2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines

Reconstructing AES Key Schedules from Decayed Memory with FPGAs

Heinrich Riebler*, Tobias Kenter*, Christian Plessl*, and Christoph Sorge†
*Department of Computer Science, University of Paderborn, 33098 Paderborn, Germany
Email: { heinrich.riebler — kenter — christian.plessl }@uni-paderborn.de
†Institute of Law and Informatics and CISPA, Saarland University, 66123 Saarbrücken, Germany
Email: christoph.sorge@uni-saarland.de

***Abstract*—In this paper, we study how AES key schedules can be reconstructed from decayed memory. This operation is a crucial and time consuming operation when trying to break encryption systems with cold-boot attacks. In software, the reconstruction of the AES master key can be performed using a recursive, branch-and-bound tree-search algorithm that exploits redundancies in the key schedule for constraining the search space. In this work, we investigate how this branch-and-bound algorithm can be accelerated with FPGAs. We translate the recursive search procedure to a state machine with an explicit stack for each recursion level and create optimized datapaths to accelerate in particular the processing of the most frequently accessed tree levels. We support two different decay models, of which especially the more realistic non-idealized asymmetric decay model causes very high runtimes in software. Our implementation on a Maxeler dataflow computing system outperforms a software implementation for this model by up to 27x, which makes cold-boot attacks against AES practical even for high error rates.**

***Keywords*-cold-boot attack; AES; key reconstruction; key schedule; branch-and-bound; FPGA; hardware acceleration**

I. Introduction

With computer users' increasing security awareness, the use of full disk encryption systems is becoming more and more common. Usually, they use AES [1] with some appropriate mode of operation. While the encryption algorithm itself is considered secure, implementations have to keep the AES key in main memory. For efficiency reasons, the so-called key schedule is stored in addition to the key itself. It contains the round keys used in the encryption and decryption process, which are derived from the key itself.

Storing the AES key and the key schedule in memory (DRAM) was not considered as problematic, as operating systems have mechanisms that control access to that memory. When the PC is turned off, DRAM modules lose the stored information. However, it has been shown [2] that, especially at low temperatures, the decay of memory contents can be slow enough to allow *cold-boot attacks* aimed at retrieving AES keys. The attacker may, for example, cool down the DRAM modules, insert them into another PC, and obtain a dump of the memory contents. In the next step, encryption keys—in essence, (pseudo) random numbers of 128, 192 or 256 bits length—have to be identified in the memory dump (key search). Halderman et al. [2] first showed how to exploit the redundant information in the key schedule to this end, even in the presence of bit errors. The same information can also be used to efficiently *correct* bit errors, thus reconstructing the original AES key. An efficient reconstruction algorithm has been presented by Tsow [3]. As most bit errors are bit flips to the memory's ground state, both key search and key reconstruction algorithms assume *perfect asymmetric decay*: all bits that are not in the ground state are considered correct.

The *contribution of this work* is to study how the algorithm for key reconstruction (recursive branch-and-bound) can be accelerated with FPGA-based custom computing machines. We present an architecture that uses highly optimized combinational datapaths for the performance-critical higher levels of the search tree and more resource-efficient pipelined ones for the less frequent and more complex lower levels. In addition to perfect asymmetric decay, we support a more realistic non-idealized asymmetric decay model, which causes very high runtimes in software. To our knowledge, our software implementations for both error models are the fastest existing for the presented reconstruction strategy. Yet our hardware implementation targeting a Maxeler dataflow computing system outperforms our software implementations for key reconstruction by up to 27x. This acceleration makes cold-boot attacks feasible for non-idealized decay models and high error rates. While FPGAs have been used for many cryptographic applications, including key search for cold-boot attacks, this is—to our knowledge—the first work that proposes to use FPGAs for accelerating key reconstruction for cold-boot attack algorithms.

The remainder of this paper is structured as follows. In Section II we discuss related work, in particular the AES algorithm and how its key schedule structure can be exploited for cold-boot attacks. In Section III we give an overview of the Maxeler data flow computing system. Section IV presents the custom computing machine for reconstructing the identified candidates. In Section V we compare the performance of our accelerators with a CPU implementation. Finally, we draw conclusions in Section VI.

II. Background and Related Work

In this section, we describe related work and the foundations of cold-boot attacks. While FPGA implementations of the AES algorithm are available (e. g. [4]), to the best of

978-1-4799-5111-6/14 $31.00 © 2014 IEEE
DOI 10.1109/FCCM.2014.67

our knowledge we are the first to target cold-boot attacks. Our previous work addresses the streaming oriented key search [5], whereas in this work we present an FPGA implementation that supports key reconstruction. We therefore focus on algorithms that have been suggested for purely software-based cold-boot attacks on AES.

A. Cold-Boot Attacks

Once a memory dump has been obtained, cryptographic keys have to be identified within that dump (*key search*). The first practical attacks on locating cryptographic keys, taking advantage of mathematical relationships within the key material, were reported by Shamir et al. [6] for RSA.

