Satisfiability-based Framework for Enabling Side-channel Attacks on Cryptographic Software

Nachiketh R. Potlapally[†], Anand Raghunathan[‡], Srivaths Ravi[‡], Niraj K. Jha[†], and Ruby B. Lee[†]
[†] Dept. of Electrical Engineering, Princeton University, Princeton , NJ 08544
[‡]NEC Laboratories America, Princeton, NJ 08544
{*npotlapa*, *jha*, *rblee*}@princeton.edu, {*anand*, *sravi*}@nec-labs.com

Abstract - *Many electronic systems contain implementations of cryptographic algorithms in order to provide security. It is well known that cryptographic algorithms, irrespective of their theoretical strength, can be broken through weaknesses in their implementation. In particular, side-channel attacks, which exploit unintended information leakage from the implementation, have been established as a powerful way of attacking cryptographic systems. All side-channel attacks can be viewed as consisting of two phases — an observation phase, wherein information is gathered from the target system, and an analysis or deduction phase in which the collected information is used to infer the cryptographic key. Thus far, most side-channel attacks have focused on extracting information that directly reveals the key, or variables from which the key can be easily deduced.*

We propose a new framework for performing side-channel attacks by formulating the analysis phase as a search problem that can be solved using modern Boolean analysis techniques such as satisfiability solvers. 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 "simply" related to the key). For example, software implementations take great care in protecting secret keys through the use of on-chip key generation and storage. However, they may inadvertently expose the values of intermediate variables in their computations. We demonstrate how to perform side-channel attacks on software implementations of cryptographic algorithms based on the use of a satisfiability solver for reasoning about the secret keys from the values of the exposed variables. Our attack technique is automated, and does not require mathematical expertise on the part of the attacker. We demonstrate the merit of the proposed technique by successfully applying it to two popular cryptographic algorithms, DES and 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 implementation, 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, but 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 encryption algorithm, deriving the n-bit key without 2^n operations). Traditionally, cryptanalysis has focused on just the mathematical function underlying the system, *e.g.*, by analyzing statistical properties of the outputs under the application of targeted inputs [1], [2]. However, many of these attacks are infeasible in practice due to the large amount of data required to implement them.

More recently, a powerful class of attacks called *side-channel attacks* has emerged, which exploits information from the implementation to substantially reduce the complexity of performing cryptanalysis [3]–[7]. Side-channel attacks can be viewed as consisting of two phases — an observation phase, wherein information is gathered by monitoring a 'side-channel' in the target system, and an analysis or deduction phase in which the collected information is used to infer the cryptographic key. Information leakage through a side-channel is an inadvertent by-product of the implementation process. Examples of side-channel information used in successful attacks are operation timing [3], power dissipation [4], [5], electromagnetic radiation [6], and behavior in the presence of induced faults [7]. Surprisingly small amounts of leaked information are sufficient to break the secret key [8]. A wide range of design techniques have been proposed to counter side-channel attacks [9]–[11]. Even with the use of such techniques, the presence of side-channels can be minimized, but it is very difficult to completely eliminate them [12].

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.* [13]. This problem of data exposure exists even in secure software implementations [14]. Recent studies have revealed the possibility of data exposure from software computations even after the computation is over [15]. 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 "simply" 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 [16]. 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 work, 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 a 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.

Acknowledgment: This work was supported by NSF under Grant No. CCR-0326372.

3-9810801-0-6/DATE06 © 2006 EDAA

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 [17], [18]. 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 a SAT solver as an automated reasoning engine in our proposed framework for enabling side-channel attacks.

A. Paper contributions

The contributions of this paper include:

- A general framework for enabling side-channel attacks by formulating the analysis of side-channel information as a search problem that can be solved using SAT solvers. This approach is fully automatic and obviates the need for mathematical expertise on the part of the attacker.
- Demonstration that a large subset of the internal variables in a cryptographic algorithm, not just the key or variables that are directly related to it, can be used to launch successful attacks.
- Application of the proposed framework to perform software side-channel attacks on the popular symmetric encryption algorithms DES and 3DES.
- Characterization of the minimal subsets of internal variables that are sufficient to break DES and 3DES given current state-of-the-art SAT solvers.

Previous work in the area of side-channel attacks has identified various side-channels, and proposed specific *ad hoc* techniques to exploit the information derived from each of them. However, due to the nature of the collected information, the analysis phase has mostly been quite simple. 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 dictate parts of the implementation that should be protected.

To the best of our knowledge, this is the first attempt to apply Boolean analysis techniques to side-channel attacks.

