An Improved Recovery Algorithm for Decayed AES Key Schedule Images

Alex Tsow

The MITRE Corporation*
atsow@mitre.org

Abstract. A practical algorithm that recovers AES key schedules from decayed memory images is presented. Halderman et al. [1] established this recovery capability, dubbed the *cold-boot attack*, as a serious vulnerability for several widespread software-based encryption packages. Our algorithm recovers AES-128 key schedules tens of millions of times faster than the original proof-of-concept release. In practice, it enables reliable recovery of key schedules at 70% decay, well over twice the decay capacity of previous methods. The algorithm is generalized to AES-256 and is empirically shown to recover 256-bit key schedules that have suffered 65% decay. When solutions are unique, the algorithm efficiently validates this property and outputs the solution for memory images decayed up to 60%.

Keywords: anti-tamper, digital forensics, decayed memory, cold-boot attack, AES, key schedule.

1 Introduction

Cold-boot attacks are another troubling example of the increasingly sophisticated threats to security and privacy. In response to these threats we investigate defensive anti-tamper techniques in the hopes of better understanding the potential of specific attack vectors. In this paper we report on our investigation of cold boot attacks and demonstrate that the problem is more serious than previously thought. We present AES key recovery techniques that handle over twice the decay rate of prior methods at comparable computational effort.

The *cold-boot attack* [1] is a serious vulnerability for software-based encryption packages—including BitLocker, FileVault and the open-source project TrueCrypt—where one can recover secret keys from decayed memory images. Decryption with decayed AES keys does not produce original plaintexts. However, the redundancy of key material inherent in the AES key schedule can rectify these faults. When combined with *asymmetric decay*, where bits overwhelmingly decay to their ground state rather than their charged state, this redundancy enables reconstruction of the original key. Heninger and Halderman have developed a recovery algorithm for AES-128 that recovers keys from 30% decayed data in less than 20 minutes about half the time.

* Approved for Public Release; Distribution Unlimited; Tracking Number 09-1872.

Our algorithm recovers keys up to several orders of magnitude faster than Heninger and Halderman's method. One case that took their algorithm more than 10 days to solve was solved by our improved method in 0.047 seconds. The speed increase enables key recovery from more severely degraded memory images. In an experimental evaluation, our algorithm recovered all keys from a 5,000 case test suite at 70% decay, with 4,927 instances recovered in less than 20 minutes—more than twice decay rate with almost double the success rate in 20 minutes. The speed increase also makes it feasible to enumerate *all* keys from which an image could have decayed, rather than halt on the first key that satisfies the decay and schedule constraints. In particular, the algorithm can determine that a solution is unique. Benchmarks demonstrate feasibility up to 60% decay where there is approximately a 2.5× slowdown compared with the halt-on-first-key search. The algorithm generalizes to 256-bit AES with only a moderate drop in the recovery capability. Empirically, the benchmarks show that AES-256 recovery begins to degrade around 65%; there are no other performance claims in the open literature about AES-256 recovery.

The AES key-schedule is the primary source of key redundancy. For the 128-bit version, the original key is bijectively mapped to 10 additional round-keys [2,3]. The mappings form a system of byte-level equations that constrain the space of likely key candidates.

The asymmetric decay property of DRAM provides a second set of constraints. When the refresh cycle of DRAM is interrupted, the data overwhelmingly decays to 0 (or 1 assuming the complementary encoding for the ground state) because the capacitance is lost over time. Occasionally bits invert to the charged state, although Halderman et al. bound these effects at 0.1%. The asymmetric decay property suggests a compatibility criterion for key candidates: if a candidate schedule subset differs from the decayed memory image only by inversions from the ground state, then it is compatible with the decayed memory image. The performance claims for our algorithm, and indeed those in [1] and [4], are based on the *perfect* asymmetry assumption, where no bit in the ground state ever inverts. The algorithm has also been adapted to accommodate inversions to the charged state by generalizing the compatibility criterion to allow a bounded number of such cases. No other logic changes are necessary.

Key reconstruction is possible because key candidates must satisfy both the asymmetric decay property and the system of equations defined by the AES key schedule. Our algorithm explores a tree of one-byte guesses. At each stage, or tree-depth, the new byte candidate and all bytes implied by the schedule equations are checked against the decayed image. The algorithm guesses bytes in an order such that guess n implies values for n schedule bytes for $n < 11$ in the 128-bit version; guesses 11-14 imply an additional 10 bytes each and the last guess implies the schedule's remaining 65 bytes.

Byte guessing proceeds in a depth-first manner. Each stage has 256 possibilities, but the schedule and decay-compatibility constraints quickly prune the possibilities, particularly in the later stages where a single byte guess implies several byte values. The selection and order of byte guessing is not unique.

Section 3.4 describes a path selection heuristic for byte guesses which improves recovery times by a factor of several hundred for decay rates over 50%, when compared with the static path implementation.

We make the following contributions: *1)* our recovery algorithm is several orders of magnitude faster than the best previously published method, *2)* the new method enables key recovery from images with significantly more decay, *3)* it enumerates all solutions to a decay image and *4)* we generalize the method to AES-256 with little loss of decay performance.

Organization: Section 2 reviews related works. Section 3 describes the algorithm in detail, including a heuristic to optimize the exploration path. Section 4 presents benchmarks for the algorithm with and without the path optimization heuristic. Benchmarks for the unique determination capability and AES-256 key recovery are also presented. Section 5 makes some observations about the benchmark results. Section 6 concludes the paper.

2 Related Work

Halderman et al. [1] established the cold-boot attack as a low-cost way to extract private key information from computers running software encryption. In particular, they extracted private keys for full-disk encryption packages such as BitLocker, FileVault, and the cross-platform open-source project TrueCrypt. Heninger and Halderman released proof-of-concept implementation that recovers 128-bit AES key-schedules.[1] It implements the algorithm from [1] which they have found to recover keys from 30% decayed memory within 20 minutes about half the time. Their archives also contain a recovery algorithm for the RSA cryptosystem.

Heninger and Shacham [4] vastly improved the ability to recover RSA private keys from decayed memory images. They improve recovery from 6% decay (running on the order of minutes) to 46% decay (running on the order of seconds) when p, q, d, d_p, and d_q are in the image. The paper further casts their recovery algorithm in terms of *known bits*, so that the bits may be randomly selected rather than simply the result of an asymmetric memory decay. We note that a perfect memory image maps to 50% known bits under the asymmetric decay assumption, since valid ground-state values theoretically could have decayed.

