Aiding Side-Channel Attacks on Cryptographic Software With Satisfiability-Based Analysis

Nakhiketh R. Potlapally, Anand Raghunathan, Srivaths Ravi, Niraj K. Jha, and Ruby B. Lee

Abstract—Cryptographic algorithms, irrespective of their theoretical strength, can be broken through weaknesses in their implementations. The most successful of these attacks are side-channel attacks which exploit unintended information leakage, e.g., timing information, power consumption, etc., from the implementation to extract the secret key. We propose a novel framework for implementing side-channel attacks where the attack is modeled as a search problem which takes the leaked information as its input, and deduces the secret key by using a satisfiability solver, a powerful Boolean reasoning technique. This approach can substantially enhance the scope of side-channel attacks by allowing a potentially wide range of internal variables to be exploited (not just those that are trivially related to the key). The proposed technique is particularly suited for attacking cryptographic software implementations which may inadvertently expose the values of intermediate variables in their computations (even though, they are very careful in protecting secret keys through the use of on-chip key generation and storage). We demonstrate our attack on standard software implementations of three popular cryptographic algorithms: DES, 3DES, and AES. Our attack technique is automated and does not require mathematical expertise on the part of the attacker.

Index Terms—AES, cryptanalysis, DES, satisfiability, security, side-channel attacks, software, 3DES.

I. INTRODUCTION

Security has emerged as a critical concern in a wide range of electronic systems. Extensive experience with the use and deployment of security technologies has shown that, in practice, most security systems are broken by exploiting weaknesses in their implementations, making it important to consider security during the complete design process.

Cryptographic primitives, such as encryption and hashing algorithms, form the basis of most security mechanisms. A cryptographic system may be abstracted as a mathematical function that performs a given mapping of its input to its output. However, in reality, it should be viewed as a specific (hardware or software) implementation of the mathematical function. Cryptanalysis refers to the process of breaking a cryptographic system without a brute-force search (e.g., for an given mapping of its input to its output) and requires only a little mathematical expertise on the part of the attacker. These attacks are broken by exploiting weaknesses in their implementations. One of the most effective of these attacks is side-channel attacks which exploit unintended information leakage, e.g., timing information, power consumption, etc., from the implementation to extract the secret key. We propose a novel framework for implementing side-channel attacks where the attack is modeled as a search problem which takes the leaked information as its input, and deduces the secret key by using a satisfiability solver, a powerful Boolean reasoning technique. This approach can substantially enhance the scope of side-channel attacks by allowing a potentially wide range of internal variables to be exploited (not just those that are trivially related to the key). The proposed technique is particularly suited for attacking cryptographic software implementations which may inadvertently expose the values of intermediate variables in their computations (even though, they are very careful in protecting secret keys through the use of on-chip key generation and storage). We demonstrate our attack on standard software implementations of three popular cryptographic algorithms: DES, 3DES, and AES. Our attack technique is automated and does not require mathematical expertise on the part of the attacker.

REFERENCES


Manuscript received March 2, 2006; revised May 27, 2006. This work was supported by the National Science Foundation under Grant CCR-0326372.
While SAT has been shown to be NP-complete, there exists, else give a proof that no such assignment is possible.

In hardware and software verification, a wide range of areas have been formulated as SAT problems. In solving techniques that have been incorporated into freely-available SAT solvers, the versatility and effectiveness of SAT solving techniques, it was a first attempt to apply Boolean analysis techniques to side-channel attacks.

While some of the early side-channel attacks targeted hardware implementations, software implementations are equally, if not more, vulnerable. Data exposure can occur in software implementations through memory bus exposure, core dump files, persistence of data in disk memory after swap, etc. [12]. This problem of data exposure exists even in secure software implementations [13]. Recent studies have revealed the possibility of data exposure from software computations even after the computation is over [14]. In some instances, even sensitive data, like passwords, were left in accessible system buffers. Software side-channels typically reveal data in bytes or larger words, making them especially attractive targets for attacks.

In this paper, we propose a framework for side-channel attacks by formulating the analysis phase as a Boolean search problem and solving it using state-of-the-art satisfiability (SAT) solvers. We demonstrate this approach in the context of software side-channel attacks. Our approach substantially enhances the scope of side-channel attacks by allowing a potentially wide range of internal variables to be exploited (not just those that are trivially related to the key). The exposure of secret keys or variables that are directly related to them leads to a trivial compromise of security. For example, in the DES algorithm, knowledge of the inputs to each S-box in a round will allow the attacker to trivially calculate the key. Therefore, secret keys and other “easy targets” are often protected from exposure, e.g., through the use of protected on-chip key generation and storage [15]. However, seemingly harmless variable values, if exposed, can be sufficient to deduce the secret keys when powerful analysis techniques, such as the SAT solver used in this paper, are employed.

