Quantifying SAT Attack Resiliency in Circuits

A Thesis

Presented in Partial Fulfillment of the Requirements for the Degree
Master of Science in the Graduate School of The Ohio State
University

By

Lucas Nestor, B.S.

Graduate Program in Department of Electrical and Computer Engineering

The Ohio State University

2023

Master's Examination Committee:

Dr. Waleed Khalil, Advisor

Dr. Eslam Tawfik

Abstract

The SAT attack is a powerful and influential hardware attack on logic locking that iteratively eliminates groups of keys until only the correct keys remain. Most locking techniques base their SAT attack resiliency on the time to compromise, or the time taken to recover a correct key. The time taken by the attack is not a consistent measure across researchers or trials as it depends on the strength of the computer used, the software implementation of the attack, the chosen SAT solver, etc. This work proposes the minimum number of iterations required by the SAT attack as a more consistent metric to be used since it algorithimically determined. The minimum number of iterations for different configurations of key gates, including a single key gate, non-interfering key gates, and directly interfering key gates, has been derived and measured. The minimum number of iterations for a single key gate can be calculated using the total number of keys, number of incorrect keys, and probability an incorrect key corrupts the circuit. When multiple key gates are present, the minimum number of iterations for each key gate separately can be used to find the minimum number for all the key gates as a group. These findings indicate that the minimum number of iterations can be used as a metric to compare results across researchers, and the placement of key gates can have an effect on the SAT resiliency of the circuit.

To Grandpa Nestor and his many stories.

Acknowledgments

I would like to thank Dr. Waleed Khalil for giving me this opportunity and providing guidance through this journey. Had he not taken a chance on me, I would not be able to present you this work today. His suggestions and insight have made me a better researcher and student and I am very grateful for his help.

Secondly, I would like to thank Dr. Chris Taylor. He willingly took time out of his schedule with no personal incentive to help guide and mentor me through my research. His recommendations have opened up new avenues to explore and saved me from doing countless hours of manual work. I greatly appreciate everything he has done for me.

I would also like to thank Dr. Eslam Tawfik for serving on my thesis committee.

Lastly, I would like to thank Lindsey Spangler, my colleague who started at the same time as me. Entering graduate school during the peak of COVID was interesting to say the least, but being able to navigate it with somebody else made it easier.

Vita

March 5, 1997	Born - Orrville, OH, USA
2020	B.S. Physics, Computer Science and Engineering The Ohio State University
2020-2023	Graduate Research Associate, The Ohio State University

Fields of Study

Major Field: Department of Electrical and Computer Engineering

Table of Contents

List of Tables

Table **Page**

List of Figures

Chapter 1: Introduction

As circuit design companies have moved to becoming fabless, they have opened themselves up to unwanted and illegal use use of their IP. This can include overproduction by an untrusted foundry, trojan insertion by a malicious employee, or reverse engineering by a competitor. To prevent this, it is desirable to protect the function of the circuit even if the netlist is available. This can be accomplished via a technique called logic locking, where additional gates and inputs, called key inputs, are added to the circuit. The correct values for the key inputs, collectively known as the key, are only known by the designer, and if an incorrect value is applied, the function of the circuit is corrupted. Thus, both the netlist and key value must be known to achieve the correct function of the circuit.

Attacks can be performed on locked circuits to recover some or all of the secret key. One such attack, the SAT attack, is a influential and powerful threat to locking techniques. As such, it is important to measure the resilience against the SAT attack for new and existing locking algorithms. Most of these metrics are based on the time that elapses while the attack is running. This, however, depends on many factors, such as the speed of the computer used and the software implementation of the attack. These can vary across research groups or between trials, which makes using only the time to compromise as a metric less effective.

Since the SAT attack is iterative in nature, the time to compromise is determined by two factors - the time taken for each iteration and the number of iterations required. This work proposes an alternative metric to quantify the run time of the SAT attack - the minimum number of iterations required to uncover the secret key. This metric represents how the attack would run if an intelligent attacker took the optimal choices and minimized redundancy between iterations. Some logic locking techniques have proven their resiliency based on the minimum number of iterations, and this work extends that concept to most locking techniques. The minimum number of iterations is deterministic and algorithmically computed. It is a constant value for each circuit and does not depend on the speed of computer or software implementation used. Therefore, it can be a useful metric when comparing results between research groups.

1.1 Organization of this Thesis

The rest of this thesis is organized as follows.

Chapter 2 will introduce the various locking techniques and attacks. Chapter 3 describes the process to calculate the minimum number of iterations needed for a circuit that contains only a single key gate. Chapter 4 then discusses how the minimum number of iterations changes as multiple key gates are added, taking into account the different ways key gates can interfere with one another. Chapter 5 contains results comparing the minimum number of iterations measured for a circuit and the actual number of iterations taken by a SAT attack. Lastly, Chapter 6 provides a conclusion and future directions for this research.

Chapter 2: Background

This section details a chronological ordering of several different logic locking techniques and the various attacks that can be used to recover the keys for those techniques. An attack is considered successful if it is able to determine part or all of the key to a circuit. The defenses and attacks presented here are only a subset of the many options available and were chosen because they form a "lineage" related to the the focus of this thesis, the SAT attack.

2.1 Random Logic Locking

The first combinational logic locking defense developed was EPIC [1], also called Random Logic Locking (RLL). In this technique, XOR/XNOR key gates are inserted on random, non-critical traces in the circuit. If w_i is the original trace in the circuit and k_i is the key bit, a modified trace $w_i^{'}$ is inserted in the circuit such that, in the case of an XOR gate, $w_i^{'} = w_i \oplus k_i$. When a correct value is applied to the key bit, the key gate passes through the original signal, so the modified trace equals the original trace. If an incorrect value is applied, the signal is flipped, corrupting the function of the circuit. If the flipped signal alters the output of the circuit, then it can be rendered unusable and protected.