Nearly a decade before the cold boot attack was demonstrated, Handschuh, Paillier, and Stern modeled *probing attacks* [5] on the square-and-multiply algorithm for modular exponentiation, DES, and RC5. They reconstruct cryptographic secrets by tracing a few critical bits over the target operation's execution. Since cold-boot attacks capture a snapshot of the execution state, these techniques only apply if a trace has been preserved in memory.

Akavia, Goldwasser, and Vaikuntanathan [6] present a model of cold-boot memory attacks in terms of experiments with probabilistic polynomial time players. The recovery player chooses a sequence of probing functions that map private

[1] `http://citp.princeton.edu/memory/code/`

keys to bit vectors; this models key material leakage. They define *adaptive* and non-adaptive variants which may or may not alter the choice of probing function in response to the results of previous probes. They further show that the Regev public key cryptosystem [7] is secure under both definitions, but with different leak parameters.

Naor and Segev [8] revisit the above formalism for memory attacks and develop a schema for constructing public key cryptosystems that are resilient against key leakage. The schema relies on the assumptions of a *universal hash proof system* [9] enabling decisional Diffie-Hellman, quadratic residuosity, and Paillier's composite residuosity problem to instantiate the cryptosystem. Alwen, Dodis, and Wichs also examine leakage resilient public key cryptosystems, including identification schemes and authenticated key agreement protocols [10]. They extend their results to the bounded retrieval model, where they consider extremely large keys and an adversary can not learn more than a predetermined bound over a lifetime. Katz further constructs a leakage resistant signature scheme in the standard model [11].

Chari et al. propose the first theoretical model for power analysis [12]. Coron, Naccache, and Kocher develop a similar formalism for characterizing leakage immunity, and present several leakage detection tests [13]. Micali and Reyzin propose a general framework for security against side-channel analysis [14]. These models do not account for memory remanence or cold-boot attacks.

Countermeasures to cold-boot attacks remain scarce within the current technology paradigms. Migrating to hardware embedded encryption, such as that proposed by the Trusted Computing Group's Opal platform [15], will mitigate cold-boot attacks on full-disk encryption. Enck et al. [16] propose an encrypting memory controller that writes only encrypted data to main memory, but decrypts it on reads into the processor or cache. There have also been attempts to manipulate the Intel x86 cache-coherence model to ensure that keys and key-derived state (such as key schedules) remain in L2 caches, but not in main memory.[2] The feasibility of this approach has yet to be demonstrated with the current architectures, however vendor modification of the instruction set may indeed make this approach a reality.

Intel has developed specialized instructions for executing AES operations [17]. There are six kinds instructions (encrypt round, encrypt last round, decrypt round, decrypt last round, inverse mix columns, and key schedule assist) which use the 128-bit XMM registers to hold round keys and block data. In addition to improving execution speed, these instructions have been designed to eliminate vulnerabilities from *cache attacks* [18], an interprocess side-channel that exploits timing differences for operations dependent upon cached and uncached data. Intel does not claim that these instructions mitigate cold-boot attacks, however we speculate that the schedule derivation assistance may improve the performance of just-in-time round-key derivation, thereby reducing the number of round keys stored in memory.

[2] Jürgen Pabel. `http://frozencache.blogspot.com/`

3 Algorithmic Description

This primary exposition details the recovery algorithm for AES-128, although the concepts generalize to the 192-bit and 256-bit cases (Section 3.5). Their differing block and key sizes create more potential for confusion when referencing schedule elements. We have implemented the 128-bit and 256-bit cases; Section 4 presents performance results for both cases.

3.1 Preliminaries

A 128-bit AES key schedule expands a four by 32-bit-word key into a 44 word sequence. Schedule components are addressed with the following notation: Let sans-serif variables, S, refer to entire key schedule. Subscripting expresses a hierarchical view of schedule components. Let S_r refer to the four words of round r. Let $\mathsf{S}_{r,w}$ refer to word w of round r. Let $\mathsf{S}_{r,w,b}$ refer to byte b of word w of round r. It is also convenient to index the schedule in a flat manner. Let $\mathsf{S^w}_i$ refer to word i of the schedule. Let $\mathsf{S^b}_i$ refer to byte i of the schedule. The notation follows the least-significant-byte-first convention. Some additional function notation is necessary to express the key schedule. Let $\mathsf{sbox}(\mathsf{S^b}_i)$ apply the AES substitution box to the byte $\mathsf{S^b}_i$. For convenience, let $\mathsf{sbox}(\mathsf{S^w}_i)$ apply the substitution box to each constituent byte when $\mathsf{S^w}_i$ is a word. Let $\mathsf{rot}(\mathsf{S^w}_i)$ rotate the word $\mathsf{S^w}_i$ by eight bit positions of increasing significance; e.g. $\mathsf{rot}(\mathsf{S}_{r,w,0}, \mathsf{S}_{r,w,1}, \mathsf{S}_{r,w,2}, \mathsf{S}_{r,w,3}) = (\mathsf{S}_{r,w,3}, \mathsf{S}_{r,w,0}, \mathsf{S}_{r,w,1}, \mathsf{S}_{r,w,2})$, in the least-significant-byte-first representation. The round constants, denoted by $\mathsf{rcon}[i]$, are the $(i-1)^{\text{th}}$ exponent of 2 in the field $GF(2^8)$ for the least significant byte and 0 for the other bytes.

For the 128-bit schedule, the first four words are the key itself. The subsequent words are prescribed by two equation schema.

$$\begin{array}{ll} \mathsf{S^w}_i = \mathsf{S^w}_{i-1} \oplus \mathsf{sbox}(\mathsf{rot}(\mathsf{S^w}_{i-3})) \oplus \mathsf{rcon}[i/4], & \text{when } i \bmod 4 = 0 \\ \mathsf{S^w}_i = \mathsf{S^w}_{i-1} \oplus \mathsf{S^w}_{i-3}, & \text{when } i \bmod 4 \neq 0 \end{array} \qquad (1)$$

The table below illustrates the indexing schema for the 128-bit key schedule. The column headings indicate the byte and word indices. Row labels indicate the round. This particular table shows how the flat byte-level references relate to the hierarchical tags. For instance, $\mathsf{S}_{8,2,3}$ refers to the same byte as $\mathsf{S^b}_{139}$.