The Boolean SAT problem is defined as follows. Given a Boolean formula made up of a conjunction of clauses, each of which is a disjunction of Boolean literals, determine whether values can be assigned to the literals such that all the clauses in the formula are satisfied, i.e., evaluate to 1. Such a literal assignment is referred to as the satisfying assignment. The function of an SAT solver is to find a satisfying assignment for any given Boolean formula, if one exists, else give a proof that no such assignment is possible. While SAT has been shown to be NP-complete, efficient heuristics exist that can solve many real-life SAT formulations. Furthermore, the many applications of SAT have motivated advances in SAT solving techniques that have been incorporated into freely-available SAT software tools [16], [17]. Many practical search problems in a wide range of areas have been formulated as SAT problems. In the field of design automation, SAT has been successfully applied in hardware and software verification and circuit testing. Given the versatility and effectiveness of SAT solving techniques, it was a natural choice to use an SAT solver as an automated reasoning engine in our proposed framework for enabling side-channel attacks.

The contributions of this paper include the following:

- Characterization of the minimal nontrivial subsets of internal variables sufficient to break DES, 3DES, and AES algorithms, and successful application of the proposed attack on standard software implementations of the three algorithms.

In the context of software attacks, existing work gives ample evidence of software data leakage. However, there is no general framework to transform these vulnerabilities into actual attacks on security software. Furthermore, when implementations take basic measures to protect the keys and other directly related variables from leakage, more powerful analysis techniques, such as the one proposed in our work, are necessary. From another perspective, a knowledge of the internal variables that can be used to launch side-channel attacks can translate into design guidelines that point to parts of the implementation that should be protected. Previously, Massacci and Marraro [18] proposed modeling of DES as an SAT formulation for studying cryptographic properties of DES, and for traditional cryptanalysis (which is based on the knowledge of only the plaintext and ciphertext). However, their results showed the inability of SAT to perform traditional cryptanalysis. To the best of our knowledge, this is the first attempt to apply Boolean analysis techniques to side-channel attacks.

The rest of this paper is organized as follows. Section II discusses the various ways in which software side-channels are created, thereby enabling side-channel attacks. Section III gives an overview of our proposed technique and illustrates the formulation of a cryptographic algorithm as a Boolean formula using DES, 3DES, and AES as examples. Section IV presents results of our comprehensive experiments with a state-of-the-art SAT solver to identify the intermediate values in DES, 3DES, and AES which enable successful inferring of the key. We conclude in Section V with our observations and directions for future work.

II. MOTIVATION

In this section, we motivate the prevalence of leakage of intermediate values from software computations. We begin by enumerating the various techniques employed in practice to obtain the intermediate values. We conclude the section by giving details of the scheme employed in our work to obtain the intermediate values required by our SAT-based cryptanalysis framework to cryptanalyze cryptographic algorithms.

A. Leakage of Software Variable Values

The architecture of a typical hardware-software system implementing a cryptographic computation is shown in Fig. 1. For a given value of the secret key, it injectively maps a plaintext to a ciphertext. The architecture is divided into software and hardware subsystems. The software subsystem comprises application programs layered on top of the system libraries and the operating system (OS). System libraries make commonly used software routines available to an application code through a function call interface while a system call interface enables an application to access hardware resources via the OS. The OS interacts with the hardware subsystem using machine instructions. The hardware component consists of the processor, input-output devices, and the hierarchical memory system made up of an on-chip cache, main memory, and magnetic disk storage. The processor has an on-chip memory which stores the cryptographic key, thereby preventing exposure of the key bits through operations which cause the key bits to be transmitted outside the processor’s secure perimeter (usually onto the main memory bus) [12]. However, the complexities involved in implementing such hardware-software systems opens avenues for intentional or unintentional leakage of data values at the various interfaces present: application-library, application-library, OS-library, and OS-software. In the following, we present specific cases of such data leakage which were observed in practice (indicated in Fig. 1).
Unintentional leakage: This happens inadvertently during normal operation due to bugs, improper policies, misconfiguration, etc. Chow et al. [14] showed the existence of program data in system buffers in main memory long after the program terminated. This included sensitive data, like passwords. Swapping of program data to disks greatly increases the probability of exposure due to the data retention property of magnetic disks. Garfinkel and Shelat [19] showed concrete evidence of persistence of data on disks and presented ways to extract the data. Core dumps occur when an application program crashes after performing an illegal operation, and are used for finding out the cause of failure. This diagnostic information can expose program data [12]. Broadwell et al. showed presence of sensitive information in the system crash reports generated by a widely used OS [20].