B. Related work

Differential and linear cryptanalysis are two well-known mathematical cryptanalytic techniques. However, they require huge amounts of input, thereby making them prohibitively expensive. For example, to break the 16-round DES, differential and linear cryptanalysis require 2^{47} chosen and known plaintexts, respectively (encrypted with the key to be computed) [1], [2]. Development of side-channel attacks enabled practical cryptanalysis of a number of popular cryptographic algorithms, *e.g.*, DES, RSA, DSS, *etc.* [3], [5], [7]. Kelsey *et al.* [8] proved the power of side-channel attacks by demonstrating that a minimal amount of side-channel information is required for breaking some popular cryptographic algorithms. Schaumuller-Bichl [19] introduced the method of formal coding in which XOR sum-of-product expressions are formulated for the DES output bits in terms of the plaintext and key bits. For a known plaintext and ciphertext pair, the equations are solved to get the key bits. However, the high complexity of the resulting equations limited the attack to being a theoretical one. Massacci and Marraro [20] proposed modeling of DES as a 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. Our work differs from theirs in terms of recognizing and demonstrating the effectiveness of SAT as a tool for enabling side-channel attacks.

The rest of the 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 and 3DES 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 and 3DES which enable successful inferring of the key. We conclude in Section V with our observations and directions for future work.

II. Exposure of Software Data

In this section, we enumerate the various ways in which side-channels that leak the values of software variables exist. We conclude the section by illustrating the manner in which we obtained the values of internal variables to cryptanalyze DES and 3DES in our experiments.

Application programs use routines provided by system libraries to implement common functionality. Through system calls, application code and system libraries interact with the operating system (OS) kernel which manages the hardware. The complexities of implementing hardware and software systems leave opportunities for data leakage at the various interfaces: application-library, application-OS, library-OS, and OS-hardware. There are two ways in which opportunities for data exposure at these interfaces occur:

- It can happen inadvertently during normal operation due to bugs, improper policies, misconfiguration, *etc*. Chow *et al.* [15] showed the existence of program data in system buffers long after the program terminated. Garfinkel and Shelat [21] showed the persistence of data on magnetic disks, and ways to extract it. Thus, swapping of processes greatly increases the chance of data exposure. Core dumps of programs are also a valuable source of program data [22].
- It can occur due to malicious hardware or software attacks that exploit system vulnerabilities, *e.g.*, hacking of the run-time stack [23], probing the cache [24], monitoring the memory traffic on the system bus [25], *etc*. Also, there exist tools for examining the contents of program memory as the program is being executed [26]. They can be used to gather the values of the required intermediate variables.

Even secure software implementations have been observed to have data exposure problems [27].

In this work, as an illustrative example, we evaluated a bus probing attack on an embedded software implementation of cryptographic algorithm. However, our technique is not restricted to any specific software side channel — any mechanism that reveals the values of some program intermediate variables can be used.

We compiled open-source FIPS-43 [28] compliant software implementations of the DES and 3DES encryption algorithms (available at: `http://www.cr0.net:8040/code/crypto/`) on the Xtensa processor, a commercial 32-bit embedded processor [29]. 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 the following section). We considered various cache configurations from 4KB upto 32KB. In all cases, we 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 begin by giving an overview of the framework, and briefly describe its constituent steps. Later, the method for representing a cryptographic algorithm as a Boolean formula is described. We use the popular DES and 3DES algorithms for illustrating this formulation process, and to discuss the results of our experiments. However, it should be noted that our technique is general and can be applied to any cryptographic algorithm.

A. Overview of the proposed framework

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 a SAT solver to solve the resulting formula. Consider a DES implementation having a side-channel that leaks values of intermediate values. For $i \in \{1,2,..,n\}$, plaintext P_i is mapped to ciphertext C_i for the secret key K. $\{V_i^{j+1}, V_i^{j+2}, \ldots, V_i^{j+k}\}$ (collectively denoted by $\{V_i^j\}$) represent the values of k intermediate variables leaked when the implementation transforms P_i to C_i.

Fig. 1 illustrates the operational flow of the proposed SAT-based framework. The first step is to obtain the Boolean formulation of the algorithm (details are presented in Section III-B). Let $\Psi(P,C,K,V^1,V^2,\ldots,V^m)$ represent the Boolean formula of the cryptographic algorithm (in our case, DES) where P, C, K and $\{V^1,V^2,\ldots,V^m\}$ represent literals corresponding to plaintext, ciphertext, secret key, and all the m internal variables, respectively.