Figure 2.1 shows an example placement of one such key gate. When a 0 is applied to key input k_i, the function of the gate becomes $w_i^{'} = w_i \oplus 0 = w_i$, and the signal is not changed. When a 1 is applied to k_i, the function of the gate becomes $w_i^{'} = w_i \oplus 1 = \overline{w_i}$, corrupting the signal. The correct key bit value for an XOR key gate is a 0, and the correct value for an XNOR key gate is a 1, since those values cause the output to equal the other input to the gate.

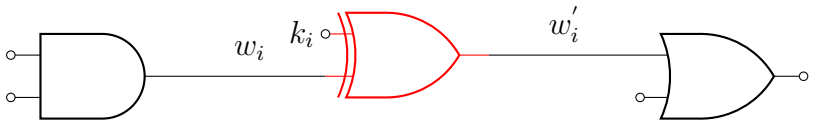

Figure 2.1: An example XOR key gate placed using RLL.

Using only an XOR or XNOR gate would not be secure, as the attacker could easily guess the value of the key based on the type of gate used. Instead, an inverter can be placed after the key gate to change the value of the correct key. For an XOR gate, an inverter would change the correct key to a 1, as that would offset the effect of the inverter. When the circuit is synthesized, the inverters can be integrated into the rest of the design to hide its trace. Thus, an attacker cannot know what the key bit should be when only looking at the type of key gate used.

2.2 Fault-Based Logic Locking

When key gates are inserted randomly, as they are in RLL, there is no way to control their effects on the output. On one extreme, the key gates could have no effect on the outputs, and the circuit behaves normally no matter the inputted key. On the other extreme, the key gates could flip every single output. Both situations are

equally as insecure because the attacker is able to completely correlate the outputs of the locked circuit with the outputs of the original circuit.

The Hamming distance (HD) between an incorrect output and the correct output can be used as a metric to measure a circuit's security. If $HD = 0$ or $HD = N$, where N is the number of outputs, the incorrect output is completely correlated with the correct output. Thus, an attacker could recover the original circuit easily. At the ideal Hamming distance of $HD = N/2$, the attacker has maximum ambiguity over which output bits are incorrect.

To allow designers to have control over a key gate's effects on the outputs of a circuit, Fault-based Logic Locking (FLL) [2] was proposed. In this technique, the key gates are iteratively placed on the traces that have the highest likelihood to modify the primary outputs of a circuit. This likelihood metric is calculated by simulating stuck-at-0 and stuck-at-1 faults on each trace. Once a key gate is inserted, an incorrect key is applied to that key input for all future iterations to prevent fault-masking - where the effects of two faults (key gates) offset each other. The result is that, for each iteration, the trace that maximally changes the output will be chosen. This process continues until the Hamming distance approaches $N/2$. Compared with random gate insertion, FLL can achieve this Hamming distance with less key gates because it choose the traces that corrupt the outputs with maximum probability.

2.3 Sensitization Attack

One method an attacker can use to exploit the weaknesses of RLL and FLL is called the Sensitization Attack [3]. In this attack, a specific input pattern is chosen such that the output of a key gate can be directly observed at a primary output of

the circuit. The same input pattern can be applied to a circuit bought off the market that already has the correct key applied. By comparing the output bits of the locked circuit and bought circuit, the correct key bit can be determined.

Figure 2.2a shown an example of a circuit vulnerable to the sensitization attack. When the input 1100 is applied to the primary inputs, the output of the key gate in red is directly propagated to the primary output of the circuit. If this input pattern is applied to an unlocked circuit, the primary output would be a 1. The attacker can correct determine that the key bit k_0 should be a 0 because a value of 1 produces a different output than the unlocked circuit for the same primary input pattern. On larger circuits, this process can be continued for all key gates to recover the key.

To protect against the sensitization attack, key gates can be placed such that the interfere with each other in a way that prevents their outputs to be directly propagated to a primary output of the circuit, as shown in Figure 2.2b. In this case, the attacker can no longer infer the key bit k_0 because they would also need to know k_1. Instead, they must search through the key space of the interfering key gates together. When many keys interfere with each other, the search space grows exponentially. RLL and FLL are particularly vulnerable to the Sensitization Attack because they do not guarantee that key gates interfere with each other nor do they give the circuit designer the control to place key gates in an interfering way.

2.4 Strong Logic Locking

To protect against the Sensitization Attack, Strong Logic Locking (SLL) [4] was created to maximize the number of interfering gates in a circuit, forcing an attacker to brute force the solution on a large number of key bits. A brute force attack is

