DE GRUYTER
OPEN

TATRA
MOUNTAINS
Mathematical Publications

DOI: 10.1515/tmmp-2015-0048
Tatra Mt. Math. Publ. **51** (2012), 187–203

USING SAT SOLVERS IN LARGE SCALE DISTRIBUTED ALGEBRAIC ATTACKS AGAINST LOW ENTROPY KEYS

Viliam Hromada — Ladislav Öllős — Pavol Zajac

ABSTRACT. In this paper we study large scale distributed algebraic attacks with SAT solvers in a specific scenarios. We are interested in the complexity of finding low entropy keys with the help of SAT solvers. Moreover, we examine how to efficiently distribute this process on multiple computing nodes. Finally, we show that the average cost of the attack per key decreases, if the attacker has access to many different encryptions with different keys.

1. Introduction

The algebraic cryptanalysis tries to solve cryptanalytic problems directly through their algebraic representation. One of the basic types of algebraic attacks is just to model an encryption with Boolean formula in conjunctive normal form (CNF). The unknown literals are mapped to internal state of the cipher during the encryption process, inputs, outputs and potential key bits. Then a SAT solver is applied to the CNF. The attacker can extract the key bits from the positive proof found by the solver (if it exists). Although the SAT problem is hard in general, there are some instances (see, e.g., [5]) when algebraic cryptanalysis using SAT solvers was found to be more efficient than brute force attacks.

In our contribution we focus on three particular practical issues arising when using SAT solvers (and other algebraic cryptanalytic tools in general). In the first part, we remark on the improvements we can get when keys are not taken

2010 Mathematics Subject Classification: 94A60, 68P25.
Keywords: algebraic cryptanalysis, SAT solvers, DES, distributed computing.
This work was supported by project VEGA 1/0173/13 and project Cryptography brings security and freedom SK06-IV-01-001. This project is co-funded by the EEA Grants and the state budget of the Slovak Republic from the EEA Scholarship Programme Slovakia. Part of the experiments were run on Slovak infrastructure for high performance computing under the project Algebraic cryptanalysis.

from a uniform random distribution, but are "password based". In the second part, we focus on issues that are involved in distributing the algebraic attacks to many computational nodes. In the final part, we remark on attacks in multiple key scenario in which the attacker wants to recover just one out of many keys used for different encryptions.

All experiments are concluded on the simple example of (round reduced) DES. We remark that basic algebraic attacks are typically successful only against already weak ciphers. However, we can apply many of these basic techniques for more advanced scenarios involving algebraic complexity reduction [6], or in combination with other types of attacks [10, 15].

2. Preliminaries

Let $\mathbb{S}$ be a deterministic encryption scheme with key space $\mathcal{K}$, message space $\mathcal{M}$, ciphertext space $\mathcal{C}$. As usual, the scheme is defined by the three algorithms:

- $Gen : \mathbb{Z} \to \mathcal{K}$ is a probabilistic key generation algorithm (with security parameter as input);
- $Enc : \mathcal{K} \times \mathcal{M} \to \mathcal{C}$ is a deterministic encryption algorithm;
- $Dec : \mathcal{K} \times \mathcal{C} \to \mathcal{M}$ is a deterministic decryption algorithm.

We assume that scheme is correct, i.e., for any $k \in \mathcal{K}, m \in \mathcal{M}$, we get

$$Dec\big(k, Enc(k, m)\big) = m.$$

We focus on a basic key recovery attack: Given N plaintext-ciphertext pairs (PC pairs in short) (m_i, c_i), attacker wants to determine the key k, such that $Enc(k, m_i) = c_i$ for $i = 1, 2, \ldots, N$, or to show that no such key exists. The simplest attack involves trying every possible $k \in \mathcal{K}$, and checking conditions $Enc(k, m_i) = c_i$. This attack is called exhaustive search (or a brute-force attack). A probabilistic version of exhaustive search involves a random selection $k \in \mathcal{K}$, instead of a systematic enumeration.

Algebraic cryptanalysis understands the conditions $Enc(k, m_i) = c_i$ as a set of equations in unknown k, and the attacker obtains the key by solving this system. Each equation $Enc(k, m_i) = c_i$ can further be expanded into a system of smaller equations involving internal state of the cipher during the encryption and related by the details of the encryption algorithm. There are many techniques of algebraic cryptanalysis, depending on the representation of the system of equations and tools used to solve this system. In this article we focus on algebraic cryptanalysis with SAT solvers.

We can use a single PC pair to determine the key from $Enc(k, m_1) = c_1$, but if $|\mathcal{K}| > |\mathcal{M}|$ there are more potential keys (so called false keys), for which the $Enc(k, m_1) = c_1$, but which were not actually used to encrypt m_1 to c_1.

Typically, to get rid of false keys we require that $|\mathcal{M}|^N \geq |\mathcal{K}|$. In our experiments we will be working with DES, where $|\mathcal{K}| = 2^{56}$, and $|\mathcal{M}| = 2^{64}$, so a single PC pair is typically enough to exclude false keys. If it is clear from the context, we will omit indexes, and talk just about system $Enc(k, m) = c$.