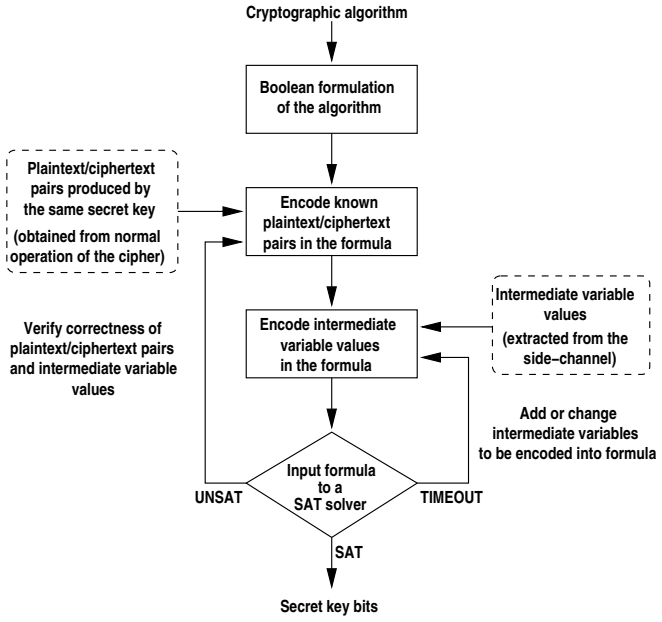


Fig. 1. High-level view of the proposed SAT-based framework

We constrain the formula based on known plaintext/ciphertext values, *i.e.*, by setting the plaintext and ciphertext literals in the formula to their observed values (*i.e.*, $(P_1,C_1),(P_2,C_2),\ldots,(P_n,C_n)$). This is done by concatenating multiple CNF formulae where each one is constrained on one known plaintext/ciphertext pair, *i.e.*, $\Psi(P_1,C_1,K,V^1,V^2,\ldots,V^m) \wedge \Psi(P_2,C_2,K,V^1,V^2,\ldots,V^m) \wedge \ldots \wedge \Psi(P_n,C_n,K,V^1,V^2,\ldots,V^m)$ $(n \geq 1)$. The next step exploits the side-channel information collected using techniques described in Section II. Here, we further constrain the formula based on the intermediate variable values observed from the side-channel (*i.e.*, $\{V_1^j\},\{V_2^j\},\ldots,\{V_n^j\}$ where set $\{V_i^j\}$ represents the values of the intermediate variables of the algorithm observed for pair (P_i,C_i)). This is represented in the formula as, $\Psi(P_1,C_1,K,\{V_1^j\},\{V_1^j\}^c) \wedge \Psi(P_2,C_2,K,\{V_2^j\},\{V_2^j\}^c,) \wedge \ldots \wedge \Psi(P_n,C_n,K,\{V_n^j\},\{V_n^j\}^c)$ ($\{V_n^j\}^c$ represents the set of intermediate variables other than $\{V_n^j\}$ which remains unassigned). It is worth mentioning that all the constraints in the formula are with respect to the same secret key, K. Note that the encoding shown in the figure assumes that a block cipher is used. For other modes (chaining or feedback modes), the feedback in the algorithm is represented by constraining the values of appropriate variables (*e.g.*, initialization vector (IV)) in adjacent copies to be the same. The resulting Boolean formula is given as an input to the SAT solver. The SAT solver can terminate its search for a literal assignment with one of the following outcomes:

- **SAT**: A satisfying assignment is found. The key value, K, can be output by identifying the values assigned to literals corresponding to the key bits in the assignment.
- **UNSAT**: There is no satisfying assignment for the Boolean formula. Assuming the Boolean formulation is correct, this implies an error in the values of the plaintext-ciphertext or the intermediate variables encoded in the formula. Hence, we backtrack, re-encode the formula with correct values, and repeat the search for an assignment.
- **TIMEOUT**: The solver is unable to find either a satisfying assignment or prove no such assignment exists for the formula within a reasonable time or memory, and therefore aborts. The time and memory limits in our experiments were usually on the order of 2000 seconds and 2 GB, respectively. In this case, an iterative loop of modifying the side-channel information (either adding more variables to or replacing variables in the set of intermediate variables whose values were used as side-channel information) can be used until the solver gives a deterministic output (SAT or UNSAT) or an upper limit on the number of loop iterations is reached.

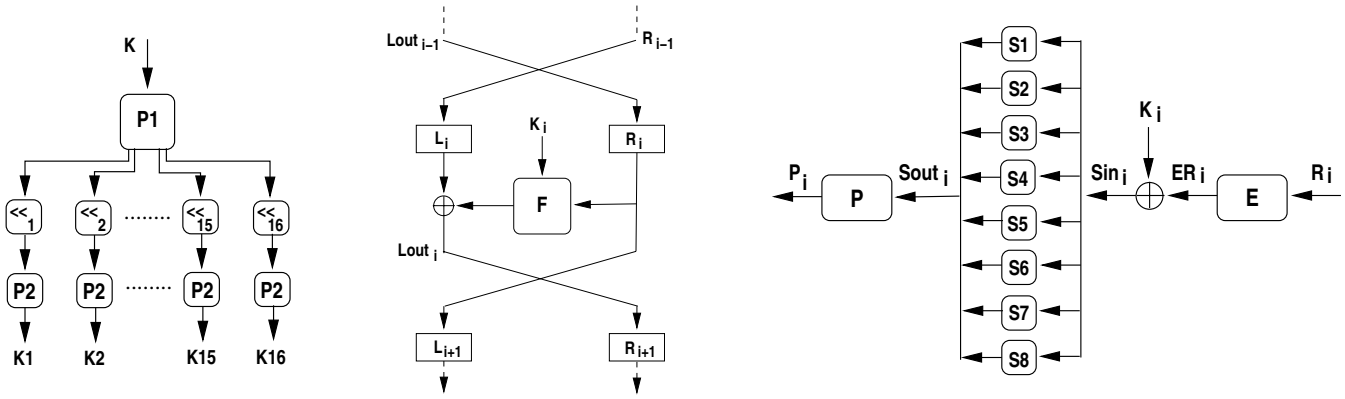
B. Boolean formulation of a cryptographic algorithm