Intentional leakage: This is precipitated by attacks which are crafted to exploit latent system vulnerabilities. A common ploy is to break a system by forcing it to implement operations other than the ones it was designed for. Examples of such malicious hardware or software attacks include hijacking the run-time stack [21], proactively probing the cache using a Trojan process [22], monitoring the memory traffic on the system bus [23], etc. Also, there exist tools for dynamically examining the contents of program memory as the program is being executed [24].

Even software systems that have been implemented with the aim of maintaining security of the data being manipulated have been shown to have security breaches [25]. Thus, it is very hard to design hardware-software systems which completely prevent any kind of data exposure.

B. Proactive Memory Bus Monitoring-Based Leakage

In this section, we illustrate the leakage technique employed by us to obtain the intermediate values which were used in our cryptanalysis framework. Our leakage technique is based on proactive manipulation of the data cache coupled with bus monitoring, in order to gather the intermediate values of the cryptographic computation. However, it should be stressed that our SAT-based framework is independent of the methods employed to leak the intermediate values of computation.

A cache is an on-chip buffer which stores recent memory accesses of a program with the aim of improving performance by exploiting spatial and temporal locality of the program. Consider an N-way set-associative cache having B-byte cache lines and M cache sets. Thus, the total size of the cache is \( MN \times B \) bytes (each of the \( M \) cache sets holds \( N \) cache lines). A main memory line having address \( A \) is mapped to the cache set given by \( [A/B] \mod M \). Given that a cache is much smaller than main memory, multiple memory lines can be mapped to the same cache line. This contention results in eviction of an existing line from the cache when a new memory line is mapped to a cache line already holding one. Also, except in case of software-controlled caches, an application has no control over where memory lines holding its data are placed. With the help of an accurate memory profiler [26], we examined the number of intermediate values evacuated from the cache onto the memory bus during the execution of a standard DES software implementation. For a 1-kB data cache, 33% of the intermediate variables are evicted onto the memory bus, and when the cache size is increased to 2 kB, this figure falls to 12%, and becomes smaller for further increases in data cache size. This motivates the need for a scheme which will proactively evict intermediate variable values from the cache.

Modern computer systems employ a variety of sophisticated hardware and software mechanisms to enforce isolation of processes running simultaneously on the system. However, one low-level hardware resource, where execution of different processes intersects, is the cache. Though a process cannot read the contents of cache lines holding data belonging to another process, it can influence the execution of the other process. For example, it can increase the number of cache misses of the other process [22]. Given memory line \( Y \) is mapped to cache line \( X = [Y/B] \mod M \) (\( B \) and \( M \) are defined as before), all the memory lines mapping to cache line \( X \) are given by \( \{ Y | Y = k \times M + B, k \in N \} \). Thus, by reading or writing data into certain memory locations in its address space, a process could cause data belonging to another process to be evicted from the cache. Symmetric cryptographic algorithms, which are targeted in this paper, iterate a set of operations (collectively termed round operation) a fixed number of times, say \( R \), over the input plaintext, in order to produce the ciphertext. The proposed cryptanalysis framework requires intermediate values generated in a round. In our method, a process interrupts the symmetric algorithm execution at the end of each round (using the system call interface), flushes the data cache, and records all the variables appearing on the bus. In order to map variables recorded on the memory bus to intermediate variables in the corresponding round, a brute-force enumeration is done. This is feasible due to two reasons: the number of values recorded for each round is not very large, and the SAT solver rejects wrong values by classifying them as unsatisfiable constraints (explained later).