2.1. SAT solver based algebraic cryptanalysis

Let $x_1, x_2, \ldots, x_n$ represent Boolean variables. Recall that conjunctive normal form (CNF) is a logical form in the form

$$F = \bigwedge_{i=1}^{m} C_i, \quad \text{where} \quad C_i = \bigvee_{j=1}^{k_i} L_{i,j}.$$

Here $L_{i,j}$ are literals that can either be a single Boolean variable, or its negation, i.e., $L_{i,j} = x_p$, or $L_{i,j} = \neg x_p$. Expressions C_i are called clauses. Formula F is satisfiable, if there exists some assignment of Boolean values to variables $x_1, x_2, \ldots, x_n$, so that the whole formula F evaluates to TRUE. Any assignment of variables such that F is satisfied is called a proof of satisfiability for F.

SAT problem is a decisional problem that asks whether given formula F is satisfiable. For CNF formulas this problem (so called CNF-SAT) is believed to be a hard problem. Its complexity depends on the number of variables n (it is easy to see it is no harder than 2^n evaluations of F), the number of clauses m, and the structure of the system. In random regular instances (each $k_i = k$) with low ratio m/n, there are typically many assignments for which each C_i is true, and it can be quickly verified that F is (most probably) satisfied. On the other hand, if the ratio m/n is high, it is almost impossible to satisfy all constraints. In practice, there are fast software tools, so called SAT solvers, that can efficiently solve SAT problems in many practical instances. A SAT solver can either provide a proof of satisfiability (SAT), decide that formula is unsatisfiable (UNSAT), or stop after running out of specified time or system resources (UNDECIDED).

Algebraic cryptanalysis with SAT solvers involves the following steps:

(1) Encode the equation system $Enc(k, m) = c$ in a CNF representation F in such a way that literals in a satisfied CNF formula translates back to a solution of the equation system. Typically we encode each internal state bit as a Boolean variable, and associate logical value TRUE with bit-value 1, and logical value FALSE with bit-value 0.

(2) Predetermine some of the literal values (so called guessing). The number of guessed variables must be balanced against the expected running time of SAT solver.

(3) Try to find a proof that F is satisfiable with SAT solver. If it is, by translating back the logical values to bit-values, we get a solution of the system. Otherwise the guess was incorrect and we try to examine other guesses, either in a systematic way or in a randomized fashion.

There are many types of SAT solvers based on different algorithms and employing various heuristics to speedup the solver. In our experiments we use SAT solvers MiniSat [8] and SharpSat [16]. They are both based on the DPLL algorithm [7]. Although we use the solvers as a blackbox, we need to present some parts of the algorithm to understand some of the experimental results and issues involved in distributed SAT-solver based cryptanalysis.

The DPLL algorithm, and its modern versions, have three principal components:

(1) **Reduction of clauses and literals.** This is accomplished by two rules:
   (a) *Unit propagation*: If some clause contains a single literal, there is only one value assign to the associated variable: TRUE for positive literal or FALSE for a negative literal. Suppose that x_i must be TRUE due to some single literal assignment. Then in other clauses every literal x_i is true, and we can remove this clause as it is already satisfied. On the other hand, if some clause contains literal $\neg x_i$, we can remove this literal from the clause as the clause can only be satisfied by the rest of the literals. If we somehow remove all literals from a clause (we get an empty clause), we know that the original formula cannot be satisfied (we get UNSAT result).
   (b) *Pure literal elimination*: If some literal is found with a single polarity (either only x_i or only $\neg x_i$), we can simultaneously satisfy (and remove from the system) all these clauses by assigning a value positive for the pure literal.

   Both of these reduction techniques can be implemented in a fast way (essentially in a linear time based on the size of the system). However they are not sufficient to find the solution in most relevant cases. We note that if we assign values for key and plaintext bits in the DES system, the rest of the values of other variables can be found just by unit propagation (following the computation by a logical circuit).

(2) **Splitting rule.** Select some variable x_i and group clauses in such a way that we get a formula in the form

$$F = (A \vee x_i) \wedge (B \vee \neg x_i) \wedge R.$$

   Formula F cannot be satisfied if both $A \wedge R$ and $B \wedge R$ cannot be satisfied. We can use this rule to define a recursive search algorithm: We assign x_i value TRUE, then recursively check whether $B \wedge R$ is consistent. If yes, than assignment $x_i = TRUE$ along with the proof for $B \wedge R$ is a proof for F. In not, we must then assign x_i value FALSE and recursively check formula $A \wedge R$. The recursive part of the algorithm can be described by a search tree, with a branch for TRUE and FALSE assignment.

(3) **Heuristics and learning.** In modern solvers, the straightforward search through the recursive assignments is replaced by more sophisticated methods of non-chronological backtracking and clause learning. These techniques exploit the information obtained from pursuing some branch of the search tree to search through the rest of the tree more efficiently. Moreover, sophisticated heuristic algorithms are used to determine which variable is assigned which value when branching is required. We say that the algorithm makes decisions, and the running time is mostly influenced by the total number of decisions the algorithm must take before it terminates.

Due to the tree-search nature of the algorithm, the expected number of decisions and thus running times have log-normal distribution, as confirmed by experiments such as [2] (and our experiments as well, see Section 4).

