A Forward-Secure Symmetric-Key Derivation Protocol

How to Improve Classical DUKPT

Eric Brier and Thomas Peyrin

Ingenico, France
{forename.name}@ingenico.com

Abstract. In this article, we study an interesting and very practical key management problem. A server shares a symmetric key with a client, whose memory is limited to R key registers. The client would like to send private messages using each time a new key derived from the original shared secret and identified with a public string sent together with the message. The server can only process N computations in order to retrieve the derived key corresponding to a given message. Finally, the algorithm must be forward-secure on the client side: even if the entire memory of the client has leaked, it should be impossible for an attacker to retrieve previously used communication keys. Given N and R , the total amount T of keys the system can handle should be as big as possible.

In practice such a forward-secure symmetric-key derivation protocol is very relevant, in particular in the payment industry where the clients are memory-constraint paying terminals and where distributing symmetric keys on field is a costly process. At the present time, one standard is widely deployed: the Derive Unique Key Per Transaction (DUKPT) scheme defined in ANSI X9.24. However, this algorithm is complicated to apprehend, not scalable and offers poor performances.

We provide here a new construction, *Optimal-DUKPT* (or O-DUKPT), that is not only simpler and more scalable, but also more efficient both in terms of client memory requirements and server computations when the total number of keys T is fixed. Finally, we also prove that our algorithm is optimal in regards to the client memory R / server computations N / number of keys T the system can handle.

keywords: key management, key derivation, DUKPT, forward-security.

1 Introduction

In information security, one of the most complicated part related to practical cryptography is the key management. Many different scenarios can exist and a different method is often required for each of them. The banking industry is well used to face strong constraints regarding key management. The financial transactions are processed in the paying terminal that reads the user magnetic card or chip card data. In order to validate the PIN input from the user, protocols based

on symmetric-key cryptography [5,6] are usually implemented. Also, because of the recent devastating attacks on back-office payment servers, the industry found strong incentives in protecting as well the user card data that is sent to the bank. Leading solutions [10,4], deploying format-preserving encryption, are also based on symmetric-key cryptography.

Of course, the symmetric key shared between the terminal and the bank must be securely distributed beforehand. In most cases, this is done by manually injecting the terminals with the key in a secure room. Clearly, such a process is costly for the stakeholders and can not be done frequently. For that reason, the symmetric keys stored in the terminal are static and not often changed.

Recently, the industry has seen the rising of side-channel attacks [8,9,13], practical and devastating methods that aims at recovering the secret key inside cryptographic modules by analyzing unusual information channels such as computation time, power consumption, electromagnetic emission, etc. As the paying terminals environment can not be considered as secure, side-channels attacks have to be taken in account seriously. For this reason, the banking industry has actively promoted an improved and more secure key management protocol: Derive Unique Key Per Transaction (DUKPT), defined in ANSI X9.24 [7].

The idea of DUKPT is to derive from the originally shared key a unique key per transaction. This feature greatly reduces the applicability of side-channel attacks, for which many measurement traces of encryption processes with the same symmetric key must be obtained. Moreover, this method is done in a forward-secure way on the client side (the servers are considered as located in a secure environment): if the internal state of the client is recovered, the attacker can not retrieve any of the transaction keys previously used. The algorithm can handle up to one million derived keys, which seems a reasonable upper bound for the number of transactions performed during a paying terminal life-cycle. Thus, the costly key injection process only has to be performed once.

DUKPT is standardized and now widely deployed in a majority of payment solutions. However, this protocol consumes a lot of memory in the devices, which are strongly memory-constrained. This is particularly problematic when a terminal has to be able to communicate with several distinct servers, and thus to handle many DUKPT instances at the same time. Moreover, DUKPT can also cause troubles on the server side, since it is costly in terms of computations required to retrieve the transaction key. Of course, this issue is even worsened by the fact that the server receives many financial transactions at the same time.

Our Contribution. In this article, we propose an improvement over the DUKPT technique described in ANSI X9.24 [7]. Our forward-secure symmetric-key derivation protocol offers scalability, simplicity and memory/computations performance gains. Yet the problem we study here is more general than just the sole case of paying terminals in the banking industry: memory-constrained clients that want to share with a computation-limited server unique symmetric keys per message sent in a forward-secure way. After having described our new proposal O-DUKPT,

we show that it is optimal in terms of client's memory requirements R / server computations N / number of keys T handled by the construction. Note that we restrict ourselves to using symmetric-key cryptography only. For forward-secure encryption schemes using public-key, see for example [3].

2 State-of-the-Art

2.1 Goals and Constraints

Derive Unique Key Per Transaction (DUKPT, see [7]) is a symmetric-key derivation scheme that offers several nice security properties for the payment industry scenarios (or any asymmetric situation where one server securely communicates with many memory-constrained clients). First, the symmetric keys used for each payment transaction are distinct. Moreover, a *forward-security* feature is incorporated: if the internal state of the paying terminal is entirely or partially compromised by any means, no useful information on the keys derived in previously processed transactions can be retrieved thanks to this leakage. Usually, the DUKPT scheme is utilized for the derivation of symmetric keys ciphering PIN blocks (for user authentication), or more recently for deriving symmetric keys that encrypts sensitive banking data such as Personal Account Number (PAN), expiration date, etc.

