CENSOR: Privacy-preserving Obfuscation for Outsourcing SAT formulas

Tassos Dimitriou, *Senior Member, IEEE* and Khazam Alhamdan

***Abstract*—We propose a novel obfuscation technique that can be used to outsource hard satisfiability (SAT) formulas to the cloud. Servers with large computational power are typically used to solve SAT instances that model real-life problems in task scheduling, AI planning, circuit verification and more. However, outsourcing data to the cloud may lead to privacy and information breaches since satisfying assignments may reveal considerable information about the underlying problem modeled by SAT.**

In this work, we develop CENSOR, a SAT obfuscation framework that resembles Indistinguishability Obfuscation. At the core of the framework lies a mechanism that transforms any formula to a random one with the same number of satisfying assignments. As a result, obfuscated formulas are indistinguishable from each other thus preserving the input-output privacy of the original SAT instance. Contrary to prior solutions that are rather adhoc in nature, we formally prove the security of our scheme. Additionally, we show that obfuscated formulas are within a polynomial factor of the original ones thus achieving polynomial slowdown. Finally, the whole process is efficient in practice, allowing solutions to original instances to be easily recovered from obfuscated ones. A byproduct of our method is that all NP problems can be potentially outsourced to the cloud by means of reducing to SAT.

***Index Terms*—SATisfiability, Obfuscation, Cloud computing/outsourcing, Verifiability, Privacy preservation, Random walks/Mixing time**

1. Introduction

Cloud computing delivers computing services (e.g. storage, servers, databases, etc.) to consumers and businesses alike, allowing them to run applications and store large amounts of data in cloud servers across the internet. Computational tasks that would otherwise require a huge amount of processing power and pose a huge burden on a client's infrastructure can now be outsourced to the cloud, offering an attractive alternative to buying or maintaining in-house servers. Cloud providers on the other hand are financially motivated to share their computational resources as meeting client demands is typically associated with a service fee. For instance, Amazon Web Services, Microsoft Azure or Google Cloud provide on-demand delivery of computing power as described above.

However, the main obstacle to outsourcing computations is privacy assurance and protection. Outsourced data may contain valuable or confidential information that can put a client's security at risk. As the cloud provider is not necessarily trustworthy, additional measures are needed to protect the confidentiality of user data. Furthermore, even if the cloud provider is trusted by the client, external attackers may gain illegitimate access to cloud servers and the data residing in them. Hence, clients need to ensure that their data remain secure before, during and after outsourcing. This problem led researchers to consider various operation- and application-oriented approaches to securing outsourced computations.

One of the main approaches to secure computation outsourcing is the use of Fully Homomorphic Encryption (FHE) [1] which aims to achieve both data confidentiality and result integrity. FHE allows the support of multiple operations on encrypted data, however this general mechanism is far from being practical. More efficient variants (however of reduced functionality) include Partially Homomorphic Encryption (PHE) (a typical example is Paillier's cryptosystem [2]) and Somewhat Homomorphic Encryption (SWHE) techniques [3]. PHE supports only one type of operation (either addition or multiplication but not both) on encrypted data while typical SWHE schemes can perform additions and a limited number of multiplications on ciphertexts, which allows handling more advanced computations compared to PHE-based schemes. Gentry et al. [4] contributed a more efficient scheme to carry homomorphic operations while Brakerski et al. [5] introduced a more efficient FHE scheme.

While FHE mechanisms can be used to protect any function that can be expressed as a Boolean circuit, they typically have large overhead that makes them unsuitable for large-scale computational tasks. This motivated researchers to look for solutions that, despite being less general and only applicable to certain types of computations, are more efficient for outsourcing specific tasks. Examples include scalar and set operations such as union, intersection and difference [6].

Another important application category involves matrix computations which can be realized as vector operations. For example, the works in [7], [8] emphasize in confidentiality and verifiability of algebraic and matrix computations. Other operations on matrices involve inversions [9] and solving systems of linear equations [10]. Most of these techniques rely on transforming or hiding the original instance by multiplying it with random matrices. Similarly, Wang et al. [11] were the first to provide practical mechanisms for securely outsourcing Linear Programs to the cloud.

Despite the wide applicability of these techniques even in advanced applications such as Machine Learning and Data Mining (for more details the reader is referred to [6] and [12]), the domain of outsourcing Boolean formulas to

• *Computer Engineering Dept., Kuwait University, Kuwait. Email: tassos.dimitriou@ieee.org, khazam.alhamdan@grad.ku.edu.kw*

the cloud has remained rather unexplored. In this respect, the present work aims to address this shortcoming.

Contributions. We investigate the security and feasibility of outsourcing large Boolean formulas to the cloud. Boolean Satisfiability (SAT) is a decision problem that asks whether a given formula can be made satisfiable under appropriate assignments to the variables of the formula. SAT has numerous applications in software testing and verification, circuit design, AI planning, task scheduling, and more [13]. However, despite the success of modern SAT solvers [14], even small sized formulas consisting of thousands of variables cannot easily be managed by them. Hence outsourcing to the cloud seems like the only workable mechanism to handle large satisfiability instances and the financial incentives for providing competitive cloud solvers are obvious.

While cloud outsourcing can be cost-effective, the whole process may suffer from privacy leaks as the structure of the formulas or the values of the solutions may reveal considerable information about the underlying problem modeled by the SAT instance. For example, many applications (AI planning, FPGA routing, circuit design, etc.) encode domain specific constraints into Boolean formulas which can then be easily extracted, as demonstrated in a number of works [15]–[18]. The solution to this is to *obfuscate* the SAT instance *prior* to outsourcing it to the cloud. However, existing techniques for obfuscating SAT formulas are rather adhoc and have been proven insecure (Section 2).

In this work, we develop CENSOR, an obfuscation framework which preserves the input-output privacy of outsourced SAT formulas. Our obfuscator deviates from traditional approaches which try to embed random "noise" in formulas. The novelty of our approach lies in the use of an algebraic substitution mechanism that turns any SAT instance into another instance with exactly the *same* number of satisfying assignments using a random walk with good mixing properties in the space of solutions.

Our first contribution is to formally describe this obfuscation mechanism by developing a framework which is similar to *Indistinguishability Obfuscation* (*iO*) [19] with the *exception* of functionality preservation; if f and $obf(f)$ agree on all assignments x as in traditional *iO*, this would leak information about the original formula. Hence a new notion of functionality preservation is required that essentially turns the scheme into *trapdoor iO* and makes it suitable for outsourcing.

Our second contribution is a basic obfuscation process which is very efficient in practice. Although the setting is not the same as in [20], we show how to obfuscate formulas with hundreds or even thousands of variables and clauses very efficiently as opposed to 64-bit conjunctions which require a very large computation overhead [21]. This happens because we don't rely on heavy cryptographic primitives and assumptions but on simple logical operations that result in a total running time of $O(m^2 \log m)$, where m is the number of clauses in the outsourced formula. Furthermore, the size of the obfuscated formula $obf(f)$ is within a factor of m of the original SAT formula, thus achieving polynomial slowdown, while recovering the solution of f from the solution of $obf(f)$ is a simple and straightforward process, thus reducing the overall impact of SAT outsourcing.

We also develop a more advanced scheme that offers stronger guarantees about the distribution of the satisfying assignments of the obfuscated formula. In particular, we prove that any k assignments remain uniformly distributed in the space of solutions. The running time of this scheme is $\tilde{O}((m + k^2)^2)$. In practice, however, dependency on k can be eliminated since, typically, hard outsourced formulas can only have few assignments discovered in polynomial time (for more on this see Sections 6.4 and 6.5).

Finally, an obvious byproduct of our method is that it is directly applicable to outsourcing other NP problems by way of reducing to SAT.

Organization. In the next section we provide a survey of works on obfuscating Boolean formulas and circuits. In Section 3, we discuss the threat model, we define our obfuscation framework and we review preliminary material. In Section 4, we define the substitution algebra that is at the heart of the obfuscation scheme which is described in Section 5. In Section 6, we prove the correctness and soundness of the obfuscator as well as the mixing time required to achieve indistinguishability of SAT formulas. The performance of the various obfuscation components is verified experimentally in Section 7. Section 8 offers a comparison with Virtual Black Box (VBB) techniques on obfuscating conjunctions [20]. Finally, Section 9 concludes this work.

2. Related Work

In this section, we review past work on SAT obfuscation. However, the majority of these works focus mostly on hiding circuit structures for intellectual property (IP) protection at the hardware level.

Logic locking [22] is a technique attempting to lock the logical functionality of a circuit unless a certain key input is provided along with the input parameters. This works by adding special 'key' gates to the paths of the circuit gates. If the correct inputs to these gates are provided, the circuit produces the correct output; otherwise, the design produces a wrong result. SLL (Strong Logic Locking) [23] attempts to overcome the weaknesses of basic logic locking. However, this and similar schemes fall prey to SAT-based attacks [24] which target specific vulnerabilities of the target logic encryption technique.

Subramanyan et al. [24] implemented an algorithm to unlock locked circuits using a SAT solver to brute-force the secret key. Given a locked logic circuit C and a working, unlocked circuit c (used as an oracle), the goal is to find the locking sequence k to unlock C. The algorithm works by generating many $\langle input, key, output \rangle$ tuples, then use them to produce a circuit of constraints that will be satisfiable if the key used is the encryption key k.

The SARLock scheme [25] was developed to enhance the circuit lock schemes which were easy targets for the SAT-based attacks implemented in [24]. However, Shamsi et al. [26] introduced an enhanced version of the SAT attack to counter the anti-SAT obfuscation schemes (such as SARlock). Existing encryption schemes assumed uniqueness of outputs for every key combination which makes them susceptible to the approximation attacks introduced in [24] and [26].

The previous schemes focus on hiding the functionality of a circuit, however the described attacks (for more the reader is referred to [27]) demonstrate the difficulty of this goal. Unfortunately, there is also very little work on hiding the structure of SAT formulas when outsourced to a cloud solver. Brun et al. [28] distributed the computation of a SAT formula on multiple machines so that every machine gets a different part of the formula to evaluate. Each evaluation is then passed to a neighbouring machine. However, the trust model is very limited as nothing prevents the servers from colluding in order to recover the hidden formula. Qin et al. [29] introduced an obfuscation scheme that first generates a formula based on a prime factorization circuit, then blends it with the original formula to achieve obfuscation. Although the idea seems interesting, the obfuscated formula has an exploitable structure which can be targeted by an XORing attack as demonstrated in [30].

This overview shows that the success of ad hoc techniques is rather doubtful and might not work well in practice as these methods rely mostly on "security via obscurity", contradicting the basic assumption that an adversary knows the details of the underlying system. Hence more formal frameworks should be used instead. Our work aims to fill this gap.

On Program Obfuscation and SAT Outsourcing. Program obfuscation is a transformation that aims to make a program unintelligible without, however, affecting its functionality or revealing anything about its description. Essentially this means that the program should work as a virtual black box (VBB).

Barak et al. [19] were the first to study the impossibility of general-purpose VBB obfuscation and suggested instead the seemingly weaker notion of indistinguishability obfuscation (*iO*) in which two different descriptions of the same program, having the same size and the same functionality (input-output behavior), should be indistinguishable from each other. The breakthrough result of Garg et al. [31] and subsequent work by Sahai and Waters [32] gave rise to a plethora of applications (for an overview see [33]) which demonstrated the huge potential of the *iO* paradigm.

One may be tempted therefore to build upon previous works on obfuscating conjunctions [20] to develop an algorithm that can be used to outsource SAT formulas to the cloud. However, by definition, such obfuscator would create obfuscated formulas that *agree* with the original formula on all assignments x, thus violating the sought for input-output privacy for the underlying SAT instance (recall the attacks in [15]–[18]).

