^{10}\ell$. After retrieving $K$, the attacker can recover the absorbed key by inverting Keccak-$f$. The second block brings message bits and $K + M$ is the input to Keccak-$f$, with $M$ spanning the first $r$ bits.

A key that is shorter than the rate would give rise to a slightly different setting, in which the key bits and message bits sit together in a first block $\text{Key} || 0^r$, with $M$ spanning $r - |\text{Key}|$ bits. We do not consider this setting in the remainder of this section, but its analysis is very similar to the one we present.

Both in the plain core and in the three-share core, the output of the round function is stored in registers. The round function of Keccak-$f$ is quadratic, hence we can apply the definitions of Section III and target the power consumed when storing the result in the registers. Subsequent rounds would make the dependencies between key and message bits more difficult to deal with.

Let $B$ be the output of the first round, $\kappa = \pi(\rho(\theta(K)))$ and $\mu = \rho(\theta(M))$. Then the output bit at coordinates $(x, y, z)$ is

$$B_{(x,y,z)} = RC_{(x,y,z)} + s_{(x,y,z)} + \mu_{(x,y,z)} + (s_{(x+1,y,z)} + \mu_{(x+1,y,z)}) + 1(\kappa_{(x+2,y,z)} + \mu_{(x+2,y,z)}).$$

The function describing the register activity is $d = K + B$ and the selection function is derived as in Section III:

$$s_{(x,y,z)}(M, \kappa^*(x+1,y,z) + \kappa^*(x+2,y,z)) =$$

$$= \mu_{(x,y,z)} + \mu_{(x+2,y,z)} + \mu_{(x+1,y,z)}\mu_{(x+2,y,z)} + \kappa^*(x+1,y,z)\mu_{(x+2,y,z)} + \kappa^*(x+2,y,z)\mu_{(x+1,y,z)}. \tag{8}$$

If the hypothesis on $\kappa^*(x+1,y,z)$ and $\kappa^*(x+2,y,z)$ is correct, then $s_{(x,y,z)}(M, \kappa^*(x+1,y,z) + \kappa^*(x+2,y,z))$ differs from $d_{(x,y,z)}(M, K)$ by a fixed (unknown) term. However, if the hypothesis is incorrect, $(\kappa_{(x+1,y,z)} + \kappa^*(x+2,y,z)) \neq (0,0)$, then $s_{(x,y,z)}(M, \kappa^*(x+1,y,z) + \kappa^*(x+2,y,z))$ is independent from $d_{(x,y,z)}(M, K)$.

Regarding the independence of the activity function between different coordinates, their sum $s_{(x_1,y_1,z_1)}(M, K) + s_{(x_2,y_2,z_2)}(M, K)$ is not necessarily balanced for all pairs $(x_1,y_1,z_1) \neq (x_2,y_2,z_2)$ because $M$ is zero in the inner part of the state. However, we can prove the following theorem.

**Theorem** Let $(x'_i, y'_i, z'_i) \in \text{GF}(2)$ for $i \in \{1, 2\}$. Then $s_{(x_1,y_1,z_1)}(M, K)$ and $s_{(x_2,y_2,z_2)}(M, K)$ are independent unless $(x'_1, z'_1) = (x'_2, z'_2)$ and $(5y_i + x_i)w + z_i \geq r$ for $i \in \{1, 2\}$ (i.e., the bits come from the same column and both from the last $c$ bits).

**Proof:** Two Boolean functions are independent if their bitwise sum is balanced, so we need to prove that the function $s_{(x_1,y_1,z_1)}(M, K) + s_{(x_2,y_2,z_2)}(M, K)$ is balanced if the conditions of the theorem are satisfied. Let us trace the bits of $M$ to those of $\mu$ through $\pi \circ \rho \circ \theta$. The bits of $M$ are balanced in the outer part of the state $((5y'_i + x'_i)w + z'_i < r)$ and zero in the inner part of the state. Let $\mu' = \theta(M)$. If the rate is not below $5w$, all bits of $\mu'$ are balanced. However, in the outer part of the state, bits in the same column are equal due to the fact that $\theta$ treats the bits of a column in the same way. So the bitwise sum of two such bits will be zero. The bit transposition $\pi \circ \rho$ just moves bits to other positions, realizing the mapping from $(x'_1, y'_1, z'_1)$ to $(x_1, y_1, z_1)$ specified in the theorem. If $(x'_1, y'_1, z'_1)$ and $(x'_2, y'_2, z'_2)$ are in different columns or not both in the outer part, the function $s_{(x_1,y_1,z_1)}(M, K) + s_{(x_2,y_2,z_2)}(M, K)$ will exhibit two balanced terms that do not cancel out. It may be that $(x_1, y_1, z_1)$ is equal to $(x_2 + 1, y_2, z_2)$ or $(x_2 + 2, y_2, z_2)$, possibly canceling out terms. In that case the balanced term $\mu_{(x_2,y_2,z_2)}$ remains, as $(x_2, y_2, z_2)$ can then not be equal to $(x_1 + 1, y_1, z_1)$ or $(x_1 + 2, y_1, z_1)$. In a similar way, equality of $(x_2, y_2, z_2)$ to $(x_1 + 1, y_1, z_1)$ or $(x_1 + 2, y_1, z_1)$ leaves $\mu_{(x_1,y_1,z_1)}$ as balanced terms.

For a focus point not covered by this theorem, there may be other activity bits correlated with the selection function. The contribution of these bits will affect the DoM values for $M_0$ and $M_1$, both for the correct and the incorrect key hypotheses and hence impact the success probability.

For focus points covered by this theorem, there may be pairs of bit positions $(x_1, y_1, z_1)$ and $(x_2, y_2, z_2)$ with activity functions that are correlated and their contributions to the variance is not independent. It is easy to see that if the correlation between the bits is $c$, their sum contributes $2(1+c)$ to the variance instead of 2. Making the plausible assumption that the sign is (near)-balanced over all non-zero correlation, allows us to predict the total variance by $b - 1$ for the correct and $b$ for the incorrect hypotheses.

**D. Experimental results for the plain core**

We performed experiments for the DoM distinguisher on Keccak variants based on all seven Keccak-$f$ versions, namely Keccak[$r = 16 \times 2^\ell, c = 9 \times 2^\ell$]. The rate ranges from 1024 bits for Keccak-$f[16|00]$ down to 16 bits for Keccak-$f[25]$. The secret value to recover is as in Section V-C.

The experiment assumes that the power consumed is equal to the number of bits that flip in the $b$-bit register that initially contains the output of the previous call to Keccak-$f$ and then the state after the first round. For each width of Keccak-$f$, we conducted distinguishing experiments for 1000 different secret key values and for each key value we took traces for a large number of $r$-bit message blocks. The focus point in this experiment is $(0, 0, 0)$, which is not in the last $c$ bits (see Theorem in Section V-C).
Figure 3 reports on the outcome of our experiments. It plots the success rate, being the ratio of correctly selected hypotheses among all keys, as a function of the number of traces. It also plots the theoretical success probability \(G_h(b/M)\), obtained in Section IV-B for Keccak-f[1600], which the experimental results follow closely. The success probability starts at \(1/2\), as the correct hypothesis could be any value, and then grows very close to 1 when \(|M|\) reaches \(20b\). It follows the scaling suggested by the Kullback-Leibler distance.

E. Experimental results for the three-share core

We performed experiments for the DoA distinguisher on Keccak\([r = 16, c = 9]\), Keccak\([r = 32, c = 18]\) and Keccak\([r = 64, c = 36]\). These toy primitives already require a high number of measurements when protected by three shares and can give an idea of the adequacy of the model and on the scaling. We again assume that the power consumed is equal to the number of bits that flip in the registers, except that the register now contains \(3b\) bits. For each instance, we conducted distinguishing experiments for 1000 different secret key values and for each key value we took traces for large numbers of message blocks. The focus point is again \((0, 0, 0)\).