In this paper, we limit our investigation to symmetric algorithms [30]. Encryption/decryption in symmetric algorithms consist of multiple iterations of a round transformation, each of which is parameterized on a different key (termed *round key*). Round key generation (also known as *key expansion*) refers to the process of generating round keys from the secret key. Thus, the operation of a symmetric algorithm is divided into two parts: round key generation and encryption/decryption process. Therefore, the Boolean formula of a cryptographic algorithm should include both key generation and encryption/decryption operations. We demonstrate this using DES and 3DES. Since 3DES is a simple extension of DES, we focus on the latter.

DES takes a 64-bit plaintext and a 64-bit secret key to produce a 64-bit ciphertext. Fig. 2(a) shows the round key generation operation. The bits of the 64-bit secret key, K, are permuted using a permutation function, $P1$. $P1$ also removes the 8 parity bits (located at bit positions 8, 16, 24, 32, 40, 48, 56 and 64), leaving a 56-bit output. The 56-bit value is rotated by a fixed offset, and passed through another permutation function, $P2$, to produce a round key. The rotate offset is different for each round (denoted by $<<_1$, $<<_2$, ..., $<<_{16}$ in Fig. 2(a)). $P2$ chooses 48 bits at pre-determined bit indices from the rotated 56-bit value, and permutes them. Thus, 16 distinct 48-bit round keys are generated from the 64-bit secret key.

Encryption in DES is done by iterating the plaintext 16 times through a round transformation where a distinct round key is used for each round. Fig. 2(b) shows the operations of a DES round transformation. Input to round i is split into two 32-bit halves: left half (L_i) and right half (R_i). R_i is transformed by the F function whose other input is the round key, K_i. The output of the F function is XORed with L_i to produce a 32-bit output, $Lout_i$. $Lout_i$ and R_i become the right (R_{i+1}) and left (L_{i+1}) halves of the next round, respectively. Fig. 2(c) expands the F function (in round i) into its constituent operations. The 32-bit right half, R_i, is expanded into a 48-bit value, ER_i, by passing it through the expansion permutation, E. ER_i is XORed with the 48-bit round key, K_i, to produce Sin_i. The 48-bit value, Sin_i, is split into eight six-bit vectors which are input to eight distinct S-boxes ($S1$, $S2$,.., $S8$). An S-box performs a table-lookup with pre-computed values and takes a six-bit input to produce a four-bit output. The four-bit outputs of the eight S-boxes are combined to form a 32-bit value, $Sout_i$. The bits of $Sout_i$ are permuted by permutation, P, to produce P_i, the 32-bit output of the F function.

Obtaining the Boolean formulation for DES round key generation is straightforward. The bits of the 64-bit secret key are permuted and rotated to obtain the round keys. Therefore, for each round, we pre-compute the bit indices (of the secret key) which form the corresponding round key. For example, the round key for round



(a) (b) (c)

Fig. 2. Functional view of DES encryption: (a) round key generation, (b) round transformation, and (c) F function used in the round transformation

five is formed by putting the 19th bit of the secret key as the first bit of the round key, 60th bit as the second bit, 43th bit as the third bit, and so on. Thus, based on this pre-computed bit index mapping between the secret key and round keys, the appropriate secret key literals can be used in the round transformation. The Boolean formulation of DES encryption requires us to deal with three types of logic functions, *i.e.*, XOR, table lookup and permute. Given any logic function, $F(.)$, its corresponding Boolean formula can be derived using the following logical relation:

$$(Z = F(.)) \equiv (Z \rightarrow F(.))(F(.) \rightarrow Z) \quad (1)$$
$$\equiv (\overline{Z} + F(.))(\overline{F(.)} + Z) \quad (2)$$

Assuming $F(.)$ and $\overline{F(.)}$ are in the product-of-sum form, the above expression can be expanded into a Boolean formula using the logic relation, $(a+bc) = (a+b)(a+c)$. The Boolean formulas of the logic functions in a DES round can be obtained as follows:

- **XOR**: The Boolean formula representing the XOR of two vectors can be obtained by the conjunction of the Boolean formula representing the XOR of individual bits. Let $z_i = x_i \oplus y_i$, where x_i and y_i are the ith input bits, and z_i the ith output bit. The Boolean formula, Φ_i, representing this operation can be derived as follows:

$$\begin{aligned}\Phi_i &= (\overline{z_i} + (x_i \oplus y_i))(z_i + \overline{(x_i \oplus y_i)}) \\ &= (\overline{z_i} + (\overline{x_i} + \overline{y_i})(x_i + y_i))(z_i + (\overline{x_i} + y_i)(x_i + \overline{y_i})) \\ &= (\overline{z_i} + \overline{x_i} + \overline{y_i})(\overline{z_i} + x_i + y_i)(z_i + \overline{x_i} + y_i)(z_i + x_i + \overline{y_i})\end{aligned}$$

- **Permutation**: The permutation functions, E and P (Fig. 2(c)), rearrange their input bits at the output (E also duplicates some of the input bits). If the jth input bit x_j is assigned to the ith output bit z_i, then the corresponding Boolean formula is given by $(\overline{z_i} + x_j)(z_i + \overline{x_j})$. We get the Boolean formula for the permutation function by the conjunction of formulae for all the output bits.
- **S-box**: The S-boxes are the only non-linear functions in the DES algorithm. An S-box maps a 6-bit input to a 4-bit output. This enables us to enumerate the behavior of an S-box using a truth table, and use a logic minimizer tool to obtain logic expressions for each of the four outputs in terms of the six inputs. Using the logic expressions, Boolean formulas can be derived for each of the four output bits (using the logic relation described in Equation (1)). The formula for a single output bit comprises 34 clauses. Conjunction of the Boolean formulae of the four output bits gives the formula for the S-box, *i.e.*, 136 clauses.

3DES is computed by using three iterations of the DES algorithm using different keys for each iteration. It consists of DES encryption with key k_1, followed by DES decryption with key k_2, and finally DES encryption with key k_3. Operations in DES decryption are similar to DES encryption except that the order of the round keys is reversed, *i.e.*, we use the 16th round key first, and go down to the first one. Thus, 3DES effectively uses a 192-bit key for encryption. The Boolean formula for 3DES is derived by conjoining Boolean formulae for DES encryption, DES decryption, and DES encryption with different key literals. Table I summarizes the literals and clauses present in the Boolean formulae for three popular cryptographic algorithms. DES, 3DES and AES. We provide results for a 128-bit key AES for comparison purposes.

TABLE I
RESULTS OF THE BOOLEAN FORMULATION

Algorithm	Literals	Clauses
DES	6904	35232
3DES	20328	104928
AES	10240	542432

IV. EXPERIMENTAL RESULTS AND ANALYSIS

In this section, we present the results of our side-channel attack method for DES and 3DES. We studied the efficacy of all the intermediate variables in DES regarding their ability to enable our technique to compute the secret key. Our exhaustive studies show that knowledge of certain sets of intermediate variables allows our technique to successfully determine the secret key. We refer to these sets of variables as *enabling sets*. We present some rules that invariably describe how the enabling sets for the DES algorithm can be formed. Thus, all the enabling sets can be enumerated by iterating through these rules. These rules also hold for 3DES where they are applied separately to its three DES segments. We performed all our experiments on a PC with a 1.6 GHz Pentium processor and a 512MB RAM running Debian Linux OS. We used the MiniSAT SAT solver from Chalmers University [18], 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 [17]

A. Cryptanalysis of DES

We present the rules characterizing the enabling sets of DES with the help of Fig. 3. This figure shows four consecutive rounds in the DES algorithm (indices i, $i+1$, $i+2$, $i+3$, $i+4$ indicate rounds). The variables in an enabling set are encapsulated within this four-round DES structure wherever this structure might occur within the 16 rounds of the DES algorithm, *i.e.*, $i \in \{1,2,..,13\}$. The rules are enumerated below:

1) **Forward L-L path**: A lower-round L value separated from a higher-round one by a single XOR operation, and the R value adjacent to the lower-round L form an enabling set. In Fig. 3, $\{L_i, L_{i+2}, R_i\}$ forms an enabling set according to this rule.
2) **Forward R-L path**: A lower-round R value separated from a higher-round L by a single XOR operation, and the L value

adjacent to the lower-round R form an enabling set. By this rule, $\{R_i, L_{i+3}, L_i\}$ is an enabling set.

3) **Reverse R-L path**: A higher-round R value separated from a lower-round L value by a single XOR operation, and the L value adjacent to the R form an enabling set. $\{R_{i+4}, L_{i+3}, L_{i+4}\}$ becomes an enabling set according to this rule.
4) **Reverse L-L path**: A higher-round L value separated from a lower-round L by a single XOR operation, and the R value adjacent to the higher-round L form an enabling set. Based on this rule, $\{L_{i+4}, L_{i+2}, R_{i+4}\}$ becomes an enabling set.
5) **Two-XOR path**: A lower-round L separated from a higher-round L by two XOR operations, and the R values adjacent to these L values form an enabling set. Therefore, $\{L_i, R_i, L_{i+4}, R_{i+4}\}$ becomes an enabling set.