Hence a new notion of functionality preservation is required that is better suited for outsourcing formulas to the cloud, one that is not yet captured by the existing model of VBB and *iO* security. Essentially, this new obfuscation framework should be used to not only hide the structure of the original formula (its description) but also the relationship of the assignments between the obfuscated and the original SAT instances. We consider this another important contribution of this work, one we formalize in the next section.

3. Threat model and assumptions

We consider a user U who is in possession of a difficult to solve SAT formula f. Due to the lack of computing power and resources, U wishes to outsource the formula to a cloud solver CS which has the capacity to solve computation-demanding problems for a fee.

Despite the obvious benefits for outsourcing the problem to the cloud, the formula may have been used to capture sensitive information since SAT has broad applications in circuit and software verification, task scheduling, etc. [13], hence it cannot be directly given to CS. As CS is not trusted by U, f has to be obfuscated prior to outsourcing in order to provide assurance that no information leakage occurs (Figure 1).

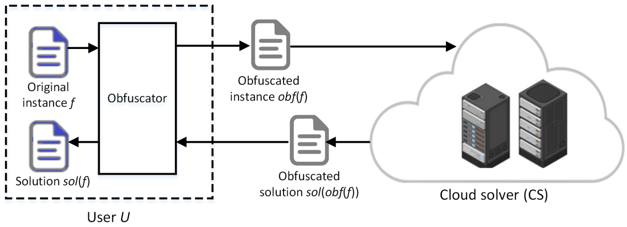

Figure 1: Flow of information between the user U and Cloud Solver CS

3.1. Security and performance requirements

Our threat model includes a CS which may be interested in analyzing the obfuscated formula or the solution produced in order to recover sensitive information about the original function. To enable secure outsourcing of SAT expressions, the following security and performance requirements are envisioned:

1) *Correctness*: Any honest CS that manages to solve the obfuscated instance $obf(f)$ should be able to produce a solution that can be de-obfuscated by U and lead to a solution of f.
2) *Verifiability*: No malicious CS should be able to produce a wrong solution that can be de-obfuscated and verified successfully by U.
3) *Privacy*: No sensitive information about the original formula should leak, other than already known, a priori information about f.
4) *Polynomial slowdown*: The size of $obf(f)$ should be within a polynomial of the size of f.
5) *Solver efficiency*. As there can be no guarantee on the time to solve hard SAT instances due to the NP-completeness of the Satisfiability problem, the burden on the cloud solver to solve $obf(f)$ should not be prohibitive. Ideally, it should be comparable to solving the original instance f.
6) *User efficiency*. The time to create $obf(f)$ and recover the solution of the original instance from $sol(obf(f))$ should be considerably less than locally solving f. In particular, it should be a polynomial on the number of variables and clauses of f, thus making solution recovery *independent* of the hardness of the original SAT instance.

3.2. Obfuscation framework

Our obfuscation scheme consists of the following operations.

- $\mathsf{KeyGen}(1^\lambda) \rightarrow key$. This is a key generation algorithm that takes as input a security parameter λ and returns a secret key that is used in the obfuscation of the original formula. The key can be thought as the seed or the randomness required during obfuscation.
- $\mathsf{Obfuscate}(key, f, n, m) \rightarrow obf(f)$. This algorithm obfuscates the input formula f using key, where n is the number of variables and m the number of clauses of f, respectively. $obf(f)$ is also a SAT formula which is outsourced to the cloud solver.
- $\mathsf{De\text{-}Obfuscate}(key, sol(obf(f)) \rightarrow \{sol(f), \perp\}$. This algorithm produces a solution for f based on the solution $sol(obf(f))$ of the obfuscated formula returned by CS. The algorithm returns $\perp$ if the validation of $sol(obf(f))$ fails or if the cloud solver returns no solution to the SAT instance.

3.3. Definition of privacy

Our definition resembles the notion of Indistinguishability Obfuscation [19]. A Probabilistic Polynomial Time (PPT) algorithm iO is said to be an indistinguishability obfuscator for a class of circuits C, if it satisfies *functionality* preservation and indistinguishability. Functionality preservation in this case means that for all circuits $c \in C$, the obfuscated circuit $iO(c)$ should match c on all inputs x, i.e. $iO(c, 1^\lambda)(x) = c(x)$.

Unfortunately this notion cannot be directly applied in our setting since the obfuscated formula $obf(f)$ should not necessarily agree with f on assignments x. Hence, we re-define functionality preservation to mean *SATisfiability preservation*. In particular,

Definition 1 (Functionality preservation for Boolean formulas)**.**

1) *For any satisfying assignment s such that $obf(f)(s) = True$, there is a unique assignment x such that $f(x)$ = True and vice versa.*
2) *If $s \neq \perp$ then $x = obf^{-1}(key, s)$, where obf^{-1} denotes the de-obfuscation algorithm.*

Part 1 in the definition means that there is a one-to-one correspondence between truth assignments of f and $obf(f)$. Hence f is satisfiable iff $obf(f)$ is satisfiable. Part 2 ensures that a satisfying assignment for f can easily be recovered from an assignment s of $obf(f)$ with the help of the trapdoor key.

This deviation from the traditional definition of iO functionality is necessitated by the fact that the output of a typical iO obfuscator on x matches the value of f on x. However, in the case of SAT this may result in a serious breach on privacy since the value of the assignment x may reveal information about the underlying problem modeled using f. In the following, the term *#SAT-equivalent* will refer to two functions with the same number of satisfying assignments.

Definition 2 (Indistinguishability for Boolean formulas)**.** *For any polynomial-size distinguisher $D = \{D_\lambda\}_{\lambda \in N}$, there exists a negligible function $\mu(\lambda)$ such that for any two #SAT-equivalent Boolean functions f_0, f_1 of the* same *size[1]:*

$$|\Pr[D(obf(f_0, 1^\lambda))] - \Pr[D(obf(f_1, 1^\lambda))]| < \mu(\lambda),$$

where the probability is over the random bits of obf.

3.4. Notation

Let $X = \{x_1, x_2 \ldots, x_n\}$ be a set of propositional variables. A *literal*, is a variable that can be in complemented ($\bar{x}$) or uncomplemented form (x). The notation $\dot{x}$ will be used to refer to a literal (x or $\bar{x}$) pertaining to a variable x without specifying its form (complemented or not). In this work, we will be working with formulas in Conjunctive Normal Form (CNF), i.e. conjunctions of one or more clauses $C_1 \cdot \ldots \cdot C_m$, where each clause is a disjunction of literals $(l_1 + \ldots + l_k)$.

The satisfiability (SAT) problem asks for an assignment of True/False (or 0/1) values to the variables of a given CNF formula f such that f evaluates to True (or 1). If there is no assignment satisfying all clauses, the formula is said to be *unsatisfiable*. Without loss of generality, we will be working with 3-SAT instances in which all clauses consist of exactly 3 literals as it is known that any formula can be converted to 3-CNF form. However, our methods can be applied *directly* to more general formulas as well. The number of variables and clauses in the formula will be denoted by n and m, respectively.

For a Boolean function of n variables, a *minterm* is the logical product (AND) of the n variables in either complemented or uncomplemented form. The k^{th} minterm (often denoted by m_k) is the minterm for which the i-th variable is negated if the i-th bit in the binary expansion of k is 0. Any function can be expressed as the sum of all minterms corresponding to the rows of its truth table where the function value is one. The *truth table* of a Boolean function on n variables is a tabular representation of the function's value on all possible assignments of the input variables. The last column is referred to as the *output* and corresponds to the formula captured by the truth table. For simplicity, we will use f_i or $f(i)$ to refer to the value of f in the i^{th} row of its output column.

4. Substitution Algebra

Definition 3. *A substitution of x by h in f (denoted by $f|_{x \leftarrow h}$) is the Boolean formula resulting by replacing any occurrence of x in f by the expression h.*

We will use substitutions to transform any function f into a function g with an *equal* number of satisfying assignments. Thus, substitutions will be the core method to transform/obfuscate a given Boolean function. Next, we define our equivalence classes with respect to the total number of True values (1's) in the output column of a formula.

1. As we will be working with formulas in d-CNF form (Section 3.4), the size of a formula on n variables is defined as the number of its clauses m.

Definition 4. *Denote by $\#(f)$ the number of satisfying assignments of a Boolean function f, i.e. $\#(f) = \sum_{i=0}^{2^n-1} f_i$, where n is the number of variables of f and $f_i \in \{0, 1\}$.*

Proposition 1. *Let f and g be two formulas. We will write $f \equiv g$ iff $\#(f) = \#(g)$. Then $\equiv$ is an equivalence relation.*

Proof. For the proof one has to show that the relation is reflexive, symmetric and transitive. However, this is straightforward and omitted as the number of 1's in the output column of a function is maintained under all these operations. Hence $\equiv$ is an equivalence relation. ∎

Thus, $\equiv$ can be used to partition the problem space (set of all formulas) into equivalent classes $[f]$, where each class contains all functions with the same number of satisfying assignments.

In the sequel, we will be working with substitutions of the form $w \leftarrow w \oplus a_1a_2 \cdots a_j$ (i.e. w will be substituted with $w \oplus a_1a_2 \cdots a_j$), where the a_i's are single literals *different* than w and '$\oplus$' is the XOR operator (s.t. $x \oplus y = x\bar{y} + \bar{x}y$). When $j = 1$, we call these *unit-substitutions*. Substitutions are very powerful as they maintain the number of satisfying assignments (Theorem 3). To prove this we need the following intermediate result.

Lemma 2. *Consider a substitution $w \leftarrow w \oplus a_1a_2 \cdots a_j$, where the a_i's are literals of the formula. Then minterms of the form $m = \dot{w}a_1a_2 \cdots a_jB$, where B is the conjunction of the remaining $n - j - 1$ literals, will have $\dot{w}$ complemented, while all others (not containing the exact form of $a_1a_2 \cdots a_j$) will remain unchanged.*

Proof. For the proof, we first consider the case where $m = \dot{w}a_1a_2 \cdots a_jB$ and compute the value of $m' = m|_{w \leftarrow w \oplus a_1a_2 \cdots a_j}$. Thus,

$$\begin{aligned} m' &= (\dot{w} \oplus a_1a_2 \cdots a_j)a_1a_2 \cdots a_jB \\ &= (\bar{\dot{w}}a_1a_2 \cdots a_j + \dot{w}\overline{a_1a_2 \cdots a_j})a_1a_2 \cdots a_jB \\ &= \bar{\dot{w}}a_1a_2 \cdots a_jB \quad (\dot{w} \text{ is } \textit{flipped}) \end{aligned}$$

Now consider the case where $m = \dot{w}\bar{a}_1\bar{a}_2 \cdots \bar{a}_ia_{i+1} \cdots a_jB$, i.e. (wlog) the first $1 \leq i \leq j$ literals appear complemented in the minterm. Then

$$\begin{aligned} m' &= (\dot{w} \oplus a_1a_2 \cdots a_j)\bar{a}_1\bar{a}_2 \cdots \bar{a}_ia_{i+1} \cdots a_jB \\ &= (\bar{\dot{w}}a_1a_2 \cdots a_j + \dot{w}\overline{a_1a_2 \cdots a_j})\bar{a}_1\bar{a}_2 \cdots \bar{a}_ia_{i+1} \cdots a_jB \\ &= \dot{w}(\bar{a}_1 + \cdots + \bar{a}_j)\bar{a}_1\bar{a}_2 \cdots \bar{a}_ia_{i+1} \cdots a_jB \\ &= \dot{w}\bar{a}_1\bar{a}_2 \cdots \bar{a}_ia_{i+1} \cdots a_jB \\ &= m \quad (\dot{w} \text{ is } \textit{unchanged}) \end{aligned}$$

Hence only the minterms that have a *matching* $a_1a_2 \cdots a_j$ will have the $\dot{w}$ literal flipped as the lemma suggests. ∎

We can now prove the following important theorem:

Theorem 3. *Substitutions maintain the number of satisfying assignments of a Boolean function.*

Proof. Let f be a Boolean function on n variables, and $\#(f)$ be the number of satisfying assignments of f. f can be represented using minterm notation as

$$f = m_{k_1} + m_{k_2} + \cdots + m_{k_L},$$

where each minterm $m_{k_l} = (\dot{u}_1\dot{u}_2 \cdots \dot{u}_n)$ is the conjuction of n literals corresponding to the *ones* of the function, and $L = \#(f)$. Thus, a substitution $f|_{w_i \leftarrow w_i \oplus a_1a_2 \cdots a_j}$ on f will be given by

$$m_{k_1}|_{w_i \leftarrow w_i \oplus a_1a_2 \cdots a_j} + \cdots + m_{k_L}|_{w_i \leftarrow w_i \oplus a_1a_2 \cdots a_j}.$$

Now consider the substitution effect on some minterm m_{k_l}. There are two cases to consider for $a_1a_2 \cdots a_j$. Either $a_1a_2 \cdots a_j$ appears in the exact same form in the minterm or not. By Lemma 2, if $a_1a_2 \cdots a_j$ appears in the minterm, $\dot{w}_i$ will be complemented. Thus the original minterm m_{k_l} will be *swapped* with another one that contains $\bar{\dot{w}}_i$. On the other hand, if some of the literals in $a_1a_2 \cdots a_j$ appear complemented in m_{k_l}, this will leave the minterm unchanged.