Figure 4 shows the success rate as a function of the number of traces for Keccak\([r = 16, c = 9]\), Keccak\([r = 32, c = 18]\) and Keccak\([r = 64, c = 36]\), both for DoA and cumulant MIA. Clearly, DoA is more efficient than cumulant MIA, confirming the intuition behind the DoA distinguisher.

The figure also plots the theoretical success probability \(G_h(9b^3/(2|M|))\) that is confirmed by the experimentally obtained values for DoA. In this case too it follows the scaling suggested by the Kullback-Leibler distance.

These simulations are for toy versions of Keccak-f. For instance actually usable in practice, e.g., from lightweight Keccak-f[200] till Keccak-f[1600] used in SHA-3, the scaling suggests a tremendous number of traces (from billions to hundreds of billions) needed to recover secret key bits.

VI. Conclusions

We presented selection functions and distinguishers dedicated to protected and unprotected hardware implementations, for any cryptographic primitive implemented as a sequence of quadratic functions. We analyzed such implementations under the presence of algorithmic noise, in particular in terms of required number of traces and success probabilities.

This analysis shows that a three-share implementation does provide security against DPA and MIA. The number of traces needed to distinguish the correct key bits grows with the third power of the algorithmic noise variance, i.e., induced by the bits being computed. This suggests that masking with a secret-sharing scheme on quadratic functions efficiently provides security that scales similarly as third-order DPA. This applies readily to the three-share architecture proposed in [7].

Our power model does not deal with glitches in the combinatorial logic. They introduce second-order effects [3], [4], although we expect them to be of lower amplitude than the register switching activity. An extension of this work includes the analysis of their effect on the success probability and adapting the results of Section IV to this seems straightforward.

REFERENCES


1 Introduction

Homomorphic encryption schemes support computation on encrypted data. Such schemes are of particular interest for various applications, such as Outsourcing of Computation [13], Electronic Voting [5], Private Information Retrieval [21], Oblivious Polynomial Evaluation [23], or Multiparty Computation [8].

The most prominent homomorphic encryption schemes, e.g., ElGamal [12], Paillier [24], Damgård-Jurik [11], are homomorphic with respect to a single algebraic operation. That is, the plaintext space forms a group \((G, \circ)\) and, given encryptions of \(m, m' \in G\), one can efficiently and securely compute an encryption of \(m \circ m'\) without revealing \(m\) and \(m'\). We will call such schemes group homomorphic encryption schemes. Although fully homomorphic schemes [14, 15, 29, 31], i.e., schemes that allow one to evaluate any circuit over encrypted data without being able to decrypt, provide a much higher flexibility compared to group homomorphic schemes, the investigation of the latter still represents an important research topic:

1. The majority of existing homomorphic schemes are group homomorphic and there are still many open questions regarding these schemes.
2. For practical applications there is currently no alternative to such schemes.\(^4\)
3. Many constructions of schemes that support more than a single algebraic operation are in particular group homomorphic as well (e.g., [4, 7]).
4. A comprehensive understanding of group homomorphic schemes leads to a better understanding of schemes that are homomorphic in a more general sense, since the underlying structures are very similar.

Over the last decades, a variety of different approaches (and according hardness assumptions and proofs of security) has been investigated for constructing group homomorphic schemes, such as the Quadratic Residuality Problem [18], the Higher Residuality Problem [5], the Decisional Diffie-Hellman Problem [12, 26], and the Decisional Composite Residuality Class Problem [24, 11]. All these schemes have been investigated separately, resulting in the fact that some of them are better understood than others. In particular, much effort has been devoted to proving existing homomorphic schemes IND-CCA1 secure (being the highest possible security level for a homomorphic scheme). For example, since the introduction of Damgård’s ElGamal [10] in 1991, many works addressed the problem of characterizing its IND-CCA1 security [17, 32]. Similarly, while the IND-CPA security of ElGamal is known for a while [30], the quest for a characterization of its IND-CCA1 security has been in the focus for many years. Only in 2010, the quest concerning these two schemes has finally found an end due to [22]. Finding similar characterizations for remaining homomorphic schemes, e.g., Paillier’s scheme, is still an open problem.

\(^4\) For example, one of the recent implementations [16] of [15] states that the largest variant (for which a security level similar to RSA-1024 is assumed) has a public key of 2.4 GB size and requires about 30 minutes to complete certain operations.
2 Our Contributions

In [1], we present a unified view both in terms of security and design on group homomorphic encryption schemes among which the most prominent encryption schemes such as ElGamal and Paillier can be found. On the one hand, this helps to access the kind of challenges mentioned above more easily (and in fact, to answer open questions) and on the other hand provides a systematic procedure for designing new schemes based on given problems. More precisely, we construct an abstract scheme that represents a large class of group homomorphic encryption schemes (that particularly encompasses all existing schemes) and prove its IND-CCA1 security equivalent to the hardness of a new abstract problem, called the Splitting Oracle-Assisted Subgroup Membership Problem (SOAP), meaning that every scheme occurs as an instantiation of the abstract scheme being IND-CCA1 secure if and only if the according instantiation of SOAP is hard. A characterization of IND-CPA security through an abstract problem, called the Subgroup Membership Problem (SMP) is an immediate byproduct of our results.

As a direct implementation, we can apply our abstract security characterizations to existing homomorphic schemes by looking at the according instantiations, to deal with the IND-CCA1 (resp. IND-CPA) security of these schemes, or to verify existing IND-CCA1 (resp. IND-CPA) security proofs.

Furthermore, our characterizations allow us to derive impossibility results. For instance, we show that there cannot exist an IND-CPA secure group homomorphic encryption scheme when the ciphertexts form a linear subspace of $\mathbb{F}^n$ for some prime field $\mathbb{F}$. This partly answers an open question whether using linear codes as ciphertext spaces yield more efficient constructions (see [15]). More importantly, in [3], we prove the general impossibility of group homomorphic encryption in the quantum world. By this we mean that for any given group homomorphic encryption scheme, we can construct a quantum adversary that efficiently breaks its IND-CPA security.

Another utilization of our results is a systematic approach for constructing provably secure group homomorphic schemes [1]. By using our abstract scheme and a concrete instantiation of SOAP resp. SMP, one can directly specify a homomorphic scheme that is IND-CCA1 resp. IND-CPA secure if and only if the respective problem is hard.

As a first example, we consider the $k$-linear problem [28] which is an alternative to DDH in groups where DDH is easy, e.g., in bilinear groups [20]. Since its introduction, it is a challenge to construct cryptographic protocols whose security is based on the $k$-linear problem (e.g., [6]). Following this task, we present the first homomorphic scheme that is based on the $k$-linear problem for $k > 2$ ($k = 1$ is ElGamal [12], $k = 2$ is Linear Encryption [6]). In addition, we introduce a new $k$-problem (an instantiation of SOAP) that we prove to be hard in the generic group model and to have the same progressive property as the $k$-linear problem. This result might be of independent interest as it can be used to construct new cryptographic protocols with unique features. For instance, we give the first homomorphic scheme that can be instantiated with groups where DDH is easy (e.g., bilinear groups) and is nevertheless provably secure in terms of IND-CCA1 due to the new $k$-problem.

Finally, in [2], we demonstrate the impact of our results to schemes that are homomorphic in a more general sense (such as fully homomorphic schemes), we first identify a certain structure that all existing homomorphic schemes have in common (so-called Shift-Type Homomorphic Encryption). In fact, this structure is the key ingredient to our IND-CPA characterization of group homomorphic schemes and allows us to extend our results to more general cases such as fully homomorphic schemes. All currently known fully homomorphic schemes arise by applying a technique that was introduced by Gentry [15] to an underlying (so-called bootstrappable) scheme. Interestingly enough, we also show that the IND-CPA security of such fully homomorphic schemes is equivalent to the 1-way KDM security of the underlying scheme (which roughly means that
the scheme remains secure even if the adversary gets to see the bits of the secret key encrypted under the corresponding public key).