The keys used for symmetric block ciphers (like AES), are chosen at random. However, the key schedule (see Section II-B), derived from the key, is usually kept in memory along with the key itself. It is much longer than the key and is comprised of round keys that are derived using a known bijective function from the AES key. Using the complete key schedule and exploiting the redundancy that is introduced because of the mathematical relations between the round keys, the search space can be considerably restricted—even in the presence of bit errors caused by memory decay. Kaplan et al. [7] were the first to propose exploiting the key schedule's structure to identify relevant memory segments.

After key candidates have been found, bit errors in the keys must be corrected (*key reconstruction*). In our work we use the reconstruction strategy proposed by Tsow [3]. Besides, Albrecht et al. [8] and Kamal et al. [9] improved the results of Tsow by solving a set of non-linear algebraic equations with noise (based on mixed integer programming) or with an off-the-shelf SAT solver.

In this paper, we focus on the reconstruction of AES keys. AES is a commonly used encryption algorithm. In particular, popular tools for the encryption of file systems, like TrueCrypt, use AES as their default block cipher.

B. AES Key Schedule

In order to achieve the desired cryptographic properties (confusion and diffusion), the initial master key is bijectively mapped to several round keys. The key size determines the number of round keys that are used in the actual encryption process. The combination of the master key and the round keys is known as the AES key schedule, which is usually pre-computed and contiguously stored in the main memory. In each round, a certain number of transformations—as defined by the known key expansion function of AES [10]—are performed. Figures 1 and 2 illustrate the generation of the key schedule. The master key is stored as the first row (round 0). To compute the remaining round keys 1–10, two different functions are applied. The operation to compute word 0 in each round key, which takes words 0 and 3 of the previous round key as input, is illustrated in Figure 2. Besides XOR operations to combine its inputs, it involves a rotate operation on one of the input words, a bytewise substitution and the combination with a round constant RCon. Depending on the AES variant, only 8 to 11 RCon values, one per round, are needed, but the substitution requires a lookup in a table with 256 entries of one byte each. Words 1–3 of each round key are the result of an XOR operation, taking the previous word of the same round key and the corresponding word of the previous round key as inputs (Figure 1). Both key search and key reconstruction algorithms take advantage of this key schedule structure.

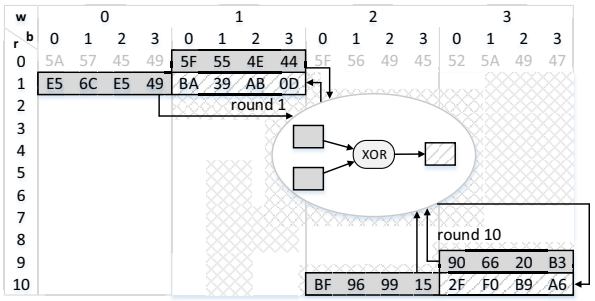

Figure 1. AES key expansion: simple XOR operation for all but the first word in each round (w: word, b: byte, r: round) [5].

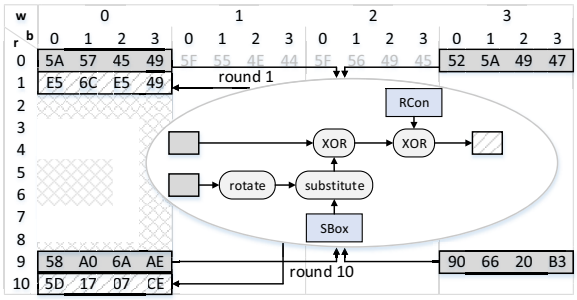

Figure 2. AES key expansion: complex operation for first word in each round [5].

C. DRAM Decay and Error Models

As mentioned before, DRAM contents of the target machine will gradually decay when powered off, which is denoted as the remanence effect [2]. Halderman et al. demonstrated the properties of memory bit decay and summarized them as decay pattern. Gruhn et al. [11] recently repeated these tests and verified the claims for SDRAMs with DDR1 and DDR2 technology. However, they could not run a successful cold-boot attack against modern DDR3 SDRAM, because their contents decay faster and produce more bit errors. This observation increases the need for more efficient search and reconstruction approaches.

According to the statistics of Halderman et al. the bit errors that flipped to the ground state of the memory cell

dominated (99.9%) over the bit errors that flipped to the opposite direction (0.1%). Thus there are two different bit error models for memory decay. *Perfect Asymmetric Decay* (PAD) ignores the second error case and assumes that there only exist bit errors in the direction of the ground state of the memory cell. All published cold-boot attack methods are based on this model, because it allows for easily testing whether a memory word obtained in a cold-boot attack is *compatible* with the original memory content (i. e. it could be a decayed version of that content). Since in reality bit errors in both directions are possible, Wang [12] proposed a threshold-based approach to take both error directions into account. His *Expected Value as Threshold* (EVT) model separates the overall rate of bitflips b in bitflips b_0 ($1 \rightarrow 0$) and b_1 ($0 \rightarrow 1$). With given b_1 and b_0 rates we can compute the expected number of bitflips in each direction by multiplying the rates with the total number of bits n in the full key schedule (see Equation 1). If the number of bits actually flipped from $1 \rightarrow 0$ or $0 \rightarrow 1$, denoted by n_0 and n_1, exceeds its expected value, then the candidate is not compatible.

$$\text{candidate compatible} = \begin{cases} n_0 < b_0 \cdot n \\ n_1 < b_1 \cdot n \end{cases} \quad (1)$$

Note that from the perspective of statistics, randomly introducing bitflips with probabilities b_0, b_1 may lead to more bitflips than the expected value. For simplicity, we select the parameters to account for this deviation.