2.2. Algebraic cryptanalysis of DES

As an object of our experimental research we have chosen the old encryption standard DES [13]. Although DES is considered insecure, suffering from various attacks and short keys, it is still used in some legacy applications. DES is useful in our research as a "benchmark" cipher as there are many results on brute-force attacks and cost of keys [11], as well as many results in algebraic cryptanalysis such as [5, 9]. We stress that most of our results should be cipher agnostic, and we will point out explicitly those results that seem to be connected to weaknesses of DES instead of general.

DES operates on 64-bit blocks with 56-bit long keys. DES has a Feistel structure with 16 rounds. After initial permutation, internal state is split into halves denoted by u_L^i, u_R^i. New state is computed as

$$u_L^{i+1} = u_R^i, \quad u_R^{i+1} = u_L^i \oplus F_{k^i}(u_R^i),$$

where F is a non-linear key-dependent function composed of bit-expansion E, subkey addition, S-box application S and bit permutation P. Subkeys k^i are derived directly from key k by taking selected 48 out of 56 bits in each round.

To convert the DES key recovery to algebraic problem we represent all key bits, input bits, and output bits with unknowns. Moreover, we add new unknowns for each state bit in each round, and for each input v^i and output w^i of S-boxes. We then construct linear equations for inputs and outputs of S-boxes

$$\begin{aligned} E(u_R^i) \oplus k^i \oplus \ v^i \ &= 0, \\ P(w^i) \oplus u_L^i \oplus u_R^{i+1} &= 0. \end{aligned}$$

Finally we write non-linear equations for each S-box

$$w^i = S(v^i).$$

All variables in vectors $u_R^i, u_L^i, v^i, w^i, k^i$ were indexed in global vector space, so we will just write x_i for any variable in the system, regardless of its original meaning (we can go back to original vectors once we know the values of x_i). Each equation was written in a simple symbol form [18], which is a tuple containing index set I and possible values of variables x_i with $i \in I$. An example symbol representation of linear equation $x_1 \oplus x_{74} \oplus x_{96} = 0$ is

x_1	x_{74}	x_{196}
0	0	0
0	1	1
1	0	1
1	1	0

An example symbol representation of the first DES S-box is (we only show the first three of all 64 lines for the sake of brevity):

x_{57}	x_{58}	x_{59}	x_{60}	x_{61}	x_{62}	x_{159}	x_{160}	x_{161}	x_{162}
0	0	0	0	0	0	1	1	1	0
0	0	0	0	0	1	0	0	0	0
0	0	0	0	1	0	0	1	0	0
⋮									

If we assign logical value FALSE to x_i with value 0, and TRUE to x_i with value 1, we can understand symbol as a representation of a logical formula in a disjunctive normal form. This formula is satisfied if any of the assignments is made according to values in a symbol. E.g., the example equation $x_1 \oplus x_{74} \oplus x_{96} = 0$ yields logical formula

$$F_j = (\neg x_1 \wedge \neg x_{74} \wedge \neg x_{196}) \vee (\neg x_1 \wedge x_{74} \wedge x_{196}) \\ \vee\ (x_1 \wedge \neg x_{74} \wedge x_{196})\ \vee (x_1 \wedge x_{74} \wedge \neg x_{196})$$

We can rewrite F_j into a CNF by making a symbol of all combinations missing in the original symbol (false assignments), negating the formula and applying DeMorgan's laws. In the example we would get

$$F_j = \neg\big((\neg x_1 \wedge \neg x_{74} \wedge x_{196})\ \vee (\neg x_1 \wedge x_{74} \wedge \neg x_{196}) \\ \vee (x_1 \wedge \neg x_{74} \wedge \neg x_{196}) \vee (x_1 \wedge x_{74} \wedge x_{196})\big)$$

and finally

$$F_j = (x_1 \vee x_{74} \vee \neg x_{196}) \wedge (x_1 \vee \neg x_{74} \vee x_{196}) \\ \wedge (\neg x_1 \vee x_{74} \vee x_{196}) \wedge (\neg x_1 \vee \neg x_{74} \vee \neg x_{196})$$

The full CNF formula for the whole DES system is a conjunction of F_j's. We can further simplify the formula by some standard logical operations, or leave this simplification to a SAT solver.

3. Algebraic attacks on low entropy keys

A typical security requirement on encryption scheme $\mathbb{S}$ is that keys are generated with uniform random distribution, i.e.,

$$\forall k \in \mathcal{K} : Prob(Gen \to k) = 1/|\mathcal{K}|.$$

In a real world situation might be different, e.g., the key generator might be wrongly implemented or deliberately compromised and produce a skewed key distribution. We focus on a different scenario. Here users are allowed to generate keys, and the software under attack uses passwords directly as keys. It is know that user generated passwords have low entropy and skewed distribution [3]. However, we will not use statistical properties, only restrict the key bytes to specific subset of ASCII encoded characters: lower case letters, upper case letters, decimal digits, and their selected combinations. These restrictions are easy to express as algebraic constraints on unknowns representing key bits using symbols: we enumerate all possible combinations of respective bits, and convert them to SAT form as described in Section 2.2.

DES key consists of eight bytes, but only seven bits of each byte are used. We will use remove most significant bit as it is always zero for selected character encodings. DES key is thus a string $k_1 k_2 \dots k_8$ of eight binary encoded numbers from specific restricted sets of seven-bit values. The restrictions are thus encoded as eight symbols involving seven unknowns. After applying restrictions to character codes used as key bytes we get a restricted key space $\mathcal{K}'$. Any key from outside $\mathcal{K}'$ does not satisfy clauses imposed by character restrictions, and thus whole system is unsatisfiable (expect possibly some rare instance where some key from $\mathcal{K}'$ is equivalent to some key from $\mathcal{K} \setminus \mathcal{K}'$).

