Forensic Cryptanalysis of the Backdoored UA-8295 Message Terminal

Stijn Maatje and Marc Stevens[0000−0002−7091−2924]

CWI Cryptology Group
{maatje,stevens}@cwi.nl

Abstract. Nation State Agencies go to great lengths to obtain signal intelligence, including backdooring cryptographic standards and equipment. Although the existence of these backdoor efforts is common knowledge, only few of known backdoored systems have been publicly analysed. In this paper we present the first detailed *forensic cryptanalysis* of the backdoored UA-8295 message terminal and we try to answer questions how the UA-8295's backdoor was designed and for which attack. Towards a better understanding of real-world backdoor design, we propose a methodology based on the *Backdoor Conjecture* that provides handles to reason about the design of backdoors and the attacks they are designed for.

Keywords: Backdoored Cryptography · Symmetric Cryptography · Forensic Cryptanalysis

1 Introduction

Cryptographic Backdoors are deliberate, concealed weaknesses in real-world cryptographic systems, inserted by a malicious adversary to spy on its targets and read their confidential communication. They introduce critical vulnerabilities to these encryption systems affecting all their users and exposing them to further risks of exploitation by others. There are many real-world examples of such backdoors with major impact including:

– **Operation Bullrun:** The 2013 Snowden files revealed some of the US backdooring efforts, and moreover that these were aimed at foreign powers as well as US citizens [4,9,12,26]. A New York Times article details the organized approach to weaken cryptographic standards and that the NSA allegedly spent $250 million a year on backdoor insertion as part of Operation Bullrun and the SIGINT enabling project [35, 36].
The NSA orchestrated the standardization of the backdoored Dual-EC Pseudorandom Number Generator by NIST (and later by ANSI and ISO), and paid to make it the default in the widely-used RSA BSAFE crypto library [30, 35, 36, 43]. Surprisingly, Juniper network equipment was hacked by another state actor to alter Dual-EC the backdoor into a backdoor for their own purposes [52].

- **Crypto AG:** The US CIA and German BND secretly controlled the Swiss company Crypto AG, a leading manufacturer of cryptographic equipment supplying to over 120 countries, and backdoored its devices for over 50 years since 1970. The operation, known internally at CIA as Operation RUBICON (or THESAURUS), was an organized effort to supply nation-states with backdoored cryptographic equipment and was publicly exposed in 2020 [19, 32]. At its height, Crypto AG controlled 80-90% of all cipher equipment used worldwide. In the 1980s, the NSA was able to read 96% of all intercepted communications sent through electronic Crypto AG machines. An internal CIA document dubbed Operation RUBICON *the intelligence coup of the century*, and deemed it the most successful CIA operation of all time [19].
- **TETRA:** TETRA is a collection of proprietary algorithms standardized by ETSI in 1995, and widely used by emergency services and in other critical infrastructure in over 100 countries. In 2023, security company Midnight Blue found critical security issues in TETRA, including a deliberately weakened encryption algorithm TEA-1 that is 'approved for general use worldwide' [28]. Further research points towards possible backdoors in TETRA, including TEA-3 and AES-56 (AES-128 with 56 bits of effective key) [1, 29].
- **USFA:** In 2019, a former top cryptographer from (now-defunct) USFA confirmed that the NSA made a deal with USFA to weaken cryptography in several instances [24]. At least three reported cases are known. First, Beroflex, a weakened copy of a NATO-approved device, was sold and used commercially in many non-NATO countries [17]. Second, the PX-1000 was a pocket telex marketed with DES symmetric encryption (1980s). By NSA request, a weakened version (dubbed the PX-1000Cr) was put on the market instead [18]. Third, the UA-8295 was a military encryption device to be used in the field marketed since 1984, originally designed by Nokia with DES encryption. The NSA supplied a weaker breakable encryption version for targeted customers [20].

The existence of real-world backdoored cryptographic systems have been known for a while, but little cryptanalytic research has been done on the exact nature of these backdoors, mostly because these backdoored encryption algorithms were (and sometimes still are) proprietary.

1.1 Prior Work on Backdoors

We distinguish between academic backdoors (publicly proposed, not used) and real-world backdoors (actually used and exposed). Most analyses of cryptographic backdoors have been at the academic level.

Academic backdoors at the implementation level. Subliminal channels are hidden communication channels for users inside cryptographic protocols [45–47]. *Kleptography* is the paradigm of 'using cryptography to subvert cryptography': a *malicious* subliminal channel where the outputs of the original and backdoored implementation are *indistinguishable* to its users [50, 51]. Such backdoors have been described for RSA key generation [16,50,51], Diffie-Hellman

key exchange [21,48,50,51], DSA [50], ElGamal encryption & signatures [50], and recently also quantum-safe schemes [22,23,25,41,49]. Bellare *et al.* formalized Algorithm Substitution Attacks (ASAs) [7,8], proving that randomized encryption is inherently vulnerable to subversion.

Academic backdoors at the specification level. These embed a hidden structural weakness in the cryptographic design itself. Early proposals include those by Rijmen-Preneel [42] and Paterson [34]. Later, Courtois adapted block ciphers to force polynomial invariants [15], and Posteuca-Ashur propose a block cipher backdoor exploiting the linear hull effect [38]. Peyrin-Wang introduce the MALICIOUS framework of a family of backdoored tweakable block ciphers [10, 37], and Baksi *et al.* propose an improved scheme ZUGZWANG [3].

Analysis of real-world backdoors. The backdoor possibility of Dual-EC was raised in 2007 [44], later works covered its standardization history [9], practical TLS exploitability [14], and the Juniper incident where an unknown actor hijacked the backdoor [13]. Meijers *et al.* presented the first public in-depth security analysis of TETRA [28] and found five serious security vulnerabilities. Alich *et al.* observe that TEA-3's 80-bit key register LFSR decomposes into two 40-bit halves, hinting at an intentional weakness, but fail to develop a practical attack [1]. For the broken 2G cellphone standard GEA-1 [2, 6], Beierle *et al.* reverse-engineered its construction using polynomial algebra over $\mathbb{F}_2$, showing the structure was unlikely to arise by chance [5]. Marsiske reverse-engineered the PX-1000Cr and claimed a successful key recovery trial via brute-force constraint solving [27]. Schneier *et al.* recognize a gap in effective methodology to study exposed real-world backdoors, and actively call for research in this area [43].

1.2 Background on the UA-8295

The paper focuses on the real-world backdoor in USFA's UA-8295 Short Burst Terminal, which has not been studied before.

The UA-8295 was sold by USFA, a now defunct subsidiary of Royal Dutch Philips founded in 1949. USFA focused on the development and production of mostly military equipment to the Dutch government, Dutch Army and other NATO countries. Some successful products spawned from this section, with the Aroflex and the Spendex voice encryption units as notable examples.

From 1977 onwards, the NSA made a deal with USFA to sell backdoored encryption equipment [31]. One example of this is the UA-8295 Short Burst Terminal (SBT), depicted in Figure 1, which is based on the DA-8520 (known as M/90 Sanomalaite or SANLA in the Finnish Army) manufactured by Nokia in the late 1970s. The original encryption algorithm on the DA-8520 used DES in Output Feedback mode [20].

As a part of this NSA-USFA deal, the NSA developed a new cryptographic algorithm and drop-in replacement ROMS. A lot of effort was put in the programming of the EPROMs by the NSA. From the outside there are virtually no differences between the original DA-8520 and the backdoored UA-8295 (beside the branding). On startup, both devices show the same version number, the user interface is the same and even the checksum of the data on the EPROMs is the



Fig. 1. The USFA UA-8295 Short Burst Terminal

same. The way to distinguish both versions is by calculating the checksum of a user-inputted key, which is shown to the user after inputting a new key.

In a publication by Dutch action group Onkruit [33] we found stolen sales records of USFA, where we can see that the UA-8295 was sold to other Philips subsidiaries in Belgium and the United Kingdom. The sale to the UK concerns the UA-8295/06, which is a normal UA-8295 unit with an Arabic keyboard, intended for use in Saudi Arabia. The UA-8295 was mentioned in the 1986 and 1988 editions of *Jane's Military Communications*, a yearly catalogue used by governments worldwide to pick equipment for their Departments of Defence. In both catalogues we can find a handheld version of the UA-8295 containing the same encryption algorithm [39, 40].

1.3 Our Contributions

In this work we study the UA-8295. However, it is not our goal to demonstrate its encryption can be broken, we already know it is weak. Our goal is to advance the understanding of the design of its backdoor and the attack it was designed for using cryptanalysis, a topic which we call *forensic cryptanalysis*. As Schneier *et al.* recognize there, is a gap in effective methodology to analyse real-world backdoors.

Hence, towards our study of the UA-8295 and future study of other real-world backdoors, we propose a forensic methodology (section 2) based on a (folklore) conjecture on real-world malicious actors aiming to produce a backdoored cryptographic scheme.

Furthermore, we present a first (forensic) study of the UA-8295. Since both the original DES implementation as well as the NSA-modified algorithm are

available to us, a simple comparison of the two allowed us to easily locate and isolate the cryptographically relevant code. We partially reverse-engineered the binary code and present the first full description of the UA-8295 encryption procedure in section 3. We identify and discuss five design issues in section 4 and section 7, of which three are cryptographic weaknesses inside the symmetric primitive SBTcipher.

We leverage these SBTcipher weaknesses to develop and implement two key recovery attacks in section 5 (these will be published open-source, and are available as auxiliary material). From our experiments (section 5.4) on modern hardware, both attacks with a single input/output block pair can easily recover the key 'on a laptop in a weekend'.

Finally, in section 6 we present a first application of the forensic methodology on the candidate attacks.

2 Forensic Cryptanalysis Methodology

For the forensic analysis of the UA-8295, we propose a new methodology for cryptographic backdoor design analysis using the (folklore) Backdoor Conjecture:

> **Conjecture 1 (Backdoor Conjecture)** *Backdoors are designed with two main goals: (i) to optimize a specific attack, and (ii) to look secure under casual analysis of design and output.*

The Backdoor Conjecture enables evaluating candidate attacks against a backdoored design, by using the conjecture's two main goals to categorize design freedoms. Class (i) design freedoms are important for optimizing the original attack, e.g., structure between key schedule and round operations. Class (ii) design freedoms are neutral to the original attack or necessary to look secure, e.g., the choice of IV or S-Boxes. Effectively the Backdoor Conjecture implies that decisions and optimizations in the designer's process embed mathematical structure that can be forensically analysed. This is the basis of the forensic methodology we propose.

More precisely, it suggests the following three steps to evaluate candidate attacks:

1. *Categorize each design freedom as class (i) or class (ii) dependent on the candidate attack:* Under the conjecture, class (i) freedoms are chosen to fully optimize the original attack, while class (ii) freedoms follow conventional design to look secure.
2. *Test class (i) freedoms*: For a candidate attack, the chosen design should be optimal, while other design variations should be suboptimal. Hence, optimality is evidence for the candidate attack, where stronger structure gives stronger evidence. Suboptimality provides evidence against the candidate attack, where a wider gap to the optimum provides stronger evidence.