w	0				1				2				3			
$r \diagdown b$	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
0	$\mathsf{S^b}_{0}$	$\mathsf{S^b}_{1}$	$\mathsf{S^b}_{2}$	$\mathsf{S^b}_{3}$	$\mathsf{S^b}_{4}$	$\mathsf{S^b}_{5}$	$\mathsf{S^b}_{6}$	$\mathsf{S^b}_{7}$	$\mathsf{S^b}_{8}$	$\mathsf{S^b}_{9}$	$\mathsf{S^b}_{10}$	$\mathsf{S^b}_{11}$	$\mathsf{S^b}_{12}$	$\mathsf{S^b}_{13}$	$\mathsf{S^b}_{14}$	$\mathsf{S^b}_{15}$
1	$\mathsf{S^b}_{16}$	$\mathsf{S^b}_{17}$	$\mathsf{S^b}_{18}$	$\mathsf{S^b}_{19}$	$\mathsf{S^b}_{20}$	$\mathsf{S^b}_{21}$	$\mathsf{S^b}_{22}$	$\mathsf{S^b}_{23}$	$\mathsf{S^b}_{24}$	$\mathsf{S^b}_{25}$	$\mathsf{S^b}_{26}$	$\mathsf{S^b}_{27}$	$\mathsf{S^b}_{28}$	$\mathsf{S^b}_{29}$	$\mathsf{S^b}_{30}$	$\mathsf{S^b}_{31}$
2	$\mathsf{S^b}_{32}$	$\mathsf{S^b}_{33}$	$\mathsf{S^b}_{34}$	$\mathsf{S^b}_{35}$	$\mathsf{S^b}_{36}$	$\mathsf{S^b}_{37}$	$\mathsf{S^b}_{38}$	$\mathsf{S^b}_{39}$	$\mathsf{S^b}_{40}$	$\mathsf{S^b}_{41}$	$\mathsf{S^b}_{42}$	$\mathsf{S^b}_{43}$	$\mathsf{S^b}_{44}$	$\mathsf{S^b}_{45}$	$\mathsf{S^b}_{46}$	$\mathsf{S^b}_{47}$
3	$\mathsf{S^b}_{48}$	$\mathsf{S^b}_{49}$	$\mathsf{S^b}_{50}$	$\mathsf{S^b}_{51}$	$\mathsf{S^b}_{52}$	$\mathsf{S^b}_{53}$	$\mathsf{S^b}_{54}$	$\mathsf{S^b}_{55}$	$\mathsf{S^b}_{56}$	$\mathsf{S^b}_{57}$	$\mathsf{S^b}_{58}$	$\mathsf{S^b}_{59}$	$\mathsf{S^b}_{60}$	$\mathsf{S^b}_{61}$	$\mathsf{S^b}_{62}$	$\mathsf{S^b}_{63}$
4	$\mathsf{S^b}_{64}$	$\mathsf{S^b}_{65}$	$\mathsf{S^b}_{66}$	$\mathsf{S^b}_{67}$	$\mathsf{S^b}_{68}$	$\mathsf{S^b}_{69}$	$\mathsf{S^b}_{70}$	$\mathsf{S^b}_{71}$	$\mathsf{S^b}_{72}$	$\mathsf{S^b}_{73}$	$\mathsf{S^b}_{74}$	$\mathsf{S^b}_{75}$	$\mathsf{S^b}_{76}$	$\mathsf{S^b}_{77}$	$\mathsf{S^b}_{78}$	$\mathsf{S^b}_{79}$
5	$\mathsf{S^b}_{80}$	$\mathsf{S^b}_{81}$	$\mathsf{S^b}_{82}$	$\mathsf{S^b}_{83}$	$\mathsf{S^b}_{84}$	$\mathsf{S^b}_{85}$	$\mathsf{S^b}_{86}$	$\mathsf{S^b}_{87}$	$\mathsf{S^b}_{88}$	$\mathsf{S^b}_{89}$	$\mathsf{S^b}_{90}$	$\mathsf{S^b}_{91}$	$\mathsf{S^b}_{92}$	$\mathsf{S^b}_{93}$	$\mathsf{S^b}_{94}$	$\mathsf{S^b}_{95}$
6	$\mathsf{S^b}_{96}$	$\mathsf{S^b}_{97}$	$\mathsf{S^b}_{98}$	$\mathsf{S^b}_{99}$	$\mathsf{S^b}_{100}$	$\mathsf{S^b}_{101}$	$\mathsf{S^b}_{102}$	$\mathsf{S^b}_{103}$	$\mathsf{S^b}_{104}$	$\mathsf{S^b}_{105}$	$\mathsf{S^b}_{106}$	$\mathsf{S^b}_{107}$	$\mathsf{S^b}_{108}$	$\mathsf{S^b}_{109}$	$\mathsf{S^b}_{110}$	$\mathsf{S^b}_{111}$
7	$\mathsf{S^b}_{112}$	$\mathsf{S^b}_{113}$	$\mathsf{S^b}_{114}$	$\mathsf{S^b}_{115}$	$\mathsf{S^b}_{116}$	$\mathsf{S^b}_{117}$	$\mathsf{S^b}_{118}$	$\mathsf{S^b}_{119}$	$\mathsf{S^b}_{120}$	$\mathsf{S^b}_{121}$	$\mathsf{S^b}_{122}$	$\mathsf{S^b}_{123}$	$\mathsf{S^b}_{124}$	$\mathsf{S^b}_{125}$	$\mathsf{S^b}_{126}$	$\mathsf{S^b}_{127}$
8	$\mathsf{S^b}_{128}$	$\mathsf{S^b}_{129}$	$\mathsf{S^b}_{130}$	$\mathsf{S^b}_{131}$	$\mathsf{S^b}_{132}$	$\mathsf{S^b}_{133}$	$\mathsf{S^b}_{134}$	$\mathsf{S^b}_{135}$	$\mathsf{S^b}_{136}$	$\mathsf{S^b}_{137}$	$\mathsf{S^b}_{138}$	$\mathsf{S^b}_{139}$	$\mathsf{S^b}_{140}$	$\mathsf{S^b}_{141}$	$\mathsf{S^b}_{142}$	$\mathsf{S^b}_{143}$
9	$\mathsf{S^b}_{144}$	$\mathsf{S^b}_{145}$	$\mathsf{S^b}_{146}$	$\mathsf{S^b}_{147}$	$\mathsf{S^b}_{148}$	$\mathsf{S^b}_{149}$	$\mathsf{S^b}_{150}$	$\mathsf{S^b}_{151}$	$\mathsf{S^b}_{152}$	$\mathsf{S^b}_{153}$	$\mathsf{S^b}_{154}$	$\mathsf{S^b}_{155}$	$\mathsf{S^b}_{156}$	$\mathsf{S^b}_{157}$	$\mathsf{S^b}_{158}$	$\mathsf{S^b}_{159}$
10	$\mathsf{S^b}_{160}$	$\mathsf{S^b}_{161}$	$\mathsf{S^b}_{162}$	$\mathsf{S^b}_{163}$	$\mathsf{S^b}_{164}$	$\mathsf{S^b}_{165}$	$\mathsf{S^b}_{166}$	$\mathsf{S^b}_{167}$	$\mathsf{S^b}_{168}$	$\mathsf{S^b}_{169}$	$\mathsf{S^b}_{170}$	$\mathsf{S^b}_{171}$	$\mathsf{S^b}_{172}$	$\mathsf{S^b}_{173}$	$\mathsf{S^b}_{174}$	$\mathsf{S^b}_{175}$

A candidate key schedule byte, $\mathsf{C^b}_i$, with ground states specified by $\mathsf{M^b}_i$ is *compatible with the decayed byte*, $\mathsf{D^b}_i$, when $\mathsf{D^b}_i$ preserves all ground-state bits in $\mathsf{C^b}_i$, or expressed equationally, when $(\mathsf{C^b}_i \oplus \mathsf{D^b}_i) \wedge (\mathsf{C^b}_i \oplus \mathsf{M^b}_i) = 0$.