In our experiments we focus primarily on three sets of different size:

(1) Set `az`: $k_i \in \{97, 98, \dots, 122\}$,
(2) Set `az09`: $k_i \in \{97, 98, \dots, 122\} \cup \{48, 49, \dots, 57\}$,
(3) Set `azAZ09`: $k_i \in \{97, 98, \dots, 122\} \cup \{65, 66, \dots, 74\} \cup \{48, 49, \dots, 57\}$.

The sizes of sets are

(1) `az`: $26^8 \doteq 2^{37.6}$,
(2) `az09`: $36^8 \doteq 2^{41.4}$,
(3) `azAZ09`: $62^8 \doteq 2^{47.6}$.

In experiments, we expect each key to be chosen only from the respective set with uniformly random probability. We can adapt exhaustive search to this case very easily: we just enumerate keys from $\mathcal{K}'$ instead of all keys from $\mathcal{K}$ (also called a dictionary attack). Thus the complexity of exhaustive search is effectively limited by size of $\mathcal{K}'$. When using algebraic attack, the situation might be different. The size of the system increases as additional clauses are added that

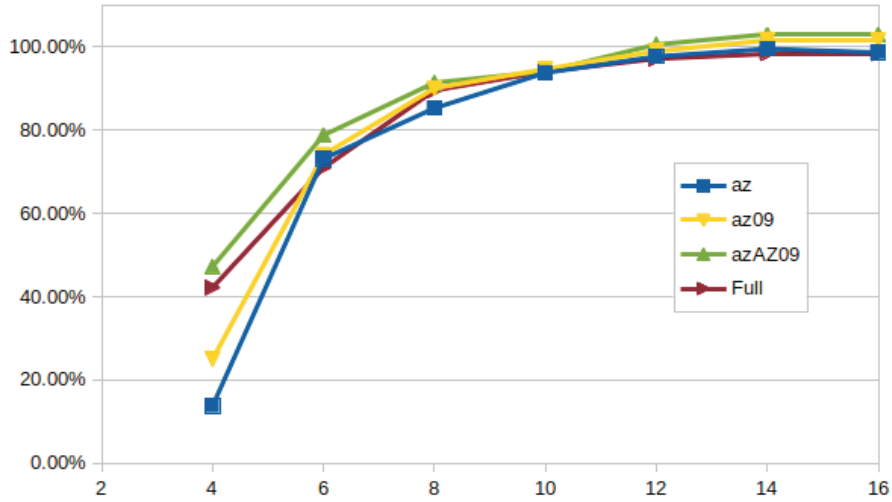

FIGURE 1. Estimated relative complexity of the algebraic attack on round reduced DES with password based keys.

model character restrictions. These additional clauses can cause a sooner termination of a DPLL branch in a search tree. The important question is whether these additional clauses are enough to reduce the practical complexity to levels comparable to reduction of keyspace. E.g. if the complexity of the efficient SAT-solver attack on some round reduced version of DES with full keyspace is equivalent to 2^{40} operations (instead of 2^{56}), does the same attack on the set `azAZ09` have complexity equivalent to $2^{34} = 2^{40/56 \cdot 48}$ operations?

Our first experiment is based on simulation using the tools to estimate complexity of algebraic attacks with local reduction and guessing used in [1, 12, 19]. We measure the average value of c for round reduced DES with N rounds, where 2^c is estimated size of optimal search tree in sylog based solver. The value 2^c is similar to the expected number of decisions in a DPLL based SAT solver [1]. We then compute a ratio

$$r(N, \mathcal{K}') = \frac{c(N, \mathcal{K}')}{\log_2 \mathcal{K}'},$$

for $\mathcal{K}'$ chosen either as whole $\mathcal{K}$ (denoted as `Full`), or one of the sets `az`, `az09`, `azAZ09`. The ratio r is plotted in Figure 1. We can see that for full DES (resp. DES with at least 12 rounds), the algebraic attacks should be as complex as respective dictionary attacks up to constant factors (depending on the efficiency of the solver), but not more. In 6-10 round DES we see similar complexity reduction in attacks on full keyspace as in attacks addressing reduced keyspace.

An interesting situation occurs with 4-round DES, where there is an observable significant difference in expected complexities between different sets. This might be caused by a weak DES key schedule, where some of the bits involved in restrictions are more important in the first four rounds. An interesting research question is whether we can deliberately poison key schedule for otherwise good cipher design to allow faster algebraic attacks for specific classes of keys?

We have conducted experiments with MiniSat to verify the simulation results. We have created a system for 4- and 6-round DES, and included the clauses to restrict key bytes to small letters in ASCII encoding (set `az`). We have run 100 instances of the problem with correct keys, e.g., those that contained only small letters, and 100 instances of random keys. There is only a negligible chance that random key is correct, in our experiments, all SAT instances from this set were found to be unsatisfiable. The results are summarised in Figure 2 and Table 1.