Fig. 2 shows the procedure for flushing values out of the data cache. The idea is to initialize static arrays where there is a one-to-one correspondence between indices of the arrays and data cache locations such that writing a value into an index of an array evicts the value stored in the corresponding cache location. First, there are as many static arrays as the associativity of the cache (line 1), and each array is of size equal to the number of cache sets (line 2). Then, memory bus monitoring is turned on (line 4), and the arrays are initialized such that the cache sets are successively evicted (lines 6–10). Finally, bus monitoring is turned off after recording all the values on the memory bus (line 11).
In our work, we compiled open-source FIPS-43 [27] compliant software implementations of the DES, 3DES, and AES encryption algorithms (available at: http://www.cr0.net:8040/code/crypto/) on the Xtensa processor, a commercial 32-bit embedded processor [28]. The software implementations of these algorithms were simulated using the Xtensa instruction set simulator (ISS), which models the processor, memory hierarchy, and system bus. The main memory trace was observed to extract values of program intermediate variables, which were then fed into the proposed SAT-based framework (described in Section III). We considered various cache configurations from 4 upto 32 kB. In all cases, were able to obtain sufficient information (some internal variable values) to discover the key using the proposed framework.

III. SAT FRAMEWORK FOR ENABLING SIDE-CHANNEL ATTACKS

In this section, we present details of our proposed SAT-based cryptanalysis framework. We end the section by showing the results of clause generation for three algorithms, DES, 3DES, and AES. However, it should be noted that our technique is general and can be applied to any cryptographic algorithm.

We wish to represent the functionality of the cryptographic algorithm being targeted as an equivalent Boolean formula in conjunctive normal form (CNF), apply constraints corresponding to the observations, i.e., plaintext, ciphertext, and internal (or intermediate) variables, produced by the secret key, to the formula, and finally, compute the secret key by using an SAT solver to solve the resulting formula. Consider an implementation of a cryptographic algorithm having a side-channel that leaks values of intermediate variables (Fig. 3). For $i \in \{1, 2, \ldots, n\}$, plaintext $P_i$ is mapped to ciphertext $C_i$ for the secret key $K$. $\{V_{i+1}, V_{i+2}, \ldots, V_{i+k}\}$ (collectively denoted by $\{V_i\}$) represent the values of $i$ intermediate variables leaked when the implementation transforms $P_i$ to $C_i$.

Let $\Psi(P, C, K, V_0, V_1, V_2, \ldots, V_m)$ be the Boolean formula of the cryptographic algorithm, where $P$, $C$, $K$, and $\{V_0, V_1, V_2, \ldots, V_m\}$ represent literals corresponding to plaintext, ciphertext, secret key, and all the $m$ internal variables, respectively. Given $n$ known plaintext/ciphertext pairs $(P_1, C_1), (P_2, C_2), \ldots, (P_n, C_n)$, and intermediate variable values leaked by the side-channel for each pair, $\{V_1', \{V_2', \ldots, \{V_m'\}\}\}$ for the same secret key $K$, we generate the formula, $\Psi(P_1, C_1, K, \{V_1\}, \{V_2\}, \ldots, \{V_m\})$ and $\Psi(P_2, C_2, K, \{V_1\}, \{V_2\}, \ldots, \{V_m\})$ and $\ldots$ and $\Psi(P_n, C_n, K, \{V_1\}, \{V_2\}, \ldots, \{V_m\})$ ($\{V_i'\}$ represents the set of intermediate variables other than $\{V_i\}$ which remains unassigned). It is worth mentioning that all the constraints (plaintext, ciphertext, and the intermediate values) in the formula are with respect to the same secret key $K$. The concatenated formula is input to an SAT solver which terminates with one of the following outputs: satisfiable (and, outputting the value of secret key $K$), unsatisfiable (indicating an error in the input constraints), or timeout (meaning that the input constraints provided are insufficient for deducing the secret key $K$ within reasonable time and memory bounds). Table I summarizes the number of variables and clauses present in the Boolean formulae for three popular cryptographic algorithms: DES, 3DES, and AES (128-bit key) [29]. We see the number of clauses needed for AES is an order of magnitude greater than that required for DES, and five times more than the number of clauses needed for 3DES. This is primarily due to the higher complexity of AES round operations compared to DES. Further information on the encoding of these algorithms can be found in [30].

IV. EXPERIMENTAL RESULTS

In this section, we present the details of our experimental setup followed by results of our side-channel attack method for DES, 3DES, and AES. We conclude this section with some observations based on the experimental results. For each of the algorithms, we give the sets of variables whose knowledge enables our framework to find the secret key. We call these sets enabling sets.