3.2 Maximizing the Implied Schedule Bytes

For the first guess, the candidate byte, $\mathsf{C^b}_{i_0}$, is only constrained by the known bits in corresponding decayed byte, $\mathsf{D^b}_{i_0}$. Yet in the second stage, for a properly chosen i_1, $\mathsf{C^b}_{i_1}$ is constrained by $\mathsf{D^b}_{i_1}$ *and* a second byte $\mathsf{D^b}_j$. A properly selected i_1 will instantiate a byte slice of one of the two schedule generating equations, (1). For example $\mathsf{C^b}_4 \oplus \mathsf{C^b}_{16} = \mathsf{C^b}_{20}$ is the first byte-slice of the generating equation for $\mathsf{C^w}_5 = (\mathsf{C^b}_{20}, \mathsf{C^b}_{21}, \mathsf{C^b}_{22}, \mathsf{C^b}_{23})$. If $i_0 = 4$ and $i_1 = 20$, then the implied byte is at index $j = 16$ and equals $\mathsf{C^b}_{20} \oplus \mathsf{C^b}_4$. Thus $\mathsf{D^b}_{16}$ constrains the implied value of $\mathsf{C^b}_{20} \oplus \mathsf{C^b}_4$.

This algorithm makes use of the following observations: *1)* Each schedule byte $\mathsf{S^b}_i, 16 \leq i < 160$ is involved in three equations. *2)* There are 256 solutions to each equation when any one variable is fixed. There is a unique solution to each equation when any two variables are fixed. *3)* Guessing a single byte at stage n implies up to n other byte values for properly structured guessing orders.

Item 1 follows from the fact that every word is generated by its preceding word and the one 4 words ago; simply limit the scope to the byte-slice of the concerned byte. For example, consider $\mathsf{S}_{1,0,0}$=$\mathsf{S^b}_{16}$, the first byte of $\mathsf{S^w}_4$.

$$\begin{array}{llcl}
\mathsf{S}_{0,0,0} & \oplus\ \mathsf{sbox}(\mathsf{S}_{0,3,1}) \oplus \mathtt{0x01} & = & \mathsf{S}_{\mathbf{1,0,0}} \\
\mathsf{S}_{0,1,0} & \oplus \qquad \mathsf{S}_{\mathbf{1,0,0}} & = & \mathsf{S}_{1,1,0} \\
\mathsf{S}_{\mathbf{1,0,0}} & \oplus\ \mathsf{sbox}(\mathsf{S}_{1,3,1}) \oplus \mathtt{0x02} & = & \mathsf{S}_{2,0,0}
\end{array} \tag{2}$$

Item 2 is implied by the fact that $\oplus$ is the field addition operation for $GF(2^8)$ and that $\mathsf{sbox}()$ is a bijection in $GF(2^8)$.

To see item 3, consider the following candidate exploration order, C, for a decayed schedule D. First choose a candidate for $\mathsf{C}_{0,0,0}$; it is only constrained by the known bits at that position. The next guess, $\mathsf{C}_{1,0,0}$, is constrained by the known bits from $\mathsf{D}_{1,0,0}$. Additionally, one can solve the equation $\mathsf{C}_{0,0,0} \oplus$

Table 1. An order for byte-guesses and consequent values in AES-128; the end of Section 3.2 describes the scripting notation

w	0				1				2				3			
$r \diagdown b$	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
0	0_0	14_{10}				13_{10}				12_{10}				1_1	14_9	
1	1_0	13_9				12_9				2_2	14_8			2_1	13_8	
2	2_0	12_8				3_3	14_7			3_2	13_7			3_1	12_7	
3	3_0	4_4	14_6			4_3	13_6			4_2	12_6			4_1	5_5	14_5
4	4_0	5_4	13_5			5_3	12_5			5_2	6_6	14_4		5_1	6_5	13_4
5	5_0	6_4	12_4			6_3	7_7	14_3		6_2	7_6	13_3		6_1	7_5	12_3
6	6_0	7_4	8_8	14_2		7_3	8_7	13_2		7_2	8_6	12_2	14_1	7_1	8_5	9_9
7	7_0	8_4	9_8	13_1		8_3	9_7	12_1	14_0	8_2	9_6	10_{10}	13_0	8_1	9_5	10_9
8	8_0	9_4	10_8	12_0	15_{10}	9_3	10_7	11_{10}	15_9	9_2	10_6	11_9	15_8	9_1	10_5	11_8
9	9_0	10_4	11_7	15_7		10_3	11_6	15_6		10_2	11_5	15_5		10_1	11_4	15_4
100	10_0	11_3	15_3			11_2	15_2			11_1	15_1			11_0	15_0	

$\mathsf{sbox}(\mathsf{C}_{0,3,1}) = \mathsf{C}_{1,0,0}$ for $\mathsf{C}_{0,3,1}$, since $\mathsf{C}_{0,0,0}$ and $\mathsf{C}_{1,0,0}$ have candidate values. Thus the second guess is constrained by the known values at $\mathsf{D}_{0,3,1}$ as well. By the same logic, $\mathsf{D}_{2,0,0}$, $\mathsf{D}_{1,3,1}$, and $\mathsf{D}_{1,2,1}$ constrain the compatible guesses for $\mathsf{C}_{2,0,0}$. Because $\mathsf{C}_{r-1,3}$ rotates when computing $\mathsf{C}_{r,0}$, continuing to propose bytes in the column $\mathsf{C}_{r,0,0}$ causes the implied byte indices increment modulo 4 when the word index wraps around modulo 4.