In practice, we have one server $\mathcal{S}$ that communicates with many paying terminals and each of these clients $\mathcal{C}_i$ must first share a symmetric key with $\mathcal{S}$. For obvious security reasons, two clients can not share the same key with the server (except by chance). This constraint could lead to memory problems for $\mathcal{S}$ if it has to deal with many clients. The issue is avoided by starting from a Base Derivation Key (BDK), directly available to $\mathcal{S}$. The k -bit key shared between a client $\mathcal{C}_i$ and the server $\mathcal{S}$ is denoted IK_i (for Initial Key) and is derived from the BDK as follows:

$$IK_i = F(BDK, i)$$

where $F : \{0, 1\}^* \mapsto \{0, 1\}^k$ is a pseudo-random function. We give in Appendix A how F is obtained in practice. Thus, the system is initialized by giving BDK to the server, and IK_i to each client $\mathcal{C}_i$. This key distribution is not in the scope of this article, but in general it is done on the client side with manual key injection in secure room or with remote key injection using public-key cryptography. Note that when a transaction is sent from the client to the server, the identity of the client $\mathcal{C}_i$ has to be sent as well, so the server can appropriately derive the key originally shared IK_i . The initialization process is depicted in Figure 1. The problem studied in this paper can be directly reduced to the case of a single client. Therefore, from now on, we will only consider two entities: a server $\mathcal{S}$ and a client $\mathcal{C}$ that initially share a key IK .

We would like to derive the unique transaction keys in a forward-secure way, using only the function F as a black-box. There is a very natural and inefficient way of achieving this: the client $\mathcal{C}$ maintains an internal state of one key, initialized with IK . The internal state after having processed the j -th transaction is

Fig. 1. Server and clients initialization

denoted S_j and we have $S_0 = IK$. Then, the key used for transaction j is simply the key stored in the internal state, S_{j-1} , and we update the internal state with $S_j = F(S_{j-1}, j)$. At each transaction, the client sends the value j so that the server can understand which key is shared for this transaction. It is clear that each key derived will be unique (except by chance) and that we have the forward-security property: F is a non-invertible process, so one can not obtain any information on the previous keys by recovering the internal state. However, it is also clear that this will be very inefficient on the server side. If one would like to handle say 1 million transactions, the server may have to go through one million computations of F to obtain the key in the worst case.

The idea of ANSI X9.24 DUKPT is to allow for the client to store more data than just one key, so as to lower the computation cost on the server side. More precisely, DUKPT allows the client to store $R = 21$ key registers and the server to compute F at maximum $N = 10$ times for one key derivation (except $IK_i = F(BDK, i)$). Overall, a total of $T = 1048575$ transactions can be handled.

In this paper we will show that DUKPT is not optimal when studying the following problem: given at maximum R key storage registers in one client $\mathcal{C}$ and N computations of F on the server $\mathcal{S}$ for one key derivation, what is the maximum number T of distinct keys the system can handle while ensuring the forward-security property if all the secret information contained in $\mathcal{C}$ is compromised? We provide an optimal algorithm, named *Optimal-DUKPT* (or O-DUKPT) that can handle up to $T = \binom{N+R}{R} - 1$ derived keys. For example, with the original parameters of DUKPT, $R = 21$ and $N = 10$, we are able to generate $T = 44352164$ keys. Otherwise, if the goal is to be able to derive about one million keys, one can use our solution with only $R = 13$ key registers. Our solution is not only more attractive in terms of memory and computations, but it is also much simpler to apprehend and to implement. Finally, *Optimal-DUKPT* is completely scalable and depending on the expected/desired memory/computation constraints of the system, it offers very valuable tradeoffs for practical applications.

2.2 ANSI X9.24 DUKPT Description

For completeness and future comparison, we provide in this section a description of the original DUKPT algorithm as defined in the ANSI X9.24-2009 document [7]. We assume that the shared symmetric key IK has been securely given

to the client and the server. We define $hw(x)$ as the hamming weight of the word x . The base 2 representation is denoted $(\cdot)_2$, i.e. 13 in base 2 is written $(1101)_2$, the most significant bit being located on the left. Also, for $x \neq 0$, we define $y = \tilde{x}$ to be the value of x with the least significant “1” bit set to zero. For example, if $x = (10110)_2$ we have $y = \tilde{x} = (10100)_2$ and $\tilde{y} = (10000)_2$.

In order to identify the key derived, for each transaction a 21-bit counter tc is sent from the client to the server. Then, for transaction j , the counter value $tc = j$ is sent and the key identified by j is used. The initial key IK is considered as a key identified by $j = 0$. DUKPT intrinsically defines a hierarchy between the keys: each key used for transaction $j \neq 0$ is the daughter of the key identified by $\tilde{j}$ (by A is the daughter of B , we mean that A is directly derived from B with F). For example, the key identified by $j = (0 \dots 010000)_2$ has four daughters identified by $j_1 = (0 \dots 010001)_2$, $j_2 = (0 \dots 010010)_2$, $j_3 = (0 \dots 010100)_2$ and $j_4 = (0 \dots 011000)_2$, since $j = \tilde{j}_1 = \tilde{j}_2 = \tilde{j}_3 = \tilde{j}_4$. More precisely, we have

$$K_j = F(K_{\tilde{j}}, j).$$

Before describing the process on the client and server side, one may ask why a 21-bit counter is needed (20 bits would suffice). The reason is that not all values of the counter and the corresponding keys will be used. Indeed, only the counter values with a non-zero hamming weight lower or equal to 10 will be considered and one can aim for a total key amount of