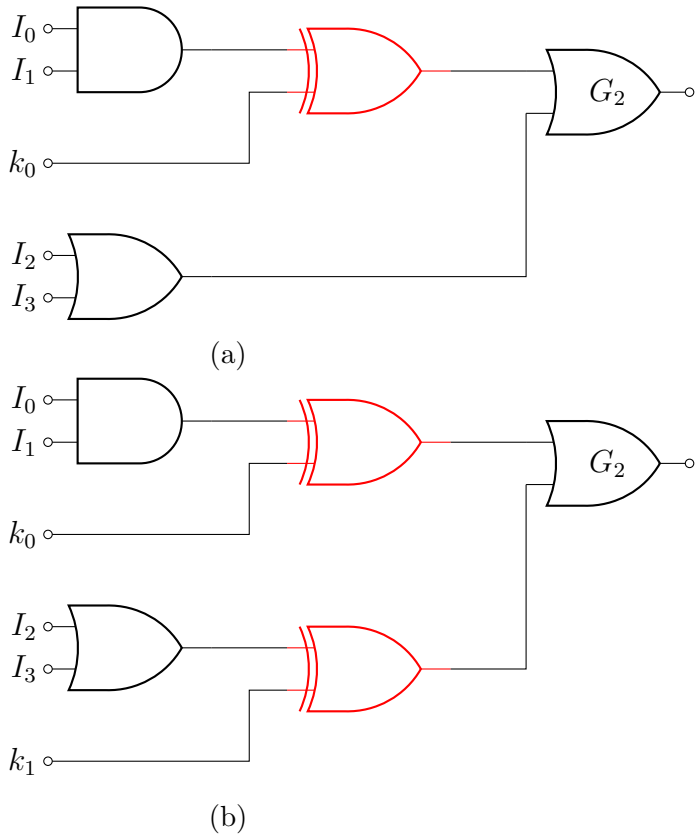

Figure 2.2: Sensitization attack performed on (a) a single key gate, and (b) two interfering key gates.

needed if two key gates are nonmutable convergent key gates. Nonmutable means that neither key gate can be muted, when the effect of the key gate is stopped before it reaches the next key gate. For example, in Figure 2.3, applying $I_4 = 1$ will mute key bit k_0, as the output of G_2 will always be 1. Thus, the key gates cannot interfere since the effects of k_0 will never reach gate G_5, meaning the sensitization attack can be used on this circuit.

The SLL algorithm iteratively places key gates such that the each key gate's relationship with all previously placed key gates is nonmutable. If no nonmutable

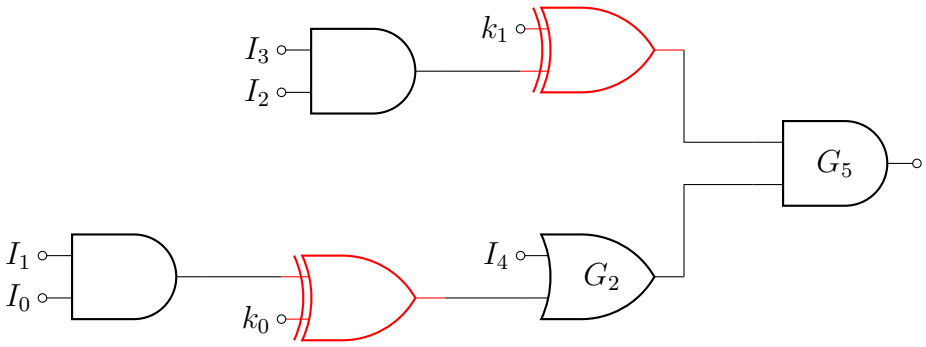

Figure 2.3: An example of a mutable key gate. Key bit k_0 can be muted by controlling primary input I_4.

net on the circuit is found, the key gate is inserted randomly. This results in groups with large numbers of nonmutable convergent key gates, forcing an attacker to brute force a large portion of the key. If the number of gates in a group is large enough, the sensitization attack is unfeasible to perform.

2.5 SAT Attack

The SAT attack [5] was developed next and reduced the time to break all existing locking techniques at the time by an order of magnitude. It uses a boolean satisfiability (SAT) solver to iteratively prune the key space by eliminating groups of incorrect keys until only correct keys are remaining. This is in contrast with a brute force attack, which only eliminates a single key at a time. It is an oracle guided attack, meaning an attacker must have access to an unlocked circuit in order to perform it.

To begin the SAT attack, a miter circuit is constructed, as shown in Figure 2.4. This circuit compares the outputs of two copies of the locked circuit. If both sets of outputs are exactly the same, then the output of the miter circuit will be a 0. If any output bit differs between the two, then the output of the miter circuit will be a 1.

The same input is fed to both copies of the circuit, while a different key is applied. Thus, the output of the miter circuit will be 1 if and only if the two key values K_1 and K_2 produce different outputs. When this happens, it is guaranteed that at least one of these sets of outputs is incorrect since they differ. However, it is possible that neither output is correct.

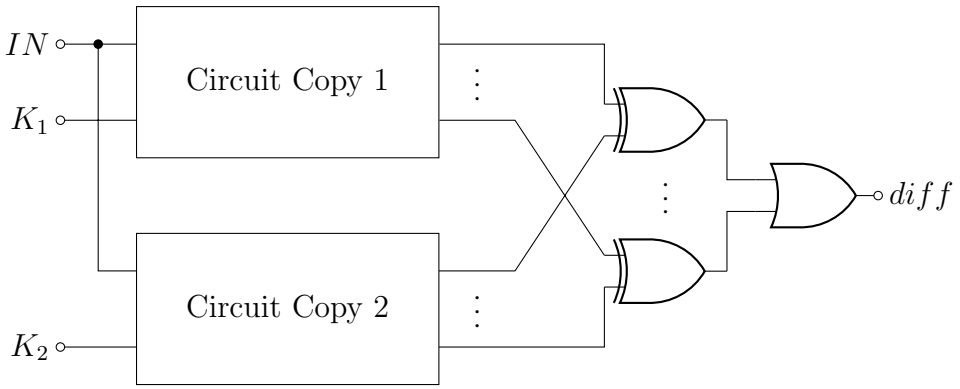

Figure 2.4: An example miter circuit used in the SAT attack.

In each iteration of the SAT attack, the miter circuit is given to the SAT solver to find an input pattern IN and two key patterns $K1$ and $K2$ such that the output of the miter circuit is 1. The input pattern IN is referred to as a distinguishing input pattern (DIP). The DIP is then applied to an oracle circuit and the correct output is observed.