3 Separation from Related Work

Aside from the related work that we have already mentioned in the previous sections, there is a substantial number of papers on the construction of IND-CPA (respectively, IND-CCA1, IND-CCA2) secure encryption schemes. In this regard, we would particularly like to mention the work by Cramer and Shoup [9] who give a generic construction of IND-CPA (respectively, IND-CCA1, IND-CCA2) secure encryption schemes through smooth (respectively, 1-universal, 2-universal) hash proof systems. Furthermore, Peikert and Waters [25] introduce the notion of Lossy Trapdoor Functions (LTFs) and give a generic construction of IND-CCA1 secure encryption schemes from such functions, while Hemenway and Ostrovsky [19] give a generic construction of IND-CCA1 secure group homomorphic encryption schemes through homomorphic hash proof systems, which are known to be constructable, e.g., from the Quadratic Residuosity Problem, the Decisional Diffie-Hellman Problem or the Decisional Composite Residuosity Problem. A somewhat different approach to the construction of IND-CCA1 secure group homomorphic encryption was presented by Prabhakaran and Rosulek [27]. Therein, they build group homomorphic encryption schemes that are secure in an even stronger sense than just being IND-CCA1, namely “homomorphic-CCA” secure.

All these works have in common that they build IND-CCA1 secure schemes from non-interactive assumptions, while we show the IND-CCA1 security equivalent to the hardness of SOAP which then naturally has to be an interactive problem, as IND-CCA1 is. Therefore, we stress that we give characterizations of the security of group homomorphic schemes. For all the above mentioned schemes this means that the underlying non-interactive assumption either implies SOAP, or is equivalent to it. In the former case, breaking the underlying assumption would not necessarily break the security of the scheme in question as it is actually equivalent to SOAP which might still be a hard problem. We do not give a generic construction of IND-CCA1 secure group homomorphic schemes from non-interactive assumptions. Concerning IND-CPA security on the other hand, this is a completely different story, as we propose a generic way to construct IND-CPA secure group homomorphic schemes from non-interactive assumptions. The latter is due to the fact that the corresponding SMP instance is always non-interactive.

References

Abstract. The indifferentiability framework of Maurer, Renner, and Holenstein (MRH) has gained immense popularity in recent years and has proved to be a powerful way to argue security of cryptosystems that enjoy proofs in the random oracle model. Recently, however, Ristenpart, Shacham, and Shrimpton (RSS) showed that the composition theorem of MRH has a more limited scope than originally thought, and that extending its scope required the introduction of reset-indifferentiability, a notion which no practical domain extenders satisfy with respect to random oracles.

In light of the results of RSS, we set out to rigorously tackle the specifics of indifferentiability and reset-indifferentiability by viewing the notions as special cases of a more general definition. Our contributions are twofold. Firstly, we provide the necessary formalism to refine the notion of indifferentiability regarding composition. By formalizing the definition of stage minimal games we expose new notions lying in between regular indifferentiability (MRH) and reset-indifferentiability (RSS).

Secondly, we answer the open problem of RSS by showing that it is impossible to build any domain extender which is reset-indifferentiable from a random oracle. This result formally confirms the intuition that reset-indifferentiability is too strong of a notion to be satisfied by any hash function. As a consequence we look at the weaker notion of single-reset-indifferentiability, yet there as well we demonstrate that there are no “meaningful” domain extenders which satisfy this notion. Not all is lost though, as we also view indifferentiability in a more general setting and point out the possibility for different variants of indifferentiability.
BLAKE2: simpler, smaller, fast as MD5

2013.02.21

https://blake2.net

Jean-Philippe Aumasson (@aumasson)
Samuel Neves (@sevenps)
Zooko Wilcox-O’Hearn (@zooko)
Christian Winnerlein (@codesinchaos)

Abstract. We present the cryptographic hash function BLAKE2, an improved version of the SHA-3 finalist BLAKE optimized for speed in software. Target applications include cloud storage, intrusion detection, or version control systems. BLAKE2 comes in two main flavors: BLAKE2b is optimized for 64-bit platforms, and BLAKE2s for smaller architectures. On 64-bit platforms, BLAKE2 is often faster than MD5, yet provides security similar to that of SHA-3. We specify parallel versions BLAKE2bp and BLAKE2sp that are up to 4 and 8 times faster, by taking advantage of SIMD and/or multiple cores. BLAKE2 has more benefits than just speed: BLAKE2 uses up to 32% less RAM than BLAKE, and comes with a comprehensive tree-hashing mode as well as an efficient MAC mode.

1 Introduction

The SHA-3 Competition succeeded in selecting a hash function that complements SHA-2 and is much faster than SHA-2 in hardware [11]. There is nevertheless a demand for fast software hashing for applications such as integrity checking and deduplication in filesystems and cloud storage, host-based intrusion detection, version control systems, or secure boot schemes.

SHA-3 does not fit these needs well—for example on Qualcomm’s Krait microarchitecture\(^1\) SHA-3-256 takes about 20% longer to hash a message than SHA-256 does, and on Intel’s Ivy Bridge microarchitecture\(^2\) SHA-3-512 takes about twice as long as SHA-512 does.

Many systems use faster algorithms like MD5, SHA-1, or a custom function to meet their speed requirements, even though those functions may be insecure. MD5 is famously vulnerable to collision and length-extension attacks [13, 23], but it is 2.53 times as fast as SHA-256 on Ivy Bridge and 2.98 times as fast as SHA-256 on Krait.

Despite MD5’s significant security flaws, it continues to be among the most widely-used algorithms for file identification and data integrity. To choose just a handful of examples, the OpenStack cloud storage system [22], the popular version control system Perforce, and the recent object storage system used internally in AOL [20] all rely on MD5 for data integrity. The venerable md5sum Unix tool remains one of the most widely-used tools for data integrity checking. The Sun/Oracle ZFS filesystem includes the option of using SHA-256 for data integrity, but the default configuration is to instead use a non-cryptographic 256-bit checksum,

\(^1\)See http://bench.cr.yp.to/results-hash.html#armeabi-h9dragon, accessed 7 Dec 2012.

Some SHA-3 finalists outperform SHA-2 in software: for example, on Ivy Bridge BLAKE-512 is 1.41 times as fast as SHA-512, and BLAKE-256 is 1.70 times as fast as SHA-256. BLAKE-512 reaches 5.76 cycles per byte, or approximately 579 mebibytes per second, against 411 for SHA-512, on a CPU clocked at 3.5GHz.

BLAKE thus appears to be a good candidate for fast software hashing. Its security was evaluated by NIST in the SHA-3 process as having a “very large security margin”, and the cryptanalysis published on BLAKE was noted as having “a great deal of depth” (see §4).

But as observed by Preneel [21], its design “reflects the state of the art in October 2008”; since then, and after extensive cryptanalysis, we have a better understanding of BLAKE’s security and efficiency properties. We therefore introduce BLAKE2, an improved BLAKE with the following properties:

- **Faster than MD5** on 64-bit Intel platforms
- **32% less RAM** required than BLAKE
- Direct support, with no overhead, of
  - **Parallelism** for many-times faster hashing on multicore or SIMD CPUs
  - **Tree hashing** for incremental update or verification of large files
  - **Prefix-MAC** for authentication that is simpler and faster than HMAC
  - **Personalization** for defining a unique hash function for each application
- **Minimal padding**, which is faster and simpler to implement
2 Description of BLAKE2

BLAKE2 comes in two flavors:

- **BLAKE2b** (or just BLAKE2) is optimized for **64-bit platforms**—including NEON-enabled ARMs—and produces digests of any size between 1 and 64 bytes.