$$T = \sum_{j=1}^{10} \binom{21}{j} = 2^{20} - 1 = 1048575.$$

On the Server Side. $\mathcal{S}$ receives the 21-bit transaction counter tc . The server will derive the transaction key with only $hw(tc)$ computations of F (since we forced $hw(tc) \leq 10$, we do have the property that at maximum $N = 10$ computations of F are required). First, $\mathcal{S}$ deduces the bit position p_1 of the most significant “1” bit of tc and computes $c_{temp} = 2^{p_1}$ and $K_{temp} = F(IK, c_{temp})$. Then, the server deduces the bit position p_2 of the second most significant “1” bit of tc and computes $c_{temp} = c_{temp} + 2^{p_2}$ and $K_{temp} = F(K_{temp}, c_{temp})$. One continues until all the $hw(tc)$ “1” bits of tc have been processed. Then, the final key stored in K_{temp} is the shared key for this transaction. One can see that the server derivation simply consists in following the key hierarchy starting from IK and ending to K_{tc} . For example, if $tc = (0 \dots 011010)_2$, the server first computes $K_{temp} = F(IK, (0 \dots 010000)_2)$, then $K_{temp} = F(K_{temp}, (0 \dots 011000)_2)$ and finally $K_{temp} = F(K_{temp}, (0 \dots 011010)_2)$.

On the Client Side. The derivation on the client side is a little bit more complicated. First, the client is initialized as follows: each register r is filled with the value $F(IK, 2^{r-1})$ with $r \in \{1, \dots, R\}$, i.e. each register r is filled with $K_{2^{r-1}}$. Then, IK is erased from the client’s memory. One can note that those $R = 21$ keys are in fact the mothers of all future keys. For the first transaction, the key corresponding to $tc = 1$ is located in the first register (since the key stored in

Table 1. Chronological accesses to the key registers on the client side for DUKPT. A value i in a cell means that the key K_i is derived and stored in the corresponding column register at the corresponding row iteration. An “X” means that for this transaction the client used the key located in the corresponding register and then erased its contents.

counter tc		R_{21}	R_{20}	R_{19}	...	R_{12}	R_{11}	...	R_5	R_4	R_3	R_2	R_1
dec	hex												
init		2^{20}	2^{19}	2^{18}	...	2^{11}	2^{10}	...	16	8	4	2	1
1	1												X
2	2											X	3
3	3												X
4	4										X	6	5
5	5												X
6	6											X	7
7	7												X
8	8									X	12	10	9
9	9												X
10	A											X	11
11	B												X
12	C											14	13
13	D												X
14	E											X	15
15	F												X
16	10				...			...	X	24	20	18	17
17	11												X
...													
2045	7FD												X
2046	7FE											X	
2048	800					X	3072		2064	2056	2052	2050	2049
2049	801												X
...													
1047552	FFCD0						X						
1048576	100000	X	$2^{20} + 2^{19}$	$2^{20} + 2^{18}$		$2^{20} + 2^{11}$	$2^{20} + 2^{10}$		$2^{20} + 16$	$2^{20} + 8$	$2^{20} + 4$	$2^{20} + 2$	$2^{20} + 1$
1048577	100001												X
...													
2095104	1FF800					X							

this register is $K_1 = F(IK, 1)$). Once the transaction completed, the content of this register is erased in order to preserve the forward-secrecy: only IK is the mother of K_1 and it has already been erased. Note that one can freely erase K_1 because it has no daughter, so one does not lose any important information for later derivation. Then, when $tc = 2$, one uses the content from register 2 as transaction key. However, since K_2 is the mother of K_3 , before erasing it, one derives $K_3 = F(K_2, 3)$ and stores this key in register 1. One continues this process until all registers contain no more information.

To summarize, for a transaction j , the client picks and uses the key K_j located in register r , where r is the bit position of the least significant “1” bit of j . Then, before erasing K_j from its memory, the client derives and stores all the $r - 1$ direct daughters of K_j in the $r - 1$ least significant registers. The forward-secrecy is always maintained since after a transaction key have been used, it is always ensured that this key and its mother (or (grand)*-mothers) will no more be

present in the client's memory. Also, remember that all counter values with hamming weight strictly greater than 10 are skipped. We give in Table 1 an illustration of the chronological accesses to the key registers.

3 A New Proposal: O-DUKPT

3.1 General Idea

Our underlying idea for improving DUKPT can be understood with the following simple observation: for the very first transaction $tc = 1$ of DUKPT, the key K_1 located in the first register is used and directly erased. Note that this key has no daughter in the key hierarchy and that its mother is IK (it is at distance 1 from the IK). Said in other words, the server can retrieve K_1 from IK with only one computation of F . Instead of erasing K_1 directly and since we are yet far from reaching 10 computations of F on the server side, we could have derived another key from K_1 and placed it in this first register. Continuing this idea, we could have generated 9 more keys only with the first register.

Now, this can be generalized to the other registers as well. Once the first register contains a key located at a distance of 10 from the IK , we can not derive it anymore. Then, we have to utilize the key located in the second register, but before erasing it from the client's memory, we can derive from it two new keys that we will place in the first and second registers. Those two new keys are at distance 2 from the IK . Again, we can derive many keys only using the first register, but one less than before since we started from a key at distance 2 (and not 1) from the IK . This idea is finally iterated to all the registers.

3.2 Description

In order to preserve the scalability of the algorithm, Optimal-DUKPT will be defined as a family of key management schemes. Each member of the family is identified by the amount R of key registers available on the client side and the number N of maximum computations required to derive one key on the server side. Moreover, we will show later that each member can handle a maximum number of keys $T = \binom{R+N}{N} - 1$.