On subsequent iterations of the SAT attack, a clause is added to the SAT solver to prevent the SAT attack from considering any key that produces an incorrect output for the previously found DIPs. This is how the SAT attack is able to eliminate groups of keys at a time - any key that produces an incorrect output for an input found in previous iterations is eliminated from consideration. This process continues,

with each iteration eliminating more keys through its DIP, until no more keys can be found that cause the output of the miter circuit to be a 1. When this happens, all remaining keys in the search space are correct.

The algorithm for the SAT attack can be seen in Algorithm 1, where the required inputs to the algorithm are the locked circuit function C and oracle circuit function *eval*. The *sat_assignment* function applies a DIP to the oracle and observes the output. After the algorithm terminates, a correct key can be found by using all DIPs found during the iterative part of the attack.

Algorithm 1 SAT Attack

Input: C, *eval*
Output: $\overrightarrow{K}_C$
  $i \leftarrow 1$
  $F_1 \leftarrow C(\overrightarrow{IN}, \overrightarrow{K}_1, \overrightarrow{OUT}_1) \bigwedge C(\overrightarrow{I}, \overrightarrow{K}_1, \overrightarrow{OUT}_1)$
  while $sat[F_i \bigwedge(\overrightarrow{OUT}_1 \neq \overrightarrow{OUT}_2)]$ **do**
    $\overrightarrow{IN}_i^d \leftarrow sat\_assignment[F_i \bigwedge(\overrightarrow{OUT}_1 \neq \overrightarrow{OUT}_2)]$
    $\overrightarrow{OUT}_i^d \leftarrow eval(\overrightarrow{IN}_i^d)$
    $F_{i+1} \leftarrow F_i \bigwedge C(\overrightarrow{IN}_i^d, \overrightarrow{K}_1, \overrightarrow{OUT}_1) \bigwedge C(\overrightarrow{IN}_i^d, \overrightarrow{K}_1, \overrightarrow{OUT}_1)$
    $i \leftarrow i + 1$
  end while
  $\overrightarrow{K}_C \leftarrow sat\_assignment[F_i]$

2.6 Anti-SAT

The key insight of the SAT attack is that it can unlock circuits in a small number of iterations (relative to the total number of key combinations) because it eliminates multiple keys per iteration. One way to increase the time taken by the SAT attack is to reduce the number of keys it can eliminate each iteration, causing an increase in the the number of iterations it performs. Anti-SAT [6], a SAT attack resistant

locking technique, achieves this by using two complementary functions, g and $\overline{g}$, as shown in Figure 2.5. In order for a key to be incorrect, the output of both g and $\overline{g}$ must be 1 (for a type-0 Anti-SAT). If there are p input patterns that make $g = 1$, then there are $2^n - p$ input patterns that make $\overline{g} = 1$, where n is the number of input to the circuit. Thus, there are a total of $p(2^n - p)$ input patterns that cause the signal in the circuit to be flipped each iteration. The value $p(2^n - p)$ is minimized when p approaches 1 or $2^n - 1$. A minimum value for this expression causes an increase in the number of iterations needed by the SAT attack.

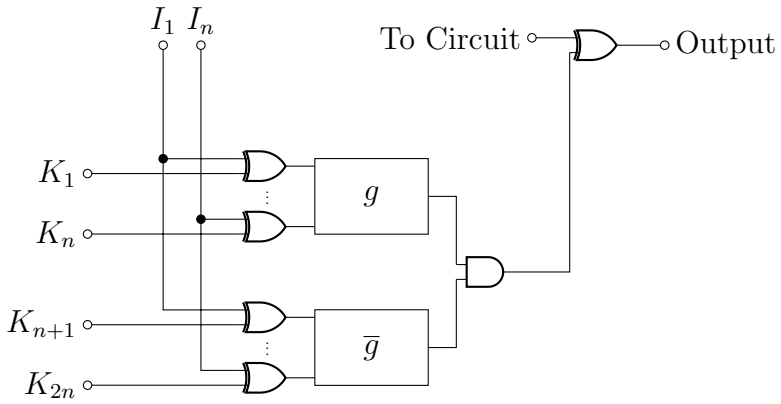

Figure 2.5: A type-0 Anti-SAT block.

One way to implement the function g is to use an AND function. Since the output of an AND gate is only asserted when all of its inputs are 1, if g is an AND gate, then $p = 1$. Similarly, g can be made a NAND gate to make $p = 2^n - 1$. When this happens, the total number of keys that can flip the signal each iteration will be $2^n - 1$, a relatively small number compared to the 2^{2n} total keys.

A correct key for this type of Anti-SAT block occurs when the output is 0 for all possible inputs. An equivalent way to state this is that one of the outputs of the g or $\overline{g}$ function needs to be 0. Since g and $\overline{g}$ are complementary, this will only occur for all possible input patterns when the inputs to both blocks are exactly the same. Thus, the correct keys are such that $K_i = K_{i+n}$, meaning the i^{th} key to g is equal to the i^{th} key to $\overline{g}$. To add variation in the key, some XOR gates can be converted into XNOR gates, which will invert the equality dependence between g and $\overline{g}$ for that bit.

2.7 SAT-Attack Resistant Locking Locking (SARLock)

While Anti-SAT increases the number of iterations by using complementary functions, SARLock [7] uses comparators to make it so only a single key corrupts the signal in the circuit each iteration. Figure 2.6 shows an example SARLock block. The primary inputs are compared to the key inputs. If they match, shown by the $IN = K$? block below, the output is asserted, and the XOR key gate flips the signal. However, when the primary input happens to equal the same pattern as the correct key, also called the protected input pattern, applying the correct key will also flip the signal. To prevent this, a mask block is inserted that will always output a 0 when the correct key is applied. Thus, for the signal to be flipped, the key must equal the input and also not be the correct key.