Table 1 illustrates this behavior by enumerating the order of byte guesses and their consequent bytes. In the table entries, the full-sized number indicates the guessing stage and the subscript indicates the sequence of implied bytes. In particular, i_0 indicates a guess for stage i and i_1 is the first implied byte from this candidate. For example, $\mathsf{C}_{5,0,0}$ is guess number 5. This value combined with the value for $\mathsf{C}_{4,0,0}$ (chosen in step 4), allows one to solve for $\mathsf{C}_{4,3,1}$. The following equations make the order of solution explicit.

$$\begin{aligned}
\mathsf{C}_{4,3,1} &= \mathsf{sbox}^{-1}(\mathsf{C}_{5,0,0} \oplus \mathsf{C}_{4,0,0} \oplus 01)\\
\mathsf{C}_{4,2,1} &= \mathsf{C}_{4,3,1} \oplus \mathsf{C}_{3,3,1}\\
\mathsf{C}_{4,1,1} &= \mathsf{C}_{4,2,1} \oplus \mathsf{C}_{3,2,1}\\
\mathsf{C}_{4,0,1} &= \mathsf{C}_{4,1,1} \oplus \mathsf{C}_{3,1,1}\\
\mathsf{C}_{3,3,2} &= \mathsf{sbox}^{-1}(\mathsf{C}_{4,0,1} \oplus \mathsf{C}_{3,0,1} \oplus 01)
\end{aligned} \tag{3}$$

After selecting candidates for bytes 0-10, there are a number of ways to guess bytes 11-15. Table 1 illustrates a choice for these positions that implies values for an additional 10 bytes for each guess. The last guess implies 65 bytes because it causes round 8 to be fully specified; the entire schedule may be derived from any complete round.

3.3 The Recovery Algorithm

The algorithm, `recoverKeyRec`, explores the candidate space one byte at a time. It exploits the constraints on guesses and their consequent bytes to prune its exploration tree. For each guess, `recoverKeyRec` considers all 256 possible values. If the candidate satisfies all constraints imposed by the decayed image, D, then it guesses a value for the next step. Exploration proceeds in a depth-first manner, so that guess i is incremented to the next compatible candidate when all of descendant candidates have been ruled out.

A breadth-first search is also possible, however this strategy greatly increases the memory utilization. The advantage of the breadth-first method is that one could track the distance from decayed data for each candidate and explore the closest options first. The first implementations of this algorithm maintained the candidates on a binary heap indexed by their cost. In practice, there were many cases in the 60-70% decay range, where the process exceeded its 31-bit address space and halted before recovering the key. On the other hand, recovery speeds of depth-first search up to the 70% rate have proven fast enough in most cases and solvable in all attempted cases. The need to solve more cases with less memory dictated a transfer to depth-first search which consumes a fixed amount of memory, about 4.2 MB in the experimental implementation. Because depth-first search has a small memory footprint, it is also inexpensive to halt tree

```
recoverKeyRec(CandidateMatrix c, DecaySchedule d):
  if (c.length()==16):
    return c.key()
  for i=0 to 255:
    if(d.isCompatible(c.guess(i))):
      key = recoverKeyRec(c.guess(i),d)
      if (key != NULL)
         return key
  return NULL
```

Fig. 1. Recursive expression of key recovery; Appendix A details the core methods

exploration and resume later. The only data necessary to save is the candidate being examined at the halting time. Breadth-first search would require one to save the binary heap of candidates.

The following describes the semantics for tokens in `recoverKeyRec` (Figure 1). For simplicity of expression assume that operations do not mutate objects, but return newly constructed objects. Italicized tokens refer to classes and teletype tokens refer to fields and methods.

Let `c` be a *CandidateMatrix* that contains candidates for schedule bytes in the order indicated by Table 1. *CandidateMatrix* maintains a `count` of how many guesses have been made (0-16) and a flat schedule representation of 176 bytes to store byte candidates and their consequent bytes. The method `guess`(*Byte* `b`), returns a new *CandidateMatrix* whose array has been updated to contain `b` at position $\texttt{count}_0$ and its consequent bytes at positions as specified by the path matrix (e.g., Table 1). In particular, guessing the 16th byte completes the entire schedule. The `key`() method returns the key once all 16 bytes have been guessed and validated. The `new CandidateMatrix()` constructor simply creates an empty array with `count` set to 0.

Let *DecaySchedule* contain the decayed key schedule and a predicate, `is-Compatible`(*CandidateMatrix*), that indicates whether or not a guess and its consequent bytes are compatible the decayed schedule. Compatibility is determined by checking that the *CandidateMatrix* contains all the known bits from the corresponding bytes of the decayed schedule.

The function `recoverKeyRec`(*CandidateMatrix*,*DecaySchedule*) returns a key whose schedule is decay-compatible and is the result of extending the incoming *CandidateMatrix*. It returns `NULL` when there is no compatible key schedule extension to the specified candidate prefix. Proper usage asserts that the starting *CandidateMatrix* and *DecaySchedule* are compatible. The initial call to `recoverKeyRec` begins with an empty *CandidateMatrix*. The recursive expression in Figure 1 makes the control logic explicit. Figure 1 halts on the first compatible key schedule, however one could modify it to halt after a full search of the key space; simply replace the `return` with a `print` on the third line. Section 4 benchmarks both variants.

3.4 Path Prioritization

The exploration path illustrated by Table 1 maximizes the number of implied schedule bytes with the goal of minimizing the number of compatible candidates at each stage. There are many ways to grow the selection path, and Table 1 illustrates just one of them. For instance, there are 3 symmetric alternatives obtained by rotating the bytes of each word. Different paths will encounter different constraints and therefore will result in varying recovery times. Within the stages 0-10, the selection order of $\mathsf{S}_{x,0,0}$ may be altered and still obtain the same consequent bytes after 10 candidate stages. Growing the path with guesses adjacent to the body of previous guesses will preserve the set of inferred bytes; this claim has been verified experimentally. The following matrix shows that guesses may be grown from the middle of the schedule:

The exploration path in Table 2 starts at $\mathsf{S}_{5,0,0}$. To maintain adjacency, the next choice may be $\mathsf{S}_{4,0,0}$ or $\mathsf{S}_{6,0,0}$. The path grows by extending either the top or bottom of previous choices in $\mathsf{S}_{x,0,0}$, where $0 \leq x < 11$. This allows the number of inferred bytes to grow by one at each stage.