As for the original DUKPT, we assume that the shared symmetric key IK has been securely given to the client and the server. In order to identify the key derived, for each transaction a public string st is sent from the client to the server. This string is composed of R integers st_i , with $1 \leq st_i \leq N$ for $1 \leq i \leq R$. An integer st_i represents the distance from IK of the key stored in register i of the client's memory before processing the transaction. For example, the string sent for the very first transaction is $1 \dots 1 \ 1$, for the second one $1 \dots 1 \ 2$, etc.

On the Client Side. The client maintains two tables. First, the classical R key registers, denoted R_i for $1 \leq i \leq R$. They are simply initialized with $R_i = F(IK, 0^R, i)$ and once this process is over, the IK is erased from the client's memory. Secondly, the client maintains a table D of R integers that

we denote D_i , where D_i represents the distance from IK of the key stored in register R_i . The content of D is exactly what is sent to the server in the string st . Naturally, it is initialized with $D_i = 1$ for $1 \leq i \leq R$.

When requested to process a new transaction, the client builds $st = D$ and looks for the least significant register having a corresponding distance D_i strictly smaller than $N + 1$. This register, that we denote R_p , contains the transaction key TK that will be used. Then, once the transaction over,

- If $D_p < N$ (i.e. more keys can be derived from TK), the client updates the p registers $R_p, R_{p-1}, \dots, R_1$ with $R_i = F(TK, D, i)$ and updates the distance table with $D_i = D_p + 1$ with $1 \leq i \leq p$.
- Otherwise, if $D_p = N$ (i.e. TK does not have any daughter), the client simply erases the content of register R_p and updates $D_p = D_p + 1 = N + 1$.

Note that in the key derivation process, the data used as input of F is always unique. Indeed, D will be different for each transaction. This guarantees the security of the system. The forward-secrecy is always maintained since after a transaction key has been used, it is always ensured that this key (and its predecessors) will no more be present in the client's memory. We give an example of the clients internal state evolution in Table 2 or an alternate tree view in Figure 2.

Fig. 2. Tree view of the client side key derivation example from Table 2, with system parameters $N = 3$ and $R = 3$. We denote TK_j the key used for the j -th iteration, and the R_i aside the circles represent the register in which each key TK_i is stored during the process.

Table 2. Example of key registers and distance tables evolution on the client side, with system parameters $N = 3$ and $R = 3$. We denote TK_i the key used for the i -th iteration. An “X” in the key registers evolution columns means that the client erases the contents from this register.

iter.	st sent	transaction key used and key registers update	key registers evolution			distance table			
			R_3	R_2	R_1		D_3	D_2	D_1
init		$R_3 = F(IK, 000, 3)$ $R_2 = F(IK, 000, 2), R_1 = F(IK, 000, 1)$	TK_{10}	TK_4	TK_1	in out	0 1	0 1	0 1
1	111	$TK_1 = R_1$ $R_1 = F(TK_1, 111, 1)$			TK_2	in out	1 1	1 1	1 2
2	112	$TK_2 = R_1$ $R_1 = F(TK_2, 112, 1)$			TK_3	in out	1 1	1 1	2 3
3	113	$TK_3 = R_1$ erase R_1			X	in out	1 1	1 1	3 4
4	114	$TK_4 = R_2$ $R_2 = F(TK_4, 114, 2), R_1 = F(TK_4, 114, 1)$		TK_7	TK_5	in out	1 1	1 2	4 2
5	122	$TK_5 = R_1$ $R_1 = F(TK_5, 122, 1)$			TK_6	in out	1 1	2 2	2 3
6	123	$TK_6 = R_1$ erase R_1			X	in out	1 1	2 2	3 4
7	124	$TK_7 = R_2$ $R_2 = F(TK_7, 124, 2), R_1 = F(TK_7, 124, 1)$		TK_9	TK_8	in out	1 1	2 3	4 3
8	133	$TK_8 = R_1$ erase R_1			X	in out	1 1	3 3	3 4
9	134	$TK_9 = R_2$ erase R_2		X		in out	1 1	3 4	4 4
10	144	$TK_{10} = R_3, R_3 = F(TK_{10}, 144, 3)$ $R_2 = F(TK_{10}, 144, 2), R_1 = F(TK_{10}, 144, 1)$	TK_{16}	TK_{13}	TK_{11}	in out	1 2	4 2	4 2
11	222	$TK_{11} = R_1$ $R_1 = F(TK_{11}, 222, 1)$			TK_{12}	in out	2 2	2 2	2 3
12	223	$TK_{12} = R_1$ erase R_1			X	in out	2 2	2 2	3 4
13	224	$TK_{13} = R_2$ $R_2 = F(TK_{13}, 224, 2), R_1 = F(TK_{13}, 224, 1)$		TK_{15}	TK_{14}	in out	2 2	2 3	4 3
14	233	$TK_{14} = R_1$ erase R_1			X	in out	2 2	3 3	3 4
15	234	$TK_{15} = R_2$ erase R_2		X		in out	2 2	3 4	4 4
16	244	$TK_{16} = R_3, R_3 = F(TK_{16}, 244, 3)$ $R_2 = F(TK_{16}, 244, 2), R_1 = F(TK_{16}, 244, 1)$	TK_{19}	TK_{18}	TK_{17}	in out	2 3	4 3	4 3
17	333	$TK_{17} = R_1$ erase R_1			X	in out	3 3	3 3	3 4
18	334	$TK_{18} = R_2$ erase R_2		X		in out	3 3	3 4	4 4
19	344	$TK_{19} = R_3$ erase R_3	X			in out	3 4	4 4	4 4