III. Maxeler Computer System

For implementing our accelerated application we use a Maxeler MPC-C platform. The overall system architecture and corresponding programming model is illustrated in Figure 3. The FPGA-based system contains four MAX3424A cards (Xilinx Virtex-6 SX475T with 24GB of on-board SDRAM memory) attached to a two socket Intel Xeon host server via PCIe. The host comprises two 6-core (12 threads) Intel Xeon X5650 CPUs (Westmere EP) running at 2.67GHz, 48GB DDR3-1333 of main memory and 3 SSDs in RAID0 configuration for fast I/O access.

The design flow for mapping applications is based on the MaxCompiler [13] programming model. It is entirely driven by Java and offers predefined APIs for specifying *data flow engines (DFE)*. Each DFE comprises of one or more *kernels*, which implement the application logic and a *manager* that controls the routing of data streams between kernels, the CPU and off-chip memory. MaxCompiler is able to take care of type conversions, can automatically perform optimizations like retiming, buffer size optimization and pipelining. Furthermore it also offers an API to describe finite state machines that can control memory streams and data paths for applications that can not be expressed as a feed-forward data path. The specified kernel and manager code is translated into a hardware design which is further

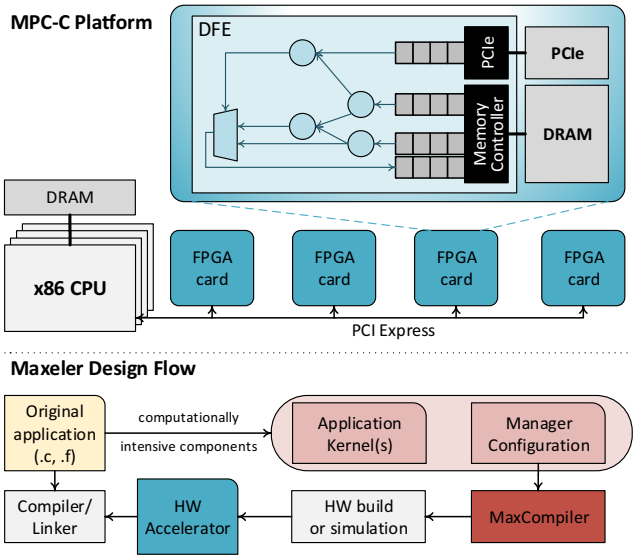

Figure 3. Maxeler MPC-C platform architecture and design flow [5].

processed by FPGA vendor tools to generate a configuration bitfile. Afterwards the bitfile is merged with additional information into a *maxfile* and linked with the original CPU application.

IV. Key Reconstruction

After the key search produces candidates for key schedules that have only a limited number of consistency errors, a valid key schedule without any errors needs to be reconstructed. In this section, we first outline the software approach to this reconstruction task, before discussing its hardware implementation with details of its two performance critical tasks.

A. Software approach

The software algorithm for key reconstruction as proposed by Tsow [3] resembles a depth-first tree search, implemented recursively. Listing 1 outlines the main recursive function of the algorithm. For the candidate schedule `CandSched C` that was obtained from memory and contains decayed bits, the initially empty recovered schedule `RecSched R` is built up by guessing an additional byte g_0 in every recursive call (`setNextGuessedByte`) and computing all bytes that can be derived from the guessed bytes (`computeDerivedBytes`). If the thus extended recovered schedule `R_next` still passes the compatibility check (`isCompatible`) with the error model, the next recursive call descends further into the search tree. Otherwise, or if no valid key schedule is found further down the tree, the next possible value for g_0 is tried. Note that for better performance, we replaced the copy operation of the recovered schedule that is indicated in line 5 by update and rollback operations. The positions of the guessed bytes are chosen

Table I
PATH FOR GUESSING BYTES FOR AES-128 [3].

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

according to a predefined path, which allows a sufficient number of bytes for reconstruction of the entire key schedule to be derived after guessing 16 bytes. Therefore, after 16 bytes are guessed and checked for compatibility along with all derived bytes, the recovered schedule `R` can be returned as valid key schedule.

```
1  recoverKeyRec(RecSched R, CandSched C):
2    if (R.getGuessedBytes() == 16):
3      return R.key();
4    for g0 = 0 to 255:
5      R_next = R;
6      R_next.setNextGuessedByte(g0);
7      R_next.computeDerivedBytes();
8      if(C.isCompatible(R_next)):
9        key = recoverKeyRec(R_next, C)
10       if (key != NULL):
11         return key;
12   return NULL;
```

Listing 1. Recursive algorithm for key reconstruction.