Because a SARLock block only flips the signal for a single key per input pattern, all $2^n - 1$ (excluding the protected input pattern) input patterns must be iterated through to eliminate every key. Therefore, the number of iterations increases exponentially with the size of the key. The key can be differentiated from the protected input pattern by not doing a bit-wise equality comparison. Instead, specified bits of the

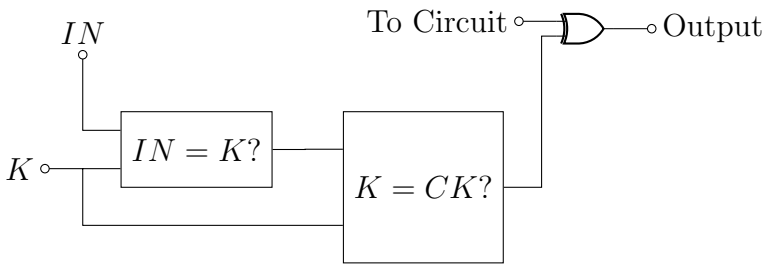

Figure 2.6: A SARLock block showing the comparator and mask.

key and primary inputs can be checked for equality, while the other bits can be checked for inequality.

2.8 Signal Probability Skew Attack

An Anti-SAT block requires g and $\overline{g}$ to have a p value as close to 1 or $2^n - 1$ as possible. This can be exploited by searching for a gate whose inputs have a very large absolute difference in probability skew. A net's signal probability skew (SPS) can be defined to be $s = P(x = 1) - 0.5$. So, for a signal that is always 1, the SPS is 0.5, while for a signal that is always 0, the SPS is -0.5. Since the outputs of the g and $\overline{g}$ block have a high skew towards 0 or 1, the absolute difference of their SPS will also be large. The signal probability skew attack [8] searches the circuit for the gates with the highest SPS difference on its inputs. If an Anti-SAT block is in the circuit, it will likely be detected by this attack. An attacker could then remove the Anti-SAT block from the circuit or netlist, recovering the original.

2.9 Removal Attack

A removal attack [9] can be used to mitigate the effects of a SARLock block in a circuit. If an attacker is able to identify which inputs are key inputs, they can trace the fanouts of those inputs to find the comparator, masking block, and single XOR key gate. With knowledge of where the original insertion point is, an attacker can recover the original netlist by removing both the comparator and mask blocks, rendering SARLock ineffective.

2.10 Traceless and Tenacious Logic Locking (TTLock)

TTLock [9] protects SARLock from the removal attack by modifying the original circuit logic so that if the locking circuitry were removed, the attacker would recover a corrupted design. In the TTLock technique, there is still a comparator that compares the key with the primary input. However, the masking circuitry, which was inserted to prevent the correct key from flipping the signal, is not present. Instead, the original circuit is modified such that, for the single protected input pattern, the function of the circuit itself is flipped. When the correct key and protected input pattern are applied, the XOR key gate will flip the signal of the corrupted circuit back to the original value. When an incorrect key is applied to the protected input pattern, the TTLock block will not flip the corrupted signal back, and the circuit will remain unusable.

Because a comparator is still used, the TTLock block only eliminates a single key in most iterations, and therefore is SAT resilient. Additionally, removing the TTLock block will result in a circuit that does not function correctly for the protected input pattern, protecting against a removal attack. However, the primary difference

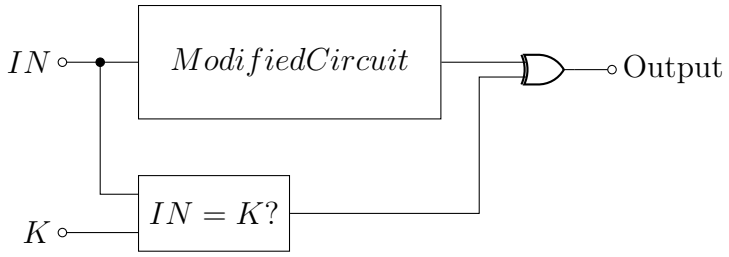

Figure 2.7: A TTLock block showing the comparator and modified circuit logic. The circuit does not produce the correct output for the protected input pattern and relies on the comparator to restore the correct value to the output.

between TTLock and SARLock comes from a SAT attack analysis of the protected input pattern. In SARLock, the protected input pattern could not eliminate any keys, rendering it useless for the SAT attack. However, in TTLock, the protected input pattern eliminates every incorrect key because only the correct key will restore the circuit's function to the proper value. Therefore, if the SAT attack happened to choose the protected input pattern as a DIP during an iteration, it would immediately solve the circuit. For a large input search space, this is unlikely.

2.11 Stripped Functionality Logic Locking (SFLL)

SFLL [10] is an extension to TTLock that allows for more than a single input pattern to be protected. While TTLock used a comparator to compare a key with the protected input pattern, SFLL compares the Hamming distance of the key with a specified input pattern. If the key is within a specific Hamming distance, the XOR key gate flips the signal. Like in TTLock, all inputs within the same Hamming distance as the central protected input result in corrupted function of the circuit. A correct

key must be applied to restore the signal to its correct value. This makes TTLock the special case of SFLL where the Hamming distance is 0.

The correct key and Hamming distance are both parameters controlled by the designer, giving them greater control of the locking scheme when compared to TTLock. The choice of Hamming distance will affect the attack resiliency of the circuit. A larger Hamming distance will decrease the number of iterations the SAT attack requires because more keys will modify the signal per input. Therefore, for a constant number of total keys, less iterations will be needed. However, it will increase the output corruptibility because more incorrect keys will cause the output to be changed from its true value. Additionally, because the logic of the circuit is changed for all inputs within the specified Hamming distance instead of a single input pattern, the circuit is more protected against the removal attack.