On the Server Side. The server receives a string st that corresponds to the table D of the client before processing the transaction. Note that the distance values memorized in this table are always increasing from most significant to least significant registers. Moreover, we recall that when the client extracted a transaction key from a register R_p , it means that the distance table was such that $D_i = N + 1$ for $1 \leq i \leq p - 1$. We denote by $g_{p,v}(D)$ the transformation that maps the distance table D to another distance table D' with

$$\begin{cases} D'_i = N + 1, & \text{for } 1 \leq i \leq p - 1 \\ D'_i = v, & \text{for } i = p \\ D'_i = D_i, & \text{for } p + 1 \leq i \leq R \end{cases}$$

The server first initializes a local distance value $d = 1$ and a register position value $p = p'$, with p' being the most significant position with $D_{p'} > 1$. Then, he computes $K = F(IK, 0^R, p')$ and keeps repeating the following process:

- While $d < D_p - 1$, compute $K = F(K, g_{p,d}(D), p)$ and $d = d + 1$.
- If $p = 1$, then $K = F(K, g_{p,d}(D), p)$ is the key shared with the client so the program can stop.
- The server finds the most significant position p' such that $D_{p'} > D_p$. If $D_{p'} \neq N + 1$, then he computes $K = F(K, g_{p,d}(D), p')$ and updates the local variables $p = p'$ and $d = d + 1$. Otherwise, $K = F(K, g_{p,d}(D), p' + 1)$ is the key shared with the client so the program can stop.

This algorithm exactly follows the implicit process performed by the client to derive the transaction key TK from the initial key IK . For example, reusing the

Table 3. Example of key derivation on the server side, with system parameters $N = 8$ and $R = 8$ and $st = 12466689$. The key is derived with 8 server computations.

iter.	key update	local values			
			d	p	p'
init	$K = F(IK, 00000000, 7)$	in out	 1	 7	 7
1	$K = F(K, 11999999, 6)$	in out	1 2	7 6	 6
2	$K = F(K, 12299999, 6)$ $K = F(K, 12399999, 5)$	in out	2 4	6 5	 5
3	$K = F(K, 12449999, 5)$ $K = F(K, 12459999, 2)$	in out	4 6	5 2	 2
4	$K = F(K, 12466669, 2)$ $K = F(K, 12466679, 2)$	in out	6	2	 1

scenario from Table 2, assume that the server receives $st = 224$. He will first set $d = 1$, $p = 3$ and compute $K = F(IK, 000, 3)$. Then, he does not enter the while loop nor the first if. He computes $p' = 1$ and since $D_{p'} = 4 = N + 1$, the key $K = F(K, 144, 2)$ is the key shared with the client. We give in Table 3 a more complicated example.

3.3 Performance Analysis

Now that we defined our construction, we would like to precisely compute its theoretical performance. It is clear that the client only needs to maintain a table of R registers and that the maximum number of computations required on the server side to derive the key is at most N calls to the function F . What is the number T of keys the system can support (note that IK is not counted as transaction key)? Since T depends on N and R , we denote $T(N, R)$ the maximum number of keys that can be generated with R client's registers and N server computations.

One can be easily convinced that $T(N, 1) = N$. Indeed, with only a single register R_1 , the derivation process will simply use and self-update the key stored in R_1 until it exceeds the maximal distance N . Also, we have $T(1, R) = R$ since with a maximal allowable distance of 1, our construction would simply fill each of the R registers with a key directly derived from the IK and that has no daughter.

Let $t(n, r)$ denote the number of distinct keys stored by register r with a distance n from IK during the entire course of the algorithm, with $1 \leq n \leq N$ and $1 \leq r \leq R$. Since the registers are ordered (a register r can only be updated by a register r' with $r' \geq r$), we deduce that $t(n, R) = 1$, because the most significant register can only be updated by itself. It is also clear that $t(1, r) = 1$, since the only keys at distance 1 are the very first keys stored in each register.

The only way for a register r to hold a key at distance $n > 1$ is to be updated from a key at distance $n - 1$ stored in a register $r' \geq r$. Thus, for $2 \leq n \leq N$ and $1 \leq r \leq R$, we have

$$t(n, r) = \sum_{i=r}^R t(n-1, i)$$

which simplifies to

$$t(n, r) = t(n-1, r) + \sum_{i=r+1}^R t(n-1, i) = t(n-1, r) + t(n, r+1).$$

We define the function $g(n, r) = \binom{n+R-r-1}{R-r}$ and it is well known that

$$\binom{a}{b} = \binom{a-1}{b} + \binom{a-1}{b-1} \quad (1)$$

$$\sum_{i=b}^a \binom{i}{b} = \binom{a+1}{b+1} \quad (2)$$

where (2) is derived by induction from (1). Thus, using (1) we obtain

$$g(n, r) = \binom{n + R - r - 2}{R - r} + \binom{n + R - r - 2}{R - r - 1} = g(n - 1, r) + g(n, r + 1).$$

Since we have $g(n, R) = 1$ and $g(1, r) = 1$, we conclude that $t(n, r) = g(n, r)$ for $1 \leq n \leq N$ and $1 \leq r \leq R$.