- **BLAKE2s** is optimized for **8- to 32-bit platforms**, and produces digests of any size between 1 and 32 bytes.

Both are believed to be highly secure and have good performance on any platform, software or hardware. Each one is portable to any CPU, but can be up twice as fast when used on the CPU size for which it is optimized; for example, on a Tegra 2 (32-bit ARMv7-based SoC) BLAKE2s is expected to be about twice as fast as BLAKE2b, whereas on an AMD A10-5800K (64-bit, Piledriver microarchitecture), BLAKE2b is expected to be more than 1.5 times as fast as BLAKE2s.

Since BLAKE2 is very similar to BLAKE, we first describe the changes introduced with BLAKE2. The specification is complete with elements shared with BLAKE in Appendix A. We refer to [https://131002.net/blake](https://131002.net/blake) for a complete specification of BLAKE.

2.1 Fewer rounds

BLAKE2b does **12 rounds**, and BLAKE2s does **10 rounds**, against 16 and 14 respectively for BLAKE. Based on the security analysis performed so far, and on reasonable assumptions on future progress, it is unlikely that 16 and 14 rounds are meaningfully more secure than 12 and 10 rounds. Recall that the initial BLAKE submission [1] had 14 and 10 rounds, respectively, and that the later increase [2] was motivated by the high speed of BLAKE.

This change gives a direct speed-up of about 25% and 29%, respectively, on long data. Speed on short data also significantly improves.

2.2 Rotations optimized for speed

The G function of BLAKE-512 performs four 64-bit word rotations of respectively 32, 25, 16, and 11 bits. **BLAKE2b replaces 25 with 24, and 11 with 63:**

- Using a 24-bit rotation allows SSSE3-capable CPUs to perform two rotations in parallel with a single SIMD instruction (namely, `pshufb`), whereas two shifts plus a logical OR are required for a rotation of 25 bits. This reduces the arithmetic cost of the G function, in recent Intel CPUs, from 18 single cycle instructions to 16 instructions, a 12% decrease.
A 63-bit rotation can be implemented as an addition (doubling) and a shift followed by a logical OR. This provides a slight speed-up on platforms where addition and shift can be realized in parallel but not two shifts (i.e., some recent Intel CPUs). Additionally, since a rotation right by 63 is equal to a rotation left by 1, this may be slightly faster in some architectures where 1 is treated as a special case.

No platform suffers from these changes. For an in-depth analysis of optimized implementations of rotations, we refer to a previous work by two co-designers of BLAKE2 [18].

Past experiments by the BLAKE designers as well as third parties suggest that known differential attacks are unlikely to get significantly better, nor worse (cf. §4).

2.3 Minimal padding and finalization flags

BLAKE2 pads the last data block if and only if necessary, with null bytes. If the data length is a multiple of the block length, no padding byte is added.

BLAKE2 introduces finalization flags \( f_0 \) and \( f_1 \), as auxiliary inputs to the compression function:

- The security functionality of the padding is transferred to a finalization flag \( f_0 \), a word set to \( ff...ff \) if the block processed is the last, and to \( 00...00 \) otherwise. The flag \( f_0 \) is 64-bit for BLAKE2b, and 32-bit for BLAKE2s.
- A second finalization flag \( f_1 \) is used to signal the last node of a layer in tree-hashing modes. When processing the last block—that is, when \( f_0 \) is \( ff...ff \)—the flag \( f_1 \) is also set to \( ff...ff \) if the node considered is the last, and to \( 00...00 \) otherwise.

The finalization flags are processed by the compression function as described in §2.4.

BLAKE2s thus supports hashing of data of at most \( 2^{64} - 1 \) bytes, that is, almost 16 exabytes (the amount of memory addressable by 64-bit processors). The upper bound for BLAKE2b is even more ridiculous, with up to \( 2^{128} - 1 \) bytes supported.

2.4 Fewer constants

Whereas BLAKE used 8 word constants as IV plus 16 word constants for use in the compression function, BLAKE2 uses a total of 8 word constants, instead of 24. This saves 128 ROM bytes and 128 RAM bytes in BLAKE2b implementations, and 64 ROM bytes and 64 RAM bytes in BLAKE2s implementations.

The compression function initialization phase is modified to:

\[
\begin{pmatrix}
  v_0 & v_1 & v_2 & v_3 \\
  v_4 & v_5 & v_6 & v_7 \\
  v_8 & v_9 & v_{10} & v_{11} \\
  v_{12} & v_{13} & v_{14} & v_{15}
\end{pmatrix}
\leftarrow
\begin{pmatrix}
  h_0 & h_1 & h_2 & h_3 \\
  h_4 & h_5 & h_6 & h_7 \\
  IV_0 & IV_1 & IV_2 & IV_3 \\
  t_0 \oplus IV_4 & t_1 \oplus IV_5 & f_0 \oplus IV_6 & f_1 \oplus IV_7
\end{pmatrix}
\]

Note the introduction of finalization flags \( f_0 \) and \( f_1 \), in place of BLAKE’s redundant counter.
The G function of BLAKE2b is defined as:

\[
\begin{align*}
    a & \leftarrow a + b + m \sigma_r (2^i) \\
    d & \leftarrow (d \oplus a) \gg 32 \\
    c & \leftarrow c + d \\
    b & \leftarrow (b \oplus c) \gg 24 \\
    a & \leftarrow a + b + m \sigma_r (2^i + 1) \\
    d & \leftarrow (d \oplus a) \gg 16 \\
    c & \leftarrow c + d \\
    b & \leftarrow (b \oplus c) \gg 63
\end{align*}
\]

Note the aforementioned change of rotation counts.

Similarly, the G function of BLAKE2s is simplified to:

\[
\begin{align*}
    a & \leftarrow a + b + m \sigma_r (2^i) \\
    d & \leftarrow (d \oplus a) \gg 16 \\
    c & \leftarrow c + d \\
    b & \leftarrow (b \oplus c) \gg 12 \\
    a & \leftarrow a + b + m \sigma_r (2^i + 1) \\
    d & \leftarrow (d \oplus a) \gg 8 \\
    c & \leftarrow c + d \\
    b & \leftarrow (b \oplus c) \gg 7
\end{align*}
\]

Omitting the constants in G gives an algorithm similar to the (unattacked) BLAZE toy version\(^3\). Constants in G initially aimed to guarantee early propagation of carries, but it turned out that the benefits (if any) are not worth the performance penalty. This change saves two xors and two loads per G, that is, 16% of the total arithmetic (addition and xor) instructions.

### 2.5 Little-endian

BLAKE, like SHA-1 and SHA-2, parses data blocks in the big-endian byte order. Like MD5, **BLAKE2 is little-endian**, because the large majority of target platforms is little-endian (AMD and Intel desktop processors, most mainstream ARM systems). Switching to little-endian may provide a slight speed-up, and often simplifies implementations.

Note that in BLAKE, the counter \( t \) is composed of two words \( t_0 \) and \( t_1 \), where \( t_0 \) holds the least significant bits of the integer encoded. This little-endian convention is preserved in BLAKE2.

### 2.6 Counter in bytes

The counter \( t \) counts **bytes rather than bits**. This simplifies implementations and reduce the risk of error, since target applications measure data volumes in bytes rather than bits. This change increases the amount of data that can be processed by 8 times, compared to BLAKE.

---

\(^3\)See [https://131002.net/blake/toyblake.pdf](https://131002.net/blake/toyblake.pdf).
2.7 Salt processing

BLAKE’s predecessor LAKE [3] introduced the built-in support for a salt, to simplify the use of randomized hashing within digital signature schemes.

In BLAKE2 the salt is processed as a one-time input to the hash function, through the IV, rather than as an input to each compression function. This simplifies the compression function, and saves a few instructions as well as a few bytes in RAM, since the salt doesn’t have to be stored anymore. Using salt-independent compression functions has only negligible, and very theoretical, impact on security, as discussed in §4.