Chapter 3: Single Key Gate

While more exotic logic locking defenses and advanced attacks are being introduced, the SAT attack still remains influential. The resiliency of all new locking techniques against the SAT attack must be measured in order to accurately characterize that technique's security. Additionally, every circuit designed should be resistant against the SAT attack or else risk being broken quickly. This remainder of this thesis will investigate using the minimum number of iterations the SAT attack is required to take as a metric for circuit resiliency. The lower bound of the attack can be determined algorithmically based on the properties of each key gate and their arrangements relative to one another.

When determining a lower bound on the SAT attack, the defender assumes a powerful attacker because the minimum security for a circuit occurs when the attacker has the maximum amount of information. Specifically, the attack model used by the defender in this work assumes the attacker:

- has access to an oracle circuit
- has access to a locked netlist
- can determine which gates are key gates
- can determine which gates are in a key block
- can determine which inputs are key inputs

- can determine which inputs go to specific key gates
- can determine which outputs are driven by specific key gates
- knows the number of correct keys for each individual key gate

3.1 Derivation

Before we consider the minimum number of iterations for an entire circuit, we should analyze the simplest locking technique: a single key gate. Consider the key gate shown in Figure 3.1, which is a generalized version of other locking techniques. It is integrated into the circuit at a single XOR gate such that one input of the XOR gate is an existing net in the original circuit, as many existing locking techniques are. The other input to the XOR gate is the output of what is called a key block. This key block performs some function on the primary inputs and key inputs of the circuit and can assert the signal to the XOR gate, corrupting the original net in the circuit. For example, in RLL, this key block would simply be a wire from the key input to the key gate. In Anti-SAT, this key block would be the complementary functions g and $\overline{g}$ that form the Anti-SAT block. In SARLock, the key block would be the comparator and key mask. As you can see, many locking techniques fit into this generalized definition of a key gate.

We will assume that the output of the key gate is a primary output of the circuit for simplicity. Because of the nature of the SAT attack, only primary input and key combinations that cause the output to be changed will be considered. Thus, it does not matter where in the circuit the key gate is placed because the SAT attack will seek out inputs that propagate errors to the output. Any input patterns that do not propagate the errors will not be considered.

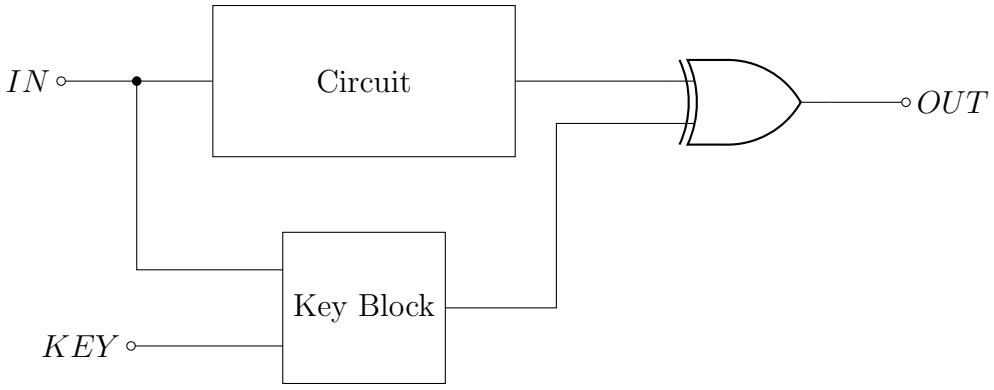

Figure 3.1: A generalized key gate.

3.2 Input-Dependent Gate

Gates that are input-dependent have a key block whose inputs are both key inputs and primary inputs of the circuit. On any given iteration i of the SAT attack, the probability a key corrupts the signal can be written as $P(change)_i$. The number of incorrect keys that can be eliminated in this iteration i, denoted by IK_i is given by equation 3.1. The number of *unique* keys that can be eliminated in iteration i is given by the inequality $UIK_i \leq IK_i$. This is because some keys may have been eliminated in previous iterations, so the unique keys eliminated in this iteration could be less.

$$IK_i = KP(change)_i \tag{3.1}$$

The sum of all unique, incorrect keys eliminated in each iteration will equal the total number of incorrect keys for the circuit. Since only unique keys are considered, all keys that have been eliminated previously will not be double counted. Thus, the total number of incorrect keys for the circuit can be given by equation 3.2.

$$\sum_i UIK_i = IK \leq \sum_i KP(change)_i \tag{3.2}$$

One simplifying assumption that we can make is to consider the probability a key is incorrect to be constant over all iterations. This assumption is valid in many common locking techniques, such as Anti-SAT and RLL, and can be used as a very good approximation for others, such as SARLock. The probability can then be pulled out of the sum over i since it no longer depends on the current iteration. This sum over i now equals the total number of iterations, λ, as shown in equation 3.3.

$$IK \leq KP(change) \sum_i 1 = KP(change)\lambda \tag{3.3}$$

This inequality can be rearranged to find a lower limit on the number of iterations required by the SAT attack, shown in equation 3.4. If each factor in the equation is varied with the others staying constant, the results of this expression logically make sense. The more incorrect keys there are while eliminating the same number of keys per iteration, the more iterations will be needed to eliminate them all. Likewise, as the number of keys increase, the number of keys eliminated each iteration will increase (assuming the elimination probability does not change), decreasing the total number of iterations. Lastly, if the probability that a key is eliminated increases, then the number of iterations will also decrease, as more keys are eliminated each iteration.

$$\lambda \geq \frac{IK}{KP(change)} \tag{3.4}$$

3.2.1 Comparison with Existing Locking Schemes

The above equation is valid for all locking techniques that have a constant $P(change)$ over each iteration and can be used as an approximation for locking schemes where $P(change)$ is near constant. It can be useful to compare to the expected results from existing locking schemes to ensure that they fit this model. In this section, equation 3.4 will be compared with Anti-SAT and SARLock.