3. *Test class (ii) freedoms:* These freedoms should follow conventional design to look secure, thus conventional design supports the candidate attack. Whereas unexpected structure in a class (ii) freedom may indicate that this freedom is class (i) for the original attack, providing evidence against the candidate attack.

Note that our forensic analysis of the UA-8295 is a first analysis and is not exhaustive. It is both guided by the proposed methodology and meant to demonstrate its applicability for future work.

2.1 Application to prior work

To illustrate the Backdoor Conjecture we can apply it to various schemes that are claimed to be backdoored. In the TETRA TEA-1 cipher, a reduction is first applied to the 80-bit key to reduce it to 32 bits [28]. It should be obvious that such a reduction compromises the security of the entire scheme and as such we can conclude that design goal (ii) does not hold for this scheme. It was more likely due to cryptographic export controls: an obvious reduction in security is a simple way of demonstrating compliance.

The GEA-1 stream cipher shows an 'exceptional interaction' between three internal LFSRs, that can be exploited for an attack [6]. The choice of LFSRs in the design is a class (i) freedom. Since the interaction between LFSRs is not noticeable at first glance of the algorithm specification, design goal (ii) holds.

The Kuznyechik block cipher, published by the Russian Federation's standardization agency, appears secure on first glance. However, after thorough analysis, Biryukov *et al.* discovered hidden structure in Kuznyechik's S-Box, allowing a decomposition of the S-Box and demonstrating poor cryptographic properties. [11].

Of course, the Backdoor Conjecture can only be applied properly if the design is available to the researcher. It is merely a tool in the cryptanalyst's toolbox, and requires a specification. Backdoored black-box systems cannot be scrutinized under the Backdoor Conjecture.

3 Description of the SBT encryption algorithm

3.1 Preliminaries

For integers $a \leq b$, $[a, b]$ (resp. $[a, b)$) denotes the set of integers from a up to including (resp. excluding) b.

Byte strings and bit strings are presented in big-endian (most significant first). Bytes can be split into two nibbles (4-bit values), which can be split into two crumbs (2-bit values).

We use natural correspondences between $\{0,1\}^8$, $\mathbb{F}_2^n$ and $\mathbb{Z}_{2^n}$ and freely switch between them. For example, a byte $B \in \{0,1\}^8$ can be represented as a pair of nibbles $(b_h, b_\ell) \in \mathbb{Z}_{2^4}^2$, as well as a vector $(b_7, b_6, \ldots, b_0) \in \mathbb{F}_2^8$.

We use bitwise OR ($\vee$), bitwise XOR ($\oplus$), and bitwise AND ($\wedge$). Let $x \in \mathbb{Z}_{2^n}$, then $x^{\ll i}$ and $x^{\gg i}$ denote bitwise left shift (multiplying by 2^i modulo 2^n) by i bits and bitwise right shift (integer division by 2^i), respectively. Similarly, $x^{\circlearrowleft i}$ and $x^{\circlearrowright i}$ denote bitwise left rotation and bitwise right rotation by i bits, respectively. I.e., $x^{\circlearrowleft i} = (x^{\ll i}) \vee (x^{\gg n-i})$ and $x^{\circlearrowright i} = (x^{\ll n-i}) \vee (x^{\gg i})$.

3.2 Binary Code and Reverse Engineering

The UA-8295 is an encrypted-message terminal capable of communicating over short-wave radio, and interacting with external printer and/or computer. It uses two Intel 80C31 8-bit microprocessors: the primary handles all main tasks, while the secondary acts as radio modem. There are four EPROM chips: two for code and a third for text (primary CPU), the last for the modem. Reverse-engineering the algorithm is only possible due to the Crypto Museum[1], who obtained both the original ROM binary code from the Nokia DA-8520 and the NSA-backdoored-version of the UA-8295. They restored their UA-8295 and modified it to be able to switch between both versions.

Comparing the two available ROMs for the primary CPU, one can easily identify the modified sections of the code, which enabled us to focus directly on where the cryptographic code is. We used the `d52`-disassembler[2] to partially reverse-engineer the cryptographic algorithms. There were many assembly level tricks in the code, which required passing specific information to the disassembler how to treat certain sections. Once we were sure what was text/binary data and how the code sections were structured, we were able to develop an algorithmic description of the cryptographic code. We reimplemented the encryption/decryption algorithms in `C++` and verified correctness by decrypting a set of encrypted messages generated using the Crypto Museum's UA-8295 device.

3.3 Encryption and Decryption Procedure

The UA-8295's symmetric encryption is based on a SPN-blockcipher-like primitive that is used in a stream cipher mode to encrypt plaintext. Here we describe on a high level how the UA-8295 code processes user inputs (plaintext and key) to generate the ciphertext.

The user inputs a 15-character user key, and can then encrypt plaintexts. For each plaintext, the first 5 characters (`A-P`) of the ciphertext are pseudo-randomly generated and used as nonce. The nonce is used together with the user key to then compute a 56-bit encryption key.

To compute the rest of the ciphertext, it generates a keystream using the encryption key and the nonce, which will be XORed with the plaintext and then encoded: 3 bytes of plaintext of which only the lower 6 bits per byte are used (for a total of 18 bits), result in 5 bytes of ciphertext with alphabet `A-P` (effectively 4 bits per character, and 20 bits in total).

[1] https://www.cryptomuseum.com/crypto/philips/ua8295/
[2] https://github.com/jblang/d52

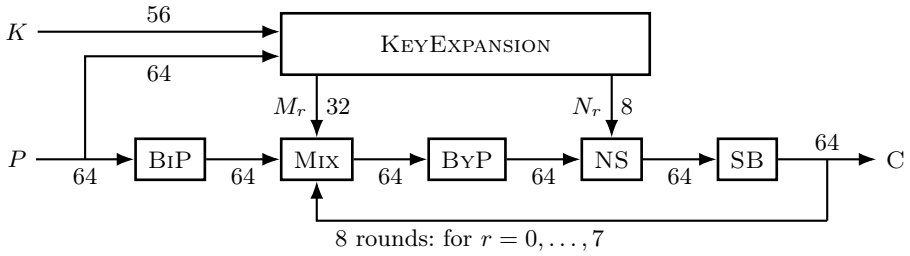

Fig. 2. Schematic view of SBTcipher

The keystream itself is generated by repeatedly generating a 64-bit input block (recursively using a LFSR, a fixed constant and the nonce), and calls the cryptographic primitive SBTcipher with the encryption key and the input block to compute one 64-bit block of the keystream.

Decryption extracts the nonce and computes the encryption key using the user key and the nonce, from which it can compute the same keystream as during encryption. It then decodes ciphertext bytes and XORs the result with the keystream to recover the plaintext bytes. Finally it uses a simple rule to recover the unique 8-bit allowed character from the decrypted lower 6 bits, i.e., bit 7 is 0 and bit 6 is the negation of bit 5.

3.4 Description of SBTcipher

SBTcipher is a SPN-blockcipher-like primitive that takes a 56-bit key K and a 64-bit input block P and outputs a 64-bit block C and is depicted in Figure 2. Note that technically it is not a block cipher: (i) it is not invertible: the key schedule *also depends on* P, (ii) it has no round key addition: key bits are used to control round operations only. Since it is used in a stream cipher mode of operation, decryption does not require it to be invertible.

SBTcipher operates on a 64-bit state initialized with P. It first applies a Bit Permutation BiP, then applies 8 rounds on the state, and finally outputs the 64-bit state. Each round consists of the following operations in order: MixNibbles Mix, Byte Permutation ByP, Nibble Switch NS and S-Boxes SB. Mix and NS use round-dependent control bits M_r (32 bits) and N_r (8 bits) respectively, which are derived from the key, *the plaintext* and the round r by KeyExpansion.

- BiP: The 64-bit state bit permutation is given in section A.1.
- Mix: This is the only round operation that mixes between bytes in the state. It is presented below.
- ByP: Applies the following permutation on the state's 8 bytes: $(S_7, S_6, \ldots, S_0) \mapsto (S_2, S_7, S_0, S_6, S_4, S_1, S_5, S_3)$.
- NS: If the i-th bit of N_r is 1 then swap the high and low half of the i-th state byte, for $i = 0, \ldots, 7$.
- SB: Applies S-Box SB_i (section A.2) on the i-th state nibble for $i = 0, \ldots, 15$.

Algorithm 1 KeyExpansion: Generate round control words M_r and N_r

Input: $K = (k_{55} \dots k_0) \in \{0,1\}^{56}$, $P \in \{0,1\}^{64}$, $r \in \{0, \dots, 7\}$
$\lambda_r = 5, 7, 9, 14, 19, 24, 26, 28$ for $r = 0, 1, 2, 3, 4, 5, 6, 7$ respectively
$\hat{K} = (k_{55}k_{54} \dots k_{28})^{\circlearrowleft \lambda_r} || (k_{27}k_{26} \dots k_0)^{\circlearrowleft \lambda_r} = (\hat{k}_{55} \dots \hat{k}_0)$
$\hat{P} = P^{\circlearrowleft 32-8r} = (\hat{p}_{63} \dots \hat{p}_0)$
$C_0 = (\hat{k}_{47}, \hat{k}_{19}, \hat{k}_{44}, \hat{k}_{16}, \hat{k}_{41}, \hat{k}_{13}, \hat{k}_{38}, \hat{k}_{10}) \oplus (\hat{p}_{56}, \hat{p}_{57}, \dots, \hat{p}_{63})$
$C_1 = (\hat{k}_{31}, \hat{k}_3, \hat{k}_{28}, \hat{k}_0, \hat{k}_{53}, \hat{k}_{25}, \hat{k}_{50}, \hat{k}_{22}) \oplus (\hat{p}_{48}, \hat{p}_{49}, \dots, \hat{p}_{55})$
$C_2 = (\hat{k}_{43}, \hat{k}_{15}, \hat{k}_{40}, \hat{k}_{12}, \hat{k}_{37}, \hat{k}_9, \hat{k}_{34}, \hat{k}_6) \oplus (\hat{p}_{40}, \hat{p}_{41}, \dots, \hat{p}_{47})$
$C_3 = (\hat{k}_{55}, \hat{k}_{27}, \hat{k}_{52}, \hat{k}_{24}, \hat{k}_{49}, \hat{k}_{21}, \hat{k}_{46}, \hat{k}_{18}) \oplus (\hat{p}_{32}, \hat{p}_{33}, \dots, \hat{p}_{39})$
$M_r = (C_3, C_2, C_1, C_0)$ ▷ MixNibbles control word
$N_r = (\hat{k}_{35}, \hat{k}_7, \hat{k}_{32}, \hat{k}_4, \hat{k}_{29}, \hat{k}_1, \hat{k}_{54}, \hat{k}_{26})$ ▷ Nibble Switch control word
Output: M_r, N_r