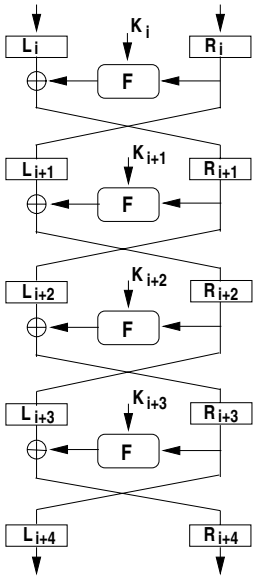


Fig. 3. DES structure illustrating invariant rules

A minimum of three intermediate variables (rules 1-4) and a maximum of four (rule 5) are needed to compute the DES secret key. Special cases arise when index i is 1 or 13. When i is 1, L_1 and R_1 (analogs of L_i and R_i in Fig. 3) are the left and right half of the plaintext which is already provided. Thus, according to rules 1 and 2, we need the knowledge of *only one* intermediate variable, either L_3 or L_4, to compute the secret key. To apply rule 5, we require two intermediate variables, L_5 and R_5, to extract the secret key. Similarly, when i is 13, L_{17} and R_{17} (analogs of R_{i+4} and L_{i+4} in Fig. 3 since there is no crossover of L and R values in the last round) are the two halves of the ciphertext which is known. According to rules 3 and 4, knowledge of either L_{16} or L_{15} is required to extract the secret key. To apply rule 5, values of both L_{13} and R_{13} are required. Thus, we see that in special cases, *i.e.*, $i \in \{1, 13\}$, the minimum number of intermediate variables required to compute the DES key reduces to one.

For time efficiency, multiple plaintexts and their corresponding ciphertexts produced by the same secret key can be encoded into the Boolean formula. In some cases, this extra information increases the power of the SAT solving process. Time taken to compute the secret key as a function of the number of plaintext/ciphertext pair values encoded into the Boolean formula for rule 3 (L_{16}) and rule 5 ($\{L_5, L_6\}$) enabling sets is shown in Figs. 4 and 5, respectively. Along with the plaintext/ciphertext pair values, the corresponding values of variables $\{L_{16}\}$ and $\{L_5, L_6\}$ are also encoded. Fig. 4 shows the time taken to compute the secret key using 2, 4, 8, 16 and 32 plaintext/ciphertext pairs when set $\{L_{16}\}$ is encoded. The SAT solver times out when values of only one plaintext/ciphertext pair is provided. Here, we can see that our technique can compute the secret key within two seconds when provided with two pairs of values, and the time taken increases as more pairs are provided. This observation is true across enabling sets found by rules 1, 2, 3 and 4. Similarly, Fig. 5 shows the average trend of time taken to compute the secret key using 1, 2, 4, 8, 16 and 32 plaintext/ciphertext pairs when set $\{L_5, L_6\}$ is encoded. Here, the time taken decreases as the number of pairs encoded increases from one to four, and then increases. In general, this observation holds for enabling sets obtained by rule 5.

Fig. 4. Time to compute the secret key with the value of variable L_{16}

There are ways in which new enabling sets can be formed by replacing variables in an enabling set by equivalent ones. Consider the enabling set produced by rule 1, $\{L_i, L_{i+2}, R_i\}$. From Fig. 3 we see that variable L_{i+2} is the same as variable R_{i+1}. Thus, $\{L_i, R_{i+1}, R_i\}$ is also an enabling set. Similarly, a rule 5 enabling set $\{L_i, R_i, L_{i+4}, R_{i+4}\}$ is equivalent to $\{L_i, L_{i+1}, L_{i+4}, L_{i+5}\}$. With respect to enabling the SAT solver to solve a Boolean formula, providing the value of variable R_{i+1} in Fig. 3 is equivalent to providing the values of variables P_i, $Sout_i$ or Sin_i in the F function (Fig. 2(b)) of round i. This can be easily explained. For example, consider the enabling set, $\{L_i, R_{i+i}, R_i\}$ (which is equivalent to the rule 1 enabling set, $\{L_i, L_{i+2}, R_i\}$). Values of L_i and R_{i+1} enable the SAT solver to find the output of the F function by a simple backward implication through the round XOR operation. This derived value can be further back-propagated through the F function until the output of the XOR operation (inside the F function) is computed. One input to this XOR can be easily derived from the forward propagation of R_i, and the other input is the round key, K_i. Thus, a simple implication will reveal the bits of the round key. By providing values of P_i, Sin_i or $Sout_i$ instead of R_{i+1}, we are directly providing the values of the output of the F function, and obviating the need for the first back implication through the round XOR. Therefore, variable sets $\{L_i, P_i, R_i\}$, $\{L_i, Sout_i, R_i\}$ and $\{L_i, Sin_i, R_i\}$ become enabling. In this manner, the collection of enabling sets can be increased.