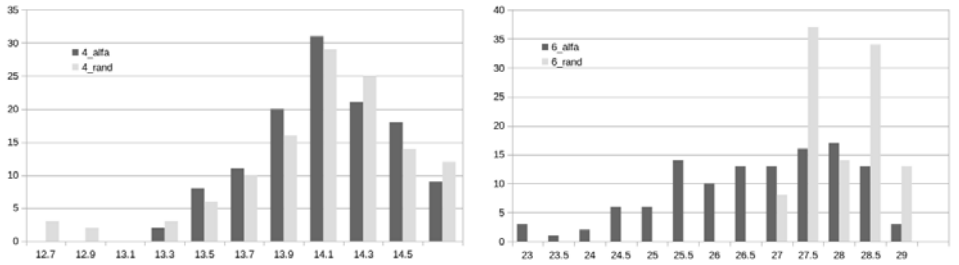

FIGURE 2. The distribution of the number of decisions (in log scale) made by MiniSat when solving instances of 4 (on the left) and 6 round DES (on the right). The system has additional clauses that restrict potential keys to small letters. The dark-colored distribution is a distribution for instances with correct keys, the light colored one with random keys (with very high probability incorrect).

TABLE 1. Base 2 logarithm of average number of decision for MiniSat solving instances of 4- and 6-round DES with constraints restricting potential keys to small letters, both for correct keys and randomly generated keys.

	4-round DES		6-round DES	
	Password	Random	Password	Random
Average Log Decisions	14	14	27	28
Relative to $\log_2 26^8$	37 %	37 %	71 %	74 %

It can be seen that 4-round instances have similar performance for both correct and incorrect keys. However, the fraction of key space covered by decisions

is higher than predicted by simulation (but comparable to attack on unrestricted keys). It is interesting to note that some instances of the problem were easier to solve than the typical ones, which can hint to insufficient heuristic selection of bits by MiniSat in comparison to simulation.

Instances of 6-round DES are harder to solve (as predicted by simulation). On the other hand, we can observe that the distribution is now different from the expected lognormal, which could mean that in larger system the heuristics employed by the solver have more pronounced effect. Moreover, the correct instances are on average slightly easier to solve, but both correct and incorrect instances have complexities similar to those predicted by the simulation.

4. Distributed algebraic attacks

It is very easy to distribute exhaustive search to multiple computers: we can just split the search space to smaller segments and push them to individual computing nodes. If we have M equivalent computing nodes, we can just split the search region into M equally sized segments, and obtain M-times speedup. If the nodes have different computing power, we create more smaller tasks, and each node is given a new task once the search of a previous task is finished. If the search space is large enough, communication and management overhead is negligible.

To distribute algebraic attack we can apply a similar strategy: Split the whole task into many small ones, and give each computing node a smaller task to complete. Once finished, a new task is given to the node. We cannot split the CNF formula, as all clauses must hold simultaneously. Instead, we select some variables corresponding to key bits, say $k_1, k_2, \ldots, k_m$. The set of variables can possible have at most $M = 2^m$ values. If we conduct an attack on low entropy keys as in Section 3, we can reduce this space further before starting the SAT solver based part. Each of these M assignments can be combined with original CNF formula. Instead of the original, we get M new SAT problems. These problems can be distributed into individual computing nodes to decide. If we expect one satisfying assignment for the original SAT problem, we will get $M-1$ UNSAT results and one SAT result with the correct solution from the computing nodes.

There are some specific issues with distributing one large SAT problem to many smaller SAT problems by guessing parts of the key. The first one is that UNSAT instances are usually slower to prove than SAT instances. If we distribute M tasks, $M-1$ are thus the slow cases. However, if we have M parallel computing nodes, this does not matter, as we can stop as soon as we obtain a positive answer. If M is much larger than the number of computing nodes, and

time difference between SAT and UNSAT instances is significant, we can adapt early stop strategy: once the SAT solver takes more time than required to find SAT instance (with high probability), we abort it. This requires some significant preparation to compute the empirical distribution, and fine tune the system, but the settings can be reused for many attacks.

The second problem is that by guessing the key bits manually, and splitting the tasks, we lose some of the heuristic speedup of the SAT solver. SAT solver's heuristics when making a decision cannot use information from the whole system (as part of it is predetermined by our guess). SAT solver instances do not share learned clauses, and cannot use information from learned clauses involving the bits we guessed manually. The more bits we guess manually, the less efficient SAT solver's heuristics we can expect. The optimal strategy seems to be to split the task at hand to exactly the same number of subtasks as the number of computing nodes. However, in some cases the resultant instances might be still too large for nodes to handle due to memory constraints (or time scheduling restrictions).

To examine the issues experimentally, we have realized a series of experiments with six-round DES and MiniSat solver in distributed computing cluster [14]. Our goal was to check the effects of the task distribution on the (estimated) total time of algebraic attack. We investigate not only the dependence of running times on the number of guessed bits, but also on which bits are guessed. We use three types of guessing strategies:

(1) *first*: We take the first m bits of the DES key. This can be thought of as a random selection of bits, as nothing special distinguishes these bits from others.

(2) *good*: We count the number of occurrences of DES key bits in the symbols used to derive the SAT formula (more precisely in the index set parts of symbols). We take the first m of the **most often used key bits**.

(3) *bad*: We count the number of occurrences of DES key bits in the symbols used to derive the SAT formula (more precisely in the index set parts of symbols). We take the first m of the **least often used key bits**.