As the total number of minterms is maintained, we have that $\#(f|_{w_i \leftarrow w_i \oplus a_1a_2 \cdots a_j}) = \#(f)$. Thus, substitutions maintain the number of satisfying assignments of the formula. ∎

This will be the basis for our obfuscation method. By applying an appropriate number of random substitutions on an input function $f \in [f]$, f will be transformed into a random function $g \in [f]$ which has the same number of satisfying assignments as f. Since all functions in $[f]$ will be indistinguishable from each other (by selecting the *number* of random substitutions appropriately), the security of the obfuscation process will follow.

5. Obfuscation Scheme

Overview: Substitutions will be used to obfuscate a given function f by turning it into a function g with exactly the same (yet unknown to the user) number of satisfying assignments. Hence g can be safely outsourced to the cloud solver without fear of leaking information about the original function f.

In the sequel we will describe a basic scheme that uses only unit-substitutions of the form $w \leftarrow w \oplus a$, where a is some literal different than w. This will guarantee that two #SAT-equivalent formulas are structurally indistinguishable (at the syntax level) but cannot fully guarantee that satisfying assignments of their obfuscated versions are uniformly distributed in the space of solutions. For this reason, in the proof of Theorem 11 (Section 6.4), we will add an extra phase that uses general type substitutions and achieves the desired result. However, in Section 6.5, we argue that the extra phase might not really be necessary in real life applications. The description of the basic scheme follows.

5.1. Obfuscation using unit-substitutions

Algorithm 1 details the basic obfuscation scheme. The algorithm applies a sequence S of N random unit substitutions to the "flattened" input formula, where N will be defined later. The concepts of flattening and Tseitin encoding will be described in the sequel.

We begin the analysis of the algorithm by a lemma that attempts to capture the structure of clauses after a series of unit-substitutions. In particular, the lemma shows that the size of each clause in the final obfuscated formula *does*

Algorithm 1: Basic Obfuscation

Input: A 3-CNF formula f
Output: $obf(f)$
Set $f' \leftarrow flatten(f)$
Let $S = [\langle w_i, a_i\rangle]_{i=1}^{N}$ be a list of random unit-substitutions
foreach *pair* $\langle w, a\rangle \in S$ **do**
  $f' \leftarrow f'|_{w \leftarrow w \oplus a}$
end
Set $f' = Tseitin(f')$ (*Optional*)
return *f'*

not depend on the number of unit substitutions applied. Then, we argue about the running time of the algorithm.

Lemma 4. *Let n be the number of variables in the formula and let $c = (u + v + w)$ be a 3-SAT clause. Then, irrespective of the number of substitutions, the final clause will have the form*

$$(u_1 \oplus \dot{\ldots} \oplus u_p \;+\; v_1 \oplus \dot{\ldots} \oplus v_q \;+\; w_1 \oplus \dot{\ldots} \oplus w_r),$$

where the dot ($\dot{}$) indicates a possible complement, u_i, v_j, w_k are variables of the formula and $p, q, r \leq n$.

Proof. The proof is by induction on the number N of substitutions. When p, q, r are equal to 1, the XOR degenerates to a single variable hence when $N = 1$ we have the original clause. So, assume that during the κ^{th} substitution the clause has the form

$$(u_1 \oplus \dot{\ldots} \oplus u_\alpha \;+\; v_1 \oplus \dot{\ldots} \oplus v_\beta \;+\; w_1 \oplus \dot{\ldots} \oplus w_\gamma),$$

for some $\alpha, \beta, \gamma \leq n$. Now consider the next unit-substitution $x \to x \oplus \dot{y}$, for some variable x and *literal* $\dot{y}$. Assume x appears in some XOR group (if not, no substitution will take place) and in particular $u_1 \oplus \dot{\ldots} \oplus u_\alpha$ (the other groups are handled similarly). Further assume that $x = u_1$ after re-arranging the literals. Then the substitution will create the new group

$$(x \oplus \dot{y}) \oplus \dot{\ldots} \oplus u_\alpha$$

Its exact form depends on whether variable y is present in the group and in which form (complemented or not). Here we consider the four cases:

- $\dot{y} = y$ and y is equal to some variable u_i in the group. Then since $y \oplus y = 0$, y will disappear and the 0 will not affect the remaining variables in the group since $u \oplus 0 = u$. Hence the group size will decrease by one.
- $\dot{y} = \bar{y}$ and y is equal to some u_i in the group. The complement in y will move on *top* of the entire group since $\overline{u \oplus v \oplus w} = \overline{u} \oplus v \oplus w = u \oplus \overline{v} \oplus w = u \oplus v \oplus \overline{w}$. As before y will disappear and the group size will decrease by one. However, the group will have an extra complement on top of it which may be cancelled if there was already one in the beginning.
- $\dot{y} = y$ and y does not appear in the group. Then the group size will increase by one.
- $\dot{y} = \bar{y}$ and y does not appear in the group. Then the group size will increase by one and the complement on y will move on top of the group which may be cancelled if there was already one in the beginning.

Hence the group will retain its form with the possible addition or elimination of one variable, and the addition or cancellation of the group's complement. Notice also that the group size can never exceed the number of variables n (as redundant variables will be cancelled), so its size is independent of the number of substitutions and always bounded by n. ∎

Since each clause consists of 3 such XOR groups and there are m clauses overall, the method achieves polynomial expansion. We conclude that

Theorem 5. *The size of the obfuscated formula is bounded by $O(mn)$, where m is the number of clauses and n the number of variables in the original formula.*

Lemma 4 also leads to a very efficient algorithm to handle substitutions. As before, a clause will consist of 3 XOR groups (connected with OR), hence we will concentrate on just one of them. In particular, we associate with every XOR group a vector of size $(n+1)$, where n is the number of variables in the formula. The i^{th}, $i = 1, \ldots, n$, position in the vector is an indicator of whether variable i exists in the group while the 0^{th} position is an indicator for the complement (shown as '$\sim$'). For example, the vector

$\sim$	x_1	x_2	x_3	x_4	x_5
1	1	0	0	1	1

corresponds to the group $\overline{x_1 \oplus x_4 \oplus x_5}$ since there is a complement and x_1, x_4, x_5 are present. Thus to perform the substitution $x_i \leftarrow x_i \oplus \dot{x}_j$, we first have to check whether x_i is present and then flip the value of x_j. If $\dot{x}_j = \bar{x}_j$ then we also flip the value of the 0^{th} position in the array. It should be clear that each substitution takes $O(1)$ time per clause and requires space $O(n)$. Hence the algorithm is very efficient in practice. Thus,

Theorem 6. *For a formula f with m clauses and n variables, each unit-substitution into f takes time $O(m)$ while the total space required is $O(mn)$.*

Remark 1. As the existence of complement on a substituted variable is equivalent to complementing the entire group of variables (recall XOR Property $\overline{u \oplus v \oplus w} = \overline{u} \oplus v \oplus w = u \oplus \overline{v} \oplus w = u \oplus v \oplus \overline{w}$), it is sufficient to consider substitutions $x_i \to x_i \oplus x_j$ that consist of uncomplemented variables only. Then, once the substitution is made, we just have to flip the complement bit in the group's vector with probability 1/2.

5.2. Flattening

Ideally, we would like to show that when we apply the obfuscation process on two functions f_0, f_1 on n variables which are #SAT-equivalent and have the same size, then $obf(f_0)$ cannot be distinguished from $obf(f_1)$. Although unit-substitutions are powerful enough to generate random looking XOR groups, an extra step is needed to ensure indistinguishablity. This is explained below by way of an example. Consider the formula

$$f = (x_1 + x_2 + x_3)(x_2 + x_4)(x_1 + x_5)$$

and a sequence of substitutions $x_1 \leftarrow x_1 \oplus x_2$, $x_1 \leftarrow x_1 \oplus x_4$, $x_1 \leftarrow x_1 \oplus x_5$ on variable x_1 applied to f. This will create a new formula in which all occurrences of x_1 will be replaced by the same group (shown underlined below):

$$f = (\underline{x_1 \oplus x_2 \oplus x_4 \oplus x_5} + x_2 + x_3)(x_2 + x_4)(\underline{x_1 \oplus x_2 \oplus x_4 \oplus x_5} + x_5)$$

More generally, all occurrences of variable x_i will be replaced by some XOR group g_i after applying a sequence of random substitutions. Thus, our function f will be transformed to

$$obf(f) = (g_1 + g_2 + g_3)(g_2 + g_4)(g_1 + g_5).$$

Hence a simple remapping of the groups $\langle g_1, ..., g_n\rangle$ to some arbitrary variables $\langle y_1, ..., y_n\rangle$ will be enough to reveal the original structure of the formula.

The previous discussion demonstrates that XOR groups originating from the same variable stay the same, hence the structure of the formula can be recovered. In the following, we describe a method (*flattening*) that ensures that all groups (even those that originate from the same variable) stay different. The intuition of this method relies on the following two lemmas.

Definition 5. *Let g_1, g_2 be any two XOR groups. Denote by $\Delta(g_1, g_2)$ the size of the symmetric difference (number of variables they differ) of the two groups.*

Lemma 7. *Consider two different XOR groups g_1, g_2 such that $\Delta(g_1, g_2) \neq 0$. Then after the application of any unit substitution, Δ cannot become zero.*

Proof. Consider a substitution $w \leftarrow w \oplus a$. Clearly, if w belongs to one of the groups, the groups will still be different after the substitution. The same will be true if w belongs to both, but the target variable a to one of them. Finally, if both w and a belong to both groups, a will be removed from both and Δ will not change. Hence in all cases, the symmetric difference cannot become zero. ■

The above lemma suggests that XOR groups that start different, stay different. The next lemma shows that not only these groups will be different, but their expected difference will be $n/2$ after applying random unit substitutions.

Lemma 8. *The expected difference Δ of any two XOR groups in the end of the random walk is $n/2$.*

Proof. This is a consequence of the stationary distribution of the random walk (Equation 6, Appendix A). This is binomial, hence the expected number of variables in any group will be equal to $n/2$. Additionally, it is a simple exercise to verify that the expected difference of two random groups will be $n/2$ since the probability that a literal appears in a group is 1/2. This important lemma has also been verified experimentally; Figure 4 in Section 7 shows the distribution of Δ for various values of n. ■

Lemmas 7, 8 suggest that all we need to prevent the previous de-obfuscation attempt is to make sure that *initially* all XOR groups are different. We will do so by simply introducing *new* variables and clauses that will achieve the desired result. Thus at the end of the random walk, all these groups will be *pairwise* different and contain about $n'/2$ variables each, where n' is the new number of variables. Before we apply this procedure (which we call *flattening*), the following pre-processing step is executed to reduce the extra variables introduced.