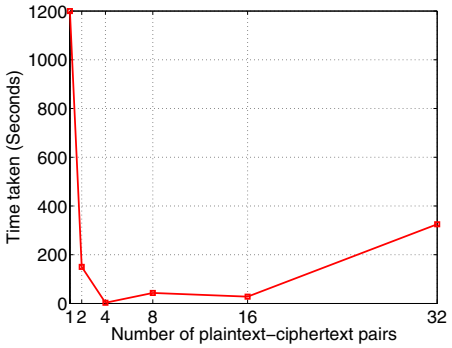


Fig. 5. Time taken to compute the secret key with the values of variables L_5 and L_6

B. Cryptanalysis of 3DES

3DES is made up of 48 DES rounds, where rounds 1 to 16 implement DES encryption with key k_1, rounds 17 to 32 implement DES decryption with key k_2, and rounds 33 to 48 implement DES

encryption with key k_3. Effectively, 3DES has a 192-bit secret key comprising three 64-bit keys, k_1, k_2 and k_3. The enabling sets for DES, which are described above, can be extended to 3DES. We get an enabling set for 3DES by combining the enabling sets of each of its three constituent DES segments. For the sake of clarity, we separate the results for 3DES into two classes below - 'rule 5 3DES enabling sets,' and 'non rule 5 3DES enabling sets':

- **Rule 5 3DES enabling sets**: These are obtained by combining rule 5 enabling sets of the three DES segments, *i.e.*, analogs of $\{L_i, R_i, L_{i+4}, R_{i+4}\}$ (Fig. 3) in the three segments. For example, we can form a rule 5 3DES enabling set by putting together the following three DES enabling sets, $\{L_9, R_9, L_{13}, R_{13}\}$ (for the first DES segment), $\{L_{25}, R_{25}, L_{29}, R_{29}\}$ (for the second DES segment), and $\{L_{41}, R_{41}, L_{45}, R_{45}\}$ (for the third DES segment). An interesting example is the one obtained by combining $\{L_5, R_5\}$ (for the first segment), $\{L_{21}, R_{21}\}$ (for the second segment) and $\{L_{45}, R_{45}\}$ (for the third segment). Here, since L_1 and R_1 constitute the plaintext, it is sufficient to provide values of only L_5 and R_5 for finding the 64-bit secret key of the first segment. Similarly, since L_{49} and R_{49} constitute the ciphertext, values of L_{45} and R_{45} are enough to compute the 64-bit key of the third segment. After finding the key of the first segment, the SAT solver can compute the outputs of this segment, L_{17} and R_{17} (which are also the inputs to the second segment), through forward implications. Thus, it is sufficient to provide only the values of L_{21} and R_{21} to compute the 64-bit key of the second segment (since set $\{L_{17}, R_{17}, L_{21}, R_{21}\}$ forms a rule 5 enabling set). Likewise, providing the values of set $\{L_{29}, R_{29}\}$ instead of $\{L_{21}, R_{21}\}$ gives the same result. In this case, after the SAT solver computes the key of the third segment, it performs backward implications from the output of the third segment (which is the ciphertext) to find the inputs of this segment, L_{33} and R_{33} (which are also the outputs of the second segment). As in the previous case, $\{L_{29}, R_{29}, L_{33}, R_{33}\}$ forms a rule 5 enabling set. For rule 5 enabling sets, our technique could derive the 192-bit 3DES key in 750 seconds on average (using four plaintext/ciphertext pairs).
- **Non rule 5 3DES enabling sets**: These are formed by combining enabling sets obtained by rules 1, 2, 3 and 4 for each of the three DES segments of 3DES. An interesting example of this enabling set is obtained by combining rule 1 enabling sets for the first and second segments, and rule 3 enabling set for the third segment: $\{L_3\}$ (L_1 and R_1 are plaintext), $\{L_{17}, L_{19}, R_{17}\}$, and $\{L_{48}\}$ (L_{49} and R_{49} are ciphertext). Likewise, 3DES enabling sets can be obtained by various combinations of enabling sets for the three DES segments. For non rule 5 enabling sets, our technique could derive the 192-bit 3DES key in 1165 seconds on average (using four plaintext/ciphertext pairs).

V. Conclusions

In this work, 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. However, the SAT solver has some limitations, *e.g.*, it cannot break the key in cases where the exposed variables are separated by more than four DES rounds. Based on our experience, we foresee scope for further research along two directions: a thorough analysis of the nature of software side-channels and their prevention, and improving SAT solving techniques with the aim of enhancing their cryptanalysis capabilities.

References

[1] E. Biham and A. Shamir, "Differential cryptanalysis of the full 16-round DES," in *Proc. Advances in Cryptology (CRYPTO '92)*, pp. 487–496, Aug. 1992.

[2] M. Matsui, "Linear cryptanalysis method for DES cipher," in *Proc. Advances in Cryptology (CRYPTO '94)*, pp. 386–397, Aug. 1994.