2.8 Parameter block

The parameter block of BLAKE2 is xored with the IV prior to the processing of the first data block. It encodes parameters for secure tree hashing, as well as key length (in keyed mode) and digest length.

The parameters are described below, and the block structure is shown in Tables 1 and 2:

- General parameters:
  - Digest byte length (1 byte): an integer in $[1, 64]$ for BLAKE2b, in $[1, 32]$ for BLAKE2s
  - Key byte length (1 byte): an integer in $[0, 64]$ for BLAKE2b, in $[0, 32]$ for BLAKE2s (set to 0 if no key is used)
  - Salt (16 or 8 bytes): an arbitrary string of 16 bytes for BLAKE2b, and 8 bytes for BLAKE2s (set to all-NULL by default)
  - Personalization (16 or 8 bytes): an arbitrary string of 16 bytes for BLAKE2b, and 8 bytes for BLAKE2s (set to all-NULL by default)

- Tree hashing parameters:
  - Fanout (1 byte): an integer in $[0, 255]$ (set to 0 if unlimited, and to 1 only in sequential mode)
  - Maximal depth (1 byte): an integer in $[1, 255]$ (set to 255 if unlimited, and to 1 only in sequential mode)
  - Leaf maximal byte length (4 bytes): an integer in $[0, 2^{32} - 1]$, that is, up to 4 GiB (set to 0 if unlimited, or in sequential mode)
  - Node offset (8 or 6 bytes): an integer in $[0, 2^{64} - 1]$ for BLAKE2b, and in $[0, 2^{48} - 1]$ for BLAKE2s (set to 0 for the first, leftmost, leaf, or in sequential mode)
  - Node depth (1 byte): an integer in $[0, 255]$ (set to 0 for the leaves, or in sequential mode)
  - Inner hash byte length (1 byte): an integer in $[0, 64]$ for BLAKE2b, and in $[0, 32]$ for BLAKE2s (set to 0 in sequential mode)

This is 50 bytes in total for BLAKE2b, and 32 bytes for BLAKE2s. Any bytes left are reserved for future and/or application-specific use, and are NULL. Values spanning more than one byte are written in little-endian. Note that tree hashing may be keyed, in which case leaf instances hash the key followed by a number of bytes equal to (at most) the maximal leaf length.
<table>
<thead>
<tr>
<th>Offset</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Digest length</td>
<td>Key length</td>
<td>Fanout</td>
<td>Depth</td>
</tr>
<tr>
<td>4</td>
<td>Leaf length</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>Node offset</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>Node depth</td>
<td>Inner length</td>
<td>RFU</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>RFU</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>Salt</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>Personalization</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>44</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>60</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1: BLAKE2b parameter block structure (offsets in bytes).

<table>
<thead>
<tr>
<th>Offset</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Digest length</td>
<td>Key length</td>
<td>Fanout</td>
<td>Depth</td>
</tr>
<tr>
<td>4</td>
<td>Leaf length</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Node offset</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>Node offset (cont.)</td>
<td>Node depth</td>
<td>Inner length</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>Salt</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>Personalization</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>28</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2: BLAKE2s parameter block structure (offsets in bytes).
Example parameter block of BLAKE2b.  We take as example an instance of BLAKE2b with

- 64-byte digests, that is, with parameter digest length set to 40,
- a 256-bit key, that is, with the parameter key length set to 20,
- a salt set to the all-55 string,
- a personalization set to the all-ee string.

BLAKE2b hashes data sequentially, thus tree parameters are set to the value specified for the sequential mode: fanout and maximal depth are set to 01, leaf maximal length is set to 00000000, node offset is set to 0000000000000000, node depth and inner hash length are set to 00.

The parameter block for this instance of BLAKE2b is thus the following:

```
40200101 00000000 00000000 00000000 00000000 00000000 00000000 00000000
55555555 55555555 55555555 55555555 eeeeeeee eeeeeeee eeeeeeee eeeeeeee
```

Example parameter block of BLAKE2s.  We take as example an instance of BLAKE2s with

- 32-byte digests, that is, with parameter digest length set to 20,
- no key, that is, with the parameter key length set to 00,
- no salt, and no personalization, that is, with all respective bytes set NULL.

BLAKE2s hashes data sequentially, thus tree parameters are set to the value specified for the sequential mode: fanout and maximal depth are set to 01, leaf maximal length is set to 00000000, node offset is set to 0000000000000000, node depth and inner hash length are set to 00.

The parameter block for this instance of BLAKE2s is thus

```
20000101 00000000 00000000 00000000 00000000 00000000 00000000 00000000
```

2.9 Keyed hashing (MAC and PRF)

When keyed (that is, when the field key length is non-zero), BLAKE2 sets the first data block to the key padded with zeros, the second data block to the first block of the message, the third block to the second block of the message, etc. Note that the padded key is treated as arbitrary data, therefore:

- The counter \( t \) includes the 64 (or 128) bytes of the key block, regardless of the key length.
- When hashing the empty message with a key, BLAKE2b and BLAKE2s make only one call to the compression function.

\(^4\)For readability we add a space between each 4-byte block, however the value represented is a string of bytes, not a sequence of 4-byte words (which makes a difference with respect to endianness).
The main application of keyed BLAKE2 is as a message authentication code (MAC): BLAKE2 can be used securely in prefix-MAC mode, thanks to the indifferentiability property inherited from BLAKE [10]. Prefix-MAC is faster than HMAC, as it saves at least one call to the compression function. Keyed BLAKE2 can also be used to instantiate PRFs, for example within the PBKDF2 password hashing scheme.

2.10 Tree hashing

The parameter block supports arbitrary tree hashing modes, be it binary or ternary trees, arbitrary-depth updatable tree hashing or fixed-depth parallel hashing, etc. Note that, unlike other functions, BLAKE2 does not restrict the leaf length and the fanout to be powers of 2.

Basic mechanism. Informally, tree hashing processes chunks of data of “leaf length” bytes independently of each other, then combines the respective hashes using a tree structure wherein each node takes as input the concatenation of “fanout” hashes. The “node offset” and “node depth” parameters ensure that each invocation to the hash function (leaf of internal node) uses a different hash function. The finalization flag $f_1$ signals when a hash invocation is the last one at a given depth (where “last” is with respect to the node offset counter, for both leaves and intermediate nodes). The flag $f_1$ can only be non-zero for the last block compressed within a hash invocation, and the root node always has $f_1$ set to $ff...ff$.

The tree hashing mechanism is illustrated on Figures 2 and 3, which show layout of trees given different parameters and different input lengths. On those figures, octagons represent leaves (i.e., instances of the hash function processing input data), double-lined nodes (including leaves) are the last nodes of a layer, and thus have the flag $f_1$ set. Labels “i:j” indicate a node’s depth $i$ and offset $j$.

We refer to [7] for a comprehensive overview of secure tree hashing constructions.
Message parsing. Unless specified otherwise, we recommend that data be parsed as contiguous blocks: for example, if leaf length is 1024 bytes, then the first 1024-byte data block is processed by the leaf with offset 0, the subsequent 1024-byte data block is processed by the leaf with offset 1, etc.

Special cases. We highlight some special cases of tree hashing:

- **Unlimited fanout**: When the fanout is unlimited (parameter set to 0), then the root node hashes the concatenation of as many leaves are required to process the message, as shown on Figure 4. That is, the depth of the tree is always 2, regardless of the maximal depth parameter. Nevertheless, changing the maximal depth parameter changes the final hash value returned. We thus recommend to set the depth parameter to 2.