5.2.1. Pre-processing. A simple (greedy) way to make initial groups different is to start with the most frequent variable x_i, generate an XOR group g_i consisting of all literals in clauses where x_i appears, replace x_i with $x_i \leftarrow x_i \oplus g_i$, simplify using the properties $P1 : u \oplus v + v = u + v$ and $P2 : u \oplus v + \overline{v} = \overline{u} + \overline{v}$, and repeat this process for the remaining variables until no more groups can be created. As an example, let's apply this process to the same function as before

$$f = (x_1 + x_2 + x_3)(x_2 + x_4)(x_1 + x_5).$$

We start with x_1 and generate $g_1 = x_2 \oplus x_3 \oplus x_5$. Then we substitute it into x_1 (i.e. $x_1 \leftarrow x_1 \oplus g_1$) to obtain $f_1 = f_{x_1 \leftarrow x_1 \oplus g_1}$,

$$\begin{aligned} f_1 = {} & (x_1 \oplus x_2 \oplus x_3 \oplus x_5 + x_2 + x_3)(x_2 + x_4) \\ & (x_1 \oplus x_2 \oplus x_3 \oplus x_5 + x_5) \end{aligned}$$

After applying property $P1$ the formula is simplified to:

$$f_1 = (x_1 \oplus x_5 + x_2 + x_3)(x_2 + x_4)(x_1 \oplus x_2 \oplus x_3 + x_5).$$

Observe that, while initially all occurrences of x_1 were equal to $x_1 \oplus g_1$, after simplification these give rise to two *different* groups in f_1: $g_{11} = x_1 \oplus x_5$ and $g_{12} = x_1 \oplus x_2 \oplus x_3$. Thus, during obfuscation, subsequent unit-substitutions will also produce different groups for these occurrences (recall Lemmas 7, 8).

Then we pick x_2 (second most frequent variable) and generate the group $g_2 = x_1 \oplus x_3 \oplus x_4$ (the group is created based on the *original* function). After substitution into x_2 (i.e. $x_2 \leftarrow x_2 \oplus g_2$), we obtain $f_2 = f_{1, x_2 \leftarrow x_2 \oplus g_2}$:

$$\begin{aligned} f_2 = {} & (x_1 \oplus x_5 + x_2 \oplus g_2 + x_3)(x_2 \oplus g_2 + x_4) \\ & (x_1 \oplus x_2 \oplus g_2 \oplus x_3 + x_5) \\ = {} & (x_1 \oplus x_5 + x_2 \oplus x_1 \oplus x_3 \oplus x_4 + x_3)(x_2 \oplus x_1 \oplus x_3 \oplus x_4 + x_4) \\ & (x_1 \oplus x_2 \oplus x_1 \oplus x_3 \oplus x_4 \oplus x_3 + x_5) \end{aligned}$$

Finally, after applying $P1$ again, we obtain

$$\begin{aligned} f_2 = {} & (x_1 \oplus x_5 + x_2 \oplus x_1 \oplus x_4 + x_3)(x_2 \oplus x_1 \oplus x_3 + x_4) \\ & (x_2 \oplus x_4 + x_5), \end{aligned}$$

in which all the groups differ in at least one literal. The process described above will be applied to all variables in the formula in an effort to make all initial groups look different.

Although promising, this procedure does not fully achieve its goal. We verified experimentally (see Figure 5) that about *half* of the groups will still be the same. However, we can now apply the flattening procedure described next to make all of them different. A better procedure to reduce the initial number of similar groups is left as future work.

5.2.2. Flattening a formula. After pre-processing[2], let $g = l_{i_1} \oplus l_{i_2} \oplus l_{i_3} \oplus \cdots \oplus l_{i_k}$ be some repeating group (two occurrences of g are shown below)

$$f = (g + \ldots + \ldots)\ldots(g + \ldots + \ldots).$$

2. Pre-processing is *optional* – if no pre-processing is performed then flattening is applied to all initial groups which consist of single literals.

We consider the first occurrence of g and we create a new variable which is set equal to a *random* subset of the group literals. For example, we can replace $l_{i_1} \oplus l_{i_3}$ by u_1 in g and add the clause $(u_1 \Leftrightarrow l_{i_1} \oplus l_{i_3})$, which is the same as $(\bar{u}_1 \oplus l_{i_1} \oplus l_{i_3})$ in f. Thus the formula becomes

$$f = (u_1 \oplus l_{i_2} \oplus l_{i_4} \cdots \oplus l_{i_k} + \ldots + \ldots) \ldots (g + \ldots + \ldots) \\ (\bar{u}_1 \oplus l_{i_1} \oplus l_{i_3})$$

Then we pick another subset of literals in g (which might involve the added variable), and set it equal to a new variable. For example, we may choose $u_1 \oplus l_{i_5} \oplus l_{i_8}$ and set it equal to u_2. Then the formula becomes

$$f = (\underline{l_{i_2} \oplus l_{i_4} \oplus u_2 \cdots \oplus l_{i_k}} + \ldots + \ldots) \ldots (g + \ldots + \ldots) \\ (\underline{\bar{u}_1 \oplus l_{i_1} \oplus l_{i_3}})(\underline{\bar{u}_2 \oplus u_1 \oplus l_{i_5} \oplus l_{i_8}}),$$

where the initial group and the added clauses are shown underlined. We repeat this procedure L times (L being a parameter that will be defined shortly), introducing L new variables and extra clauses for the *first* occurrence of the group g. Then we do the same for the second occurrence of the group, introducing *another* L variables $v_1, v_2, \cdots, v_L$ and set them equal to random subsets of the second occurrence of g, and so on.

For a 3-SAT formula with m clauses, there can be a maximum of $3m$ repeating groups, and a total of $3mL$ new variables and clauses introduced (L for each group). To reverse engineer an attacker would have to guess the group positions. In particular, the attacker would have to choose L clauses and XOR them together in the hope to reconstruct the first occurrence of group g (observe how XORing the underlined groups produces g), compare it against the second occurrence of g (which also had to be reconstructed in the same manner), and so on. The amount of work required would be at least $\binom{3mL}{L}^2 \geq (\frac{3mL}{L})^{2L} = (3m)^{2L}$. Thus, setting $L = \log n$ would make flattening secure against any polynomial-time adversary,[3]

The impact of flattening on the running time and resources of the obfuscation algorithm is small. After flattening, the number of variables in a 3-SAT formula becomes $n' = 3m$, while the number of clauses is bounded by $m' = 4m \log n$. Applying Theorem 5 to the flattened formula, we see that polynomial slowdown is still maintained.

5.3. Obfuscation tool

The basic obfuscation tool will apply the results of the previous sections to obfuscate a given 3-CNF function f (see Algorithm 1). Through a series of random unit-substitutions, f will be mapped to a function in the same equivalence class, thus maintaining its original satisfiability.

In more detail, $\mathsf{KeyGen}(1^\lambda)$ will produce a *key* which can be used as a seed to a cryptographically secure random number generator to generate a sequence $S = [\langle w_i, a_i \rangle]$, $i = 1 \ldots N$, of unit substitutions $w_i \to w_i \oplus a_i$ of length N. Alternatively, we can think of the *key* as the set S of random substitutions. The value of N depends on the number of variables of the formula and will be determined in Theorem 9 to guarantee indistinguishability and security.

3. Initially we have chosen $L = 1$, which led to a relatively easy de-obfuscation attack as pointed to us by an astute anonymous reviewer.

To recover the solution of the original function given the solution of $obf(f)$ algorithm $\mathsf{De\text{-}Obfuscate}(key, Sol(obf(f)))$ is used. First, the algorithm checks the validity of the solution, i.e. whether it satisfies $obf(f)$. If not, $\perp$ is returned. Otherwise, using the series of substitutions but in *reverse* order, the algorithm recovers the values of each variable, thus reverting the effect of the substitutions.

The Tseitin encoding [34] mentioned in Algorithm 1 is used to convert the final formula (currently consisting of XOR groups) into an appropriate 3-SAT form. This increases the formula size by a factor of $O(n)$, however, this step is optional. Notice that the Tseitin transform does *not* add to the security of the scheme as it can easily be reverse-engineered. It is merely used to produce a formula in 3-SAT form. We may as well outsource the XOR groups to the cloud solver. In fact, this is exactly what we do in the experimental section using CryptoMiniSat, a state-of-the art solver that directly works with XOR groups.

6. Security Analysis

In this section we analyze the security properties of our scheme. We start by considering the number of unit substitutions required in order to obtain a random XOR group starting from any such one. This will help us later prove the indistinguishability properties of the obfuscator. We also argue about correctness and functionality preservation.

6.1. Mixing time

The proof of the following result can be found in Appendix A.

Theorem 9. *The number of substitutions required in order to obtain a random XOR group is bounded by $\frac{3}{2} n \log n + O(n)$, where n is the number of variables in the formula.*

6.2. Correctness and Soundness

Theorem 10. *Our scheme is a correct and verifiable 3-SAT obfuscation scheme.*

Proof. For any formula f and its obfuscated version $obf(f)$, a satisfying assignment $s = sol(obf(f))$ computed by an honest cloud server can always be verified successfully by the user.

Next, we show that a correctly verified assignment s always corresponds to a satisfying assignment $x = obf^{-1}(s)$ of the original formula f. By way of contradiction, assume this is not the case. Then there is some clause $c = (u + v + w)$ of f which is not satisfied under x. Now, apply to c the same series of substitutions $S = [(x_i, a_i)]$, $i = 1, \ldots, N$, that led to the creation of $obf(f)$. Ignoring the Tseitin encoding, c will be transformed into the following three groups of XORs connected together with OR:

$$(u_1 \oplus \therefore \oplus u_p \; + \; v_1 \oplus \therefore \oplus v_q \; + \; w_1 \oplus \therefore \oplus w_r).$$

As this is part of $obf(f)$, it will be satisfied by s. Hence at least one of the groups, say $u_1 \oplus \therefore \oplus u_p$, will be equal to 1. Now consider the *subsequence* T_u of S that, starting from u, ends up in $u_1 \oplus \therefore \oplus u_p$. Collecting all target variables and substitutions in T, we obtain the combined substitution $u \leftarrow u_1 \oplus \therefore \oplus u_p$. Since $u_1 \oplus \ldots \oplus u_p$ is 1 in $obf(f)$, u will also be 1. Hence clause c should be satisfied as well. ■

6.3. Maintaining functionality

We have defined functionality as SATisfiability of the boolean function. Using Theorem 3, the functionality of the original function is preserved as both f and $obf(f)$ have the same number of satisfying assignments. Additionally, as we showed in the previous theorem, an assignment x for f can be derived from an assignment s of $obf(f)$.

Polynomial slowdown is also achieved since by Theorem 5 and the impact of flattening (Section 5.2), the size of the obfuscated formula is $\tilde{O}(m^2)$. Finally, combining Theorems 6 and 9 along with the new variables and clauses introduced in flattening, the total running time of the algorithm is also $\tilde{O}(m^2)$.

6.4. Indistinguishability

To prove indistinguishability we need to argue that the obfuscated formula produced is a random one in the space of all formulas with the same number of satisfying assignments. The result in Theorem 9 gives the number of substitutions required to create a random XOR group when starting even in a single variable. This result will be leveraged below to show that the resulting formula is also random.

Theorem 11. *For any two #SAT-equivalent Boolean functions f_0, f_1 of the same size, $obf(f_0)$ is indistinguishable from $obf(f_1)$.*

Proof. For the proof we will consider two #SAT-equivalent functions f_0 and f_1 on n variables and m clauses and show that the probability of distinguishing their obfuscated versions is zero.