A first heuristic for choosing the best path might be to count up the known bits in the exploration path and its consequent bytes. However within a set of guesses and implied bytes, the selection order can also make a difference. Past a certain threshold, adding more constraints does not prune the exploration anymore because the byte is already uniquely determined (or its parent has been ruled out). Consider the case when the decayed data in the candidate position is `0xFF`. If the last guess corresponds to this position, then all of the consequent bytes are wasted constraints because the byte is uniquely determined. On the other hand, that byte position would make an excellent initial position for stage 0 because the first guess never produces any consequent bytes. The final algorithm's path comparison heuristic estimates the number of branches at each stage. At a stage, the estimate simply counts the number of known bits in the initial and consequent positions. The intuition is that each known bit will reduce the number of valid branches by a factor of 2. Thus, the branch estimate for a stage i is $2^{min(8,k_i)}$ where k_i is the number of known bits in the i^{th} stage's initial and consequent byte positions. This is clearly only an estimate, as some

Table 2. An exploration path starting in round 5; script notation parallels the example at the end of Section 3.2

w	0				1				2				3			
$r\diagdown^{b}$	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
0	10_0													10_1		
1	7_0									10_2				7_1		
2	6_0					10_3				7_2				6_1		
3	3_0	10_4				7_3				6_2				3_1	10_5	
4	2_0	7_4				6_3				3_2	10_6			2_1	7_5	
5	0_0	6_4				3_3	10_7			2_2	7_6			1_1	6_5	
6	1_0	4_4	10_8			4_3	7_7			4_2	6_6			4_1	5_5	10_9
7	4_0	5_4	8_8			5_3	8_7			5_2	8_6	10_{10}		5_1	8_5	9_9
8	5_0	8_4	9_8			8_3	9_7			8_2	9_6			8_1	9_5	
9	8_0	9_4				9_3				9_2				9_1		
10	9_0															

Table 3. AES-256 exploration path template; script notation parallels the example at the end of Section 3.2

0				1				2				3				0				1				2				3			
0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
0_0	15_7	23_7	31_7		14_7	22_7	30_7		13_7	21_7	29_7		12_7	20_7	28_7		10_6	18_6	26_6		9_6	17_6	25_6		8_6	16_6	24_6	31_6	1_1	15_6	23_6
1_0	14_6	22_6	30_6		13_6	21_6	29_6		12_6	20_6	28_6		10_5	18_5	26_5		9_5	17_5	25_5		8_5	16_5	24_5	31_5	2_2	15_5	23_5	30_5	2_1	14_5	22_5
2_0	13_5	21_5	29_5		12_5	20_5	28_5		10_4	18_4	26_4		9_4	17_4	25_4		8_4	16_4	24_4	31_4	3_3	15_4	23_4	30_4	3_2	14_4	22_4	29_4	3_1	13_4	21_4
3_0	12_4	20_4	28_4		10_3	18_3	26_3		9_3	17_3	25_3		8_3	16_3	24_3	31_3	4_4	15_3	23_3	30_3	4_3	14_3	22_3	29_3	4_2	13_3	21_3	28_3	4_1	12_3	20_3
4_0	10_2	18_2	26_2		9_2	17_2	25_2		8_2	16_2	24_2	31_2	5_5	15_2	23_2	30_2	5_4	14_2	22_2	29_2	5_3	13_2	21_2	28_2	5_2	12_2	20_2	26_1	5_1	10_1	18_1
5_0	9_1	17_1	25_1		8_1	16_1	24_1	31_1	6_6	15_1	23_1	30_1	6_5	14_1	22_1	29_1	6_4	13_1	21_1	28_1	6_3	12_1	20_1	26_0	6_2	10_0	18_0	25_0	6_1	9_0	17_0
6_0	8_0	16_0	24_0	31_0	7_7	15_0	23_0	30_0	7_6	14_0	22_0	29_0	7_5	13_0	21_0	28_0	7_4	12_0	20_0	27_6	7_3	11_6	19_6	27_5	7_2	11_5	19_5	27_4	7_1	11_4	19_4
7_0	11_3	19_3	27_3		11_2	19_2	27_2		11_1	19_1	27_1		11_0	19_0	27_0																

bits may constrain portions of the guess that have already been determined. The total branching estimate is the product of the stage estimates. Since the stage estimates are all powers of two, it suffices to sum their exponents. Thus, the heuristic ranks paths by the scalar value $\sum_{i=0}^{15} min(8, k_i)$.

The algorithm considers variants of Table 1 along two axes. One axis is byte slice selection; Table 1 initiates on slice 0, although three other may be obtained by rotating all schedule words by the same amount. The other axis is byte guessing order within stages 0-10 as described above; choose a initial round and then extend the guesses by adding to the adjacent round on the top or the bottom. There are other paths not reached by these variables; e.g., growth may be rooted in words 1-3 of the round key rather than 0. Performance may well improve by selecting these paths from a larger space, however no additional path analysis is investigated in this work.

3.5 Generalizing to Other Instances of AES

Generalizing the algorithm to operate on 192-bit and 256-bit variants of AES is straightforward. One needs to construct a different path template and update the `isCompatible`() method to incorporate the schedule generating equations of the larger keys. Table 3 illustrates the path template used by the 256-bit implementation. The heuristic considers paths on the same axes as the 128-bit version. Since the key size is twice the length of the block size, the matrix is only 8 deep and therefore the heuristic considers fewer paths.

4 Benchmarks

Test cases are generated with OpenSSL's `RAND_bytes`() function. For a given decay rate d, the test generator derives a key schedule from randomly selected key bytes and then randomly zeroes $d\%$ of the bits. Tests assume a ground-state encoding of 0. Performance was evaluated on Dell Precision Workstation 7400 running a 3.4 Ghz quad-core Xeon processor with 4GB of RAM. The C99 reference implementation of the algorithm is compiled with the MinGW version of `gcc-3.4.5` at the highest level of optimization, `-O4`. All computations cases were run to completion in a serial manner. Time was measured by entry and exit calls to `clock()` from the `time.h` library; clock resolution is 64 Hz. The original

Table 4. Run-time results for four versions of the algorithm; each decay rate test suite contains 10,000 cases

Case	Key size	Path selection	Halting condition
PathOpt-128	128 bits	Heuristically chosen (Section 3.4)	First match
PathOpt-256	256 bits	Heuristically chosen (Section 3.4)	First match
Basic-128	128 bits	Fixed to Table 1	First match
Exhaust-128	128 bits	Heuristically chosen (Section 3.4)	End of key space

PathOpt-128 — Run-time (seconds)

Decay	30%	40%	50%	60%
Total	90.120	93.559	142.322	1,736.321
Avg.	0.009	0.009	0.014	0.174
Med.	0.015	0.015	0.015	0.031
Max	0.015	0.015	0.078	2.094
Min	0.000	0.000	0.000	0.000
St.Dev	0.007	0.008	0.015	0.772

PathOpt-256 — Run-time (seconds)

Decay	30%	40%	50%	60%
Total	17.046	26.185	123.250	6,954.231
Avg.	0.002	0.003	0.012	0.695
Med.	0.000	0.000	0.000	0.062
Max	0.016	0.062	2.125	352.015
Min	0.000	0.000	0.000	0.000
St.Dev.	0.005	0.006	0.044	5.920

Basic-128 — Run-time (seconds)

Decay	30%	40%	50%	60%
Total	219.204	1,526.308	32,551.469	1,638,788.166
Avg.	0.022	0.153	3.255	163.879
Med.	0.015	0.015	0.078	1.968
Max	9.562	266.390	3,354.890	343,656.375
Min	0.000	0.000	0.000	0.000
St.Dev.	0.140	2.994	55.563	3,753.608

Exhaust-128 — Run-time (seconds)

Decay	30%	40%	50%	60%
Total	96.403	112.350	258.568	4,497.599
Avg.	0.010	0.011	0.026	0.450
Med.	0.015	0.015	0.015	0.110
Max	0.031	0.468	0.875	75.203
Min	0.000	0.000	0.000	0.000
St.Dev.	0.007	0.009	0.036	1.921

keys were found for all test cases, using the heuristically chosen path and halting on the first match.