- **Dealing with saturated trees**: If a tree hashing instance has fanout \( f \geq 2 \), maximal depth \( d \geq 2 \), and leaf maximal length \( \ell \geq 1 \) bytes, then up to \( f^{d-1} \cdot \ell \) can be processed within a single tree. If more bytes have to be hashed, the fanout of the root node is extended to hash as many digests as necessary to respect the depth limit. This mechanism is illustrated on Figure 5. Note that if the maximal depth is 2, then the value does not affect the layout of the tree, which is identical to that of a tree hash with unlimited fanout (see Figure 4).

Generic tree parameters. Tree parameters supported by the parameter block allow for a wide range of implementation trade-offs, for example to efficiently support updatable hashing, which is typically an advantage when hashing many (small) chunks of data.

Although optimal performance will be reached by choosing the parameters specific to one’s application, we specify the following parameters for a **generic tree mode**: binary tree (i.e., fanout 2), unlimited depth, and leaves of 4 KiB (the typical size of a memory page).
Figure 4: Tree hashing with unbounded fanout (0) and arbitrary maximal depth (de facto, 2).

Figure 5: Tree hashing with maximal depth 3, fanout 2, but a root with larger fanout due to the reach of the maximal depth.

**Updatable hashing example.** Assume one has to provide a digest of a 1-tebibyte filesystem disk image that is updated every day. Instead of recomputing the digest by reading all the $2^{40}$ bytes, one can use our generic tree mode to implement an updatable hashing scheme:

1. Apply the generic tree mode, and store the $2^{40}/4096 = 2^{28}$ hashes from the leaves as well as the $2^{28} - 2$ intermediate hashes

2. When a leaf is changed, update the final digest by recomputing the 28 intermediate hashes

If BLAKE2b is used with intermediate hashes of 32 bytes, and that it hashes at a rate of 500 mebibytes per second, then step 1 takes approximately 35 minutes and generates about 16 gibibytes of intermediate data, whereas step 2 is instantaneous.

Note however that much less data may be stored: For many applications it is preferable to only store the intermediate hashes for larger pieces of data (without increasing the leaf size), which reduces memory requirement by only storing “higher” intermediate values. For example, storing intermediate values for 4 MiB chunks instead of all 4 KiB leaves reduces the storage to only 16 MiB. Indeed, using 4 KiB leaves allows applications with different piece sizes (as long as they are powers-of-two of at least 4 KiB) to produce the same root hash, while allowing them to make different granularity vs. storage trade-offs.
2.11 Parallel hashing: BLAKE2sp and BLAKE2bp

We specify 2 parallel hash functions (that is, with depth 2 and unlimited leaf length):

- **BLAKE2bp** runs 4 instances of BLAKE2b in parallel
- **BLAKE2sp** runs 8 instances of BLAKE2s in parallel

These functions use a different parsing rule than the default one in §§2.10: The first instance (node offset 0) hashes the message composed of the concatenation of all message blocks of index zero modulo 4; the second instance (node offset 1) hashes blocks of index 1 modulo 4, etc. Note that when the leaf length is unlimited, parsing the input as contiguous blocks would require the knowledge of the input length before any parallel operation, which is undesirable (e.g. when hashing a stream of data of undefined length, or a file received over a network).

When hashing one single large file, and when incrementability is not required, such parallel modes with unlimited leaves length seems the most appropriate, since

- They **minimize the computation overhead** by doing only one non-leaf call to the sequential hash function
- They **maximize the usage of the CPU** by keeping multiple cores and instruction pipelines busy simultaneously
- They require **realistic bandwidth and memory**

Note that parallel hashes have exactly the same interfaces as their sequential counterparts; for example, for BLAKE2bp one can define

```
blake2bp( uint8_t *out, uint8_t *in, uint8_t *key, int outlen, int inlen, int keylen )
```

Within a parallel hash, the same parameter block, except for the node offset, is used for all 4 or 8 instance of the sequential hash. For example, with no key, no salt, and no personalization, a version of BLAKE2sp producing 32-byte digests uses the eight following parameters blocks for the eight leaves:

```
20000802 00000000 00000000 00000020 00000000 00000000 00000000 00000000
20000802 00000000 01000000 00000020 00000000 00000000 00000000 00000000
20000802 00000000 02000000 00000020 00000000 00000000 00000000 00000000
20000802 00000000 03000000 00000020 00000000 00000000 00000000 00000000
20000802 00000000 04000000 00000020 00000000 00000000 00000000 00000000
20000802 00000000 05000000 00000020 00000000 00000000 00000000 00000000
20000802 00000000 06000000 00000020 00000000 00000000 00000000 00000000
20000802 00000000 07000000 00000020 00000000 00000000 00000000 00000000
```

Here the fanout is set to 08, the depth is set to 02, the leaf length is set to 00 (unlimited), the node depth is set to 00 (leaves), and the inner hash length is set to 20. The node offset ranges from 0 to 7, and is little-endian encoded to, e.g., the byte string 07000000 (representing the integer 00000007). Note that the last node (offset 7) sets the finalization flag $f_1$ in its last call to the compression function.
Finally, the root hash function in this version of BLAKE2sp uses the following parameter block (note the node depth set to 01):

```
  20000802 00000000 00000000 00000120 00000000 00000000 00000000 00000000
```

### 3 Performance

BLAKE2 is much faster than BLAKE, mainly due to its reduced number of rounds. On long messages, the BLAKE2b and BLAKE2s versions are expected to be approximately 25% and 29% faster, ignoring any savings from the absence of constants, optimized rotations, or little-endian conversion. The parallel versions BLAKE2bp and BLAKE2sp are expected to be 4 and 8 times faster than BLAKE2b and BLAKE2s on long messages, when implemented with multiple threads on a CPU with 4 or more cores (as most desktop and server processors: AMD FX-8150, Intel Core i5-2400S, etc.). Parallel hashing also benefits from advanced CPU technologies, as previously observed [19, §5.2].

#### 3.1 Why BLAKE2 is fast in software

BLAKE2, along with its parallel variant, can take advantage of the following architectural features, or combinations thereof:

**Instruction-level parallelism.** Most modern processors are superscalar, that is, able to run several instructions per cycle through pipelining, out-of-order execution, and other related techniques. BLAKE2 has a natural instruction parallelism of 4 instructions within the G function; processors that are able to handle more instruction-level parallelism can do so in BLAKE2bp, by interleaving independent compression function calls. Examples of processors with notorious amount of instruction parallelism are Intel's Core 2, i7, and Itanium or AMD's K10, Bulldozer, and Piledriver.

**SIMD instructions.** Initially designed to speed up multimedia tasks, many modern processors contain vector units, which enable SIMD processing of data. Again, BLAKE2 can take advantage of vector units not only in its G function, but also in tree modes (such as the mode proposed in §§2.11), by running several compression instances within vector registers. Microarchitectures with SIMD capabilities are found in recent Intel and AMD CPUs, NEON-extended ARM-based SoC, PowerPC and Cell CPUs.

**Multiple cores.** Limits in both semiconductor manufacturing processes, as well as instruction-level parallelism have driven CPU manufacturers towards yet another kind of coarse-grained parallelism, where multiple independent CPUs are placed inside the same die, and enable the programmer to get thread-level parallelism. While sequential BLAKE2 does not take advantage of this, the parallel mode described in §§2.11, and other tree modes, can run each intermediate hashing in its own thread. Candidate processors for this approach are recent Intel and AMD chips, the IBM Cell, and recent ARM, UltraSPARC and Loongson models.
### 3.2 64-bit CPUs

We have submitted optimized BLAKE2 implementations to eBACS [6], that take advantage of the AVX and XOP instruction sets. Table 3 reports the timings obtained in two key architectures: Intel’s Sandy Bridge (hydra7) and AMD’s Bulldozer (hydra6). The full set of results is available at [http://bench.cr.yp.to/results-hash.html](http://bench.cr.yp.to/results-hash.html). Furthermore, Table 4 reports the actual hashing speeds experienced on those CPUs, when running at their default frequency.