The total amount of key handled by the system is computed by

$$T(N, R) = \sum_{i=1}^N \sum_{j=1}^R t(i, j) = \sum_{i=1}^N \sum_{j=1}^R \binom{i + R - j - 1}{R - j} = \sum_{i=0}^{N-1} \sum_{j=0}^{R-1} \binom{i + j}{j}$$

Finally, using identities (1) and (2) we obtain

$$\begin{aligned} T(N, R) &= \sum_{i=0}^{N-1} \sum_{j=0}^{R-1} \binom{i + j}{j} = \sum_{i=0}^{N-1} \sum_{j=0}^{R-1} \binom{i + j}{i} = \sum_{i=0}^{N-1} \binom{i + R}{i + 1} = \sum_{i=0}^{N-1} \binom{i + R}{R - 1} \\ &= -1 + \sum_{i=-1}^{N-1} \binom{i + R}{R - 1} = \binom{R + N}{N} - 1 = \binom{R + N}{R} - 1 \end{aligned}$$

3.4 Optimality Proof

In this section we show that when N and R are fixed, the amount of keys T handled by our algorithm is the maximal reachable value. First, note that we do not count the initial key in this amount. Indeed, as potentially sensitive in practice, we consider that the initial key must not be stored on the client side after the initialization process has been performed. Of course, if needed, our algorithm O-DUKPT can be trivially modified in order to also utilize IK as one of the transaction keys: the initialization process only stores the IK in one of the register, then the first transaction key is this initial key and the first registers update simply performs the initialization process from O-DUKPT.

We assume that the server and the client can only call the non-invertible function F to derive the keys, in a black-box manner. This pseudo-random function manipulates an input of arbitrary length and outputs a key.

After the initialization phase and during the protocol run, the client can only store R keys in its memory. We do not count here the temporary memory required for the key derivations. Those R registers represent the number of keys that can be memorized in the client's memory once the transaction is over. Once the transaction key has been deduced by the client, he processes the transaction with this key and sends the public data st to the server. Once st received, the server can only use a maximum of N calls of F in order to derive the transaction key from the initial key IK .

The key generation algorithm must be forward-secure on the client side. That is, when a transaction has been performed, it must be impossible for an attacker that just recovered the R internal registers to retrieve any transaction

key previously utilized. We call such an algorithm a *forward-secure DUKPT algorithm* and we denote by T the maximum number of distinct keys the system can handle.

At each transaction i , the client can first do some computations from the R registers contents to deduce the transaction key TK_i , then he stores TK_i in its local memory in order to use it, and he finally updates its R internal registers. Because F is as pseudo-random function, there is no need to use several keys on its input. One can generate as many distinct outputs as needed from F with a single key input by using distinct additional data (such as a counter for example). Thus, when deriving keys or transaction keys with F , we can assume that only one key is used on its input.

Now, since we would like to preserve the forward-security on its side (and since TK_i only depends on one key from the R registers), the client model can be simplified: at each transaction, he picks the transaction key TK from one of its internal registers R_i , he stores it in its local memory and finally updates the R registers (i.e. the computation phase can be merged with the update). Moreover, since the forward-security forces only the TK to be erased, the client only uses this key for derivation (there is no advantage in doing a derivation from a key that we do not have to erase yet). Therefore, for each transaction, the client picks the transaction key TK from one of its internal registers R_i , stores it in its local memory and finally updates the R registers exclusively from it.

When studying theoretically the optimality of a DUKPT algorithm, there is no need to consider the server behavior. Indeed, the only requirement for the server is that it must be able to compute the transactions with at most N calls to F . Since F is as pseudo-random function, this only depends on how the client generated the transaction keys. This constraint is modeled on the client side with a distance value assigned to each key, representing the number of calls to F required to reach this key from IK . Obviously, no transaction key can have a distance strictly greater than N (and it is useless to memorize any key with a distance strictly greater than N).

Theorem 1. *A forward-secure DUKPT algorithm with R client registers and N maximal server computations can derive at most T distinct keys, with*

$$T = \binom{R+N}{N} - 1 = \binom{R+N}{R} - 1.$$

Let $\mathcal{A}$ be an optimal algorithm, i.e. reaching the maximum value T of keys handled. We prove this theorem with several very simple lemmas concerning $\mathcal{A}$.

Lemma 1. *After the initialization process of $\mathcal{A}$, the R registers of the client are filled with R new distinct keys.*

Proof. It is clear that not filling all R registers during the initialization phase is not an optimal method. Let $\mathcal{B}$ be such a forward-secure DUKPT algorithm. We can trivially build another forward-secure DUKPT algorithm $\mathcal{B}'$ that generates strictly more keys than $\mathcal{B}$: during the initialization phase, $\mathcal{B}'$ memorizes one more

key derived from IK in one of the registers left blank by $\mathcal{B}$. It uses this key for the very first transaction and erases the contents of the corresponding register. Once this key is used and erased from the client's memory, $\mathcal{B}'$ behaves identically as $\mathcal{B}$. Overall, one more key has been generated in the process. $\square$

Lemma 2. *When $\mathcal{A}$ derives keys on the client side during the registers update, it only memorizes newly derived keys in empty registers.*

Proof. Indeed, let $\mathcal{B}$ be a forward-secure DUKPT algorithm that memorizes a newly derived key in a non empty register during one transaction. We can trivially build another forward-secure DUKPT algorithm $\mathcal{B}'$ that generates strictly more keys than $\mathcal{B}$: $\mathcal{B}'$ behaves identically as $\mathcal{B}$, but instead of directly erasing this particular register, it first uses the key stored in it and erases the register contents once the transaction is over. Overall, one more key has been generated in the process. $\square$

Lemma 3. *When $\mathcal{A}$ derives keys on the client side during the registers update, all previously empty registers are filled at the end of the process.*

Proof. Let $\mathcal{B}$ be a forward-secure DUKPT algorithm that does not fill all empty registers when deriving new keys during one transaction. We can trivially build another forward-secure DUKPT algorithm $\mathcal{B}'$ that generates strictly more keys than $\mathcal{B}$: $\mathcal{B}'$ behaves identically as $\mathcal{B}$, but instead fills one of the empty register that $\mathcal{B}$ left blank with a new distinct key K (this is possible since we already assumed that $\mathcal{B}$ possess some key content to derive from at this moment). Then, during the next transaction, $\mathcal{B}'$ will use K , erase it and finally continue as $\mathcal{B}'$ in the previous transaction. Overall, one more key has been generated in the process. $\square$

The direct corollary of the two last lemmas is that the update derives keys in every empty register only.