In Anti-SAT, the total number of keys is $K = 2^{2n}$, where n is the number of primary inputs used. The number of incorrect keys is given by $2^{2n} - 2^n$. For each iteration, a given key is incorrect if both the g and $\overline{g}$ function output a 1 (for a type-0 Anti-SAT block). If p input vectors to the g block cause it to output a 1, then $2^n - p$ input vectors cause the $\overline{g}$ block to also output 1. Thus, there are a total of $p(2^n - p)$ input vectors to the Anti-SAT block that cause it to be incorrect on a single iteration. The total number of input vectors for the Anti-SAT block is 2^{2n}, so $P(change) = p(2^n - p)/2^{2n}$. Using these values with equation 3.4, the lower bound of the number of iterations is given by equation 3.5. This result is the same lower bound as found in [6].

$$\lambda \geq \frac{2^{2n} - 2^n}{2^n \frac{p(2^n - p)}{2^n}} = \frac{2^{2n} - 2^n}{p(2^n - p)} \tag{3.5}$$

In SARLock, the total number of keys is equal to the number inputs used, n, while the number of incorrect keys is given by $n - 1$. In this case, though, the probability a key changes the signal is not constant for every iteration. In $n - 1$ iterations, the probability is given by $P(change) = 1/2^n$. However, for a single input pattern, the protected input pattern, the signal can never be flipped, so $P(change) = 0$. This analysis will make the approximation that all iterations have the same probability

by ignoring the protected input pattern. Thus, the minimum number of iterations required by the SAT attack is given in equation 3.6. Note that, while this was done using an approximation, the result derived is the exact result quoted in [7].

$$\lambda \geq \frac{2^n - 1}{2^n \frac{1}{2^n}} = 2^n - 1 \tag{3.6}$$

3.3 Input-Independent Gate

Key gates that are input-independent have a key block whose inputs are only the key bits for that specific gate. Thus, for a circuit only containing a single input-independent gate, the SAT attack is able to choose any input on any iteration since the primary inputs have no effect on the key gate. On the first iteration, all keys that cause the output to be flipped will be eliminated. Since flipping the signal is only a function of the key inputs, this will eliminate every incorrect key. Thus, each input-independent gate only takes a single iteration of the SAT attack to solve.

Chapter 4: Multiple Key Gates

While the previous chapter discussed how to lower bound the SAT attack for a single key gate, this chapter considers when multiple key gates are present. The way the key gates are arranged and affect each other must be taken into account when analyzing such a circuit. We will call the way key gates interact with one another interference. This chapter will analyze three types of interference: no interference, direct interference, and indirect interference.

Two key gates do not interference, or exhibit no interference, when both their fan-in and fan-out cones are disjoint sets of gates. Gates directly interfere with one another when one key gate is present in the fan-in cone of another key gate. Lastly, indirect interference occurs when the fan-out of two key gates converge at some other net in the circuit. Figure 4.1 shows a simple example of all three types of interference.

4.1 No Interference

To determine a lower bound for non-interfering key gates, we consider the simplest possible case: only two key gates present in the circuit. Since they do not interfere, both the inputs that drive each gate and the outputs each gate affects do not overlap. Therefore, an intelligent attacker can manipulate the inputs to one key gate and observe the outputs for that key gate independently of the other. Take the inputs in

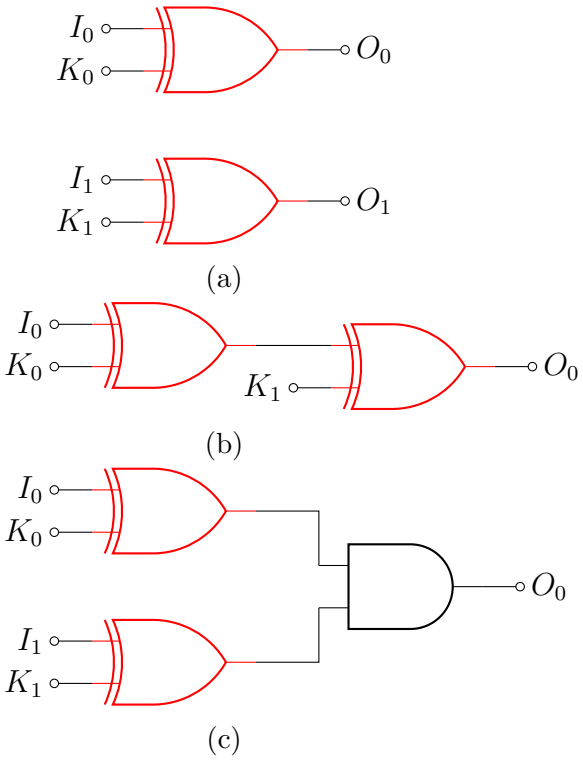

Figure 4.1: (a) No Interference: The two key gates have disjoint fan-out cones and affect different outputs. (b) Direct Interference: One key gate is in the fan-in cone of another. (c) Indirect Interference: The outputs of both key gates converge at a third gate.

the fan-in of key gate KG_1 to be I_1 and the outputs in the fan-out of KG_1 to be O_1, and similarly for KG_2. If the value of the outputs O_1 are different than those of the corresponding outputs on the oracle, then the attacker knows that the key provided to KG_1 was incorrect because KG_2 cannot affect O_1.