The guessing strategies come from the heuristic experience obtained in other attacks. The *good* and *bad* strategies are motivated by the fact that more often a variable is used in a system, the more of the system it influences.

In Figure 3 we can see the empirical distributions of logarithms of running times of SAT solvers for both *good* and *bad* strategy (the *first* strategy is similar to the *bad* one). The more bits we guess, the lower the time needed to solve the small distributed instance of the SAT problem. As expected, the running times have a distribution similar to a log-normal one (we do not have enough precision to show this in a statistical way).

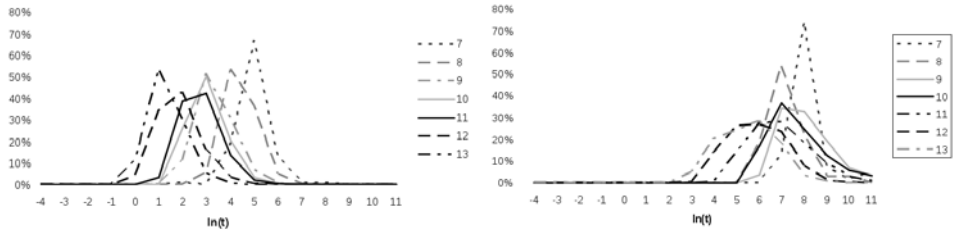

FIGURE 3. Histogram of logarithms of computing times depending on the number of guessed bits. On the left are times for a "good" selection of bits, on the right for a "bad" selection of bits.

On the other hand, we can see that there is an order of magnitude difference between running times in the *good* and the *bad* case for the same number of guessed bits. The difference is better seen in Figure 4, which shows a computed estimated total running time to solve the whole system based on empirical statistics obtained in the first experiment (including the *first* strategy and a comparison with exhaustive search on the same platform).

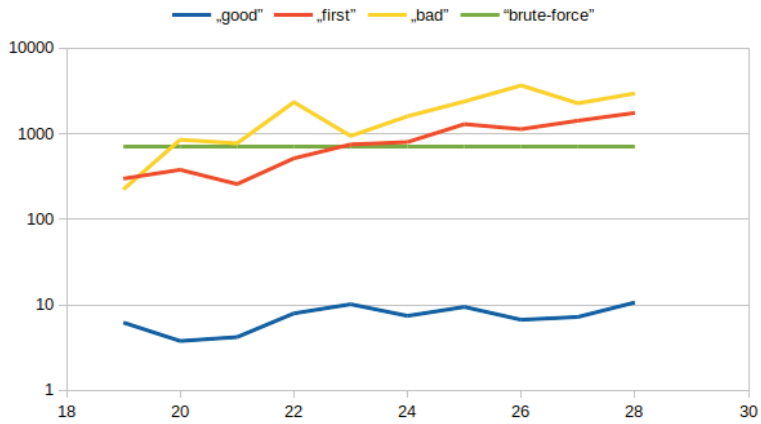

FIGURE 4. Estimated time complexity for the whole attack on 56-bit six-round DES for different selections and numbers of guessed bits.

We can see that the *bad* and the *first* strategy produce the expected result of growing total time per number of distributed tasks. On the other hand, our *good* strategy performs relatively consistently across different number of guessed bits with only a slow decline in performance when many tasks are generated. We note that if we extrapolate the trendlines for the *bad* and the *first* strategy to a single task where no bits are guessed, the expected running time should be comparable to the *good* strategy.

Thus, the overall impact of the distribution strategy is very significant. Between the worst and the best choice of guessed bits (including the number of bits) we get a thousand-fold increase in performance. In practice (as it is seen in Figure 4), the distribution strategy can also impact whether our SAT solver based algebraic attack is more efficient than exhaustive search, or not.

5. Algebraic attacks in multiple key scenario

Classical key recovery attacks focus on obtaining a single key used to encrypt a set of known P-C pairs. On the other hand, in the real world situations, the attacker is able to obtain many P-C pairs encrypted by many different keys. Suppose that each of these P-C pairs is based on the same plaintext. Attacker can modify the exhaustive search algorithm. He sorts the ciphertexts (or prepares a hash table). Then he tries to encrypting the plaintext with each key, and checks whether he gets any ciphertext from the set. This allows the attacker to obtain all the keys, or just one of them, much faster on average than by running individual exhaustive searches for each of the keys. The situation does not hold when each plaintext and ciphertext is different, as the attacker must test each key against each P-C pair.

As already observed by Courtois in [4], algebraic attacks in multikey scenario can be more efficient in the singlekey scenario. In [4], this was caused by special properties of GOST leading to a set of weak keys for which the attack is much faster than for an average key. However, as we will show, the algebraic cryptanalysis can provide a speedup even in generic key search in the multikey scenario.

For the sake of simplicity, let us suppose that one P-C pair is enough to identify the corresponding key. Furthermore, let the attacker collect M P-C pairs $(m^{(i)}, c^{(i)})$, each encrypted with a different key $k^{(i)}$ (unknown to the attacker). The attacker can construct a system for the cipher where plaintext and ciphertext bits are represented by unknowns, we will call them u, and v to differentiate them from generic unknowns x. The attacker can construct M individual SAT problems by adding additional unit clauses consisting of literals u_j, if $m_j^{(i)} = 1$, or $\neg u_j$, if $m_j^{(i)} = 0$, respectively. Similarly he adds unit clauses for v_j. With these assignments, the attacker can try to solve M individual systems. If he only wants to obtain any of the keys, he can be lucky (or have enough parallel computing nodes) and solve some of the systems faster. However, this can also be done with exhaustive search, and algebraic cryptanalysis does not lead to any special advantage in this case (if there are no special instances of weak key such as in [4]).