Lemma 4. *The transaction key chosen by $\mathcal{A}$ is always one of the keys at the maximal available distance from IK (different from $N + 1$).*

Proof. Let $\mathcal{B}$ be a forward-secure DUKPT algorithm that extracts a transaction key TK from a register R_i containing a key at distance $d < d_{max}$ from the IK , where d_{max} denotes the maximal distance available among the R registers. From previous lemmas we know that, after erasure of R_i , all empty registers must be filled with keys derived from TK . We can trivially build another forward-secure DUKPT algorithm $\mathcal{B}'$ that generates strictly more keys than $\mathcal{B}$: $\mathcal{B}'$ behaves identically as $\mathcal{B}$, but instead does one more transaction. First $\mathcal{B}'$ extracts a transaction key TK_+ among the set of registers containing keys located at distance d_{max} from IK . We denote by R_+ this register. Then, the update simply consists in erasing R_+ . For the next iteration, $\mathcal{B}'$ extracts the transaction key TK from R_i and performs the update exactly as $\mathcal{B}$. The only difference for $\mathcal{B}'$ is that R_+ will be updated as well, because it is now empty. The update is done with TK , located at distance $d < d_{max}$: we make $(d_{max} - d)$ calls to F to perform the

derivation, so that R_+ finally contains a key at distance d_{max} . Thus, at this point, $\mathcal{B}'$ generated one more key (i.e. TK^+) than $\mathcal{B}$, while reaching the same distance table situation (since R_+ has distance d_{max} for both $\mathcal{B}$ and $\mathcal{B}'$). $\square$

We showed that an optimal algorithm $\mathcal{A}$ must fulfill several properties stated in the previous lemmas. Namely, the initialization phase must fill all the R client's registers. Then, for each transaction, the client must use (and erase) one of the keys stored with the maximal distance from IK and derive exclusively from it distinct keys that will be stored in each and every empty register only. This already almost completely specifies what is an optimal algorithm. The only freedom remaining concerns which key is picked when several have the same maximal distance from IK and this has absolutely no incidence on the maximum number T of keys one can generate. Thus, all algorithms verifying the lemmas are equivalent and optimal. Since our proposal O-DUKPT does fulfill those conditions, we can conclude that we reach the optimal value of T :

$$T = \binom{R+N}{N} - 1 = \binom{R+N}{R} - 1.$$

4 Discussions

Knowing the maximum number of computations N on the server is a good guarantee of minimal performance (note that the maximal number of computations on the client side is equivalent for both algorithms: $R-1$ for DUKPT and R for O-DUKPT.). However, one could also estimate the average number of computations and for this we need to know the relative amount of keys at distance i from the IK . Let $A(i)$ represent the number of keys at distance i . Of course, we have $T = \sum_{i=1}^N A(i)$. The more keys we have at a large distance, the bigger will be the average number of computations per key on the server side. The average number of computations on the server side is

$$C_S = \left(\sum_{i=1}^N i \cdot A(i) \right) / T$$

and on the client side it is

$$C_C = \left(\sum_{i=1}^N A(i) \right) / T = 1.$$

In the case of O-DUKPT, we have

$$A(i) = \sum_{j=1}^R t(i, j) = \sum_{j=1}^R \binom{i + R + j - 1}{R - j} = \binom{R + i - 1}{i}$$

and for classical DUKPT, we have $A(i) = \binom{21}{i}$, with $i \leq 10$.

Table 4. Performance comparison between DUKPT (parameters $R = 21/N = 10$) and O-DUKPT (for parameters $R = 21/N = 7$, $R = 13/N = 10$ and $R = 17/N = 8$)

	DUKPT ($R = 21, N = 10$)	O-DUKPT ($R = 21, N = 7$)	O-DUKPT ($R = 13, N = 10$)	O-DUKPT ($R = 17, N = 8$)
T	1048575	1184039	1144065	1081574
$A(1)/T$	$2^{-15.6}$	$2^{-15.8}$	$2^{-16.4}$	$2^{-16.0}$
$A(2)/T$	$2^{-12.3}$	$2^{-12.3}$	$2^{-13.6}$	$2^{-12.8}$
$A(3)/T$	$2^{-9.6}$	$2^{-9.4}$	$2^{-11.3}$	$2^{-10.1}$
$A(4)/T$	$2^{-7.4}$	$2^{-6.8}$	$2^{-9.3}$	$2^{-7.8}$
$A(5)/T$	$2^{-5.7}$	$2^{-4.5}$	$2^{-7.5}$	$2^{-5.7}$
$A(6)/T$	$2^{-4.3}$	$2^{-2.4}$	$2^{-5.9}$	$2^{-3.9}$
$A(7)/T$	$2^{-3.2}$	$2^{-0.4}$	$2^{-4.5}$	$2^{-2.1}$
$A(8)/T$	$2^{-2.4}$		$2^{-3.2}$	$2^{-0.6}$
$A(9)/T$	$2^{-1.8}$		$2^{-2.0}$	
$A(10)/T$	$2^{-1.6}$		$2^{-0.8}$	
C_S	8.65	6.68	9.28	7.56

As a comparison with classical DUKPT, if we use the same amount of registers $R = 21$ for O-DUKPT, we only need to do at maximum $N = 7$ computations to handle an equivalent number of keys: $\binom{21+7}{21} - 1 = 1184039$. If we allow the same amount of maximum computations $N = 10$ for O-DUKPT, then we only need to maintain $R = 13$ key registers to handle an equivalent number of keys: $\binom{13+10}{13} - 1 = 1144065$. The Table 4 gives the numerical application for A_i , C_S and C_C . Thus, the performance improvement is twofold: for T and R fixed, not only O-DUKPT has a lower maximum number of computations on server side, but the average number of computations is also lower. Finally, we give an example for which O-DUKPT provides better results in regards to every performance aspects ($R = 17$ and $N = 8$ gives $T = 1081574$ and $C_S = 7.56$).