First notice that after flattening, *both* functions will consist of *exactly* $n' = dm$ variables and $m' = (d+1)m \log n$ clauses (case for d-CNF formulas). As all initial XOR groups in each clause are different, at the end of the random walk, Lemmas 7, 8 will ensure that the resulting groups in the clauses will remain different and on average will consist of $n'/2$ random literals each. Furthermore, at the end of the substitution process, both functions $obf(f_0)$ and $obf(f_1)$ will be members of the same equivalence class $[f_0]$ (Theorem 3). We now have to argue that the resulting formulas are randomly distributed in $[f_0]$. This is needed because the assignments of a formula f may reveal considerable information about f, as explained in the introduction and the definition of privacy. Hence the assignments of $obf(f)$ must be uniformly distributed in the space of solutions. (For ease of exposition, in the remaining section we use n instead of n'.)

Define the n-dimensional hypercube where each node corresponds to a minterm and two nodes are connected by an edge if they differ in a single bit. Thus the hypercube consists of all 2^n minterms. Now consider a minterm m that corresponds to a '1' in the output column of the truth table of the function (a satisfying assignment). By Lemma 2, a unit substitution $w \leftarrow w \oplus a$ will flip variable w in m, if m contains the exact form of literal a. Since the substitution is random and a is a random literal, w will be flipped with probability $1/2$. Thus the '1' corresponding to m will be moved to a neighboring minterm in the hypercube. We can visualize this by having a particle ('1') placed on a node of the hypercube and moving this particle in random locations as dictated by the substitutions. Essentially this corresponds to a lazy random walk, where the particle is moved to a neighboring location with probability $1/2$. The mixing time of this walk is bounded by $\frac{1}{2}nlogn + O(n)$ (see for example Theorem 18.3 in [35]). Since this is *less* than the number of random substitutions (and hence moves) performed by the obfuscation algorithm, the '1' particle will be moved to a random location in the n-dimensional hypercube.

One may be tempted to argue that all particles (i.e. all satisfying assignments of the formula) are randomly distributed in the hypercube as these particles move in parallel with each other. However, this analysis does not exclude the possibility that particles "move in sync", thus they depend on each other.

6.4.1. 2-wise Independence. In the following, we argue that for any two such particles their symmetric difference (number of bits they differ) will be binomially distributed. Hence any two of them will end up in random locations in the hypercube. For a proof let Δ be their symmetric difference and $w \to w \oplus a$ a random substitution. Denote by D the set of positions where these particles are different (hence $|D| = \Delta$) and by I the positions which are identical. If $a \in I$ then no matter what w is, Δ will not change. However, if $a \in D$ then with probability $\frac{\Delta-1}{n-1}$, w also be in D and Δ will decrease by 1. Otherwise, with probability $\frac{n-\Delta}{n-1}$, w will be in I and Δ will increase by 1. Hence the effect on Δ is captured by the following transition probabilities:

$$P(\Delta, j) = \begin{cases} \Delta(n-\Delta)/n(n-1) & \text{if } j = \Delta + 1, \\ \Delta(\Delta-1)/n(n-1) & \text{if } j = \Delta - 1, \\ 1 - \Delta/n & \text{if } j = \Delta. \end{cases} \tag{1}$$

This is exactly the urn experiment described in the proof of Theorem 9 and Δ will be binomially distributed (Equation 6, Appendix A).

Hence the resulting locations of any two particles/satisfying assignments will be random. Typically, this would be sufficient for any real life SAT instance as assignments are hard to find, thus it would be difficult to show dependency of solutions and break the indistinguishability game. However, it is conceivable that in artificial examples this might be possible. Hence in the next section, we describe a more advanced obfuscation scheme that can be applied to achieve complete k-wise independence of assignments using a phase of more general substitutions.

6.4.2. From 2-wise to k-wise independence. Consider an input formula whose assignments are located in arbitrary nodes of the hypercube. We would like to ensure that these assignments are close to being k-wise independent after a sufficient number of substitutions. The question is how many and what type of substitutions are needed.

This is equivalent to asking what is the mixing rate of a random walk on the graph whose nodes consist of k tuples of distinct n-bit strings, and whose edges are induced by substitution operations (see also [36] for a similar approach). We define the δ-closeness of the walk to be the total variation distance between the distribution of k-tuples obtained by the random walk as opposed to the uniform distribution on random k-tuples (see Definition 6). Our main result is given below

Theorem 12. *The number of random substitutions required to make assignments δ-close to k-wise independence, where $\delta \leq 1$, is*

$$O(n \log n (\log \frac{k^2}{\delta})).$$

To determine the mixing rate and show k-wise independence, we leverage 2-way independence as in [36]. We denote by $T(n, k, \delta)$ the mixing time (number of steps/substitutions) required to achieve δ-closeness to k-wise independence. It is known (see for example [35]) that the mixing time satisfies

$$T(n, k, \delta) \leq \log(\frac{1}{\delta}) T(n, k, \frac{1}{4}) \tag{2}$$

To prove Theorem 12, we will consider three phases of substitutions. Phase 1 of the obfuscation process consists of a series of unit-substitutions of length $T(n, 2, \delta/2k^2)$, for $\delta \leq 1$. Hence any *two* assignments of the obfuscated formula will be $\frac{\delta}{2k^2}$-close to pairwise independence. Using Equation (2) and Theorem 9, $O(n \log n \log \frac{k^2}{\delta})$ unit-substitutions are sufficient for this. Next, we consider how we can make any k assignments resulting from Phase 1, k-wise independent.

Let $a = \log(2k^2/\delta)$ and consider the first a bits of each assignment. The probability that there exists a pair of assignments whose first a bits are identical is bounded by

$$\binom{k}{2} \cdot (2^{-a} + \frac{\delta}{2k^2}) \leq \frac{k^2}{2} \cdot (\frac{\delta}{2k^2} + \frac{\delta}{2k^2}) \leq \delta/2.$$

Hence with probability at least $1 - \delta/2$, all these k assignments will differ in their first a bits.

We now begin Phase 2 by performing the following substitutions:

for $i = a + 1$ **to** n **do**

$$\text{Set } x_i \leftarrow x_i \oplus m_{\alpha_1} \oplus m_{\alpha_2} \oplus \cdots \oplus m_{\alpha_t} \tag{3}$$

where each m_{α_j} is a *minterm* on the first a variables appearing with probability 1/2, so on average there are $2^a/2 = k^2/\delta$ of them in (3). The crucial point is that since the first a bits of all assignments are different by Phase 1, the last $n - a$ bits of each assignment will have *uniform* distribution. To see why, consider the i-th bit of an assignment whose first a bits are equal to some $\alpha \in \{0, 1\}^n$. When we apply the substitution, this bit will be flipped only if minterm m_α appears in (3) (by Lemma 2 the other terms will have no effect on the bit). Since this happens with probability 1/2, every such bit will assume a random value *independently* of all the others.

We end the obfuscation by Phase 3 shown below:

for $i = 1$ **to** a **do**

$$\text{Set } x_i \leftarrow x_i \oplus m'_{\alpha_1} \oplus m'_{\alpha_2} \oplus \cdots \oplus m'_{\alpha_t} \tag{4}$$

where again each m'_{α_j} is a minterm on the *last* a variables appearing with probability 1/2, so on average there are $2^a/2 = k^2/\delta$ of them in (4).

As the last $n - a$ bits of each assignment are uniformly distributed by Phase 2, the probability that the last a bits are not distinct is bounded by $\binom{k}{2} 2^{-a} \leq \frac{k^2}{2} \frac{\delta}{2k^2} \leq \frac{\delta}{4}$. Hence with probability at least $1 - \delta/2 - \delta/4 > 1 - \delta$ and using the properties of substitutions as before, the first a bits will also have a uniform distribution. Thus any k assignments of the obfuscated function will be δ-close to be uniformly distributed in $[f]$. ■

The algorithm described above has to include a last flattening (recall Section 5.2) and random walk phase (Phase 4) as the variables appearing in Equations 3 and 4 have a very special structure and can easily be recovered. However, unless this is done carefully, the resulting formula size will increase by a factor of $\Omega(k^4 \log^2 k)$. To this end, the following additional transformations need to be considered.

- In Equation 3, we set $x_i \leftarrow x_i \oplus w_i$, where each w_i is a new variable equal to $y_{\alpha_1} \oplus y_{\alpha_2} \oplus \cdots \oplus y_{\alpha_t}$ and each y_{α_j} is equal to the corresponding minterm on the first a variables. Thus at this point, the obfuscated formula is expanded with $n - a$ clauses of the form $(w_i \iff y_{\alpha_1} \oplus y_{\alpha_2} \oplus \cdots \oplus y_{\alpha_t})$ and another 2^a clauses of the form $(y_{\alpha_j} \iff m_{\alpha_j})$.
- Similarly, in Equation 4, we set $x_i \leftarrow x_i \oplus v_i$, where each v_i is a new variable equal to $y'_{\alpha_1} \oplus y'_{\alpha_2} \oplus \cdots \oplus y'_{\alpha_t}$ and each y'_{α_j} is equal to the corresponding minterm on the last a variables. Thus the obfuscated formula is expanded with a more clauses of the form $(v_i \iff y'_{\alpha_1} \oplus y'_{\alpha_2} \oplus \cdots \oplus y'_{\alpha_t})$ and another 2^a clauses of the form $(y'_{\alpha_j} \iff m'_{\alpha_j})$.

This results in a formula with $C = O(m + k^2 \log k)$ clauses and $V = O(m + k^2 \log k)$ variables. When flattening and the final set of $O(VlogV)$ unit substitutions is performed (Phase 4), the size of the formula becomes $O(CV)$ due to the creation of XOR groups of size $O(V)$ each. Hence the running time of the advanced algorithm becomes $O(CVlogV)$, compared to $O(m^2 logm)$ of the basic scheme.

An interesting research question here is whether a substitution mechanism exists that can eliminate this dependency on k. This would result in an obfuscator that uniformly distributes the assignments of a formula, irrespective of their number.

6.5. Discussion

The results of the previous section suggest that if a formula f has k assignments, $obf(f)$ will be a random formula in $[f]$. Hence if k is polynomial, the algorithm remains polynomial as well. At the same time, ensuring k-wise independence imposes an overhead on the running time and length of the obfuscated formulas, so one might ask, what are the benefits of using the more advanced scheme in practical situations?

Arguably, the benefits are not that many. Outsourcing a formula f to the cloud makes sense when f is hard, hence finding even a single satisfying assignment is very difficult, let alone finding a large number of them in order to study dependency of solutions – in this case the exponential behavior of the problem would definitely show up. Hence, for all real life applications the basic scheme can be used. Alternatively k can be selected to be constant, or larger than the expected number of assignments that can be found by the solver within a reasonable (polynomial) amount of time since enumerating all assignments is a #P-complete problem and hence intractable. For a concrete example, consider Alice who wants to factor an RSA modulus and models this as a SAT instance to be outsourced. As this instance has only two solutions (the primes $\langle p, q\rangle$ or $\langle q, p\rangle$ that make up N), the basic scheme can be used. In conclusion, although other interesting problems modelled by SAT may not necessarily have a small total number of satisfying assignments, they can only have a small number of assignments discovered by a polynomial time solver.

In the next section, we study the performance of the basic scheme.

7. Performance

In this section we study the various stages of the obfuscation process (Algorithm 1). The validation of theoretical parameters (length of random walk, symmetric differences of groups, etc.) can be found in Appendix B. All experiments were performed on an Intel core i7-6700HQ CPU @ 2.60GHz with 16GB of memory. CryptoMiniSat [37] was used to compare the solving time between the obfuscated formula and the original one for the formulas contained in the SATlib benchmark [38].