MixNibbles. The 64-bit state has 16 nibbles. The MixNibbles function Mix operates on the nibbles in a specific order: from low to high byte order and in each byte first update the high nibble then the low nibble. Each of the 16 nibbles is updated using 2 selected control bits from M_r, and in certain cases will also use the value from another selected nibble. More specifically, let $S = (n_{15}, \dots, n_0)$ then for $i = 0, \dots, 15$:

$$\begin{aligned}
k &= (M_r^{\gg 2i} \bmod 4) && \in \mathbb{Z}_4 && \text{(2 control bits)} \\
a &= i \oplus 1 && \in \mathbb{Z}_{16} && \text{(nibble index to update)} \\
b &= a \oplus 2^{3-k} && \in \mathbb{Z}_{16} && \text{(other selected nibble index)} \\
n_a &\leftarrow \text{MixN}(n_a, n_b, k) && \in \mathbb{Z}_{16} && \text{(update } n_a\text{)}
\end{aligned}$$

Here MixN splits $n_a = (c_1, c_0)$ and $n_b = (d_1, d_0)$ in high and low 2-bit crumbs. Depending on $k = (k_1, k_0)$ it will increase or decrease either c_1 or c_0 modulo 4 ($4 = 2 \times 2$ choices). Only when c_1 (or c_0) 'wraps around', $d_1 + d_0$ is added to c_0 (resp. c_1) modulo 4. Embedding bits (k_0, k_1, or their complement) also as 0 or 1 in $\mathbb{Z}_4$, we can define MixN : $\mathbb{Z}_4^2 \times \mathbb{Z}_4^2 \times \mathbb{Z}_4 \to \mathbb{Z}_4^2$ as:

$$\begin{aligned}
&(c_1, c_0, d_1, d_0, k_1, k_0) \mapsto \\
&(c_1, c_0) + (-1)^{\overline{k_0}} \cdot (\overline{k_1}, k_1) + \delta_{[k_1 c_0 + \overline{k_1} c_1 \equiv 3k_0 \bmod 4]} \cdot (d_1 + d_0) \cdot (k_1, \overline{k_1}).
\end{aligned}$$

Note that (1) the bits from n_b only affect the new n_a if the predicate $[k_1 c_0 + \overline{k_1} c_1 \equiv 3k_0 \bmod 4]$ is true, which happens with probability $\frac{1}{4}$ (assuming n_a uniformly distributed), (2) only the 2 computed bits $d_1 + d_0 \bmod 4$ from n_b are used. Hence, on average n_a uses only $2/4 = 1/2$ bit from n_b.

Key expansion. The SBTcipher function uses key expansion to generate two control words for every round: a 32-bit control M_r for MixNibbles and an 8-bit control N_r for the Nibble Switch, which are generated as described in Algorithm 1.

3.5 Encryption

Given an encryption key K, a nonce $L \in \{\mathtt{A}, \ldots, \mathtt{P}\}^5$ and a plaintext $P \in (\mathbb{Z}_{2^8})^*$, $\textsc{Encrypt}_{K,L}(P)$ generates a ciphertext of the form $L||\{\{\mathtt{A}, \ldots, \mathtt{P}\}^5\}^*$. The encryption is lossy: bits 6 and 7 of every plaintext byte are lost and will be replaced in decryption using a rule to map to the desired character subset. P is first padded with the character '_' to the next multiple of 3. Let $n = |P|$ the length of the padded plaintext in bytes, and $m = \lceil n/8 \rceil$ be the number of 64-bit keystream blocks needed to encrypt P.

Encrypt uses a Fibonacci LFSR with polynomial $1 + x^{31} + x^{63}$ to shift a new MSB bit in, which is clocked 64 times on each call. Let $L = (l_0, l_1, l_2, l_3, l_4)$. Then a sequence of input blocks $B_0, \ldots, B_{m-1} \in (\mathbb{Z}_{2^{64}})^m$ is generated as follows:

$$\begin{aligned}
B_0 &= \mathrm{LFSR}^{64}(\mathtt{CF17AF8A107AC0F5}_{16}) && \text{(IV, no nonce)}\\
B_1 &= \mathrm{LFSR}^{64}((B_0 \wedge (2^{40}-1)) + 2^{40}l_0 + 2^{48}l_1 + 2^{56}l_2) && \text{(nonce mixed in)}\\
B_i &= \mathrm{LFSR}^{64}(B_{i-1}) && \text{for } i = 2, \ldots, m-1
\end{aligned}$$

It then encrypts the padded plaintext $P = (p_0, \ldots, p_{n-1})$ as follows:

$$\begin{aligned}
(z_7^i, \ldots, z_0^i) &= \mathrm{SBT}\textsc{cipher}(K, B_i) && \text{for } i = 0, \ldots, m-1\\
x_j &= p_j \oplus z^{(j/8)}_{(j \bmod 8)} && \text{for } j = 0, \ldots, n-1
\end{aligned}$$

This mode of operation is not unlike Counter mode: input blocks are a function of nonce and position, without plaintext/ciphertext contribution. The first 5 bytes of the ciphertext are exactly L, then for every group of 3 plaintext bytes it encodes the corresponding $(x_{3j}, x_{3j+1}, x_{3j+2})$ into 5 ciphertext bytes $(c_{5j+5}, c_{5j+6}, c_{5j+7}, c_{5j+8}, c_{5j+9})$ as follows:

$$\boxed{\begin{aligned}
x_j &= (b_7\ b_6\ b_5\ b_4\ b_3\ b_2\ b_1\ b_0)\\
x_{j+1} &= (b_{15}\ b_{14}\ b_{13}\ b_{12}\ b_{11}\ b_{10}\ b_9\ b_8)\\
x_{j+2} &= (b_{23}\ b_{22}\ b_{21}\ b_{20}\ b_{19}\ b_{18}\ b_{17}\ b_{16})
\end{aligned}} \leftrightarrow \boxed{\begin{aligned}
c_{5j+5} &= \mathtt{'A'} + (0\ 0\ 0\ 0\ b_0\ \ b_8\ \ b_{16}\ b_1)\\
c_{5j+6} &= \mathtt{'A'} + (0\ 0\ 0\ 0\ b_9\ \ b_{17}\ b_2\ \ b_{10})\\
c_{5j+7} &= \mathtt{'A'} + (0\ 0\ 0\ 0\ b_{18}\ b_3\ \ b_{11}\ b_{19})\\
c_{5j+8} &= \mathtt{'A'} + (0\ 0\ 0\ 0\ b_4\ \ b_{12}\ b_{20}\ b_5)\\
c_{5j+9} &= \mathtt{'A'} + (0\ 0\ 0\ 0\ b_{13}\ b_{21}\ b_6\ \ b_{14})
\end{aligned}}$$

3.6 Decryption

Decrypt generates the sequence z_j^i in the same manner as Encrypt, however it then computes the sequence x'_j (x_j with some bits lost) by reversing the above encoding into the ciphertext bytes. The plaintext bytes are then recovered as:

$$p_j = \textsc{fixchar}\left(x'_j \oplus z^{(j/8)}_{(j \bmod 8)}\right), \text{where}$$

$$\textsc{fixchar}(b_7b_6b_5b_4b_3b_2b_1b_0) \coloneqq (0\overline{b_5}b_5b_4b_3b_2b_1b_0).$$

3.7 Key Derivation

The nonce is freshly generated for every encryption as described in section 7.2. After that, the encryption key K is derived from the 15-character user key $U = (u_0, \ldots, u_{14})$ and the 5-character nonce $L = (l_0, \ldots, l_4)$ as described in Algorithm 2. Note that compared to the original 8-bit assembly in our description we chose to reverse the bit in each byte of the key, as then KeyExpansion admits a much nicer description over the 56-bit key.

Algorithm 2 KeyDerivation

Input: $U = (u_0, \ldots, u_{14}) \in (\mathbb{Z}_{2^8})^{15}, L = (l_0, \ldots, l_4) \in (\mathbb{Z}_{2^8})^5$
$P' = (u_7, u_6, u_5, u_4, u_3, u_2 \oplus l_2, u_1 \oplus l_1, u_0 \oplus l_0)$
$K' = (\phi(u_{14}), \phi(u_{13}), \ldots, \phi(u_8))$, where $\phi(b_7, \ldots, b_0) \coloneqq (b_0, \ldots, b_7)$
$(c_7, \ldots, c_0) = \mathrm{SBTcipher}(K', \mathrm{LFSR}^{64}(P'))$
Output: $K = (\phi(c_6), \phi(c_5), \phi(c_4), \phi(c_3), \phi(c_2), \phi(c_1), \phi(c_0))$

4 Cryptanalysis of SBTcipher

In this work we aim to forensically cryptanalyse the UA-8295 to gain a better understanding of the backdoor as well as the methods used to design it.

Once we obtained the reverse-engineered description of the encryption and decryption procedures and of SBTcipher, we first performed a careful analysis of potential weaknesses and interesting abnormalities in proper cryptographic design. We found the following issues:

(i1) *Flawed mode-of-operation*: The mode of operation generates a sequence of input blocks which depends on the nonce, *except* the very first block which is fixed and the same for every encryption key and nonce. A fixed first plaintext block may aid attacks with precomputation.

(i2) *Leaky nonce generation*: The nonce is generated by combining the previous encryption's first 5 key stream bytes together with sampling a hardware counter. If the hardware counter is sufficiently predictable then the nonce leaks information about the previous key stream. *Surprisingly, this issue is also present in the original Nokia DES-based ROM.*

(i3) *Weak mixing in SBTcipher*: the only round operation that mixes bytes is Mix, however there are 2 issues: First, whenever a 4-bit nibble is updated with the value of another nibble, it actually uses only a 2-bit crumb derived from it instead of the whole nibble. Second, with probability $\frac{3}{4}$ the nibble update does not involve another nibble at all, only with probability $\frac{1}{4}$ will it use the 2-bit crumb.

(i4) *The control bits ('round keys') also depend on the plaintext*: In SBTcipher, the KeyExpansion generates control bits that depend on both key and plaintext block, which makes it hard to efficiently invert.

(i5) *Strong interaction between* KeyExpansion *and* ByP: The weak mixing makes it interesting to follow the path a byte follows through the byte permutation ByP. Each of the 8 'byte paths' through ByP uses exactly 40 control bits that depend on 40 key bits, however each byte path actually only uses 19 to 25 *unique* key bits.

Somewhat surprisingly, there is no obvious backdoor in the form of a reduction to an effective key of say only 40 bits, as is frequent in weakening cryptographic primitives under export control. In this case, the backdoor of the UA-8295 is more carefully hidden in the design of the encryption procedure, although the found issues (i1)-(i4) easily stand out for cryptographers. The issue (i5) was only found after performing preliminary cryptanalysis searching for hidden structures. Issue (i4) is deemed harmless for now, while issues (i1) and (i2) will be covered later in section 7.

It is clear that more complete cryptanalysis and developing actual attacks based on the observed issues (i3) and (i5) is necessary to gain more insights. As stated before: a malicious designer would strongly prefer low-data attacks over large-data attacks for practical surveillance, which is even more limiting for the short messages this device was designed for. Hence, in this paper, we focus on low-data attacks, though we have superficially looked at linear and differential attacks and both are poor fits for the round operations Mix and NS.