The path used to guess bytes is illustrated in Table I, where the guessed bytes are represented in bold as $\mathbf{0_0}$ to $\mathbf{15_0}$. The table also illustrates, which other bytes can be derived after each byte is guessed. For example, after byte $\mathbf{15_0}$ is guessed, the bytes 15_1 to 15_{10} can be derived from the combination of $\mathbf{15_0}$ and all previously guessed and derived bytes. Note that we also used the inverse variants of the expansion rules, resulting in a backward calculation inside the key schedule. After all derivations of the guessed byte $\mathbf{15_0}$ are computed, the entire round 8 of the key schedule is complete. From this, the complete key schedule and thus the actual key in round 0 of the schedule can be reconstructed afterwards.

B. HW Implementation

The first challenge is to translate this recursive branch-and-bound algorithm to hardware. Since the recursion depth is limited to 16, we are able to design a finite state machine (see Figure 4), where the current search tree node, which is implicitly represented by the call stack of the recursive function in software, is translated into explicit tree nodes that are pushed to and popped from a 16 entry stack in the fast local memory on the FPGA (BRAM). Each of those tree nodes on the stack stores the position inside the search tree along with all already guessed and derived bytes of the reconstructed schedule. For the EVT model we additionally need to store the sums of previous (consumed) bitflips. In contrast to software, since our stack has a sufficient bitwidth to access an entire tree node in parallel, we would not gain performance by using update and rollback mechanisms. The `NEXT_STAGE` state corresponds to a descent in the search tree, keeps track of the current stage (level of the search tree) and determines which byte is to be guessed next accordingly to the search path.

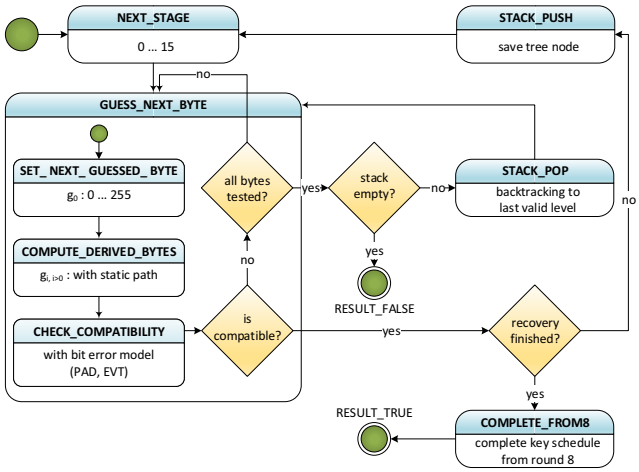

Figure 4. FSM that implements the recursive key reconstruction algorithm.

The superstate `GUESS_NEXT_BYTE` corresponds approximately to the `for` loop of the recursive algorithm and tries all possible values 0 to 255 for the guessed byte g_0 sequentially. It contains substates to set the guessed byte, to compute all derived bytes and to perform a consistency check. The latter two consume most runtime in software and thus are crucial in order to gain performance, which was the second challenge for the hardware implementation of this algorithm. The tasks

inside those substates `COMPUTE_DERIVED_BYTES` and `CHECK_COMPATIBILITY` differ in each stage not only by their inputs, but also in the type and number of derivation steps to apply for the former state and in the number of new bytes to check for the latter stage. The number of derived bytes and consequently of bytes to check depends on the stage n and follows Equation 2.

$$\#\text{ derived bytes} = \begin{cases} n & \text{if } n < 11 \\ 10 & \text{if } 11 \leq n \leq 14 \\ 65 & \text{if } n = 15 \end{cases} \quad (2)$$

Thus for the mentioned two states we decided to implement a separate sub-circuit for each reconstruction stage. This saves the latencies for selecting the inputs and enables optimization of their datapaths for their specific tasks. Additionally we gathered statistics about how often each stage is reached depending on the bitflip rate (see Figure 5) and found that stages 8 and higher are typically reached at least three orders of magnitude less frequently than the most frequent ones (stages 2 to 4). Therefore the more frequent stages are optimized with elaborate combinational datapaths whereas multi-cycle implementations are chosen for stages 8 and higher in order to minimize their impact on clock frequency.

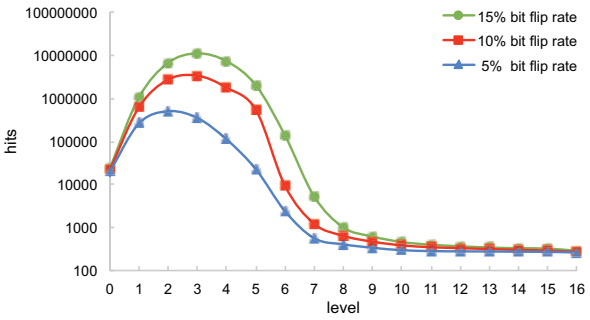

Figure 5. Number of times each level is reached for 256 test cases.