Table 4 summarizes the benchmark results for four variants of the algorithm: PathOpt-128, PathOpt-256, Basic-128, and Exhaust-128. There are 10,000 cases for each of the four decay rates, 30%, 40%, 50%, and 60%. The 128-bit variants have been run on the same test cases, so their results are directly comparable. All times are measured in seconds. A time of 0.000 means that the computation finished in less than 1/64 second, or about 53 million processor cycles.

Additional testing (Table 5) was performed to estimate the maximum recoverable decay rates for PathOpt-128 and PathOpt-256. Only 5,000 cases were examined due to extended recovery times.

5 Analysis

PathOpt-128 solves all cases at 50% decay and less in under half a second. At 60% decay, PathOpt-128 recovered the worst case in 35.500 seconds while solving the average case in 0.174 seconds. At the extended decay rate of 70%, recovery time averages grew to just over 6 minutes with the median time at just under five seconds. Nearly half of the 17.4 day run was consumed by solving the worst case of the test suite; the second slowest case was over six times faster. 4927 cases were recovered in less than 20 minutes.

PathOpt-128 runs faster than Basic-128 across the board and the speedup quickly grows as the decay rate increases. The speedups for 30%, 40%, 50%, and

Table 5. Extended decay rate runs; each decay rate test suite has 5,000 cases

Case	Total	Avg.	Med.	Max	Min	St.Dev.
PathOpt-128 @ 70% decay	1,504,487.119 s	300.897 s	4.938 s	737,266.687 s	0.000 s	10,677.913 s
PathOpt-256 @ 65% decay	446,879.849 s	89.376 s	0.875 s	194,410.875 s	0.000 s	2,843.061 s

60% are 2.43×, 16.3×, 228×, and 943×, respectively. At 70%, only 10 cases had completed after a week when the experiment was terminated. The path selection heuristic makes 70% decay a feasibly solvable problem in the test environment. Even for the low decay rates, Basic-128 has a much higher standard deviation; their worst cases with Basic-128 are several orders of magnitude worse than their worst cases with PathOpt-128.

The profound impact of heuristic path selection at high decay rates suggests that a more thorough search for the best path could further extend the maximum feasible decay capacity. Only a small subset of possible paths are considered. The current analysis takes less than 1/64 second, as evidenced by the 0.000 timing results, so there is ample room for more startup analysis.

Full search of the key space appears to be a small factor slower than stopping at the first compatible key. It widens as the decay increases, but by 60% the Exhaust-128 only takes 2.590 times longer. We note that all 90,000 test cases had precisely one solution, so the exhaustive search seems unnecessary at the tested decay rates.

PathOpt-256 performs well up 60% decay rates, solving cases in an average of 0.695 seconds and in no more than 352.015 seconds. At 70%, no cases were solved in the test suite during a 1 week trial. At 65% the results are promising: the 5,000 cases have been solved in an average of 89.676 seconds. The longest case took 2.25 days to recover, while 99.4% of the cases have been recovered in less than 20 minutes.

Interestingly, the PathOpt-256 is slightly faster than PathOpt-128 on decay rates at 50% and below. The average solution times for these cases is within two units of the 64 Hz clock resolution. We conjecture that the heuristic path analysis takes less time with the 256-bit version since there are fewer paths to consider, due to the flatness of the path matrix (Section 3.5).

As a point of comparison, the original algorithm [1] was compiled in the same environment and run against the same test suite of 30% and 40% decayed AES-128 schedules. After three weeks of execution, only the first four cases at 30% had been solved and the first case at 40% had not yet finished.

6 Conclusion

We presented a new class of recovery capability that is several orders of magnitude faster than previous methods, particularly for higher decay rates. It more than doubles the decay rate recovery feasibility of prior work. The tree-pruning constraints enable efficient and exhaustive key space searches to determine solution uniqueness. We have generalized the implementation to AES-256 while maintaining excellent performance up to 65% decay.

Acknowledgements

I owe a special debt of gratitude to Dr. Adam L. Young for his insight and wisdom regarding writing, presentation, and the scientific community. I have additionally benefited from the thoughtful review of Kerry A. McKay. I am further indebted to Charles C. Howell, Steven M. Godin, and Eileen M. Boettcher for their strident advocacy—without which this paper would not be possible.