A. Experimental Setup

We performed all our experiments on a PC with a 1.6-GHz Pentium processor and a 512–MB RAM running Debian Linux OS. We used the MiniSAT SAT solver from Chalmers University [17], since it has been benchmarked to be one of the best performing publicly available SAT solvers (http://www.satlive.org). However, similar results were also obtained using other state-of-the-art solvers such as zChaff [16]. Given a cryptographic algorithm, we obtain its structural RTL description, and convert that to CNF. The values of intermediate variables obtained from memory bus monitoring are plugged into the CNF description along with the plaintext and ciphertext values, and this modified CNF is input to the MiniSAT solver. Fig. 4 illustrates the experimental setup.

B. Cryptanalysis of DES, 3DES, and AES

In this section, we present the results of cryptanalyzing DES, 3DES, and AES using the proposed method. As part of the results, we enumerate the minimum set of intermediate values required by the proposed technique for each algorithm in order to derive the secret key. The results are as follows.

- **DES: DES algorithm transforms a 64-bit plaintext into a 64-bit ciphertext by iterating it through 16 rounds, parameterized on a 56-bit key. Every round $i$ takes a 64-bit input which is split into two 32-bit inputs, $L_i$ and $R_i$, i.e., $\{L_i, R_i\}$ is the plaintext and $\{L_{i+1}, R_{i+1}\}$ is the ciphertext. The enabling sets for DES are $\{L_i, R_{i+1}, R_{i+2}\}$, $\{L_i, L_{i+1}, R_{i+2}\}$, and $\{L_i, R_i, L_{i+4}, R_{i+4}\}$, where $1 \leq i \leq 13$. A minimum of two plaintext-ciphertext
pairs (and the corresponding intermediate values) are required to obtain the secret key of DES.

- **3DES**: 3DES consists of three copies of DES, each parameterized on a different key, placed end to end. Thus, 3DES comprises 48 rounds which map a 64-bit plaintext to a 64-bit ciphertext, parameterized on a 168-bit key. An enabling set for 3DES is a union of the enabling sets for each copy of DES. A minimum of four plaintext-ciphertext pairs (along with the corresponding enabling sets) are required for breaking 3DES.

- **AES**: AES encrypts a 128-bit plaintext to a 128-bit ciphertext by iterating it through ten rounds, parameterized on a 128-bit key. An enabling set for AES consists of 128-bit input and output of any of the rounds. This pair is sufficient for breaking the 128-bit key of AES.

### C. Observations

Prior to our work, it was observed that hardness of the DES CNF increases abruptly beyond three rounds for SAT solvers, and becomes unsolvable for four rounds and higher (irrespective of the number of plaintext and ciphertext values encoded into the formula) [18], [31]. However, our experimental results demonstrate that a four-round DES can be solved by an SAT solver when only two plaintext and the corresponding ciphertext values are encoded into the CNF. The SAT solver aborts for five- (and higher) round DES. An interesting question to resolve is whether five-round DES can be solved by making suitable enhancements to the SAT solving algorithms or it represents a provably insurmountable barrier? Researchers in SAT have used a metric called clause-to-variable ratio, as a way of quantifying the hardness of solving 3-CNF instances [32] (3-CNF comprise only three literal clauses). Based on this metric, they have empirically shown the existence of a threshold phenomenon, a ratio of about 4.3 separates under-constrained 3-CNF (which can be proven to be easily satisfiable) from over-constrained 3-CNF (which can be easily proven to be unsatisfiable). The 3-CNF instances lying in the neighborhood of the threshold value, i.e., 4.3, are extremely hard to solve. Table II shows the number of variables, clauses, and the clause to variable ratio for one- to five-round DES coded as 3-CNF. We see that the clause-to-variable ratio hovers around 1.37 in all five cases (the ratio is around this value for all the 16 rounds). At least based on the threshold metric, the five-round DES does not show an abrupt increase in hardness of solving.