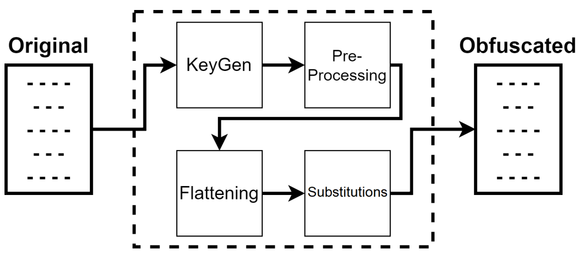

Figure 2: Flow of the algorithm

7.1. Time analysis of various stages

In this section we measure the time needed for each stage of the algorithm to complete. The stages include: i) pre-processing, ii) flattening, and iii) application of random unit substitutions. We have used the Satlib benchmark [38] which contains the following types of formulas: (i) Uniform Random (*uf*) SAT which consists of 3700 hard satisfiable instances of uniformly generated 3-CNF formulas with clauses-to-variables ratio near the phase transition $(m/n = 4.25)$, (ii) Large random (LRAN) which consists of large uniformly generated formulas at the phase transition, and (iii) Logistics Planning which contains instances generated from encoding logistics problems (scheduling the delivery of packages between locations without exceeding some cost L) into SAT. Table 1 presents our findings along with the sizes (n, m) and (n', m') of the original and obfuscated formulas, respectively. As it can be seen, formulas with thousands of variables can be obfuscated. The dominant factor is the number of unit substitutions which depends on the number of extra variables introduced due to flattening. Hence a better method to ensure dissimilarity of initial groups will have a strong positive impact in the obfuscation process.

7.2. User efficiency

In this section we consider the time required to solve an obfuscated formula vs. the original formula f. We also study the time needed to recover the solution to f from the solution of $obf(f)$.

Here we used CryptoMiniSAT [37], a SAT solver that is specialized in solving cryptographic problems. It can accept SAT formulas in ordinary CNF but also formulas containing XOR relations (which are typical of stream ciphers – hence its use in breaking cryprographic algorithms). This characteristic of CryptoMiniSAT makes it ideal in our case as ordinary SAT solvers break down (take considerable more time) when trying to solve them.[4]

Table 2 shows a comparison between the time (in ms) to solve original instances contained in the SAT benchmark [38] as opposed to the time needed to solve their obfuscated variants. We first solved all the instances in each *uf* category (each containing hundreds of formulas) and found the median formula f_m to reduce variability of solving times. Then we obfuscated f_m many times and we depicted the average solving time. The table also shows the ratio between the two times. There are no safe conclusions that can be drawn from this as we are dealing with random formulas and the hardness of both the original and the obfuscated formulas can clearly influence solution times. However, we believe that cloud solvers with lots of computational power can easily handle the obfuscation times shown. Furthermore, this variability in solving times is basically caused by the increased number of clauses and variables in the obfuscated instance. Hence a better flattening/preprocessing procedure can reduce this number and thus improve solution time even further. We leave this as an important future research direction.

Remark 2. *Comparing the total time for obfuscation (preprocessing + flattening + unit subs) in Table 1 with the time to solve the original formula in Table 2, one may conclude that the burden on the users for obfuscation is higher than solving the formulas themselves. Hence outsourcing seems pointless. However, this conclusion is erroneous because obfuscation time (and hence outsourcing) is always bounded by a polynomial while solving a*

4. Typically, there are two traditional SAT-solving paradigms: i) the *portfolio*, and ii) the *Divide & Conquer* approach. In the portfolio approach. a collection of SAT solvers is applied in the same instance. All solvers run in parallel, terminating when one finds a satisfying assignment. Typical solvers in this category include ManySAT [39], CryptoMiniSAT, HordeSAT [40], to name a few. D&C solvers on the other hand, solve an instance by partitioning the search space or by reducing the original formula into several sub-formulas which then try to solve in parallel. Example solvers in this category include Cube & Conquer [41], Paracooba [42], ggSAT [43], etc.

A cloud solver may have developed its own software to solve SAT instances or use one or more of the techniques mentioned above. Affording massive parallelism and computing resources may offer an increased speedup in solving obfuscated instances [14]. Here we are using CryptoMiniSAT in a single machine with moderate capabilities which may have an impact on the number of variables (and time) we can handle to solve obfuscated instances.

TABLE 1: Problem size and stages analysis (time in ms)

Problem set	Preprocessing	Flattening	Unit subs	n	m	n'	m'	Recovery	Verification
uf20	7	0	134	20	91	180	251	2	≈ 0
uf50	59	1	681	50	218	387	555	13	≈ 0
uf75	160	2	1308	75	325	556	806	29	≈ 0
uf100	365	3	2455	100	430	716	1046	60	≈ 0
uf125	618	4	3882	125	538	885	1298	121	1
uf150	1064	6	5871	150	645	1045	1540	210	1
uf175	1641	9	8544	175	753	1211	1789	317	2
uf200	2486	12	11751	200	860	1368	2028	388	2
uf225	3388	13	15601	225	960	1514	2249	632	3
uf250	4317	15	20800	250	1065	1671	2486	911	4
LRAN-f600	45983	56	105040	600	2550	3773	5723	802	18
LRAN-f1000	215368	219	450880	1000	4250	6164	9414	3774	53
LRAN-f2000	1526537	546	2543397	2000	8500	12116	18616	28699	202
logistics.a	104450	233	2222166	828	6718	12119	18009	12357	55
logistics.b	106509	343	2592536	843	7301	12987	19445	13166	64
logistics.c	407769	491	5999134	1141	10719	19263	28841	39374	146
logistics.d	587699	2327	17564516	4713	21991	34249	51527	551891	977

TABLE 2: Comparison between original and obfuscated solving times (time in ms)

Problem set	Original	Obfuscated	Ratio
uf20	≈ 0	72	-
uf50	≈ 0	531.5	-
uf75	≈ 0	1134.5	-
uf100	≈ 0	2388	-
uf125	12	4440	370
uf150	49	8126.5	166
uf175	216.5	13826.5	64
uf200	259.5	17051.5	66
uf225	2905	57885.5	20
uf250	1356.5	753723	556

SAT formula depends on the hardness of the particular instance and may require exponential time in the worst case. Furthermore, the obfuscation time can be dropped using a better prepossessing or flattening phase, thus reducing considerably the number of random substitutions required.

Next, we consider the time needed to completely recover the solutions of the original formulas from the solutions of the obfuscated ones (De-obfuscation). The time is broken down into the following parts:

- *Recovery*, which denotes the time needed to recover the values of the original variables. This is done by reversing the unit substitution steps and expressing each variable as the XOR of the obfuscated ones.
- *Solution verification*, which denotes the time needed to verify the validity of the recovered solution.

The time needed for each part is presented in the last two columns of Table 1, measured in milliseconds. Recovery time is bounded by $O(n^2)$ as each variable is the XOR of $O(n)$ other variables while that for verification is $O(m)$. We should stress here that both recovery time and solution verification are *independent* of the hardness of the formula, they require polynomial time in the number of variables and clauses, and thus can easily be handled by user machines. This further shows the practicality of our approach.

8. Comparison with VBB Obfuscation [20]

We conclude this work by attempting a comparison with the framework developed by Bartusek et al. [20] for obfuscating conjunctions. A conjunction is any Boolean function $f(x_1, \dots, x_n) = \bigwedge_{i \in S} l_i$, for some $S \subseteq [n]$, where each l_i can be x_i or $\bar{x}_i$. This is similar to looking for an input string $x \in \{0,1\}^n$ that matches a pattern $pat \in \{0,1,\star\}^n$ in all non-wildcard positions ($\star$ denotes a wild-card). For example $x = 1010$ matches $pat = \star 01 \star$ but not $pat = 1 \star\star 1$. This problem has interesting applications in hiding secrets inside programs and the authors were able to guarantee distributional virtual black-box obfuscation (VBB). However, as was demonstrated in [21], implementation under Entropic Ring LWE of the work in [44] is not straightforward. Securely obfuscating 64-bit conjunction programs required major design and system-level advances and many hours of processing, resulting in obfuscated program sizes of about 750GB.

Unfortunately, conjunction obfuscation is not interesting from an outsourcing/satisfiability point of view which is the focus of this work. Given a function $f(x_1, \dots, x_n) = \bigwedge_{i \in S} l_i$, it is straightforward to find a satisfying assignment, hence there is no need for outsourcing f to a cloud solver. Furthermore, it would not be secure for the reasons explained in Section 3.3. However, for the purpose of exposition we will describe how conjunctions can be outsourced in a way that the underlying pattern can only be discovered with negligible probability. Hence, in this case we will consider obfuscation to be broken if the indices of the literals or the positions of the wildcards are discovered by an attacker.

In the following, we will adapt the mechanisms (unit substitutions and flattening) we developed for general 3-SAT formulas for the case of simple conjunctions. So, let $f = x_{i_1} \dots x_{i_k}$, where k is the number of variables appearing in the conjunction (wlog assume these are not negated) and $n - k$ is the number of wildcards. Let's consider first the extreme case where $f = x_{i_1}$.

After applying unit substitutions, f will be obfuscated to a single XOR group g, which hides the original variable. Unfortunately, this is not enough as we can just guess the index of the original variable with probability $1/n$. This improves if the number k of variables appearing in the conjunction increases, however our goal is to make this *independent* of k. Another source of leakage is the number of satisfying assignments. As our framework maintains the number of assignments between the original formula and the obfuscated one, a solver that discovers all assignments can immediately deduce the number of wildcards in the conjunction.

Now consider the general case $f = x_{i_1} \dots x_{i_k}$. We will show how to obfuscate this by a series of transformations as follows:

$$
\begin{aligned}
f &= x_{i_1} ... x_{i_k} \\
&= (x_{i_1} + x_{i_1}) ... (x_{i_k} + x_{i_k})(x_{j_1} + \bar{x}_{j_1}) ... (x_{j_{n-k}} + \bar{x}_{j_{n-k}})
\end{aligned}
$$

where in the second step we substituted each x_{i_l} with a clause $(x_{i_l} + x_{i_l})$ and introduced extra clauses for the missing variables x_{i_j}. Now we can apply flattening to rename the second occurrence of each variable. Thus f becomes

$$
\begin{aligned}
f = {} & (x_{i_1} + u_{i_1}) ... (x_{i_k} + u_{i_k})(x_{j_1} + \bar{u}_{j_1}) ... (x_{j_{n-k}} + \bar{u}_{j_{n-k}}) \\
& (x_1 \oplus \bar{u}_1) ... (x_n \oplus \bar{u}_n)
\end{aligned}
$$

Once we apply unit substitutions, all groups will be different (recall Lemma 8) and will be binomially distributed irrespective of the starting variable. Furthermore, the obfuscated formula will consist of $2n$ clauses, n of which are single groups and another n are clauses of two groups ORed together. Thus no information can be deduced from the structure of f about the indices of the original variables. Notice, however, that the number of assignments is still 2^{n-k} (despite the addition of new variables). Assuming that a malicious solver can discover all of them in time significantly less than $O(2^{n-k})$, the probability of deducing the original variables becomes $1/\binom{n}{k}$.

The final corrective step is to add some extra clauses on new variables so that the number of assignments is independent of k. The clauses to be added will be equal to

$$
y_{j_1} \dots y_{j_{n-k}} (y_{i_1} + \bar{y}_{i_1}) \dots (y_{i_k} + \bar{y}_{i_k}),
$$

thus the number of satisfying assignments will be multiplied by 2^k, for a total of 2^n. Flattening the new variables, we obtain the final obfuscated formula:

$$
\begin{aligned}
& (x_{i_1} + u_{i_1}) \dots (x_{i_k} + u_{i_k})(x_{j_1} + \bar{u}_{j_1}) \dots (x_{j_{n-k}} + \bar{u}_{j_{n-k}}) \\
& (y_{j_1} + v_{j_1}) \dots (y_{j_{n-k}} + v_{j_{n-k}})(y_{i_1} + \bar{v}_{i_1}) \dots (y_{i_k} + \bar{v}_{i_k}) \\
& (x_1 \oplus \bar{u}_1) ... (x_n \oplus \bar{u}_n) \\
& (y_1 \oplus \bar{v}_1) \dots (y_n \oplus \bar{y}_n)
\end{aligned}
$$

Once we apply unit substitutions, all groups will be different and randomly distributed, the obfuscated formula will consist of $4n$ clauses ($2n$ single groups and another $2n$ groups of two), and the total number of assignments will be 2^n. As there is no dependency on k any more, the probability of guessing the indices of the original variables is the same as picking a random subset out of n elements (but not the empty set). Thus the probability of a successful attack is

$$
\frac{1}{(2^n - 1)}.
$$

This analysis shows that our conjunction obfuscation does not require the pattern to have high entropy to be secure as is the case for the VBB obfuscators.