[3] P. Kocher, "Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems," in *Proc. Advances in Cryptology (CRYPTO '96)*, pp. 104–113, Aug. 1996.

[4] P. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," in *Proc. Advances in Cryptology (CRYPTO '99)*, pp. 388–397, Aug. 1999.

[5] T. Messerges, E. A. Dabbish, and R. H. Sloan, "Examining smart-card security under the threat of power analysis attacks," *IEEE Trans. Computers*, vol. 51, pp. 541–552, May 2002.

[6] W. van Eck, "Electromagnetic radiation from video display units: An eavesdropping risk," *Computers and Security*, vol. 4, pp. 269–288, 1985.

[7] D. Boneh, R. A. Demillo, and R. J. Lipton, "On the importance of checking cryptographic protocols for faults," in *Proc. Advances in Cryptology (Eurocrypt '97)*, pp. 37–51, May 1997.

[8] J. Kelsey, B. Schneier, D. Wagner, and C. Hall, "Side channel cryptanalysis of product ciphers," *J. Computer Security*, vol. 8, no. 2, pp. 141–158, 2000.

[9] L. Benini, A. Macii, E. Macii, E. Omerbegovic, M. Poncino, and F. Pro, "A novel architecture for power maskable arithmetic units," in *Proc. Great Lakes Symp. VLSI*, pp. 136–140, Apr. 2003.

[10] H. Saputra, N. Vijaykrishnan, M. Kandemir, M. J. Irwin, R. Brooks, S. Kim, and W. Zhang, "Masking the energy behavior of DES encryption," in *Proc. Design Automation & Test in Europe Conf.*, pp. 10084–10089, Mar. 2003.

[11] K. Tiri and I. Verbauwhede, "Securing encryption algorithms against DPA at the logic level: Next generation smart card technology," in *Proc. Cryptographic Hardware and Embedded Systems*, pp. 125–136, 2003.

[12] J. S. Coron, D. Naccache, and P. Kocher, "Statistics and information leakage," *ACM Trans. Embedded Comput. Systems*, vol. 3, pp. 492–508, Aug. 2004.

[13] J. Viega, *Protecting Sensitive Data in Memory*. http://www-106.ibm.com/developerworks/security/library/, 2001.

[14] J. Whittaker and H. H. Thompson, "Security bugs exposed: A systematic approach to uncovering software vulnerabilities," *Software Testing and Quality Engineering*, pp. 28–32, Mar. 2003.

[15] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum, "Understanding data lifetime via whole system simulation," in *Proc. USENIX Security Symp.*, pp. 321–336, Aug. 2004.

[16] Safenet Inc., *Better Hardware Security Modules Through Better Design*. http://www.safenetinc.com/library/8/HSM_Design_principles.pdf, 2002.

[17] L. Zhang and S. Malik, "The quest for efficient Boolean satisfiability solvers," in *Proc. Int. Conf. Computer-Aided Verif.*, pp. 17–36, July 2002.

[18] N. Een and N. Sorenson, "An extensible SAT solver," in *Proc. Int. Conf. Theory & Appl. Satisfiability Testing*, May 2003.

[19] I. Schaumuller-Bichl, "Cryptanalysis of the data encryption standard by the method of formal coding," in *Proc. Advances in Cryptology (Eurocrypt '82)*, pp. 235–255, May 1982.

[20] F. Massacci and L. Marraro, "Logical cryptanalysis as a SAT problem," *J. Automated Reasoning*, vol. 24, pp. 165–203, Feb. 2000.

[21] S. Garfinkel and A. Shelat, "Remembrance of data passed," *IEEE Security and Privacy*, vol. 1, pp. 17–27, Feb. 2003.

[22] P. Broadwell, M. Harren, and N. Sastry, "Scrash: A system for generating secure crash information," in *Proc. USENIX Security Symp.*, Aug. 2003.

[23] V. Paretsky, *The Role of Hardware in Exposing Security Breaches*. http://www.ddj.com/print, 2005.

[24] C. Percival, *Cache Missing for Fun and Profit*. http://www.daemonology.net/papers/htt.pdf, 2005.

[25] R. J. Anderson and M. G. Kuhn, "Tamper resistance - A cautionary note," in *Proc. USENIX Wkshp. Electronic Commerce*, 1996.

[26] D. Farmer and W. Venema, *The Coroner's Toolkit*. http://www.porcupine.org/forensics/tct.html, 2005.

[27] J. Whittaker, "Why secure applications are difficult to write," *IEEE Security and Privacy*, vol. 1, pp. 81–83, Mar. 2003.

[28] National Institute of Standards and Technology, *Federal Information Processing Standard 46-3*. http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf, 1999.

[29] *Xtensa Application Specific Microprocessor Solutions - Overview Handbook*. Tensilica Inc. (http://www.tensilica.com), 2001.

[30] B. Schneier, *Applied Cryptography: Protocols, Algorithms and Source Code in C*. John Wiley and Sons, 1996.