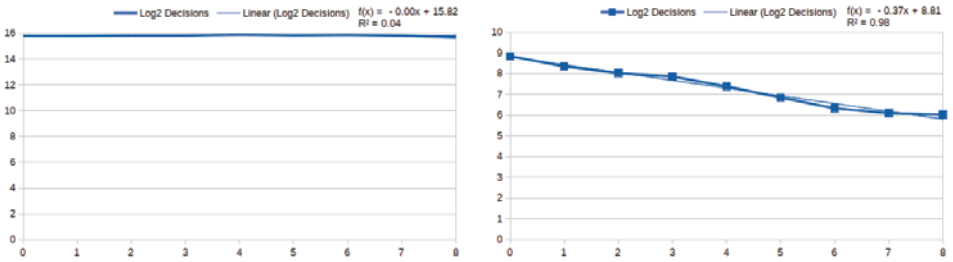

FIGURE 5. The dependence of the logarithm of SAT solver decisions on the logarithm of the number of PC pairs. On the left, average number of decisions before the first key is found by MiniSat. On the right, average number of decisions per key for SharpSat

On the other hand, the attacker that wants any key does not really need to construct M individual systems. He can encode the M known P-C pairs as one formula, which can be expressed in symbolic form as

$$\Phi = \left(\left(u^{(1)} = m^{(1)}\right) \wedge \left(v^{(1)} = c^{(1)}\right)\right) \vee \left(\left(u^{(2)} = m^{(2)}\right) \wedge \left(v^{(2)} = c^{(2)}\right)\right) \vee \cdots \\ \cdots \vee \left(\left(u^{(M)} = m^{(M)}\right) \wedge \left(v^{(M)} = c^{(M)}\right)\right).$$

Each of the expressions $(u^{(i)} = m^{(i)})$ can be expressed as a conjunction of literals u_j, if $m_j^{(i)} = 1$, or $\neg u_j$, if $m_j^{(i)} = 0$, respectively. Similarly for $(v^{(i)} = c^{(i)})$. Formula Φ is not in CNF, but it can easily be rewritten to this form by Tseitin transformation [17].

If F expresses the original system for the cipher, formula $F \wedge \Phi$ is satisfied for any key that encrypts some $m^{(i)}$ to $u^{(i)}$. Moreover, each proof of satisfiability gives us not only the value of the key, but also the corresponding plaintext and ciphertext. The set of all solutions of $F \wedge \Phi$ covers the whole set of keys. To find all solutions of some SAT instance, it is possible to modify the DPLL algorithm to exclude the solution that was found and continue the search. This is not directly supported by MiniSat solver, but there are solvers such as SharpSat that are able to provide all proofs of satisfiability within one computation.

We were increasing the size of the P-C set exponentially, so $M = 2^m$ for $m = 0, 1, 2, \ldots, 8$. In our experiments we were interested how M influences the average number of decisions required to find the first solution of the system (this can be done by MiniSat), and the average number of decisions per solution of the system (we needed SharpSat for this experiment). The results are presented in Figure 5. We can see that we have no advantage if we only need one key, but on the other hand, there is no specific penalty in the number of decisions required to find the solution. This does not mean that the time does not increase, but it only increases linearly with the size of the system, instead of exponentially.

On the other hand, if we are interested in many solutions, each new solution requires on average less number of decisions. This means an average speedup in exponent with the growing number of P-C pairs obtained by attacker, which leads to a significant savings per key obtained. E.g., if the attacker collects million P-C pairs, the average number of decisions is 170-times lower per each key. The savings of the attacker depend on how effectively he can encode formula Φ for such a large set of P-C pairs in comparison to the size of the original system F. We can, e.g., expect some savings, if each plaintext in the set of plaintexts shares some common bits.

6. Conclusions

Our experiments with algebraic cryptanalysis with SAT solvers show that:

(1) Low entropy keys from a set with an algebraic representation can be identified quickly, with the complexity based on the size of the set of low entropy keys, not the whole key space. This does not mean that SAT-solver based attacks on, e.g., a full DES are expected to be more efficient than brute-forcing the expected low-entropy key space when searching for a single key on a generic PC or specialized DES hardware. On the other hand, a specialized SAT-solving hardware does not lose the the advantage of the reduced key space. Moreover, for some weak instances of ciphers, the additional algebraic structure of the set of expected weak keys can significantly improve the algebraic attacks compared to results obtained when we do not impose any structure on the expected set of keys.

(2) When performing distributed attacks with SAT solvers, we lose the efficiency of solvers' heuristics if we split the system into very small parts by guessing parts of the key. On the other hand, if we carefully select the key bits to guess, we can observe a thousand fold decrease in the expected total running time when compared to a wrong choice (including the number of bits to guess). The dedicated attacker that wants to solve many instances of the problem can analyze the problem beforehand and select suitable distribution after initial simulation to gain this speedup in subsequent attacks. This could also mean that some of the experimental results based on guessing some bits and extrapolating results might in fact underestimate the complexity of the attack for a dedicated attacker.