Below we present a detailed cryptanalysis of the (i3) and (i5) weaknesses of SBTcipher in sections 4.1 and 4.2 and we develop two key-recovery attacks given one or more known input/output block pairs:

1. A *List Merge* (LM) attack (sec. 5.1): for every byte path we look at the k unique key bits it uses and compute a List of k-bit candidate partial keys that are 'valid' for a given input and output block. Then we merge these Lists one-by-one by considering every allowable combination (i.e., where the two partial keys agree under their keymask's overlap) and filtering out the resulting larger candidate partial keys for which an inconsistency arises.
2. A *Branch-and-Prune* (BnP) attack (sec. 5.2): a depth-first search tree over the key space. For every node we check whether an inconsistency can be found, in which case we prune that subtree and backtrack.

4.1 SBTcipher's Weak Mixing

To cryptanalyse the weak mixing of SBTcipher, we use the notion of a *byte path*: the path a byte takes through the SBTcipher evaluation. We essentially view all round operations as operating on bytes: NS and SB update each byte individually, Mix updates each byte (potentially using other bytes) and notably ByP which simply permutes the bytes. Given a byte index $a \in [0, 7]$ at the start of the first round of SBTcipher, the byte path corresponding to this byte consists of all round operations that update that byte, following its byte position changes by ByP, throughout the eight rounds.

To illustrate issue (i3) above, we can look at the expected number of bits a byte path uses from other bytes. First, let Z_{nibble} be the random variable

denoting the number of bits a nibble uses from other bytes. Recall that a nibble only needs an external crumb whenever $[k_1c_0 + \overline{k_1}c_1 \equiv 3k_0 \bmod 4]$ (see sec. 3.4), which occurs with probability $\frac{1}{4}$ if k is fixed.

Note that if MIX is currently operating on nibble at position a, and if $k = 3$, the other selected nibble has position $b = a \oplus 1$, which is exactly the other nibble in the byte containing a. If we assume the 2 control bits k are uniformly distributed, we have that the expected number of bits used *from other bytes* is:

$$\mathbb{E}(Z_{nibble}) = \left(\frac{3}{4} \cdot 0 + \frac{1}{4} \cdot 2\right) \cdot \frac{3}{4} = \frac{3}{8} \text{ bits.}$$

Now let Z_{BP} be the random variable denoting the number of bits a byte path uses from other byte paths. Since there are a total of 16 MIXN evaluations during evaluation of a byte path (8 rounds, 2 nibbles per byte), a byte path uses expectedly $\mathbb{E}(Z_{BP}) = \mathbb{E}(Z_{nibble}) \cdot 8 \cdot 2 = 6$ bits from other byte paths. This is the only interaction that occurs between byte paths during the evaluation of SBTCIPHER.

4.2 Candidate Partial Key Lists

Towards the List Merge attack, we first describe the definition of a *List*. While we initially look only at candidate partial key Lists for individual byte paths, we already define Lists for any combination of byte paths below:

Definition 1. *A* List L_A*, with a byte path selection* $A \subseteq [0, 7]$*, is a 2-tuple containing the following information:*

- $\textsc{keymask}_A \in \{0,1\}^{56}$*. The* i*-th bit of* $\textsc{keymask}_A$ *is set if and only if the* i*-th bit is used by the byte paths in* A*,*
- *A set of all valid keys restricted to* $\textsc{keymask}_A$*, i.e.,*

$$\left\{K \in \{0,1\}^{56} \mid (K \wedge \overline{\textsc{keymask}_A}) = 0 \text{ and } \textsc{valid}(K, A) = 1\right\}.$$

Since the keys in this last set are restricted according to $\textsc{keymask}_A$*, we call them* partial keys*. With* L_A *we typically refer directly to its partial key set, thus* $|L_A|$ *denotes the size of this set.*

Given an input and output block to the SBTCIPHER, VALID is a to-be-defined (non-perfect) oracle function that outputs 1 if the given key could have been used to generate the output from the input, when only considering the byte paths selected in A. Crucially, VALID should not return false negatives, but it can be expected to output many false positives especially for small A. For brevity, we introduce shortened notations: $L_{3210} \coloneqq L_{\{3,2,1,0\}}$ and $L_{AB} \coloneqq L_{A \cup B}$, we also use $\mathcal{L} \coloneqq L_{76543210}$, the List containing only the actual key K.

It is useful to analyse the expected sizes of these Lists towards building a cryptanalytic attack. Let us first consider single byte path cases, i.e., $|A| = 1$. To estimate List size, we calculate the number of bits we need to guess when executing the algorithm on a given input.

Table 1. Number of unique key bits used by every byte path

Byte path number (a)	7	6	5	4	3	2	1	0
Unique key bits used (U_a)	24	25	22	20	22	20	19	22

Table 2. Expected List sizes for every byte path

Byte path (a)	$\log_2(\mathbb{E}[\lvert L_a\rvert])$	$\mathbb{E}[\lvert L_a\rvert]$	$\mathbb{E}$(storage)
7	22	$4.2 \cdot 10^6$	32 MiB
6	23	$8.4 \cdot 10^6$	64 MiB
5	20	$1.0 \cdot 10^6$	8 MiB
4	18	$2.6 \cdot 10^5$	2 MiB
3	20	$1.0 \cdot 10^6$	8 MiB
2	18	$2.6 \cdot 10^5$	2 MiB
1	17	$1.3 \cdot 10^5$	1 MiB
0	20	$1.0 \cdot 10^6$	8 MiB

Let $a \in [0, 7]$ be a single byte path. Let U_a be the number of unique key bits used by byte path a, which is fixed and can be found in Table 1. The expected size of the List L_a can be approximated as follows: there are 2^{U_a} candidate partial keys. For each candidate partial key, one can expect the byte path needs $\mathbb{E}(Z_{BP})$ bits from unknown other bytes, so starting from the given input byte there are expectedly $2^{\mathbb{E}(Z_{BP})} = 2^6$ different output bytes possible and we have a probability of $2^{6-8} = 2^{-2}$ that the given output byte is possible. In all, we can express this in $\log_2$ as:

$$\log_2(\mathbb{E}[|L_a|]) = U_a + \mathbb{E}(Z_{BP}) - 8 = U_a + 6 - 8 = U_a - 2.$$

Table 2 contains the expected List sizes for all $a \in [0, 7]$ as well as the expected amount of storage needed. To calculate the amount of expected storage needed, we calculate the expected List size, $\mathbb{E}[|L_a|]$, and use 8 bytes to store a key.

We can now also calculate the expected List sizes when considering multiple byte paths. Let $A \subseteq [0, 7]$ be a selection of byte paths and let $n = |A|$. Let U_A denote the number of unique key bits used by the byte paths in A.

Let Z_n be the random variable denoting the number of bits n byte paths use from other byte paths. When considering $n > 1$ byte paths, Mix does not use an average of $6n$ unknown neighbouring bits. This is because the probability that the other selected nibble index b in MixN falls in one of non-selected byte paths is no longer the factor $\frac{3}{4}$ used in $\mathbb{E}(Z_{BP})$, but $\frac{3}{4} \cdot \frac{8-n}{7}$, which leads to the following equation:

$$\mathbb{E}(Z_n) = n \cdot \frac{8-n}{7} \cdot \mathbb{E}(Z_{BP}) = \frac{6}{7} n(8-n).$$

Then, to estimate the size of L_A we need to subtract a total of $8n$ bits, since we are filtering on $8n$ given output bits. In all we get

$$\log_2(\mathbb{E}[|L_A|]) = U_A + \mathbb{E}(Z_n) - 8n = U_A + \frac{6}{7} n(8-n) - 8n.$$

5 Key Recovery Attacks on SBTcipher

5.1 The List Merge Attack

We will now sketch a first approach to attack SBTcipher. First, we generate all single byte path Lists and save them in memory. Given some order of the Lists, we pick an element pk_a out of the first List, say L_A, then we pick an element pk_b out of the second Lists, say L_B and we check their compatibility in two ways. For this we first define the overlap of their respective key masks, $\text{OVERLAP}_{AB} = \text{KEYMASK}_A \wedge \text{KEYMASK}_B$, and then check if the partial keys agree under this overlap: $pk_a \wedge \text{OVERLAP}_{AB} = pk_b \wedge \text{OVERLAP}_{AB}$. Then we check if the new combined partial key is still considered valid for the byte paths a and b, i.e., if $\text{VALID}(pk_a \vee pk_b, \{a, b\}) = 1$.

There are now two ways to proceed. (1) Either perform a breadth-first search and check compatibility of all partial keys in L_a with all partial keys in L_b to obtain a new List L_{ab} (the naive merging strategy). Then proceed to merge L_{ab} with L_c to obtain L_{abc}, etc. Or (2) we could continue in a depth-first search: given pk_{ab} pick elements from the next List L_c checking compatibility, then for every resulting element pk_{abc} we recursively continue with the next List L_d, etc.

We use the latter strategy which we refer to as the *streaming List merge*. In this manner we effectively only operate on a single element of each merged List, and we avoid storing entire merged Lists which significantly reduces the memory complexity.

We estimate the time complexity of this algorithm in the number of calls to the VALID oracle. Naturally, this complexity changes when we consider different orders of the Lists. In particular, the complexity is in direct proportion with the increase of U_A. When merging Lists L_A and L_B (w.l.o.g. let $|L_A| > |L_B|$) into L_{AB}, the complexity to calculate List L_{AB} is approximately

$$|L_A| \cdot 2^{U_{AB} - U_A - 2}.$$

To minimize this quantity, we want U_{AB} to be minimal. To achieve this, we have to consider all selections of byte paths of some given size $n \in [0, 7]$ which minimizes U_{AB}. The second column of Table 3 shows all such byte path combinations.

Optimizing the merging order. Generating the single byte path Lists L_7, ..., L_0 can be done as a precomputation, hence the main attack costs consists of merging them, which needs to be done *in the most optimal manner*.

To find the most efficient path of combining Lists, we need to analyse the key bits every byte path uses in SBTcipher. A single byte path uses 4 key bits in Mix and 1 key bit in NS every round, meaning every byte path can depend on at most $(4 + 1) \cdot 8 = 40$ key bits.

Table 1 shows the number of unique key bits each of the byte paths use. Rather curiously we see that all byte paths use around half the optimal key bit dependency as observed in issue (i5).

Table 3. The combinations of byte paths that use the least amount of unique key bits per n active byte paths. Underlined is the inclusion chain we use. After merging 5 byte paths it is more efficient to exhaustively enumerate the remaining key space.

n	Byte paths (A)	U_A	$\mathbb{E}(Z_n)$	$\log_2(\mathbb{E}(\lvert L_A\rvert))$	$\log_2$(compl.)	ρ
1	1	19	6	17	-	2
2	21	31	72/7	25.29	27	5.71
3	721, 521	41	90/7	29.86	33.29	11.14
4	7321, 7642, 6541	47	96/7	28.71	33.86	18.29
5	73210, 765421	51	90/7	23.86	30.71	27.14
6	763210, 76421, 765320, 764321, 754320, 753210	54	72/7	16.29	24.86	37.71
7	7653210, 7654320	55	6	5	15.29	50
8	76543210	56	0	0	4	56

To find the most optimal merging path for obtaining $\mathcal{L}$, we first need to consider which combinations of byte paths use the least amount of unique key bits for a given number of byte paths.