C. Computing derived bytes

Computing the derived bytes follows the fixed pattern of Table I. As motivated before, we hardcoded for each stage all its implications into one sub-circuit. This also allowed us to hardcode the round constants RCon (required in the complex key expansion rule) into the sub-circuits to save accesses to BRAM, since in each stage at most a few known round constants are used. Note that all derived bytes of each stage depend on each other, so for example after guessing $\mathbf{15_0}$, first 15_1 needs to be derived from $\mathbf{15_0}$ and 11_4, before 15_2 can be derived from 15_1 and 11_5. All derivation operations that only apply the simple XOR operation are easily combined into a single cycle combinational path, for example from $\mathbf{15_0}$ all the way left to 15_3. After hardcoding the round constants RCon for each stage, the operations of the complex expansion rule can also be merged into this combinational path. However, this additionally requires to store the data for the substitution operation SBox in distributed LUT RAM that can be accessed without a latency cycle. With 256 bytes, the SBox table is slightly larger than a typical use case for LUT RAM, but easily tolerable for the short and frequent stages of the `COMPUTE_DERIVED_BYTES` state. With this optimization, we were able to implement single-cycle combinatorial datapaths for the stages 0 to 7. For the longer and conveniently less frequent stages 8 and higher, we instead access BRAM for the SBox lookup, splitting the state into a chain of substates, each one ending with a read request `SBox-Req` and starting with the corresponding read in the next cycle, indicated by the clock signal between the substages. For example 15_4 is then computed as first byte of the next substate, along with 15_5 to 15_7 which follow combinatorially. Afterwards, the substate ends with the next read request to compute 15_8. In Figure 6, we illustrate stage 3 as example for a single cycle datapath with complex expansion rule and stage 15 as example for a multi cycle datapath.

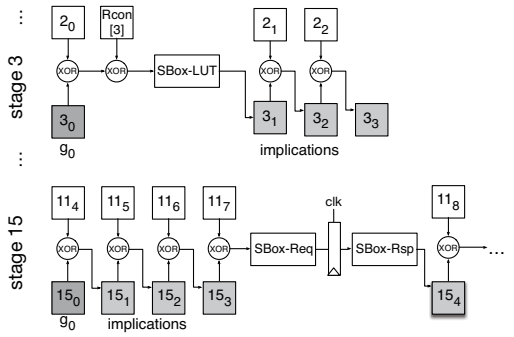

Figure 6. Illustration of datapaths for stage 3 and stage 15 of `COMPUTE_DERIVED_BYTES`.

D. Checking compatibility

We implemented the `CHECK_COMPATIBILITY` state in two variants, one reflecting the Perfect Asymmetric Decay (PAD) model and the other one implementing the Expected Value as Threshold (EVT) model (see Section II-C).

For the PAD model, each byte in the recovered schedule `R` can be checked independently for compatibility with the candidate schedule `C` from memory. It is compatible if the two bytes match, or if between the byte from `R` and the byte from `C`, bits are only flipped towards the ground state of the memory cell. In the `CHECK_COMPATIBILITY` state, this can be checked in parallel and in a single cycle. According to Tsow [3] a candidate byte r_i is compatible with the decayed byte c_i, when r_i preserves all ground-state bits m_i. This can be expressed as $(r_i \oplus c_i) \wedge (r_i \oplus m_i) = 0$.

For the EVT model, at each guess the bitflips of g_0 and all derived bytes need to be summed up and compared to the expected values, which we compute on the host CPU and add as parameters $\texttt{exp\_n}_0$ and $\texttt{exp\_n}_1$ to the search. This summation of bitflips is similar to a balanced adder tree (see Figure 7). We need to compute two separate sums for bits flipped in either direction $1 \rightarrow 0$ or $0 \rightarrow 1$, respectively, which can be done in parallel. We use adders with the specific bit-widths required to represent the highest possible bitflip value at each level. Afterwards, we add the sums from the previous search stage $\texttt{prev\_n}_0$ ($\texttt{prev\_n}_1$) to the bitflips of those bytes that were guessed or derived in the current stage. If each value is less or equal than the expected values, the candidate key schedule is compatible and passed to the next stage.

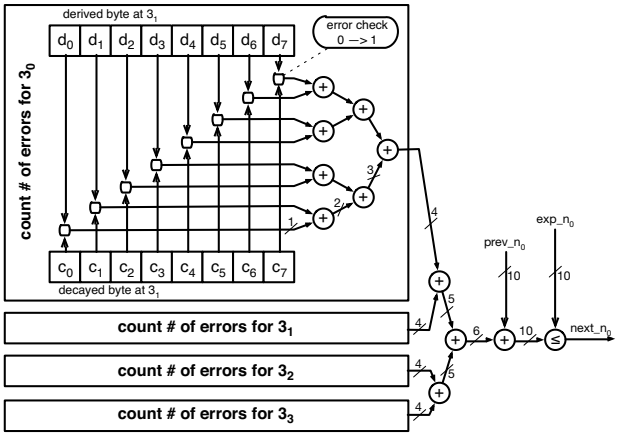

Figure 7. Checking compatibility with EVT error model for bit flips $0 \rightarrow 1$ for stage 3.