Compared to the best known timings for BLAKE [19],

- On Sandy Bridge, BLAKE2b is 71.99% faster than BLAKE-512, and BLAKE2s is 40.26% faster than BLAKE-256,
- On Bulldozer, BLAKE2b is 30.25% faster than BLAKE-512, and BLAKE2s is 43.78% faster than BLAKE-256.

Due to the lack of native rotation instructions on SIMD registers, the speedup of BLAKE2b is greater on the Intel processors, which benefit not only from the round reduction, but also from the easier-to-implement rotations.

On short messages, the speed advantage of the improved padding on BLAKE2 is quite noticeable. On Sandy Bridge, no other cryptographic hash function measured in SUPERCOP\(^5\) (including MD5 and MD4) is faster than BLAKE2s on 64-byte messages, while BLAKE2b is as fast as MD4.

As expected, the parallel versions provide a speed-up of a factor close to the parallelization degree: for example, using our tool b2sum on Bulldozer, the file `ubuntu-12.04-beta1-desktop-amd64.iso` is hashed in 1.16s with BLAKE2b, 0.33s with BLAKE2bp (that is, 3.51 times faster), in 1.72s with BLAKE2s, and in 0.27s with BLAKE2sp (that is, 6.37 times faster). Similarly, on Sandy Bridge BLAKE2bp is 3.76 times faster than BLAKE2b (1.58s vs 0.42s) hashing the same file, while BLAKE2sp is 3.68 times faster than BLAKE2s (2.21s vs 0.60s). Enabling hyperthreading (8 virtual cores) increases the latter speedup to 5.66, hashing the

---

\(^5\) [http://bench.cr.yp.to/results-hash.html#amd64-hydra7](http://bench.cr.yp.to/results-hash.html#amd64-hydra7)
file in 0.39s. We expect these speedups to converge to 4 and 8 respectively, as implementations (and CPUs) improve.

### 3.3 Low-end platforms

A typical implementation of BLAKE-256 in embedded software stores in RAM at least the chaining value (32 bytes), the message (64 bytes), the constants (64 bytes), the permutation internal state (64 bytes), the counter (8 bytes), and the salt, if used (16 bytes); that is, 232 bytes, and 248 with a salt. BLAKE2s reduces these figures to 168 bytes—recall that the salt doesn’t have to be stored anymore—that is, a gain of respectively 28% and 32%. Similarly, BLAKE2b only requires 336 bytes of RAM, against 464 or 496 for BLAKE-512.

### 3.4 Hardware

Hardware directly benefit from the 29% and 25% speed-up in sequential mode, due to the round reduction, for any message length. Parallelism is straightforward to implement by replicating the architecture of the sequential hash. BLAKE2 enjoys the same degrees of freedom as BLAKE to implement various space-time tradeoffs (horizontal and vertical folding, pipelining, etc.). In addition, parallel hashing provides another dimension for trade-offs in hardware architectures: depending on the system properties (e.g. how many input bits can be read per cycle), one may choose between, for example, BLAKE2sp based on 8 high-latency compact cores, or BLAKE2s based on a single low-latency unrolled core.

## 4 Security

BLAKE2 aims to provide the highest security level, be it in terms of classical notions as (second) preimage or collision resistance, or of theoretical notions as pseudorandomness (a.k.a. indistinguishability) or indifferentiability.

BLAKE2 builds on the high confidence built by BLAKE in the SHA-3 competition. Although BLAKE2 performs fewer rounds than BLAKE, this does not imply a lower security, as explained below.

### 4.1 BLAKE legacy

The security of BLAKE2 is closely related to that of the SHA-3 finalist BLAKE, since they rely on a similar core permutation originally used in Bernstein’s ChaCha stream cipher [4] (itself a variant of Salsa20 [5], co-winner in the eSTREAM project\(^6\)).

The final SHA-3 report [11, p5] comments that, like Keccak, BLAKE has a “very large security margin”, and that the cryptanalysis performed on BLAKE to date “appears to have a great deal of depth, while the cryptanalysis on Keccak has somewhat less depth”.

Indeed, since 2009, at least 14 research papers have described cryptanalysis results on reduced versions of BLAKE. The most advanced attacks on the BLAKE as hash function—as opposed to its building blocks—are preimage attacks on 2.5 rounds by Ji and Liangyu, with respective complexities \(2^{241}\) and \(2^{481}\) for BLAKE-256 and BLAKE-512 [15]. Most research

---

\(^6\)See [http://www.ecrypt.eu.org/stream/](http://www.ecrypt.eu.org/stream/).
actually considered reduced versions of the compression function or core permutation of BLAKE, regardless of the constraints imposed by the IV. The most recent results of this type are the following

- A “distinguisher” on 6 rounds of the permutation of BLAKE-256, with complexity $2^{456}$, by Dunkelman and Khovratovich [12];
- A “boomerang distinguisher” on 8 rounds of the core permutation of BLAKE-512, with complexity $2^{242}$, by Biryukov, Nikolic, and Roy [9] (recent works question the correctness of this result [17]).

The exact attacks as described in research papers may not directly apply to BLAKE2, due to the changes of rotation counts (typically, differential characteristics for BLAKE do not apply to BLAKE2). Nevertheless, we expect attacks on reduced BLAKE with $n$ rounds to adapt to BLAKE2 with $n$ rounds, though with slightly different complexities.

### 4.2 Implications of BLAKE2 tweaks

We have argued that the reduced number of rounds and the optimized rotations are unlikely to meaningfully reduce the security of BLAKE2, compared to that of BLAKE. We summarize the security implications of other tweaks:

**Salt-independent compressions.** BLAKE2 salts the hash function in the IV, rather than each compression. This preserves the uniqueness of the hash function for any distinct salt, but facilitates theoretical multicollision attacks relying on offline precomputations (see [8,16]). However, this leaves fewer “controlled” bits in the initial state of the compression function, which complicates the finding of fixed points.

**Many valid IVs.** Due to the high number of valid parameter blocks, BLAKE2 admits many valid initial chaining values. For example, if an attacker has an oracle that returns collisions for random chaining values and messages, she is more likely to succeed in attacks on the hash function because she has many valid targets, rather than a valid one. However, such a scenario assumes that collisions can be found efficiently, that is, that the hash function is already broken.

**Simplified padding.** The new padding does not include the message length of the message, unlike BLAKE. However, it is easy to see that the length is indirectly encoded through the counter, and that the padding preserves the unambiguous encoding of the initial padding. That is, the padding simplification does not affect the security of the hash function.

### 5 Legal statements

We, the designers of BLAKE2, do hereby declare that

- BLAKE2 is free for everyone to use;
• We are aware of no patent applications that may cover the practice of the BLAKE2 algorithm (or any version constructed as specified in the present document), reference implementation, or optimized implementations;

The reference source code package of BLAKE2, as available on https://blake2.net/, is published under the CC0 licence. The notice included on top of each source code file states the following:

To the extent possible under law, the author(s) have dedicated all copyright and related and neighboring rights to this software to the public domain worldwide.

References


http://creativecommons.org/choose/zero/


A Specification complements

To make the document self-contained, we complete the specification of BLAKE2b and BLAKE2s, describing mechanisms inherited from BLAKE and referring to the new features introduced in §2.