### Table II

<table>
<thead>
<tr>
<th>Round</th>
<th>Variables</th>
<th>Clauses</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4766</td>
<td>6560</td>
<td>1.376</td>
</tr>
<tr>
<td>2</td>
<td>8788</td>
<td>12096</td>
<td>1.376</td>
</tr>
<tr>
<td>3</td>
<td>12802</td>
<td>17632</td>
<td>1.377</td>
</tr>
<tr>
<td>4</td>
<td>16816</td>
<td>23168</td>
<td>1.377</td>
</tr>
<tr>
<td>5</td>
<td>20830</td>
<td>28704</td>
<td>1.378</td>
</tr>
</tbody>
</table>

### V. Conclusion

In this paper, we have presented a novel framework for performing side-channel attacks on cryptographic software. We have argued and demonstrated the dangers of software side-channels in compromising secret keys. Also, we have developed an automated SAT-based framework for exploiting the vulnerabilities of the software side-channel. An interesting avenue of research would be to modify SAT solvers to enable them to find secret keys with smaller amounts of side-channel information.

### REFERENCES


High-Speed Recursion Architectures for 
MAP-Based Turbo Decoders
Zhongfeng Wang

Abstract—The maximum a posteriori probability (MAP) algorithm has been widely used in Turbo decoding for its outstanding performance. However, it is very challenging to design high-speed MAP decoders because of inherent recursive computations. This paper presents two novel high-speed recursion architectures for MAP-based Turbo decoders. Algorithmic transformation, approximation, and architectural optimization are incorporated in the proposed designs to reduce the critical path. Simulations show that neither of the proposed designs has observable decoding performance loss compared to the true MAP algorithm when applied in Turbo decoding. Synthesis results show that the proposed Radix-2 recursion architecture can achieve comparable processing speed to that of the state-of-the-art recursion (Radix-4) architecture with significantly lower complexity while the proposed Radix-4 architecture is 32% faster than the best existing design.

Index Terms—Error correction codes, high-speed design, maximum a posteriori probability (MAP) decoder, Turbo code, VLSI.

I. INTRODUCTION

Turbo code [1] invented in 1993, has attracted tremendous attentions in both academics and industry for its outstanding performance, rich applications can be found in wireless and satellite communications [2], [3]. Practical Turbo decoders usually employ serial decoding architectures [4] for area efficiency. Thus, the throughput of a Turbo decoder is highly limited by the clock speed and the maximum number of iterations to be performed. To facilitate iterative decoding, Turbo decoders require soft-input soft-output decoding algorithms, among which the maximum a posteriori probability (MAP) algorithm [5] is widely adopted for its excellent performance.

Due to the recursive computations inherent with the MAP algorithm, the conventional pipelining technique is not applicable for raising the effective processing speed unless one MAP decoder is used to process more than one Turbo code blocks or sub-blocks as discussed in [6]. Among various high-speed recursion architectures in [6]–[10], the designs presented in [7] and [10] are most attractive. In [7], an offset-adding-compared-select (OACS) architecture [8] is adopted to replace the traditional add-compared-select-offset (ACSO) architecture. In addition, the lookup table (LUT) is simplified with only 1-bit output, and the computation of absolute value is avoided through introduction of the reverse difference of two competing path metrics. An approximate 17% speedup over the traditional Radix-2 ASCO architecture was reported. With one-step look-ahead operation, a Radix-4 ACSO architecture can be derived. Practical Radix-4 architectures such as those presented in [9] and [10] always involve approximations in order to achieve higher effective speed-ups. For instance, the following approximation is adopted in [10]:

\[
\text{max} \ast (\text{max} \ast (A, B), \text{max} \ast (C, D)) \\
\approx \text{max} \ast (\text{max}(A, B), \text{max}(C, D))
\]  

(1)

where

\[
\text{max} \ast (A, B) \equiv \text{max}(A, B) + \log \left(1 + e^{-1-\alpha} \right).
\]  

(2)

This Radix-4 architecture can generally improve the processing speed (equals twice of its clock speed) by over 40% over the traditional Radix-2 architecture, and it has de facto the highest processing speed among all existing (MAP decoder) designs found in the literature. However, the hardware will be nearly doubled compared to the traditional ACSO [7] architecture.

The contributions of this paper include the following: 1) an advanced Radix-2 recursion architecture based on algorithmic transformation, approximation and architectural level optimization, which can achieve comparable processing speed as the state-of-the-art Radix-4 design with significantly lower complexity and 2) an improved Radix-4 architecture that is 32% faster than the best existing approach.

This paper is organized as follows. Section II presents an advanced Radix-2 recursion architecture for MAP decoders. Section III studies an improved Radix-4 recursion architecture. Section IV presents bit-error-rate performance comparisons between the original MAP algorithm and various approximations when applied in Turbo decoding. The synthesis results about hardware complexity and processing speed for various architectures are also provided in this section.