The size of the required adder tree follows the number of derived bytes depending on the stage n according to Equation 2 as outlined above. Using this information, we added additional substates into the `CHECK_COMPATIBILITY` state to split the summation for stages 8 and higher into more cycles. Otherwise those stages become the critical path of our design and reduce the maximal achievable frequency. Nevertheless, the EVT error model still has a strong impact on our achieved clock frequencies, as shown in the synthesis results of our reconstruction kernels for the two error models in Table II. Our implementation achieves 175 MHz for the PAD error model, but only 90 MHz for the EVT model.

V. Results

In this section, we discuss some details of our test case generation and of our software implementation of the key reconstruction. We then use this software to evaluate the performance of our hardware implementation. We performed all hardware tests on the Maxeler system presented in Section III with *one* of its four accelerator cards. For the software implementations, unless otherwise stated, we evaluated the performance on a single, exclusively used cluster node with two 2.6 GHz octa-core Xeon E5-2670 processors with 64GB of main memory.

Table II
Synthesis results of two key reconstruction kernels targeting a Virtex-6 SX475T FPGA.

	AESKeyFixPAD	**AESKeyFixEVT**
Used LUTs (%)	6.65	8.74
Used FFs (%)	5.06	5.04
Used BRAMs (%)	1.41	1.41
Used DSPs (%)	0.00	0.00
Achieved Frequency (MHz)	175	90

A. Test Case Design

Independently of the concrete decay model, there are two different metrics to describe the decay state of a found candidate schedule. Tsow's [3] tests, which are limited to the PAD model, are designed on the basis of a decay rate d that specifies for each bit in the schedule the probability to switch to its ground state. Let the ground state be 0 throughout this section. Then a decay rate of e.g. $d = 60\%$ means that 60% of all bits are decayed, so if they have been 1 before they became 0 and if they have been 0 before, they remained 0. Due to the cryptographic properties of AES, a key schedule has on average an equal amount of $1s$ and $0s$. Thus, at a decay rate of $d = 60\%$, on average 50% of all bits have been $0s$ in the first place and remained $0s$, either after decay or not. Further 30% of all bits are flipped $1 \rightarrow 0$ and the last 20% remain $1s$. However, this rate of on average 30% bitflips $1 \rightarrow 0$ varies a lot depending on how the actual distribution of $1s$ and $0s$ in each concrete schedule is and how they are struck by the random decay process, which may affect the difficulty of the actual reconstruction task a lot. Therefore, Wang [12] decided to instead fix this rate of bitflips $1 \rightarrow 0$ as b_0 and generate test cases accordingly. This also allows to easily add a second rate b_1 for bitflips $1 \rightarrow 0$ for the EVT model, with the property that the total rate of bitflips b is computed by $b = b_0 + b_1$. Conceptually, Tsow's metric is closer to the actual decay process, whereas Wang's model better captures the bitflip rates of candidate schedules retrieved from actual key search in memory. We use Tsow's decay rate d only for the comparisons with Tsow's results and otherwise utilize Wang's bitflip rates.

For our performance tests, we generate 10000 random keys by reading 16 bytes from /dev/urandom for each key. After expanding the full key schedule we simulate a memory decay according to one of the presented methods. Since our accelerator follows the path of Table I, we first need to determine a compatible byte at the first position $\mathbf{0_0}$ of the key schedule. If the searched value is large, the reconstruction needs more effort, because our state machine incrementally tries every possible value (0-255). Hence, we need a sufficiently large number of test cases to measure representative

average runtimes. Conveniently, 10000 candidate schedules can also represent a realistic workload for a real cold boot attack. At a biflip rate of 30%, 422 bits of a AES-128 key schedule flip. In a concrete test, searching 1 GB of real memory contents for key schedule candidates with up to 422 bitflips resulted in 554 candidates. Extrapolating this to 16 GB of RAM, 8864 candidates might occur in this example. However, these numbers should only be considered as anecdotal evidence and may greatly vary with decay rates, memory size and software stack of the system under attack.

B. Software Implementation

Since reference software was only partially available and had serious shortcomings for our tests, we implemented our own software reference. Tsow's [3] software is written in C, but only supports the PAD error model. Wang's [12] software supports both error models, but is written in Java and around one order of magnitude slower than Tsow's implementation. In order to achieve good performance for our software, we implemented the reconstruction as an iterative algorithm to avoid large copy operations of the call stack during recursion steps. Additionally we introduced a sophisticated guess procedure which only needs one instance of a candidate key schedule (array) to perform. As mentioned before, we extend the candidate key schedule according to the path on a valid assignment of g_0 (and its derived bytes) and use a rollback function respectively, if the computation on some level is exhausted.