Table 3 contains the results of this analysis. The second column shows which combination of byte paths use the least amount of unique key bits, given the size of the byte path selection A. For all these selections of byte paths, we can calculate $\mathbb{E}(Z_n)$ and combine these results to calculate the expected List sizes for this attack and the time complexity of the merge.

This table also shows the reduction factor ρ, which illustrates how well this attack reduces the size of the full List, i.e., $\rho = U_A - \log_2(\mathbb{E}(|L_A|))$. One can see that the reduction factor grows more quickly than the 2-bits advantage each byte path List gives individually. The effect is entirely due to the VALID-oracle calls inside the merging: without it the 8 byte path Lists combined would only result in a 16-bit reduction of the key space.

Surprisingly, a unique optimal merging strategy emerges: the underlined entries form the only optimal merging path.

The overall attack. The overall attack basically consists of the streaming List merge and using the optimal merge order, of course immediately stopping when the correct key has been recovered. However, we can slightly further optimize the attack by only using the streaming List merge on the first 5 Lists. After that there are 51 key bits guessed and only 5 key bits remain unknown, on which we can do a simple exhaustive search testing each with a simple call to SBTCIPHER. This trick effectively removes the need to compute and store 3 byte path Lists, including the largest single byte path: byte path 6.

The depth first exhaustive search takes a maximum of 2^5 SBTCIPHER calls, in contrast to the 2^3 VALID calls it would have taken without the cut-off. In our implementation, one VALID call is 3.36 times *slower* than one SBTCIPHER call, meaning the 2^3 VALID calls execute in roughly the same time as 2^5 SBTCIPHER calls. Faster implementation methods can make the gap between these smaller, but the main benefit lies in reducing the memory cost. Switching to exhaustive

search earlier, say after merging the first 4 Lists, actually results in a noticeable slower attack, thus 5 Lists is deemed optimal.

The overall attack is thus as follows:

1. Generate all single byte path Lists;
2. Using the optimal merging order, start the streaming List merge over the first 5 Lists;
3. When 51 key bits have been guessed, do exhaustive search over the remaining 5 key bits.

Enumerating partial keys by likelihood. One further optimization that is possible is to enumerate the partial key candidates in the first List based on the order of their likelihood.

This likelihood can be computed during the precomputation of the single byte path Lists, and arises from the uncertainty of crumb values in MixN. When generating a single byte path List, it can occur that a crumb is needed from a non-selected byte path which is an 'unknown crumb'. This is solved by exhaustively trying all four possible values for such an 'unknown crumb'.

Thus during the precomputation of the single byte path Lists, one can for any partial key compute all possible 'byte path solutions' consisting of choosing specific values for 'unknown crumbs'. Here we can assign each possible crumb value the same probability of 1/4. Thus the probability of each 'byte path solution' is $(1/4)^i$ where i is the number of 'unknown crumbs' with a chosen value. To assign a likelihood to a partial key, we add all probabilities of the found 'byte path solutions' for that partial key.

These key likelihoods can easily be calculated during the generation phase of the single byte path Lists. Storing the partial key likelihoods in a double-precision float next to the partial key, essentially doubles the memory needed to store the single byte path List.

5.2 Branch-and-Prune Attack

We developed another attack which needs very little memory. The core idea is to maintain sets of potential values for each internal nibble, also referred to as *domains*, and for each internal operation apply *constraint propagation* (CP) to reduce the involved domains. If any of the domains is reduced to the empty set then an *inconsistency* is found and no solution is possible.

At a high-level we apply this idea in a *Branch-and-Prune* (BnP) strategy: we explore the keyspace in a depth-first search tree guessing key bit by key bit. After guessing a key bit we apply constraint propagation and check for inconsistency: one of the domains is empty. If an inconsistency is found then we backtrack, otherwise we continue with the next key guess.

Constraint Propagation. More specifically, the variables we maintain domains for are: key bits (1 bit), neighbour crumbs (2 bits), state nibbles (4 bits).

Hence for each variable X (of b-bits) we maintain a domain $\mathcal{D}_X$ that is a subset of all possible values: $\mathcal{D}_X \subset \{0,1\}^b$, for $b = 1, 2, 4$. Initially all domains are maximal, i.e., consist of all possible values, except the domains corresponding to the nibbles of the known input and output block which are of size 1. The known input block also ensures that the input block bits used within the KeyExpansion are fully known and the uncertainty of the control bits value lies only in the uncertainty of the key bit values.

The high-level algorithm may adjust a key-bit domain to a single value, after which constraint propagation is applied as follows:

1. *Key pass*: for every control bit (40 per round) its domain is jointly updated with the domain of key bit it depends on.
2. *Forward pass*: propagate through all round functions for all rounds $r = 0, \ldots, 7$:
   (a) *MixN*: for each individual MixN call to update a nibble on position a in SBTcipher, we're going to propagate constraints from the corresponding input domain $\mathcal{D}_{in,a}$ into the output domain $\mathcal{D}_{out,a}$. We first combine the two individual control bit domains into a 2-bit domain $\mathcal{D}_k \subset \{0,1,2,3\}$ for the control value k. Then for each possible control value k we can determine the 'neighbour' nibble position b and compute a temporary output domain $\mathcal{D}_{out,a,k}$ given the input domains of nibbles a and b. If $\mathcal{D}_{out,a,k} \cap \mathcal{D}_{out,a} = \emptyset$ then that value of k can be removed from $\mathcal{D}_k$. Finally we update $\mathcal{D}_{out,a}$ as:
   $$\mathcal{D}_{out,a} \leftarrow \mathcal{D}_{out,a} \cap (\cup_{k \in \mathcal{D}_k} \mathcal{D}_{out,a,k}).$$
   (b) *ByP*: this is a simple permutation of the nibble domains corresponding to how the byte permutation operates on the nibbles.
   (c) *NS*: for each byte i, we consider the corresponding control bit domain $\mathcal{D}_k \subset \{0,1\}$ and the low and high nibble input domains $\mathcal{D}_{in,0}, \mathcal{D}_{in,1}$ to propagate into the corresponding output domains $\mathcal{D}_{out,0}, \mathcal{D}_{out,1}$:
   $$\mathcal{D}_{out,0} \leftarrow \mathcal{D}_{out,0} \cap (\cup_{k \in \mathcal{D}_k} \mathcal{D}_{out,0+k}), \quad \mathcal{D}_{out,1} \leftarrow \mathcal{D}_{out,1} \cap (\cup_{k \in \mathcal{D}_k} \mathcal{D}_{out,1-k}).$$
   (d) *SB*: For nibble i, we simply propagate forward using:
   $$\mathcal{D}_{out} \leftarrow \mathcal{D}_{out} \cap \{SB_i(x) : x \in \mathcal{D}_{in}\}.$$
3. *Backward pass*: propagate backwards through all round functions for rounds $r = 7, \ldots, 0$. We use the same approaches described for the forward pass, but revert the direction and update the input domain from the output domain.
4. *Forward pass 2*: we repeat the forward pass once more, as this improves the probability of finding an inconsistency earlier. Another backward pass was also considered, as well as adaptively repeating forward and backward passes until the domains stabilize, but the benefits did not outweigh the additional cost in the overall attack.

If during any step of the above passes a resulting domain becomes empty then the system is found to be inconsistent: no solution is possible. In that case, a global fail flag is set and constraint propagation immediately returns, without completing the rest of the passes.

Branch-and-Prune. The overall attack uses a simple Branch-and-Prune strategy: a depth-first search tree of the keyspace. At each node, it first makes a copy of the domains from its parent, then applies the key guess as constraint to its own copy of the domains and calls the constraint propagation. If an inconsistency (empty domain) is found then it backtracks, otherwise continues with the 2 children nodes corresponding to the next key bit guess.

The order in which the key bits are guessed is important, and we directly determine them from the earlier attack's byte path merge order: 1, 2, 7, 3, 0, 6, 5, 4. We first add all key bits belonging to byte path 1, then those of byte path 2, then 7, etc. For each byte path we add the key bits by looking at their order of occurrence in Mix and NS in round order 0, 7, 1, 6, 2, 5, 3, 4. This results in the following key bit order:

$$
\begin{aligned}
(BP1:)&\ 11, 39, 14, 42, 7, 12, 40, 15, 43, 55, 18, 46, 3, 19, 4, 32, 35, 45, 13,\\
(BP2:)&\ 17, 20, 48, 16, 44, 47, 21, 9, 37, 5, 23, 51,\\
(BP7:)&\ 22, 50, 26, 25, 53, 49, 34, 0, 28, 10,\\
(BP3:)&\ 54, 6, 30, 31, 2, 33,\\
(BP0:)&\ 8, 36, 24, 38,\\
(BP6:)&\ 41, 52, 27,\\
(BP5:)&\ 1,\\
(BP4:)&\ 29.
\end{aligned}
$$

5.3 Implementations

Both attacks are fully implemented in `C++` to measure attack costs. We will publish all our implementations open-source under the MIT license.

Below we first discuss some implementation aspects, and then present the experimental results on the attacks.

The Branch-and-Prune attack. We represent domains in bit masks, such that computing the intersection and union of domains is simply a binary AND and OR operation. To speed up the implementation, we heavily rely on look-up tables to replace entire loops over all values in domains in the constraint propagation. The attack supports using more then one pair of input/output blocks in which case it maintains separate instances of the domains for each such pair. At every key search tree node it applies the key bit guess and constraint propagation to each instance and backtracks as soon as at least one instance finds an inconsistency.

Note that the whole BnP attack only effectively needs memory for the code, to store copies of the domains for each search tree level, and some look-up tables to speed-up the implementation, which together should easily fit 640KiB that was commonly available in computers in the mid-1980s. The current implementation effectively needs more memory (3.3MiB) due to the use of `C++` and not spending time to minimize other memory usage.

The List Merge attack. The high-level attack is straightforward to implement for the 1 input/output block pair case. It should be noted that given more than 1 input/output block pairs (IO-pairs), for each extra IO-pair the attack does the following: (1) it computes the single byte path Lists for that IO-pair by only enumerating over those partial keys present in the corresponding main IO-pair's List. (2) filters the main IO-pair's List to the intersection of both Lists. Although the extra IO-pairs are effectively only used to filter the main IO-pair's single byte path Lists, and not during the merging step, the List sizes shrink rapidly with each extra IO-pair.

Now the main missing component is how to implement VALID. We have separate implementations for VALID for single byte paths and for VALID during the List merging. For single byte paths, VALID exhaustively enumerates all possible evaluation paths, and whenever a crumb from another byte is needed it exhaustively tries all 4 possible values. To optimize the cost we use a meet-in-the-middle strategy: it produces a List of all attainable values of the byte from the input byte after 4 rounds, as well as a List of all attainable values of the byte working backwards 4 rounds from the output byte. Finally, the partial key is deemed valid if and only if the intersection of the two sets is non-empty.