Recall that BLAKE2b works with 64-bit words, and BLAKE2s works with 32-bit words, and that both parse byte streams as word arrays in a little-endian way.
A.1 BLAKE2b

BLAKE2b supports data of any byte length \(0 \leq \ell < 2^{128}\). Data is first padded as per §§2.3 to form a sequence of \(N = \lceil \ell/128 \rceil \) 16-word blocks \(m^0, m^1, \ldots, m^{N-1}\), and then hashed by doing

\[
\begin{align*}
    h^0 & \leftarrow IV \oplus P \\
    \text{for } i = 0, \ldots, N - 1 & \quad h^{i+1} \leftarrow \text{compress}(h^i, m^i, \ell^i) \\
\end{align*}
\]

return \(h^N\)

where \(\ell^i\) denotes the number of data bytes in \(m^0, m^1, \ldots, m^i\) (that is, not counting any padding byte), \(P\) is the parameter block specified in §§2.8, and \(IV\) is (as in BLAKE and SHA-512) the following 64-bit words:

\[
\begin{align*}
    IV_0 &= 6a09e667f3bcc908 & IV_1 &= bb67ae8584caa73b \\
    IV_2 &= 3c6ef372fe94f82b & IV_3 &= a54ff53a5f1d36f1 \\
    IV_4 &= 510e527fade682d1 & IV_5 &= 9b05688c2b3e6c1f \\
    IV_6 &= 1f83d9abfb41bd6b & IV_7 &= 5be0cd19137e2179 \\
\end{align*}
\]

The compression function \textit{compress} takes as input

- a 64-byte chain value \(h = h_0, \ldots, h_7\)
- a 128-byte message block \(m = m_0, \ldots, m_{15}\)
- a counter \(t = t_0, t_1\), and finalization flags \(f_0, f_1\)

First, \textit{compress} initializes a 16-word internal state \(v_0, \ldots, v_{15}\) as per §§2.4, that is

\[
\begin{pmatrix}
    v_0 & v_1 & v_2 & v_3 \\
    v_4 & v_5 & v_6 & v_7 \\
    v_8 & v_9 & v_{10} & v_{11} \\
    v_{12} & v_{13} & v_{14} & v_{15}
\end{pmatrix}
\leftarrow
\begin{pmatrix}
    h_0 & h_1 & h_2 & h_3 \\
    h_4 & h_5 & h_6 & h_7 \\
    IV_0 & IV_1 & IV_2 & IV_3 \\
    t_0 \oplus IV_4 & t_1 \oplus IV_5 & f_0 \oplus IV_6 & f_1 \oplus IV_7
\end{pmatrix}
\]

where \(f_0\) and \(f_1\) are the finalization flags defined in §§2.3.

The internal state \(v\) is then transformed through a sequence of 12 rounds, where a round does

\[
\begin{align*}
    G_0(v_0, v_4, v_8, v_{12}) & \quad G_1(v_1, v_5, v_9, v_{13}) & \quad G_2(v_2, v_6, v_{10}, v_{14}) & \quad G_3(v_3, v_7, v_{11}, v_{15}) \\
    G_4(v_0, v_5, v_{10}, v_{15}) & \quad G_5(v_1, v_6, v_{11}, v_{12}) & \quad G_6(v_2, v_7, v_8, v_{13}) & \quad G_7(v_3, v_4, v_9, v_{14})
\end{align*}
\]

That is, a round applies a \(G\) function to each of the columns in parallel, and then to all of the diagonals in parallel. The \(G\) function of BLAKE2b is defined in §§2.4, and uses the constants in Table 5.
Table 5: Permutations of \{0, \ldots, 15\} used by the BLAKE2 functions.

| \(\sigma_0\) | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| \(\sigma_1\) | 14 | 10 | 4 | 8 | 9 | 15 | 13 | 6 | 1 | 12 | 0 | 2 | 11 | 7 | 5 | 3 |
| \(\sigma_2\) | 11 | 8 | 12 | 0 | 5 | 2 | 15 | 13 | 10 | 14 | 3 | 6 | 7 | 1 | 9 | 4 |
| \(\sigma_3\) | 7 | 9 | 3 | 1 | 13 | 12 | 11 | 14 | 2 | 6 | 5 | 10 | 4 | 0 | 15 | 8 |
| \(\sigma_4\) | 9 | 0 | 5 | 7 | 2 | 4 | 10 | 15 | 14 | 1 | 11 | 12 | 6 | 8 | 3 | 13 |
| \(\sigma_5\) | 2 | 12 | 6 | 10 | 0 | 11 | 8 | 3 | 4 | 13 | 7 | 5 | 15 | 14 | 1 | 9 |
| \(\sigma_6\) | 12 | 5 | 1 | 15 | 14 | 13 | 4 | 10 | 0 | 7 | 6 | 3 | 9 | 2 | 8 | 11 |
| \(\sigma_7\) | 13 | 11 | 7 | 14 | 12 | 1 | 3 | 9 | 5 | 0 | 15 | 4 | 8 | 6 | 2 | 10 |
| \(\sigma_8\) | 6 | 15 | 14 | 9 | 11 | 3 | 0 | 8 | 12 | 2 | 13 | 7 | 1 | 4 | 10 | 5 |
| \(\sigma_9\) | 10 | 2 | 8 | 4 | 7 | 6 | 1 | 5 | 15 | 11 | 9 | 14 | 3 | 12 | 13 | 0 |

After the 12 rounds, the new chain value \(h'_0, \ldots, h'_7\) is defined as

\[
\begin{align*}
h'_0 &\leftarrow h_0 \oplus v_0 \oplus v_8 \\
h'_1 &\leftarrow h_1 \oplus v_1 \oplus v_9 \\
h'_2 &\leftarrow h_2 \oplus v_2 \oplus v_{10} \\
h'_3 &\leftarrow h_3 \oplus v_3 \oplus v_{11} \\
h'_4 &\leftarrow h_4 \oplus v_4 \oplus v_{12} \\
h'_5 &\leftarrow h_5 \oplus v_5 \oplus v_{13} \\
h'_6 &\leftarrow h_6 \oplus v_6 \oplus v_{14} \\
h'_7 &\leftarrow h_7 \oplus v_7 \oplus v_{15}
\end{align*}
\]

Note the absence of the salt, compared to BLAKE.

A.2 BLAKE2s

BLAKE2s supports data of any byte length \(0 \leq \ell < 2^{64}\). It works similarly to BLAKE2b, but on 32-bit words instead of 64-bit words (the byte length of a chaining value, a message block, a counter or finalization flag are thus divided by two).

BLAKE2s uses the following IV:

\[
\begin{align*}
IV_0 &= 6a09e667 \\
IV_1 &= bb67ae85 \\
IV_2 &= 3c6ef372 \\
IV_3 &= a54ff53a \\
IV_4 &= 510e527f \\
IV_5 &= 9b05688c \\
IV_6 &= 1f83d9ab \\
IV_7 &= 5be0cd19
\end{align*}
\]

BLAKE2s does 10 rounds, and uses the G function defined in §§2.4.
Index

Abed Farzaneh, 27
Andreeva Elena, 111
Armknecht Frederik, 107
Aumasson Jean-Philippe, 112
Babbage Steve, 47
Bertoni Guido, 98
Biryukov Alex, 68
Blondeau Céline, 26
Boureanu Ioana, 52
Daemen Joan, 98
Debande Nicolas, 98
Dinur Itai, 66
Dunkelman Orr, 65, 66
Forler Christian, 6, 27
Gagliardoni Tommaso, 107
Hermans Jens, 48
Katzenbeisser Stefan, 107
Keller Nathan, 65
Le Thanh-Ha, 98
Lehmann Michael, 67
Leurent Gaëtan, 25
List Eik, 27
Lucks Stefan, 6, 27
Luykx Atul, 111
Maitra Subhamoy, 67
McGrew David, 6
Meier Willi, 67
Mennink Bart, 111
Mitrokotsa Aikaterini, 52
Neves Samuel, 112
Nyberg Kaisa, 61
Paul Goutam, 67
Peeters Michaël, 98
Peeters Roel, 48
Peter Andreas, 107
Preneel Bart, 111
Sarkar Santanu, 67
Shamir Adi, 66
Stefan Lucks, 5
Van Assche Gilles, 98
Vaudenay Serge, 49, 52
Velichkov Vesselin, 68
Wenzel Jakob, 6, 27
Wilcox-O’Hearn Zooko, 112
Winnerlein Christian, 112