(3) In multikey scenario, the attacker can encode many instances of the problem as a single SAT instance and try to find each solution of the system. In this case he gets a decrease in the average number of decisions per key depending on the number of P-C pairs he can obtain. To get some real time savings, the attacker must be able to efficiently encode the large formula based on the set of P-C pairs, and to efficiently find all solutions of SAT problem in one search.

The results taken together lead to a hypothesis that algebraic cryptanalysis with dedicated fast solvers might be more efficient than simple brute force in massive surveillance efforts, when the attacker can collect many unrelated instances of the problem (many P-C pairs with distinct plaintexts and ciphertexts), especially if he looks for some low entropy keys in these sets (produced by known compromised random number generators, or based on passwords).

REFERENCES

[1] ADAMČEK, P.—LODERER, M.—ZAJAC, P.: *A comparison of local reduction and SAT-solver based algebraic cryptanalysis of JH and Keccak*, Tatra Mt. Math. Publ. **53** (2012), 1–20.

[2] BARD, G.—COURTOIS, N.—JEFFERSON, C.: *Efficient methods for conversion and solution of sparse systems of low-degree multivariate polynomials over GF(2) via SAT-solvers*, Cryptology ePrint Archive, Report 2007/024, 2007, `http://eprint.iacr.org/`.

[3] BONNEAU, J.: *The science of guessing: analyzing an anonymized corpus of 70 million passwords*, in: IEEE Security & Privacy (Oakland) 2012, San Francisco, CA, USA, pp. 538–552.

[4] COURTOIS, N.: *Cryptanalysis of GOST in the multiple-key scenario*, Tatra Mt. Math. Publ. **57** (2013), 45–63.

[5] COURTOIS, N.—BARD, G.: *Algebraic cryptanalysis of the Data Encryption Standard*, in: Cryptography and Coding (Steven Galbraith, ed.), Lect. Notes in Comput. Sci., Vol. 4887, Springer, Berlin, Heidelberg, 2007, pp. 152–169.

[6] COURTOIS, N.—HULME, D.—MOUROUZIS, T.: *Solving circuit optimisation problems in cryptography and cryptanalysis*, Cryptology ePrint Archive, Report 2011/475, 2011.

[7] DAVIS, M.—LOGEMANN, G.—LOVELAND, D.: *A machine program for theorem proving*, Comm. ACM **5** (1962), no. 7, 394–397.

[8] EEN, N.—SÖRENSSON, N.: *The MiniSat page*, `http://minisat.se`.

[9] FAUGÈRE, J.-C.—PERRET, L.—SPAENLEHAUER, P.-J.: *Algebraic-differential cryptanalysis of DES*, in: Western European Workshop on Research in Cryptology-WEWoRC 2009, pp. 1–5.

[10] GĄSECKI, A.: *Low data complexity differential-algebraic attack on reduced round DES*, Tatra Mt. Math. Publ. **57** (2013), 35–43.

[11] KLEINJUNG, T.—LENSTRA, A.—PAGE, D.—SMART, N.: *Using the cloud to determine key strengths*, Cryptology ePrint Archive, Report 2011/254, 2011, `http://eprint.iacr.org/`.

[12] LACKO-BARTOŠOVÁ, L.: *Algebraic cryptanalysis of PRESENT based on the method of syllogisms*, Tatra Mt. Math. Publ. **53** (2012), 201–212.

[13] NIST: *Data Encryption Standard (DES)*, FIPS PUB 46-2, January 1988.

[14] ÖLLŐS, L.: *Algebraic cryptanalysis on GRID*, Master's Thesis, Slovak University of Technology, 2014. (In Slovak)

[15] RENAULD, M.—STANDAERT, F.-X.— NICOLAS—VEYRAT-CHARVILLON, N.: *Algebraic side-channel attacks on the AES: Why time also matters in DPA*, in: Cryptographic Hardware and Embedded Systems-CHES 2009, Springer, Berlin, 2009, pp. 97–111.

[16] THURLEY, M.: *sharpSAT-counting models with advanced component caching and implicit BCP*, in: Theory and Applications of Satisfiability Testing-SAT 2006. Lect. Notes in Comput. Sci., Vol. 4121, Springer, Berlin, 2006, pp. 424–429.

[17] TSEITIN, G. S.: *On the complexity of derivation in propositional calculus*, in: Automation of Reasoning, Springer, Berlin, 1983, pp. 466–483.
[18] ZAJAC, P.: *On the use of the method of syllogisms in algebraic cryptanalysis*, in: Proceedings of the 1st Plenary Conference of the NIL-I-004: Bergen, Norway, 2009, pp. 21–30.
[19] ZAJAC, P.—ČAGALA, R.: *Local reduction and the algebraic cryptanalysis of the block cipher GOST*, Period. Math. Hungar. **65** (2012), 239–255.

Received September 9, 2015

Institute of Computer Science and Mathematics
Faculty of Electrical Engineering and
Information Technology
Slovak University of Technology in Bratislava
Ilkovičova 3
SK–812-19 Bratislava
SLOVAKIA

E-mail: xhromadav@stuba.sk
xollos@stuba.sk
pavol.zajac@stuba.sk