9. Conclusions

Outsourcing SAT computations to cloud solvers is necessary in order to deal with the complexity of real world problems modeled by large SAT formulas. However, naive outsourcing may leak sensitive information and put the user's data at risk. Existing techniques based on program obfuscation primitives [44] have an extremely large overhead hence they are not considered practical. Furthermore they are not suitable for outsourcing (see also previous discussion).

In this work we presented a formal framework to obfuscate SAT formulas prior to outsourcing them to the cloud. At the heart of our approach lies a random walk in the space of solutions which is implemented with simple logical operations on Boolean formulas. The rapid mixing of the random walk ensures the polynomial character of our framework and the creation of obfuscated formulas that are within a factor m of the original ones. Most importantly, our framework, which resembles Indistinguishability Obfuscation, maintains satisfiability and guarantees that obfuscated formulas remain indistinguishable and secure against de-obfuscation attacks without relying on any hard problem or cryptographic assumption.

Experimental evaluation shows that the overhead added by our SAT obfuscator is within the practical abilities of cloud solvers while recovering the original solution from the obfuscated one is a very simple and straightforward process. An important consequence of our work is that all problems in NP can be outsourced to the cloud by way of reducing to SAT. This further ensures the usefulness and wide applicability of our approach.

Acknowledgements

The authors would like to thank Evita Nestoridi, Andrea Collevecchio, Timothy Gowers and Alistair Sinclair for useful discussions about mixing time of random walks.

References

[1] C. Gentry and D. Boneh, *A fully homomorphic encryption scheme*. Stanford university Stanford, 2009, vol. 20, no. 9.

[2] P. Paillier, "Public-key cryptosystems based on composite degree residuosity classes," in *International conference on the theory and applications of cryptographic techniques*. Springer, 1999, pp. 223–238.

[3] F. Armknecht and A.-R. Sadeghi, "A new approach for algebraically homomorphic encryption." *IACR Cryptol. ePrint Arch.*, vol. 2008, p. 422, 2008.

[4] C. Gentry and S. Halevi, "Implementing gentry's fully-homomorphic encryption scheme," in *Annual international conference on the theory and applications of cryptographic techniques*. Springer, 2011, pp. 129–148.

[5] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(leveled) fully homomorphic encryption without bootstrapping," *ACM Transactions on Computation Theory (TOCT)*, vol. 6, no. 3, pp. 1–36, 2014.

[6] Y. Yang, X. Huang, X. Liu, H. Cheng, J. Weng, X. Luo, and V. Chang, "A comprehensive survey on secure outsourced computation and its applications," *IEEE Access*, vol. 7, pp. 159 426–159 465, 2019.

[7] D. Benjamin and M. J. Atallah, "Private and cheating-free outsourcing of algebraic computations," in *2008 Sixth Annual Conference on Privacy, Security and Trust*. IEEE, 2008, pp. 240–245.

[8] Y. Zhang and M. Blanton, "Efficient secure and verifiable outsourcing of matrix multiplications," in *International Conference on Information Security*. Springer, 2014, pp. 158–178.

[9] P. Mohassel, "Efficient and secure delegation of linear algebra." *IACR Cryptol. ePrint Arch.*, vol. 2011, p. 605, 2011.

[10] C. Wang, K. Ren, J. Wang, and Q. Wang, "Harnessing the cloud for securely outsourcing large-scale systems of linear equations," *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 6, pp. 1172–1181, 2012.

[11] C. Wang, K. Ren, and J. Wang, "Secure and practical outsourcing of linear programming in cloud computing," in *2011 Proceedings Ieee Infocom*. IEEE, 2011, pp. 820–828.

[12] Z. Shan, K. Ren, M. Blanton, and C. Wang, "Practical secure computation outsourcing: A survey," *ACM Computing Surveys (CSUR)*, vol. 51, no. 2, pp. 1–40, 2018.

[13] J. Marques-Silva, "Practical applications of boolean satisfiability," in *2008 9th International Workshop on Discrete Event Systems*. IEEE, 2008, pp. 74–80.

[14] T. Balyo, N. Froleyks, M. J. Heule, M. Iser, M. Järvisalo, and M. Suda, "Proceedings of sat competition 2020: Solver and benchmark descriptions," 2020.

[15] R. Ostrowski, É. Grégoire, B. Mazure, and L. Sais, "Recovering and exploiting structural knowledge from cnf formulas," in *International Conference on Principles and Practice of Constraint Programming*. Springer, 2002, pp. 185–199.

[16] J. A. Roy, I. L. Markov, and V. Bertacco, "Restoring circuit structure from sat instances," in *proceedings of international workshop on Logic and synthesis*. Citeseer, 2004, pp. 663–678.

[17] Z. Fu and S. Malik, "Extracting logic circuit structure from conjunctive normal form descriptions," in *20th International Conference on VLSI Design held jointly with 6th International Conference on Embedded Systems (VLSID'07)*. IEEE, 2007, pp. 37–42.

[18] A. Biere, D. Le Berre, E. Lonca, and N. Manthey, "Detecting cardinality constraints in cnf," in *International Conference on Theory and Applications of Satisfiability Testing*. Springer, 2014, pp. 285–301.

[19] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang, "On the (im)possibility of obfuscating programs," in *Annual International Cryptology Conference*. Springer, 2001, pp. 1–18.

[20] J. Bartusek, T. Lepoint, F. Ma, and M. Zhandry, "New techniques for obfuscating conjunctions," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2019, pp. 636–666.

[21] D. B. Cousins, G. Di Crescenzo, K. D. Gür, K. King, Y. Polyakov, K. Rohloff, G. W. Ryan, and E. Savas, "Implementing conjunction obfuscation under entropic ring lwe," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 354–371.

[22] R. S. Chakraborty and S. Bhunia, "Harpoon: An obfuscation-based soc design methodology for hardware protection," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 10, pp. 1493–1502, 2009.

[23] M. Yasin, J. J. Rajendran, O. Sinanoglu, and R. Karri, "On improving the security of logic locking," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 9, pp. 1411–1424, 2015.

[24] P. Subramanyan, S. Ray, and S. Malik, "Evaluating the security of logic encryption algorithms," in *2015 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, 2015, pp. 137–143.

[25] M. Yasin, B. Mazumdar, J. J. Rajendran, and O. Sinanoglu, "Sarlock: Sat attack resistant logic locking," in *2016 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, 2016, pp. 236–241.

[26] K. Shamsi, M. Li, T. Meade, Z. Zhao, D. Z. Pan, and Y. Jin, "Appsat: Approximately deobfuscating integrated circuits," in *2017 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, 2017, pp. 95–100.

[27] K. Zamiri Azar, H. Mardani Kamali, H. Homayoun, and A. Sasan, "Threats on logic locking: A decade later," in *Proceedings of the 2019 on Great Lakes Symposium on VLSI*, 2019, pp. 471–476.

[28] Y. Brun and N. Medvidovic, "Keeping data private while computing in the cloud," in *2012 IEEE Fifth International Conference on Cloud Computing*. IEEE, 2012, pp. 285–294.

[29] Y. Qin, S. Shen, and Y. Jia, "Structure-aware cnf obfuscation for privacy-preserving sat solving," in *2014 Twelfth ACM/IEEE Conference on Formal Methods and Models for Codesign (MEMOCODE)*. IEEE, 2014, pp. 84–93.

[30] K. Alhamdan, T. Dimitriou, and I. Ahmad, "Attack on a scheme for obfuscating and outsourcing SAT computations to the cloud," in *Proceedings of the 16th International Joint Conference on e-Business and Telecommunications, ICETE 2019 - Volume 2: SECRYPT, Prague, Czech Republic, July 26-28, 2019*.

[31] S. Garg, C. Gentry, S. Halevi, M. Raykova, A. Sahai, and B. Waters, "Candidate indistinguishability obfuscation and functional encryption for all circuits," *SIAM Journal on Computing*, vol. 45, no. 3, pp. 882–929, 2016.

[32] A. Sahai and B. Waters, "How to use indistinguishability obfuscation: deniable encryption, and more," in *Proceedings of the forty-sixth annual ACM symposium on Theory of computing*, 2014, pp. 475–484.

[33] M. Horváth and L. Buttyán, *Cryptographic Obfuscation: A Survey*. Springer, 2020.

[34] A. Biere, M. Heule, and H. van Maaren, *Handbook of satisfiability*. IOS press, 2009, vol. 185.

[35] D. A. Levin and Y. Peres, *Markov chains and mixing times*. American Mathematical Soc., 2017, vol. 107. [Online]. Available: http://pages.uoregon.edu/dlevin/MARKOV/

[36] S. Hoory, A. Magen, S. Myers, and C. Rackoff, "Simple permutations mix well," *Theoretical computer science*, vol. 348, no. 2-3, pp. 251–261, 2005.

[37] M. Soos, K. Nohl, and C. Castelluccia, "Extending SAT solvers to cryptographic problems," in *Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings*, ser. Lecture Notes in Computer Science, O. Kullmann, Ed., vol. 5584. Springer, 2009, pp. 244–257. [Online]. Available: https://doi.org/10.1007/978-3-642-02777-2_24

[38] H. H. Hoos and T. Stützle, "Satlib: An online resource for research on sat," *Sat*, vol. 2000, pp. 283–292, 2000.

[39] Y. Hamadi, S. Jabbour, and L. Sais, "Manysat: a parallel sat solver," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 6, no. 4, pp. 245–262, 2010.

[40] T. Balyo, P. Sanders, and C. Sinz, "Hordesat: A massively parallel portfolio sat solver," in *Theory and Applications of Satisfiability Testing–SAT 2015: 18th International Conference, Austin, TX, USA, September 24-27, 2015, Proceedings 18*. Springer, 2015, pp. 156–172.

[41] M. J. Heule, O. Kullmann, S. Wieringa, and A. Biere, "Cube and conquer: Guiding cdcl sat solvers by lookaheads," in *Hardware and Software: Verification and Testing: 7th International Haifa Verification Conference, HVC 2011, Haifa, Israel, December 6-8, 2011, Revised Selected Papers 7*. Springer, 2012, pp. 50–65.

[42] M. Heisinger, M. Fleury, and A. Biere, "Distributed cube and conquer with paracooba," in *Theory and Applications of Satisfiability Testing–SAT 2020: 23rd International Conference, Alghero, Italy, July 3–10, 2020, Proceedings 23*. Springer, 2020, pp. 114–122.

[43] A. Ozdemir, H. Wu, and C. Barrett, "Sat solving in the serverless cloud," in *2021 Formal Methods in Computer Aided Design (FMCAD)*. IEEE, 2021, pp. 241–245.

[44] Z. Brakerski, V. Vaikuntanathan, H. Wee, and D. Wichs, "Obfuscating conjunctions under entropic ring lwe," in *Proceedings of the 2016 ACM Conference on Innovations in Theoretical Computer Science*, 2016, pp. 147–156.

[45] A. Ben-Hamou, Y. Peres *et al.*, "Cutoff for a stratified random walk on the hypercube," *Electronic Communications in Probability*, vol. 23, 2018.