Our software implementation supports both PAD and EVT error models. It is compiled with gcc-4.4.7 at the highest optimization level, -O4. We compared its PAD variant with Tsow's software that is limited to this variant. Tests were executed with the path of Table I for 10000 test cases for each decay rate 30%, 40%, 50% and 60% likewise to Tsow and according to his definition of error rate. Table III summarizes the results. Note that the listed values (marked with asterisk) are obtained from the publication of Tsow and only serve for a rough comparison, since his test cases were not available. So both computations were performed on different test cases, but with a sufficiently large and equal number of tests. Due to our code optimization and probably to some degree also due to our faster CPU, our software implementation achieves an overall speedup of around 3x over Tsow's implementation and thus is to our knowledge the fastest one using the presented reconstruction technique.

C. Performance Comparison of Software to Hardware

We compare our software implementation to our presented hardware accelerator, also supporting both error models. The results for PAD are summarized in Table IV. The hardware achieves a speedup of around 6x for all bitflip rates except for the lowest tested bitflip rate, where the individual runtimes are small enough to be affected by call overheads to the hardware, which limits the speedups to 3x in this case.

Table III
RUNTIME [S] OF OUR C IMPLEMENTATION COMPARED TO TSOW.

decay d		**Σ**	**Ø**	**St.Dev.**	**Speedup**
30%	Tsow*	219.204	0.022	0.140	
	our impl.	76.454	0.008	0.124	2.9
40%	Tsow*	1,526.308	0.153	2.994	
	our impl.	474.249	0.047	0.471	3.2
50%	Tsow*	32,551.469	3.255	55.563	
	our impl.	11,602.430	1.160	25.752	2.8
60%	Tsow*	1,638,788.166	163.879	3,753.608	
	our impl.	472,374.406	47.237	850.187	3.5

Table IV
RUNTIME [S] WITH PAD FOR 10000 TEST CASES EACH.

bitflips b_0		**Σ**	**Ø**	**St.Dev.**	**Speedup**
5%	CPU	14.901	0.001	0.007	
	FPGA	4.956	0.000	0.001	3.0
10%	CPU	118.731	0.012	0.032	
	FPGA	20.095	0.002	0.008	5.9
15%	CPU	174.841	0.017	0.080	
	FPGA	28.437	0.003	0.015	6.1
20%	CPU	659.96	0.066	0.772	
	FPGA	110.835	0.011	0.136	6.0
25%	CPU	13,895.451	1.390	21.399	
	FPGA	2,187.083	0.219	3.179	6.4
30%	CPU*	2,599,493.566	259.949	3,912.294	
	FPGA	418,751.305	41.875	602.368	6.2

For EVT, we performed two test series, one with a bitflip rate into the opposite direction of the ground state of $b_1 = 0.1\%$ and one with $b_1 = 0.2\%$. The results are summarized in Table V and Table VI. The overall runtimes increase with higher bitflip rates both for CPU and FPGA. However, the FPGA implementation apparently scales better, so the speedups increase from 6.5x for $b_0 = 4.9\%$ and $b_1 = 0.1\%$ to 27x for $b_0 = 24.8\%$ and $b_1 = 0.2\%$. As a result, the advantage of the FPGA implementation grows specifically when the runtimes become a practical issue.

Table V
RUNTIME [S] WITH EVT ($b_1 = 0.1\%$) FOR 10000 TEST CASES EACH.

b_0		**Σ**	**Ø**	**St.Dev.**	**Speedup**
4.9%	CPU	329.714	0.033	0.728	
	FPGA	50.499	0.005	0.027	6.5
9.9%	CPU	1,483.592	0.148	1.132	
	FPGA	161.574	0.016	0.109	9.2
14.9%	CPU	10,428.147	1.043	10.305	
	FPGA	881.636	0.088	0.832	11.8
19.9%	CPU	95,249.41	9.525	142.731	
	FPGA	6,030.007	0.603	8.397	15.8
24.9%	CPU*	1,996,589.189	199.659	3,071.691	
	FPGA	98,269.133	9.827	117.238	20.3

D. Practical Considerations

In the last rows of Tables V to VI, the CPU runtimes are marked with an asterisk. In contrast to the other measurements they were not performed on the aforementioned workstation, because after running the FPGA tests and given the observed speedups from the earlier tests, we expected very long software runtimes. Therefore the tests

Table VI
RUNTIME [S] WITH EVT ($b_1 = 0.2\%$) FOR 10000 TEST CASES EACH.

b_0		Σ	Ø	St.Dev.	Speedup
4.8%	CPU	2501.573	0.250	2.845	
	FPGA	260.108	0.026	0.257	9.6
9.8%	CPU	12335.19	1.234	21.820	
	FPGA	1089.177	0.109	1.776	11.3
14.8%	CPU	75171.739	7.517	83.372	
	FPGA	5095.195	5095.195	6.53	14.8
19.8%	CPU	773118.366	77.312	909.462	
	FPGA	36250.274	3.625	30.053	21.3
24.8%	CPU*	10636553.810	1063.655	13973.626	
	FPGA	392347.162	39.235	335.88	27.1

were executed on a cluster of 900 nodes, all of the same type as described before. Each of the 10000 test cases for this bitflip rate was allocated to a central scheduler, which assigned them to a free node. Whenever a job received a free resource, the computation was performed exclusively without any concurrent task. For the presented runtimes, we summed up the total runtimes of all reconstruction jobs. With a total runtime of 23 days for $b_1 = 0.1\%$ and 123 days for $b_1 = 0.2\%$, not only our evaluation but also any real cold boot attack with similarly decayed memory would have been seriously delayed by a sequential software reconstruction, where respective sequential FPGA reconstructions finished in 29 hours or four and a half days, respectively.