References

1. Halderman, J.A., Schoen, S.D., Heninger, N., Clarkson, W., Paul, W., Calandrino, J.A., Feldman, A.J., Appelbaum, J., Felten, E.W.: Lest we remember: cold boot attacks on encryption keys. In: USENIX Security Symposium, pp. 45–60. USENIX Association, Berkeley (2008)
2. Daemen, J., Rijmen, V.: The block cipther Rijndael. In: Schneier, B., Quisquater, J.-J. (eds.) CARDIS 1998. LNCS, vol. 1820, pp. 277–284. Springer, Heidelberg (2000)
3. Daemen, J., Rijmen, V.: The Design of Rijndael. Springer, Heidelberg (2002)
4. Heninger, N., Shacham, H.: Reconstructing RSA private keys from random key bits. In: Halevi, S. (ed.) CRYPTO 2009. LNCS, vol. 5677, pp. 1–17. Springer, Heidelberg (2009)
5. Handschuh, H., Paillier, P., Stern, J.: Probing attacks on tamper-resistant devices. In: Koç, Ç.K., Paar, C. (eds.) CHES 1999. LNCS, vol. 1717, pp. 303–315. Springer, Heidelberg (1999)
6. Akavia, A., Goldwasser, S., Vaikuntanathan, V.: Simultaneous hardcore bits and cryptography against memory attacks. In: Reingold, O. (ed.) TCC 2009. LNCS, vol. 5444, pp. 474–495. Springer, Heidelberg (2009)
7. Regev, O.: On lattices, learning with errors, random linear codes, and cryptography. In: STOC, pp. 84–93. ACM, New York (2005)
8. Naor, M., Segev, G.: Public-key cryptosystems resilient to key leakage. In: Halevi, S. (ed.) CRYPTO 2009. LNCS, vol. 5677, pp. 18–35. Springer, Heidelberg (2009)
9. Cramer, R., Shoup, V.: Universal hash proofs and a paradigm for adaptive chosen ciphertext secure public-key encryption. In: Knudsen, L.R. (ed.) EUROCRYPT 2002. LNCS, vol. 2332, pp. 45–64. Springer, Heidelberg (2002)
10. Alwen, J., Dodis, Y., Wichs, D.: Leakage-resilient public-key cryptography in the bounded-retrieval model. In: Halevi, S. (ed.) CRYPTO 2009. LNCS, vol. 5677, pp. 36–54. Springer, Heidelberg (2009)
11. Katz, J.: Signature schemes with bounded leakage resilience. Cryptology ePrint Archive: Report 2009/133 (March 22, 2009)
12. Chari, S., Jutla, C.S., Rao, J.R., Rohatgi, P.: Towards sound approaches to counteract power-analysis attacks. In: Wiener, M. (ed.) CRYPTO 1999. LNCS, vol. 1666, pp. 398–412. Springer, Heidelberg (1999)
13. Coron, J.S., Naccache, D., Kocher, P.: Statistics and secret leakage. ACM Trans. Embed. Comput. Syst. 3(3), 492–508 (2004)
14. Micali, S., Reyzin, L.: Physically observable cryptography. In: Naor, M. (ed.) TCC 2004. LNCS, vol. 2951, pp. 278–296. Springer, Heidelberg (2004)

15. TCG storage security subsystem class: Opal version 1.0, `http://www.trustedcomputinggroup.org/` (January 27, 2009)
16. Enck, W., Butler, K., Richardson, T., McDaniel, P., Smith, A.: Defending against attacks on main memory persistence. In: ACSAC, Washington, DC, USA, pp. 65–74. IEEE Computer Society, Los Alamitos (2008)
17. Gueron, S.: Advanced encryption standard (AES) instructions set (April 27, 2009), `http://www.intel.com/`
18. Osvik, D.A., Shamir, A., Tromer, E.: Cache attacks and countermeasures: the case of AES. In: Pointcheval, D. (ed.) CT-RSA 2006. LNCS, vol. 3860, pp. 1–20. Springer, Heidelberg (2006)

A Extended Pseudocode

```
path = ((  0),
        ( 16, 13),
        ( 32, 29, 25),
        ( 48, 45, 41, 37),
        ( 64, 61, 57, 53, 49),
        ( 80, 77, 73, 69, 65, 62),
        ( 96, 93, 89, 85, 81, 78, 74),
        (112,109,105,101, 97, 94, 90, 86),
        (128,125,121,117,113,110,106,102, 98),
        (144,141,137,133,129,126,122,118,114,111),
        (160,157,153,149,145,142,138,134,130,127,123),
        (173,169,165,161,158,154,150,146,143,139,135),
        (131,119,107, 95, 82, 70, 58, 46, 33, 21,  9),
        (124,115,103, 91, 79, 66, 54, 42, 30, 17,  5),
        (120,108, 99, 87, 75, 63, 50, 38, 26, 14,  1),
        (174,170,166,162,159,155,151,147,140,136,132))

class CandidateMatrix:
    Int count
    Byte m[176]

    def guess (Byte b):
        c = copy(self)
        c.count = count + 1
        c.m[path[count][0]] = b
        if count == 0:
            return c
        for i = 1 to len(path[count]):
            if defined(c.m[path[count][i]-16]):
                b4 = path[count][i-1]
                b3 = path[count][i]
                b0 = b4-16
                if inFirstWordOfRoundKey(b4):
                    c.m[b3] = unsbox(c.m[b4] XOR c.m[b0]
                              XOR rcon[getRound(b4)][getBytePos(b4)]
                else:
                    c.m[b3] = c.m[b0] XOR c.m[b4]
            else:
                b0 = path[count][i]
                b3 = path[count][i-1]
                b4 = b0 + 16
                if inFirstWordOfRoundKey(b4):
                    c.m[b0] = sbox(c.m[b3]) XOR c.m[b4]
                              XOR rcon[getRound(b4)][getBytePos(b4)]
                else:
                    c.m[b0] = c.m[b3] XOR c.m[b4]
        if c.count == 16:
            c = deriveFullScheduleFromRound8(c)
        return c

    def key ():
        return m[0:16]

class DecaySchedule:
    Byte decaySched[176]
    Byte gndEnc[176]
    def isCompatible (CandidateMatrix candidate):
        for i = 0 to 176:
            if defined(candidate.m[i]):
                if (candidate.m[i] XOR decaySched[i])
                   AND (candidate.m[i] XOR gndEnc[i]):
                    continue
                else:
                    return FALSE
        return TRUE

def recoverKeyRec(CandidateMatrix c, DecaySchedule d):
    if (c.length()==16):
        return c.key()
    for i=0 to 255:
        if(d.isCompatible(c.guess(i))):
            key = recoverKeyRec(c.guess(i),d)
        if (key != NULL):
            return key
    return NULL
```

The pseudocode follows a Python-like syntax, but with some additional explicit typing and field declaration. Variables may hold their declared types or undefined values—a property checked by `defined()`.

The `path` variable encodes the guessing and inference order of Table 1 in the flat byte-level schedule view. The recursive exploration function, `recoverKeyRec()`, is the same as in Fig. 1.

The `CandidateMatrix` class is dominated by the `guess` method which guesses a value for the position determined by `path[count][0]` and infers the consequent bytes. The inference logic splits into two cases: when the unknown byte is in the first word, $\mathsf{S^w}_{i-4}$, of the schedule generating equations (see (1)) or in the middle word, $\mathsf{S^w}_{i-1}$. The variable names, `b0`, `b3`, and `b4` reflect the relative position of their encapsulating words in the schedule; if `b0` comes from an arbitrary word in the first 10 rounds, then `b3` and `b4` come from the words three and four words ahead of `b0`, respectively. The inference sequence in `path` has been chosen to account for the necessary rotations in byte slices when solving the s-box version of the generating equations. Upon completion of the 16 guesses the eighth round becomes fully specified, implying the remainder of the schedule. The `key()` method simply returns the first 16 bytes of completed key schedule.

`DecaySchedule` holds the observed decayed data and ground state encoding for each byte. Its one method `isCompatible()` checks that each defined bit of the candidate schedule equals the decayed data or the ground state.