This procedure can be enhanced to produce a set of all possible entire *byte path solutions*: each consisting of specific intermediate byte values and the specific external crumb values that were used. The set of byte path solutions is useful for two reasons: (1) we can use it to compute the probability of each partial key which is used to speed up the attack, (2) we can reuse the set of byte path solutions to implement VALID during the List merging.

We implemented two variants for VALID to use during List merging: The first 'merge-VALID' reuses the byte path solutions: to merge two partial keys, each with corresponding sets of byte path solutions, VALID produces a merged set of solutions by checking every combination for consistency. The merged partial key is deemed valid if and only if the merged set of solutions is non-empty. Unfortunately, the cost of a VALID call depends on the number of combinations it needs to check, which grows with the number of byte paths involved.

For the second 'merge-VALID' we simply reused the constraint propagation of the BnP attack. By restricting the domains for input/output bytes and guessed key bits to those specific values, and keeping all other domains maximal, we can apply constraint propagation to effectively check each partial key with a fixed cost. Although this variant may find not inconsistencies that the byte-path-solution-based VALID above would, a call to this VALID implementation is much cheaper and independent of the number of byte paths involved.

While we were able to implement and benchmark the byte-path-solution-based VALID, the CP-based VALID variant proved to be much better and is used for the remainder of the paper.

The current implementation uses up to 300 MiB, which is a lot for mid-1980s-standards. Though this is likely higher than needed due to memory fragmentation, especially during the generation of the single byte path Lists. Actually precomputing the Lists and loading them would probably already reduce mem-

Table 4. Experimental results for different attacks and for 0,1,2 or 3 extra pairs (EP).

			Precomp. (s)			**Attack time (s)**			**Memory (MiB)**		
		EP	Q1	med.	Q3	Q1	med.	Q3	Q1	med.	Q3
List Merge	Lexicographic	0	120	131	139	19442	45692	69831	220	227	235
	Key order	1	156	169	183	323	592	998	253	276	292
		2	166	181	195	5	13	21	271	295	314
		3	169	184	198	0	0	0	274	299	321
	Key likelihood	0	120	131	139	3981	12429	48018	220	227	236
	order	1	155	169	183	20	103	342	253	277	292
		2	166	182	196	0	1	3	270	295	314
		3	169	183	200	0	0	0	275	299	321
BnP		0	-	-	-	18960	33311	66215	3	3	3
		1	-	-	-	179	365	520	3	3	3
		2	-	-	-	7	15	22	3	3	3
		3	-	-	-	2	5	8	3	3	3

ory usage. We also did not spend any time on memory optimization, so a further reduction is quite likely.

5.4 Results

We have benchmarked three attacks:

- the List Merge attack without key-likelihood-sorting,
- the List Merge attack with key-likelihood-sorting,
- the Branch-and-Prune attack.

We also benchmarked using multiple input/output block pairs. The first input block is always the fixed B_0, whereas the extra input blocks are sampled uniformly at random. The corresponding output blocks are simply computed by calling SBTcipher.

We generated a list of 20 random encryption keys, and for each key ran the three attacks with 0,1,2 and 3 extra input/output block pairs, resulting in a total of 12 runs per key. This enables us to compare the two main attacks, as well as evaluate the effect of key sorting and of using extra input/output pairs.

The machine we ran all benchmarks on is our server with two Intel Xeon Gold 6248 CPUs (20 cores at 2.5 GHz each) and 1.5 TB RAM, attacks were run single-threaded and leaving at least one core per CPU free. Table 4 contains the summary results of the experiments. The table shows statistics on the runtime in seconds (median and the first and third quartile (Q1, Q3, resp.)), as well as the memory used by the attack. For the List Merge attack we distinguish between precomputation (generating the single byte path Lists), and the main attack (the streaming merge).

Enumerating the partial keys according to likelihood has a significant effect on the List Merge attack, gaining a factor 3 on the median time. Large outliers do exist, though, as the Q3 time is only 30% lower. In fact the List Merge attack

with key-likelihood-sorting is the clear winner in attack time complexity of the three attacks for every choice of number of extra pairs. The only aspects where the Branch-and-Prune attack has a clear advantage is that no precomputation and almost no memory are necessary.

Adding extra plaintext-ciphertext pairs has a very significant effect on the attack time. Adding even a single extra pair reduces the median attack time by two orders of magnitude, while only increasing the memory usage by 30-60 MiB for the List Merge attack. Memory usage for the Branch-and-Prune attack remains constant at around 3 MiB. Adding another pair reduces attack time to mere seconds and this is reduced to milliseconds when adding a third extra pair.

One may wonder how well these attacks would perform on mid 1980s high end hardware, and that is a difficult question to answer. First, the current memory usage of the List Merge attack is quite high. But with further optimizations we think it is not unreasonable to expect the attack could be made to fit on a Cray X-MP (released 1982), or Cray 2 (1985). The Branch-and-Prune attack even fits 640KiB that personal computers commonly had since the mid 1980s.

Second, the attack running times with 0 extra pairs is quite feasible on modern hardware, but hard to imagine an effective key recovery attack to run daily potentially against several different encryption keys in the 1980s. However, attack running times quickly drop by adding extra pairs, and the attack with even a single extra pair can be quite plausible both from the runtime as from an intelligence gathering standpoint.

6 Forensic Cryptanalysis

In this section we present a first application of the proposed forensic methodology in section 2. This is really a first application building on top of the reverse-engineering and cryptanalysis. Our goal here is try to answer the question whether the methodology can be used to gather more evidence about the backdoor and the original attack, trying to exploit the mathematical structure embedded that is predicted by the Backdoor Conjecture. A more exhaustive application to UA-8295 would cost significant effort beyond the scope of this paper and left to future work, as well as application to other real-world backdoors.

We have two candidate attacks described in section 5: List Merge and Branch-and-Prune. Both attacks depend on two hidden weaknesses: (i3) weak mixing in Mix, and (i5) strong interaction between KeyExpansion and ByP. Thus in particular, we have looked at two design freedoms: (1) Choice of byte permutation ByP and (2) Choice of IV that are treated below, as well as special properties of the attack in the attack design space.

Our prediction is that that ByP is a class (i) design freedom: optimized for the original attack and that IV is a class (ii) design freedom: not attack relevant and chosen to look secure.

6.1 Choice of Attack Design

In the design of the List Merge attack we evaluated which sets of n byte paths are optimal: i.e., having a minimum amount of active key bits and expected List size. It was surprising that there exists a clear optimal inclusion chain of these local optima (cf. Table 3), which is due to the strong interaction between KeyExpansion and ByP (issue (i5)). We thus consider these to be class (i) design freedoms, and the unique global optimum consisting of local optima is an unexpectedly strong structure in the attack design space that it is strong evidence that this was by design.

6.2 Choice of ByP

As a forensic experiment, we have also analysed how different choices of ByP affect the List Merge attack, whose estimate cost is the easiest to calculate. We only consider ByP and attack design spaces that match properties of the current ByP, denoted by ByP_{org}, and List Merge attack. In particular, we're looking at the cost of List Merge attacks built from inclusion chains of local List-size optima for a given permutation (cf. Table 3).

Exploring a large design space (ByP and KeyExpansion) together to design a hidden weaknesses may require a cheap cost predictor to find good candidates. We explored several cost predictors, including the exact expected attack cost:

- M_1: exact total expected number of valid calls
- M_2: sum of unique key bits per byte path
- M_3: maximum of unique key bits per byte path

Especially for M_2 and M_3 we consider the rank under those metrics, but also the rank when discounting higher-ranked permutations that actually lead to worse attacks.

We consider different sets of permutations that ByP_{org} is a member of:

- A: permutations on 8 elements (S_8)
- D: permutations without fixed points on 8 elements (derangements, D_8)
- C: permutations that are 8-cycles
- N: permutations $\sigma \in S_8$ s.t. $|\sigma(i) - \sigma(i+1)| \neq 1$ for all i

The first is a control set, while the latter three are arguably more realistic candidate sets for a secure looking byte permutation. Note that restricting to attacks built from inclusion chains of local List-size optima reduces the effective number of permutations under consideration.

The ranking results are presented in Table 5. Within each permutation set there are sufficiently many better permutations under the exact metric M_1. However, under M_2 ByP always surfaces as the highest ranked permutation. So while ByP is not the optimal choice under the exact attack cost metric, it is under a realistic cheap predictive metric.

Table 5. Ranking of BYP_{org} in sets of permutations under three metrics.

Permutation set	Metric	Rank	Rank no-worse
A	M_1	73/3435	73/73
D	M_1	59/1462	59/59
C	M_1	37/844	37/37
N	M_1	23/485	23/23
A	M_2	1/3435	1/73
D	M_2	1/1462	1/59
C	M_2	1/844	1/37
N	M_2	1/485	1/23
A	M_3	7/3435	6/73
D	M_3	7/1462	6/59
C	M_3	7/844	6/37
N	M_3	4/485	3/23

6.3 Choice of IV Analysis

As an experiment, we altered SBTcipher by choosing the constant IV from one of two classes of values and measure attack time.

The fixed value $IV = \texttt{CF17AF8A107AC0F5}_{16}$ is used as input for the mode of operation as described in section 3.5. This fixed value determines $B_0 = \text{LFSR}^{64}(IV)$, which determines the fixed first state $\text{BiP}(B_0)$ as it enters Mix in the first round. Recall that the weak mixing is caused by that a nibble update by Mix is only affected by another nibble's value if the predicate $[k_1 c_0 + \overline{k_1} c_1 \equiv 3k_0 \bmod 4]$ is true, which happens with say probability 1/4. Hence, it might be possible to choose the IV such that $\text{BiP}(\text{LFSR}^{64}(IV))$ biases the probability of nibble mixing to improve the attack. Moreover, even if there is any structure at the first Mix call, any visible structure in $\text{BiP}(B_0)$ is obfuscated in IV by both LFSR^{64} and BiP, which makes this essentially a free choice for the designers. Let IV_{org} denote the original value $\texttt{CF17AF8A107AC0F5}_{16}$. We define the *max-prob-wrap* class as all IV-values that maximizes the probability of needing an unknown crumb in MixN during the first round. I.e. all IV such that $\text{BiP}(\text{LFSR}^{64}(IV)) \in \{0,3\}^{32} \subset (\mathbb{Z}_{2^2})^{32}$. The *min-prob-wrap* class does the opposite and minimizes the probability of needing an unknown crumb in MixN during the first round, i.e., $\text{BiP}(\text{LFSR}^{64}(IV)) \in \{1,2\}^{32} \subset (\mathbb{Z}_{2^2})^{32}$.

Results of experiments done where the value of IV is changed can be found in Table 6. We see that choosing IV from the *max-prob-wrap* class indeed slightly increases attack time. Furthermore, choosing IV in *min-prob-wrap* also decreases attack time. We also see that this effect disappears when increasing the number of extra pairs that are given to the attack, since these are chosen at random.

In general the differences are relatively small within a factor 1.25. So technically the IV affects the attack complexity and it may not have been chosen optimally. However the effect is rather small and there does not seem to be structure in the chosen IV. It does provide some evidence against the Branch-and-Prune attack, but given the small effect that would be quite weak evidence.