Variants. We proved in a previous section the optimality of our algorithm. However, one may derive variants concerning its implementation and most specifically concerning how the client communicates the identity of the key to the server and how the server processes its corresponding key derivation.

Our O-DUKPT implementation proposed requires the client to send a string st of R integers in $[1, \dots, N]$. This could be coded on $\log_2((N-1)^R)$ bits. The algorithm is simple to understand and implement, but it is not optimal in terms of message size since $T < (N-1)^R$. For example, classical DUKPT requires to send a 21-bit counter, while O-DUKPT (with parameters $R = 17, N = 8$) requires 48 bits.

One can think of several variants if message size is an issue in practice. For example, a very easy way to lower the message size is to leverage the memory available at the server side. For example, instead of sending the D table before

processing the transaction, the client could simply send the transaction counter (thus coded on $\log_2(T)$ bits, the smallest possible message size). The server would have to recover the corresponding table D from the transaction counter received. This could be done very simply by a table lookup. This table, filled during initialization of the system, would require $T \cdot \log_2((N-1)^R)$ bits (roughly 5MB for O-DUKPT with parameters $R = 17$ and $N = 8$).

Acknowledgments

The authors would like to thank the ASIACRYPT 2010 committee for their helpful comments and for pointing us the reference [3].

References

1. Bellare, M.: New Proofs for NMAC and HMAC: Security Without Collision-Resistance. Cryptology ePrint Archive, Report 2006/043 (2006), <http://eprint.iacr.org/>
2. Bellare, M., Kilian, J., Rogaway, P.: The Security of Cipher Block Chaining. In: Desmedt, Y.G. (ed.) CRYPTO 1994. LNCS, vol. 839, pp. 341–355. Springer, Heidelberg (1994)
3. Canetti, R., Halevi, S., Katz, J.: A Forward-Secure Public-Key Encryption Scheme. In: Biham, E. (ed.) EUROCRYPT 2003. LNCS, vol. 2656, pp. 255–271. Springer, Heidelberg (2003)
4. Brier, E., Peyrin, T., Stern, J.: BPS: a Format Preserving Encryption Proposal. NIST submission (April 2010), http://csrc.nist.gov/groups/ST/toolkit/BCM/modes_development.html
5. American National Standards Institute. ISO 9564-1:2002 Banking – Personal Identification Number (PIN) management and security – Part 1: Basic principles and requirements for online PIN handling in ATM and POS systems (2002)
6. American National Standards Institute. ANSI X9.8-1:2003 Banking - Personal Identification Number Management and Security - Part 1: PIN protection principles and techniques for online PIN verification in ATM and POS systems (2003)
7. American National Standards Institute. ANSI X9.24-1:2009 Retail Financial Services Symmetric Key Management Part 1: Using Symmetric Techniques (2009)
8. Kocher, P.C.: Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In: Koblitz, N. (ed.) CRYPTO 1996. LNCS, vol. 1109, pp. 104–113. Springer, Heidelberg (1996)
9. Kocher, P.C., Jaffe, J., Jun, B.: Differential Power Analysis. In: Wiener, M. (ed.) CRYPTO 1999. LNCS, vol. 1666, pp. 388–397. Springer, Heidelberg (1999)
10. Bellare, M., Rogaway, P., Spies, T.: Format-preserving Feistel-based Encryption Mode. NIST submission (April 2010), http://csrc.nist.gov/groups/ST/toolkit/BCM/modes_development.html
11. NIST. FIPS 198 – The Keyed-Hash Message Authentication Code, HMAC (2002)
12. National Institute of Standards and Technology. SP800-67: Recommendation for the Triple Data Encryption Algorithm (TDEA) Block Cipher (May 2004), <http://csrc.nist.gov>
13. Quisquater, J.-J., Samyde, D.: ElectroMagnetic Analysis (EMA): Measures and Counter-Measures for Smart Cards. In: E-smart, pp. 200–210 (2001)

Appendix A: How to Instantiate F in Practice?

In ANSI X9.24, the DUKPT implementation described is intended to derive 112-bit TDES keys [12] (128-bit keys with 8 bits of parity checks). The F function is therefore itself based on TDES. A 128-bit incoming key K (the first input) is divided into two 64-bit parts K_L and K_R , and the new key $K' = K'_L || K'_R$ is derived with

$$\begin{aligned} K'_L &= \text{TDES}_{K_L}(C \oplus K_R) \oplus K_R \\ K'_R &= \text{TDES}_{K_L}(C' \oplus C \oplus K_R) \oplus K_R \end{aligned}$$

where C is a known value depending on the counter (the second input), and C' is a fixed constant. The parity bits of K'_L and K'_R are then appropriately adjusted.

As F function, we advice to use commonly deployed MAC algorithms such as CBC-MAC [2] or HMAC [11,1] with the incoming key as MAC key and transaction related input as MAC message input.