As discussed before, the reconstruction difficulty of the various candidate schedules differs a lot even for the same error rates. This is also reflected in the high standard deviations in all result tables. Therefore, even when large amounts of parallel resources are available, in our case 900 software nodes, the achievable runtimes for an entire workload of 10000 candidates to reconstruct have a lower limit given by the longest running individual reconstruction. In our experiments, this was about three days for $b_1 = 0.1\%$ and twelve days for $b_1 = 0.2\%$ in software, which is almost three times more than the entire runtime for all reconstructions on one single FPGA accelerator. So in this regard, a single FPGA system beats the utilized CPU cluster not only by orders of magnitude in asset cost and energy consumption, but still delivers more practical usefulness.

VI. CONCLUSION

In this paper, we have presented our approach to accelerate key reconstruction for cold-boot attacks using a hardware accelerator. We have adapted a branch-and-bound algorithm for this task and implemented it on an FPGA. Previous results were limited to a simplified error model, assuming Perfect Asymmetric Decay. With the achieved acceleration, a more realistic error model can be supported, even at high error rates. We have achieved speedups of up to 6x for the ideal PAD model and 27x for the more realistic EVT model.

In our future work, we plan to improve the performance by using all 4 DFE cards in our Maxeler system in parallel. Since every key reconstruction is independent, we could use an allocation system like for the long running software tests on the cluster. Moreover, we will investigate how the tree search process in the key reconstruction algorithm can be parallelized. One direction we plan to explore is a work stealing approach that has been successfully used for parallelizing search in unbalanced trees in software.

ACKNOWLEDGEMENT

This work was partially supported by the German Research Foundation (DFG) within the Collaborative Research Centre "On-The-Fly Computing" (SFB 901), the European Union Seventh Framework Programme under grant agreement no. 610996 (SAVE), and the Maxeler university program MAXUP.

REFERENCES

[1] "Specification of the Advanced Encryption Standard (AES)," Federal Information Processing Standards, NIST Standard 197, 2001.

[2] A. J. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten, "Lest we remember: Cold boot attacks on encryption keys," in *Proc. Conf. on Security Symposium (SS)*. USENIX Association, 2008, pp. 45–60.

[3] A. Tsow, "An improved recovery algorithm for decayed AES key schedule images," in *Proc. Int. Workshop on Selected Areas in Cryptography (SAC)*. Springer, 2009, pp. 215–230.

[4] J. Zambreno, D. Nguyen, and A. Choudhary, "Exploring Area/Delay Tradeoffs in an AES FPGA Implementation," in *Proc. Int Conf. on Field Programmable Logic and Application (FPL)*. Springer, 2004, pp. 575–585.

[5] H. Riebler, T. Kenter, C. Sorge, and C. Plessl, "FPGA-accelerated key search for cold-boot attacks against AES," in *Proc. Int. Conf. on Field Programmable Technology (ICFPT)*. IEEE, 2013.

[6] A. Shamir and N. van Someren, "Playing hide and seek with stored keys," in *Proc. Int. Conf. on Financial Cryptography (FC)*. Springer, 1999, pp. 118–124.

[7] B. Kaplan and M. Geiger, "RAM is key: Extracting disk encryption keys from volatile memory," Master's thesis, Carnegie Mellon University, 2007.

[8] M. Albrecht and C. Cid, "Cold boot key recovery by solving polynomial systems with noise," in *Proc. Int. Conf. on Applied cryptography and network security (ACNS)*. Springer, 2011, pp. 57–72.

[9] A. A. Kamal and A. M. Youssef, "Applications of SAT solvers to AES key recovery from decayed key schedule images," in *Proc. Int. Conference on Emerging Security Information, Systems and Technologies (SECUWARE)*. IEEE Computer Society, Jul. 2010, pp. 216–220.

[10] J. Daemen and V. Rijmen, "The block cipher Rijndael," in *Proc. Int. Conference on Smart Card Research and Applications (CARDIS)*. Springer, 2000, pp. 277–284.

[11] M. Gruhn and T. Müller, "On the practicability of cold boot attacks," in *Proc. Int. Workshop on Frontiers in Availability, Reliability and Security*. IEEE Computer Society, 2013.

[12] Q. Wang, "Localization and extraction of cryptographic keys from memory images and data streams," Master's thesis, University of Paderborn, 2012.

[13] O. Pell and V. Averbukh, "Maximum performance computing with dataflow engines," *IEEE Computing in Science & Engineering*, vol. 14, pp. 98–103, 2012.