Table 6. Experimental Branch-and-Prune attack results when considering different classes of values for IV and 0,1,2, or 3 extra pairs (EP).

		IV_{org}			**min-prob-wrap**			**max-prob-wrap**		
		Attack time (s)			**Attack time (s)**			**Attack time (s)**		
	EP	Q1	med.	Q3	Q1	med.	Q3	Q1	med.	Q3
BnP	0	18960	33311	66215	15687	33877	59379	22573	39444	73914
	1	179	365	520	157	264	412	179	290	498
	2	7	15	22	6	12	16	7	13	19
	3	2	5	8	2	5	7	2	5	7

The most likely explanation is that this was considered a class (ii) design freedom by the designers and chosen to look secure.

7 Other UA-8295 Cryptographic Design Issues

As described in section 4, we found several cryptographic design issues in the UA-8295. The most severe, (i3) weak mixing and (i5) interaction between KeyExpansion and ByP, were used in the previous sections to mount an attack on SBTcipher. We consider issue (i4) (control bits also depend on the plaintext bits) an uncommon design choice, but see no clear security risk. Perhaps it was used to obfuscate statistical aberrations arising from (i3) and (i5). Below we cover the two remaining issues in more detail.

7.1 Nonce Independence

The first eight bytes of key stream are $(z_7^0, \ldots, z_0^0) = \text{SBTcipher}(K, B_0)$, where $B_0 = \text{LFSR}^{64}(\texttt{CF17AF8A107AC0F5}_{16})$ is constant. These first eight key stream bytes thus only depend on the encryption key. This is a classic setting to apply a brute-force key recovery time-memory trade-off attack like Hellman's attack. However, the SBTcipher weaknesses and attacks we presented above make it seem unlikely such a brute-force attack was the goal.

Nevertheless, this issue does enable precomputations. In particular for the List Merge attack one can precompute single byte path Lists for every applicable combination of input and output byte values. Making Lists for all 256×256 combinations for 5 byte positions results in $327,680$ precomputed Lists, demanding a massive amount of storage. Whereas if the input byte is always fixed, one only needs $1,280$ precomputed Lists.

7.2 Nonce Generation

For every message, the UA-8295 pseudorandomly generates a 5-byte nonce L, which is sent as the first block of letters. The first three bytes are actually used within the mode of operation, while the last two bytes are dummy values.

The i-th nonce byte is generated as $L_{5-i} = tl_0 \oplus \hat{z}_i$, where tl_0 is the name of a hardware counter register and $\hat{z}_i$ is the i-th byte of the keystream of the *previous* message. The 5 nonce bytes are generated in a tight loop, with a fixed number of CPU instructions between every read of the hardware counter tl_0. Therefore one can expect the amount of hardware counter ticks between generating adjacent nonce bytes is very predictable, and consequently, the difference values $L_{i-1} - L_i$ should exhibit strong biases for $i = 1, \ldots, 4$.

To analyse this, we consecutively encrypted sample messages on the UA-8295 with user key 15× '`A`' and plaintext '`AAA`' and calculated the difference between nonce characters. In 75.5% of the obtained 388 cases, the difference was exactly 15 and in a further 14.9% of all cases the difference was 12. These two values alone already account for over 90% of all nonce differences.

Curiously, this nonce generation issue was already present in the Nokia DA-8520 on which the UA-8295 is based. A question remains why this was the case. At least this issue was not deliberately inserted, but potentially deliberately kept, by the NSA into the UA-8295.

7.3 Fixed Format Messages

One direction we did not explore was that of the fixed format message. It is possible on the UA-8295 to send a message that conforms a user selected format, with the possibility of appending a free-format message.

The manual does not detail how to create a new format. The UA-8295 comes pre-programmed with a format to communicate coordinates, bearing and range:

```
COORDINATES X:???? Y:???? BEARING:???? RANGE:????,
```

where every '`?`' can be substituted for a number. At present, we do not know how such a fixed format message is encrypted. It is possible that the entire string is used as plaintext, or that the format is encoded and used as plaintext appended to the user-inserted data. In either case a fixed-format message could act as a crib, further increasing practicability for known-plaintext attacks such as the ones described in section 5, but more research on this has to be carried out. In particular, that part of the binary code is not yet reverse-engineered.

References

1. Alich, J., Askeland, A., Banik, S., Beyne, T., Canteaut, A., Felke, P., Leander, G., Meier, W., Stennes, L.: Observations on TETRA encryption algorithm TEA-3. IACR Trans. Symm. Cryptol. **2025**(1), 276–308 (2025). https://doi.org/10.46586/tosc.v2025.i1.276-308
2. Amzaleg, D., Dinur, I.: Refined cryptanalysis of the GPRS ciphers GEA-1 and GEA-2. In: Dunkelman, O., Dziembowski, S. (eds.) EUROCRYPT 2022, Part III. LNCS, vol. 13277, pp. 57–85. Springer, Cham (May / Jun 2022). https://doi.org/10.1007/978-3-031-07082-2_3
3. Baksi, A., Bhattacharjee, A., Breier, J., Isobe, T., Nandi, M.: Big brother is watching you: A closer look at backdoor construction. Cryptology ePrint Archive, Report 2022/953 (2022), https://eprint.iacr.org/2022/953
4. Ball, J., Borger, J., Greenwald, G.: Revealed: how US and UK spy agencies defeat internet privacy and security. The Guardian (September 2013), https://www.theguardian.com/world/2013/sep/05/nsa-gchq-encryption-codes-security
5. Beierle, C., Beyne, T., Felke, P., Leander, G.: Constructing and deconstructing intentional weaknesses in symmetric ciphers. In: Dodis, Y., Shrimpton, T. (eds.) CRYPTO 2022, Part III. LNCS, vol. 13509, pp. 748–778. Springer, Cham (Aug 2022). https://doi.org/10.1007/978-3-031-15982-4_25
6. Beierle, C., Derbez, P., Leander, G., Leurent, G., Raddum, H., Rotella, Y., Rupprecht, D., Stennes, L.: Cryptanalysis of the GPRS encryption algorithms GEA-1 and GEA-2. In: Canteaut, A., Standaert, F.X. (eds.) EUROCRYPT 2021, Part II. LNCS, vol. 12697, pp. 155–183. Springer, Cham (Oct 2021). https://doi.org/10.1007/978-3-030-77886-6_6
7. Bellare, M., Jaeger, J., Kane, D.: Mass-surveillance without the state: Strongly undetectable algorithm-substitution attacks. In: Ray, I., Li, N., Kruegel, C. (eds.) ACM CCS 2015. pp. 1431–1440. ACM Press (Oct 2015). https://doi.org/10.1145/2810103.2813681
8. Bellare, M., Paterson, K.G., Rogaway, P.: Security of symmetric encryption against mass surveillance. In: Garay, J.A., Gennaro, R. (eds.) CRYPTO 2014, Part I. LNCS, vol. 8616, pp. 1–19. Springer, Berlin, Heidelberg (Aug 2014). https://doi.org/10.1007/978-3-662-44371-2_1
9. Bernstein, D.J., Lange, T., Niederhagen, R.: Dual EC: A standardized back door. In: Ryan, P.Y.A., Naccache, D., Quisquater, J. (eds.) The New Codebreakers - Essays Dedicated to David Kahn on the Occasion of His 85th Birthday. Lecture Notes in Computer Science, vol. 9100, pp. 256–281. Springer (2016). https://doi.org/10.1007/978-3-662-49301-4_17, https://doi.org/10.1007/978-3-662-49301-4_17
10. Beyne, T., Li, C.: Cryptanalysis of the MALICIOUS framework. Cryptology ePrint Archive, Report 2020/1032 (2020), https://eprint.iacr.org/2020/1032
11. Biryukov, A., Perrin, L., Udovenko, A.: Reverse-engineering the S-box of Streebog, Kuznyechik and STRIBOBr1. In: Fischlin, M., Coron, J.S. (eds.) EUROCRYPT 2016, Part I. LNCS, vol. 9665, pp. 372–402. Springer, Berlin, Heidelberg (May 2016). https://doi.org/10.1007/978-3-662-49890-3_15
12. Boon, F., Derix, S., Modderkolk, H.: Document Snowden: Nederland al sinds 1946 doelwit van NSA. NRC (November 2013), https://www.nrc.nl/nieuws/2013/11/23/nederland-sinds-1946-doelwit-van-nsa-a1429490
13. Checkoway, S., Maskiewicz, J., Garman, C., Fried, J., Cohney, S., Green, M., Heninger, N., Weinmann, R.P., Rescorla, E., Shacham, H.: A systematic analysis of the juniper dual EC incident. In: Weippl, E.R., Katzenbeisser, S., Kruegel,

C., Myers, A.C., Halevi, S. (eds.) ACM CCS 2016. pp. 468–479. ACM Press (Oct 2016). https://doi.org/10.1145/2976749.2978395
14. Checkoway, S., Niederhagen, R., Everspaugh, A., Green, M., Lange, T., Ristenpart, T., Bernstein, D.J., Maskiewicz, J., Shacham, H., Fredrikson, M.: On the practical exploitability of dual EC in TLS implementations. In: Fu, K., Jung, J. (eds.) USENIX Security 2014. pp. 319–335. USENIX Association (Aug 2014), https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/checkoway
15. Courtois, N.T.: On the existence of non-linear invariants and algebraic polynomial constructive approach to backdoors in block ciphers. Cryptology ePrint Archive, Report 2018/807 (2018), https://eprint.iacr.org/2018/807
16. Crépeau, C., Slakmon, A.: Simple backdoors for RSA key generation. In: Joye, M. (ed.) CT-RSA 2003. LNCS, vol. 2612, pp. 403–416. Springer, Berlin, Heidelberg (Apr 2003). https://doi.org/10.1007/3-540-36563-X_28
17. Crypto Museum: Aroflex - UA-8116 - Rapid offline encryption device (August 2009), https://www.cryptomuseum.com/crypto/philips/aroflex/index.htm
18. Crypto Museum: PX-1000 - Pocket telex with encryption (April 2011), https://www.cryptomuseum.com/crypto/philips/px1000/
19. Crypto Museum: Operation RUBICON (THESAURUS) - The secret purchase of Crypto AG by BND and CIA (December 2019), https://www.cryptomuseum.com/intel/cia/rubicon.htm
20. Crypto Museum: UA-8295 - Short-Burst Message Terminal (January 2022), https://www.cryptomuseum.com/crypto/philips/ua8295/index.htm
21. Dorey, K., Chang-Fong, N., Essex, A.: Indiscreet logs: Persistent Diffie-Hellman backdoors in TLS. Cryptology ePrint Archive, Report 2016/999 (2016), https://eprint.iacr.org/2016/999
22. Hemmert, T.: How to backdoor LWE-like cryptosystems. Cryptology ePrint Archive, Report 2022/1381 (2022), https://eprint.iacr.org/2022/1381
23. Hemmert, T., May, A., Mittmann, J., Schneider, C.R.T.: How to backdoor (classic) McEliece and how to guard against backdoors. In: Cheon, J.H., Johansson, T. (eds.) PQCrypto 2022. pp. 24–44. Springer, Cham (Sep 2022). https://doi.org/10.1007/978-3-031-17234-2_2
24. Jaspers, H., Terlingen, S.: Former Philips top cryptographer admits to compromising encryption devices. Argos (February 2020), https://argos.vpro.nl/artikelen/former-philips-top-cryptographer-admits-to-compromising-encryption-devices-
25. Kwant, R., Lange, T., Thissen, K.: Lattice klepto - turning post-quantum crypto against itself. In: Adams, C., Camenisch, J. (eds.) SAC 2017. LNCS, vol. 10719, pp. 336–354. Springer, Cham (Aug 2017). https://doi.org/10.1007/978-3-319-72565-9_17
26. MacAskill, E., Dance, G.: NSA files: Decoded. What the revelations mean for you. The Guardian (November 2013), https://www.theguardian.com/world/interactive/2013/nov/01/snowden-nsa-files-surveillance-revelations-decoded#section/1
27. Marsiske, S.: NSA's backdoor of the PX1000-Cr. PoC||GTFO **21**, 59–66 (February 2022), https://archive.org/details/pocorgtfo21/page/59/mode/1up
28. Meijer, C., Bokslag, W., Wetzels, J.: All cops are broadcasting: TETRA under scrutiny. In: Calandrino, J.A., Troncoso, C. (eds.) USENIX Security 2023. pp. 7463–7479. USENIX Association (Aug 2023), https://www.usenix.org/conference/usenixsecurity23/presentation/meijer