Appendix A. Proof of mixing time – Theorem 9

Recall that *each* group inside a clause can be represented with a $(n+1)$-bit vector, where a single bit corresponds to a complement operation while the rest correspond to the n variables of the formula. As the complement bit can be set independently in the end of the substitution process, here we will concentrate on the n remaining bits.

The substitution procedure can be thought as a walk in a n-dimensional hypercube whose nodes are tuples in $\{0,1\}^n$. Two such nodes are connected by an edge if they differ in exactly one coordinate. A substitution $x \leftarrow x \oplus a$, where both x, a are single variables, moves from a node $\{x_1, \ldots, x_j, \ldots, x_n\}$ to $\{x_1, \ldots, 1-x_j, \ldots, x_n\}$, iff the bit corresponding to x is already set and $x_j = a$ (this is equivalent to testing if x is present in the group and then flipping the bit corresponding to a). Our goal next is to analyze such a walk. In particular, we want to answer the question: *how many steps (substitutions) are needed in order to end up in a random node (XOR group) starting from any initial node (XOR group) in the hypercube?*

We now consider a process (a variant of the *Ehrenfest urn*) which appears quite different from the above walk. Suppose n numbered balls are distributed between two urns, 1 and 0, where Urn 1 is never allowed to become empty. At each move, a ball is selected uniformly at random. If the ball belongs to Urn 1 then we pick one of the *remaining* $n-1$ balls and transfer it from its current urn to the other urn. If W_t is the number of balls in Urn 1 at time t, then the (single step) transition probability matrix for W_t is given by

$$P_W(i,j) = \begin{cases} i(n-i)/n(n-1) & \text{if } j = i+1, \\ i(i-1)/n(n-1) & \text{if } j = i-1, \\ 1 - i/n & \text{if } j = i. \end{cases} \tag{5}$$

This process captures the mechanics of substitutions. Consider for example a node in the hypercube consisting of i ones (i.e. i of the variables have been set). A random substitution $x \leftarrow x \oplus a$ will force a move to a node with $i+1$ ones if x is one of the i variables already set (probability i/n) *and* a is one of the remaining variables (probability $(n-i)/(n-1)$). Similarly, a move to a node with $i-1$ ones will take place if a is one of $i-1$ variables that have already been set (x is excluded because the substitution $x \leftarrow x \oplus a$, for $a = x$, is not allowed as this would eliminate variables in a clause).

Thus W_t is a Markov chain with state space $\{1, 2, \ldots, n\}$ that either moves by ± 1 or stays put according to probabilities $P_W(i,j)$. The distribution of balls after t moves, where $t \to \infty$, is called the *stationary distribution* π_W and satisfies the equation $\pi_W = \pi_W P$. As it turns out, the stationary distribution for the above chain is binomial and is given by the expression

$$\pi_W(i) = \frac{\binom{n}{i}}{2^n - 1}. \tag{6}$$

The urn chain is essentially a projection of the hypercube random walk on the numbers $\{1, 2, \ldots, n\}$. This is unsurprising given the standard bijection between $\{0,1\}^n$ and subsets of a set with n elements. The term $2^n - 1$ in the denominator simply accounts for the fact that we are never allowed to visit node $\{0, 0, \ldots, 0\}$ in the hypercube as this would result in an empty group.

Checking whether Equation 6 satisfies $\pi_W = \pi_W P$ is cumbersome, however it is enough to verify whether π_W satisfies the *detailed balance* equations

$$\pi_W(i) P_W(i,j) = \pi_W(j) P_W(j,i), \; \forall i,j \in \{1, 2 \ldots, n\}$$

as it is known that any distribution satisfying the balance equations is stationary for P_W (see for example Proposition 1.20 from [35]). This is clearly the case here hence π_W is stationary.

Our goal in the following would be to quantify exactly the number of steps t required for the hypercube random walk to converge to its stationary distribution. It turns out that all we have to do is study the same question for the W_t Markov chain. If $X_t = \{x_1^t, \ldots, x_n^t\}$ is the position of the random walk in the hypercube at time t then set $W_t = \sum_{i=1}^n x_i^t$ equal to the Hamming weight $W(X_t)$ of the vector X_t. Clearly, this is the Markov chain defined on the urns. The bijection between numbered balls and n-bit vectors allows us to reduce the study of X_t to the study of W_t. Hence bounding the maximal distance $d(t)$ between the t-step probability distribution P_W^t and π_W (the stationary distribution of W_t) is our primary objective.

Definition 6. *The total variation distance between two distributions μ, ν on space $\mathcal{X}$ is given by*

$$||\mu - \nu||_{TV} = \frac{1}{2} \sum_{x \in \mathcal{X}} |\mu(x) - \nu(x)| \tag{7}$$

We define $d(t) = ||P_W^t - \pi_W||_{TV}$ as the total variation distance between the two distributions on the urn, and the mixing time t_{mix} as

$$\tau_{mix} := min_t\{d(t) < \epsilon\}, \tag{8}$$

for some constant ϵ (typically $\epsilon = 1/4$). The following lemma explains why the study of the hypercube walk can be reduced to the study of the urn chain: the t-step distributions of both have the same variation distance.

Lemma 13. *Let $X_t = \{x_1^t, \ldots, x_n^t\}$ be the position of the random walk in the hypercube at time t, and let $W_t = \sum_{i=1}^n x_i^t$. Then*

$$||P_{H,\mathbf{1}}^t - \pi_H||_{TV} = ||P_{W,n}^t - \pi_W||_{TV},$$

where $P_{H,\mathbf{1}}^t$ denotes the t-step distribution the hypercube starting from a vertex with all ones, and $P_{W,n}^t$ denotes the t-step distribution of the urn chain starting with all the balls in Urn 1.

Proof.

$$\begin{aligned}
||P^t_{H,\mathbf{1}} - \pi_H||_{TV} &= \frac{1}{2}\sum_x |P^t_{H,\mathbf{1}}(x) - \pi_H(x)| \\
&= \frac{1}{2}\sum_w \sum_{\substack{x \\ W(x)=w}} |P^t_{H,\mathbf{1}}(x) - \pi_H(x)| \\
&= \frac{1}{2}\sum_w \left| \sum_{\substack{x \\ W(x)=w}} P^t_{H,\mathbf{1}}(x) - \pi_H(x) \right| \\
&= \frac{1}{2}\sum_w |P^t_{W,n}(w) - \pi_W| \\
&= ||P^t_{W,n} - \pi_W||_{TV}
\end{aligned}$$

■

In a recent result, Ben-Hamu and Peres [45] have studied a different walk on the hypercube whose transition probability exhibits the same behavior as $W(t)$. They were able to show that the chain W_t exhibits a sharp cutoff at $\frac{3}{2}nlogn$ with a window of $O(n)$. This means that τ_{mix} is bounded by $\frac{3}{2}nlogn + O(n)$. Combining this with Lemma 13, we conclude that the hypercube walk converges to stationarity equally fast. Hence, the number of unit-substitutions required in order to obtain a random group is bounded by $\frac{3}{2}nlogn + O(n)$.

Experimental validation of this theoretical result is demonstrated in Appendix B.

Appendix B. Experimental validation of theoretical parameters

In Theorem 9, we have established that the number of substitutions required is $\frac{3}{2}nlogn + O(n)$. In this section, we start by first validating the length of the random walk then we study the effects of flattening and preprocessing as applied to SAT formulas of [38].

B.1. Length of random walk

An alternative characterization of the mixing time in a hypercube is the first time when all coordinates have been updated. In our case this translates to starting with a group of size 1 and performing N unit substitutions until all group positions have been updated. Our goal is to show that N matches the theoretical value $\frac{3}{2}nlogn + O(n)$ determined in Theorem 9.

This is depicted in Figure 3. For all values of $n =$ 1000 to 30000, we started with a group of size 1 and we marked the first time all group positions have been updated through a unit substitution. This corresponds to a *strong stationary time* which constitutes a bound on the mixing time of the random walk [35]. As it can be seen in the figure, the result of the experiment matches the theoretical result $\frac{3}{2}nlogn + O(n)$ found in Theorem 9. The value of the constant hidden in the linear term was also found experimentally to be approximately equal to 11. Hence, in all experiments, the length of the random walk was set to $\frac{3}{2}nlogn + 11n$, where n is the number of variables in the formula.

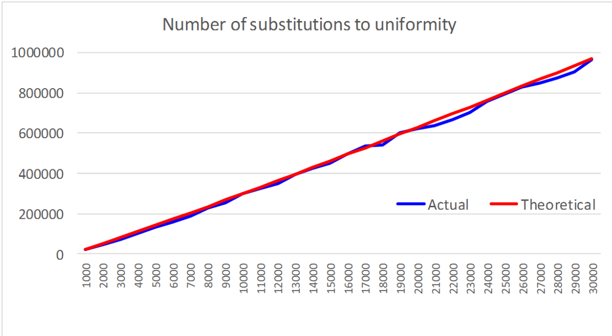

Figure 3: Validating length of random walk.

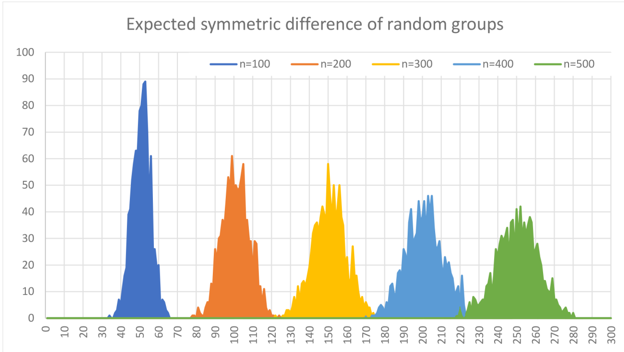

Figure 4: Distribution of symmetric differences

B.2. Expected difference of random groups

In this experiment we used the value N determined in the previous experiment and computed the symmetric difference at the end of the random walk. The results validate the conclusions drawn in Lemma 8 and can be seen in Figure 4.

For each value of $n = 100, 200, \ldots, 500$, we generated two random groups with $\Delta = 1$ and applied $\frac{3}{2}nlogn+11n$ random unit substitutions. Each experiment was repeated 1000 times to increase the confidence of the results. The horizontal axis shows the symmetric difference at the end of the random walk, while the vertical axis shows the count of pairs with the same symmetric difference. As it can be seen in the figure, the distribution of the symmetric difference follows a binomial distribution with mean $n/2$ which matches the result of Lemma 8.

B.3. Pre-processing to reduce number of extra variables and clauses

In this experiment we ran the pre-processing method described in Section 5.2.1. Recall that the goal of pre-processing is to create as many as possible initial groups that are different in order to minimize the number of extra variables used in flattening. As a benchmark, we used the set of satisfiable uniform random (uf) 3-SAT instances found in [38]. The results are displayed in Figure 5. Recall that the total number of groups that can be generated in a 3-CNF formula is at most $3m$, where m is the number of clauses in the formula.

Figure 5 shows that the total number of variables is reduced by an average of 55% if the pre-processing stage

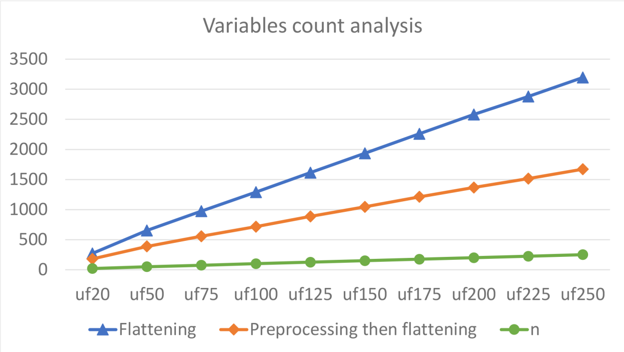

Figure 5: Variables added in random formulas after applying the flattening stage with and without pre-processing.

is applied. If not (i.e only flattening is used), the number of variables is maximized to $3m$.