29. Meijer, C., Bokslag, W., Wetzels, J.: 2 Cops 2 Broadcasting: TETRA End-To-End and Broader Ecosystem Insecurities (2025), https://www.midnightblue.nl/research/2tetra2burst
30. Menn, J.: Exclusive: Secret contract tied NSA and security industry pioneer. Reuters (December 2013), https://www.reuters.com/article/world/exclusive-secret-contract-tied-nsa-and-security-industry-pioneer-idUSBRE9BJ1C5/
31. Metze, M.: Ingelijfd door de NSA [geannoteerde versie]. De Groene Amsterdammer **5** (2014)
32. Miller, G.: The intelligence coup of the century. The Washington Post (February 2020), https://www.washingtonpost.com/graphics/2020/world/national-security/cia-crypto-encryption-machines-espionage/
33. Onkruit: Dossier: Philips USFA/Crypto, wapenfabrikant in moeilijkheden (June 1992)
34. Paterson, K.G.: Imprimitive permutation groups and trapdoors in iterated block ciphers. In: Knudsen, L.R. (ed.) FSE'99. LNCS, vol. 1636, pp. 201–214. Springer, Berlin, Heidelberg (Mar 1999). https://doi.org/10.1007/3-540-48519-8_15
35. Perlroth, N., Larson, J., Shane, S.: N.S.A. Able to Foil Basic Safeguards of Privacy on Web. New York Times (September 2013), https://www.nytimes.com/2013/09/06/us/nsa-foils-much-internet-encryption.html
36. Perlroth, N., Larson, J., Shane, S.: Secret Documents Reveal N.S.A. Campaign Against Encryption. New York Times (September 2013), https://archive.nytimes.com/www.nytimes.com/interactive/2013/09/05/us/documents-reveal-nsa-campaign-against-encryption.html
37. Peyrin, T., Wang, H.: The MALICIOUS framework: Embedding backdoors into tweakable block ciphers. In: Micciancio, D., Ristenpart, T. (eds.) CRYPTO 2020, Part III. LNCS, vol. 12172, pp. 249–278. Springer, Cham (Aug 2020). https://doi.org/10.1007/978-3-030-56877-1_9
38. Posteuca, R., Ashur, T.: How to backdoor a cipher. Cryptology ePrint Archive, Report 2021/442 (2021), https://eprint.iacr.org/2021/442
39. Raggett, R. (ed.): Jane's Military Communications. Jane's Information Group (1986), iSBN 0-7106-0824-1
40. Raggett, R. (ed.): Jane's Military Communications. Jane's Information Group (1988), iSBN 0-7106-0526-X
41. Ravi, P., Bhasin, S., Chattopadhyay, A., Aikata, Roy, S.S.: Backdooring post-quantum cryptography: Kleptographic attacks on lattice-based KEMs. Cryptology ePrint Archive, Report 2022/1681 (2022), https://eprint.iacr.org/2022/1681
42. Rijmen, V., Preneel, B.: A family of trapdoor ciphers. In: Biham, E. (ed.) FSE'97. LNCS, vol. 1267, pp. 139–148. Springer, Berlin, Heidelberg (Jan 1997). https://doi.org/10.1007/BFb0052342
43. Schneier, B., Fredrikson, M., Kohno, T., Ristenpart, T.: Surreptitiously weakening cryptographic systems. Cryptology ePrint Archive, Report 2015/097 (2015), https://eprint.iacr.org/2015/097
44. Shumow, D., Ferguson, N.: On the possibility of a back door in the NIST SP800-90 Dual EC PRNG. Crypto'07 Rump Session (2007), https://rump2007.cr.yp.to/15-shumow.pdf
45. Simmons, G.J.: The prisoners' problem and the subliminal channel. In: Chaum, D. (ed.) CRYPTO'83. pp. 51–67. Plenum Press, New York, USA (1983). https://doi.org/10.1007/978-1-4684-4730-9_5
46. Simmons, G.J.: Authentication theory/coding theory. In: Blakley, G.R., Chaum, D. (eds.) CRYPTO'84. LNCS, vol. 196, pp. 411–431. Springer, Berlin, Heidelberg (Aug 1984). https://doi.org/10.1007/3-540-39568-7_32

47. Simmons, G.J.: The practice of authentication. In: Pichler, F. (ed.) EUROCRYPT'85. LNCS, vol. 219, pp. 261–272. Springer, Berlin, Heidelberg (Apr 1986). https://doi.org/10.1007/3-540-39805-8_31
48. Wong, D.: How to backdoor Diffie-Hellman. Cryptology ePrint Archive, Report 2016/644 (2016), https://eprint.iacr.org/2016/644
49. Yang, Z., Xie, T., Pan, Y.: Lattice klepto revisited. In: Sun, H.M., Shieh, S.P., Gu, G., Ateniese, G. (eds.) ASIACCS 20. pp. 867–873. ACM Press (Oct 2020). https://doi.org/10.1145/3320269.3384768
50. Young, A., Yung, M.: The dark side of "black-box" cryptography, or: Should we trust capstone? In: Koblitz, N. (ed.) CRYPTO'96. LNCS, vol. 1109, pp. 89–103. Springer, Berlin, Heidelberg (Aug 1996). https://doi.org/10.1007/3-540-68697-5_8
51. Young, A., Yung, M.: Kleptography: Using cryptography against cryptography. In: Fumy, W. (ed.) EUROCRYPT'97. LNCS, vol. 1233, pp. 62–74. Springer, Berlin, Heidelberg (May 1997). https://doi.org/10.1007/3-540-69053-0_6
52. Zetter, K.: Researchers Solve Juniper Backdoor Mystery; Signs Point to NSA. WIRED (December 2015), https://www.wired.com/2015/12/researchers-solve-the-juniper-mystery-and-they-say-its-partially-the-nsas-fault/

A Supplementary Material

A.1 Bit Permutation

Let P be the input 64-bit state to SBTcipher. The Bit Permutation BiP permutes its bits according to Table 7. The highest bit 63 of the output is the 54-th input bit, and the lowest bit 0 of the output bit is the 19-th input bit.

Table 7. Bit permutation BiP

(54,	4,	9,	56,	31,	23,	32,	43,
40,	58,	12,	28,	38,	49,	0,	17,
53,	34,	30,	42,	15,	22,	60,	2,
21,	52,	39,	6,	46,	63,	29,	10,
45,	24,	1,	35,	14,	55,	18,	59,
61,	27,	11,	41,	3,	20,	50,	33,
26,	7,	57,	8,	51,	37,	44,	16,
36,	13,	25,	62,	5,	48,	47,	19)

A.2 S-Box Substitution Tables

Table 8. S-Boxes in SBTcipher (in hexadecimal)

	x															
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
$SB_0(x)$	1	5	4	6	C	A	9	F	3	E	8	0	D	7	2	B
$SB_1(x)$	8	D	7	E	5	0	B	A	2	3	C	1	F	4	9	6
$SB_2(x)$	5	2	1	4	D	E	0	9	F	B	6	C	3	A	7	8
$SB_3(x)$	9	7	6	D	B	F	4	C	0	8	2	E	A	1	3	5
$SB_4(x)$	B	A	E	0	9	D	3	2	6	C	F	7	8	5	4	1
$SB_5(x)$	3	E	D	9	1	4	8	6	A	0	B	5	2	F	C	7
$SB_6(x)$	4	9	8	5	0	6	A	E	B	2	7	F	1	3	D	C
$SB_7(x)$	7	F	0	C	A	8	1	B	9	D	5	3	E	2	6	4
$SB_8(x)$	2	4	5	C	9	B	7	8	F	E	D	A	3	1	0	6
$SB_9(x)$	7	D	6	8	1	3	0	4	5	F	2	E	A	C	B	9
$SB_{10}(x)$	B	7	9	5	A	1	F	6	2	C	4	D	E	8	3	0
$SB_{11}(x)$	D	2	C	9	E	7	3	1	4	8	0	F	6	A	5	B
$SB_{12}(x)$	1	8	E	A	7	4	9	D	6	3	B	5	F	0	2	C
$SB_{13}(x)$	8	B	3	E	D	A	4	F	9	0	C	6	5	7	1	2
$SB_{14}(x)$	F	A	8	D	3	0	E	2	C	6	9	1	4	B	7	5
$SB_{15}(x)$	4	F	A	1	B	2	8	0	D	5	6	C	7	3	9	E

A.3 Key Checksum

The UA-8295 can be used in two modes. One can use a user-set key or use a hard coded fixed key. The fixed key found in the ROM is `2V58RGKLXN49TKD`. Note that this fixed key can be changed by the customer and will probably not be the same for different users, unless they belong to the same network. After the user has entered the key, the full key cannot be displayed any more, but a four-character checksum is displayed instead. Algorithm 3 details the computation of this checksum. It computes the first keystream block that ENCRYPT produces with an all zero nonce ($L = 0^{24}$).

Algorithm 3 Checksum algorithm on the UA-8295

Input: $U \in (\mathbb{Z}_{2^8})^{15}$
$K = \text{KEYDERIVATION}(U, (0,0,0,0,0))$
$B_0 = \text{LFSR}^{64}(\texttt{CF17AF8A107AC0F5}_{16})$
$(n_{15}, \ldots, n_0) = \text{SBTCIPHER}(K, B_0)$ ▷ Read SBTCIPHER output as 16 nibbles
$\text{CHECKSUM} = (\texttt{'A'}, \texttt{'A'}, \texttt{'A'}, \texttt{'A'}) + (n_{12} \oplus n_{15}, n_8 \oplus n_{11}, n_4 \oplus n_7, n_0 \oplus n_3)$
Output: CHECKSUM