Since an attacker knows when an individual gate KG_1 or KG_2 corrupts the signal and can also manipulate I_1 and I_2 independently, that attacker can solve for keys K_1 and K_2 simultaneously. They can apply input patterns to I_1 that will solve KG_1 as if KG_2 was not present in the circuit. The set of DIPs that can be used to solve both

gates individually can be applied to each gate's respective subset of all the primary input bits.

Table 4.1 shows an example of solving two gates simultaneously. KG_1 uses bits 0-3 of the primary inputs and can be solved in 4 iterations independently. KG_2 uses input bits 4-7 and can also be solved in 4 iterations. The first two columns display a set of DIPs that solves the circuit when the respective key gate is the only one present. The bold part of the input pattern shows which bits are specific to that individual key gate. The last column shows a set of DIPs where each gate's respective range of the input pattern uses the values that solve that gate independently in the first two columns. Because each key gate is still receiving the same input patterns as in the first two columns, this circuit will still be solved in four iterations, even when both key gates are present.

DIP 1	DIP 2	Combined DIP
0000 1101 10	0101 **0000** 11	0000 0000 00
0001 1100 01	1110 **0001** 10	0001 0001 00
0010 1101 00	0010 **0010** 01	0010 0010 00
0011 1110 10	0101 **0011** 00	0011 0011 00

Table 4.1: The inputs for each key gate can be chosen from the DIPs used to solve those key gates individually.

From this derivation, the number of iterations for non-interfering key gates can be lower bounded when the SAT attack chooses the same input patterns for each key gate that would solve that gate independently. Since the attack must still iterate through every input pattern needed by all key gates, the gate requiring the largest

number of iterations will set the lower bound for the entire circuit. Table 4.2 shows an example where the gates require different numbers of iterations. Here, KG_2 requires 6 iterations, while KG_1 still requires 4. The SAT attack is lower bounded to 6 iterations because that it what is required to solve KG_2. During iterations 5 and 6, the input to KG_1 has no function and can be any value.

DIP 1	DIP 2	Combined DIP
0000 1101 10	0101 **0000** 11	0000 0000 00
0001 1100 01	1110 **0001** 10	0001 0001 00
0010 1101 00	0010 **0010** 01	0010 0010 00
0011 1110 10	0101 **0011** 00	0011 0011 00
0000 0010 00	1101 **0100** 00	0000 0100 00
0001 0110 10	1111 **0101** 00	0001 0101 00

Table 4.2: The minimum number of iterations for non-interfering gates is determined by the gate with the largest number of individual iterations.

4.2 Direct Interference

When two key gates are directly interfering, they can no longer be solved independently like was possible for non-interfering gates. To see why, consider the four possible cases that can occur with two key gates: neither key gate flips the signal, only key gate 1 flips the signal, only key gate 2 flips the signal, and both key gates flip the signal. When neither gate flips the signal, the output will not be changed, and thus no information can be discerned about the correctness of the key applied. When either gate 1 or gate 2 flips the signal, the output will be changed, and the key applied is known to be incorrect. When both gates flip the signal, the second gate

will revert the signal back to its correct value. Thus, from the perspective of the SAT attack, the output has not been changed, and no information can be determined. The two gates have offset each other's effects.

Next, consider applying an single input pattern to the locked circuit. That input pattern will map to a set of keys that produce an incorrect output for that input. In many locking schemes, this set of keys is mutually exclusive from the set of keys mapped to by every other input, meaning an applied key will corrupt the signal if and only if the corresponding primary input pattern is applied. We can denote this mapping as $I \rightarrow K(I)$, where I is the primary input pattern and $K(I)$ is the set of keys that change the output when that input pattern is applied. When I is applied along with a key from $K(I)$, the output will be changed. However, when I is applied with a key not in $K(I)$, the output will not be changed. Likewise, applying $K(I)$ with an input that is not I will not change the output.

This mapping can be visualized using a plot such as Figure 4.2. In this figure, the rows represent different primary input patterns, while the columns represent different key input patterns. A square is dark when that combination of key and primary input corrupts the signal. Light squares occur when the key gate does not flip the signal. As can be seen in this figure, each input maps to a specific set of keys since no column contains two dark squares. The plot shown is for an Anti-SAT block that requires 8 iterations to be solved.

The concept of 1-to-n primary input to key mappings can be used to see why directly interfering key gates cannot be solved independently. A primary input pattern I for a circuit with two key gates can be split into subsets $I = (I_a, I_b)$, where I_a and I_b are the bits of the input that go to key gate A and key gate B, respectively. Likewise,

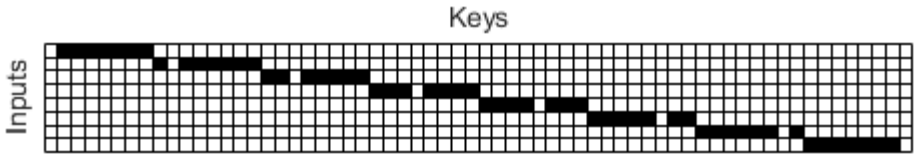

Figure 4.2: Each input pattern I (Y-axis) for this example Anti-SAT block maps to a unique set of keys $K(I)$ (X-axis).

the total key for the circuit K can be split in this manner. We denote the key set $K(I_a, I_b)$ as the keys that would cause both gates to flip the signal when input (I_a, I_b) is applied. Similarly, the key set $K(\ I_a,\ I_b)$ represents all keys that cause neither gate to corrupt the signal for that input. This notation cam be extended to any number of gates.

Now, once again consider an iteration in the SAT attack where the DIP applied is (I_a, I_b). The keys in sets $K(I_a,\ I_b)$ and $K(\ I_a, I_b)$ would both flip a single key gate, ultimately changing the output of the circuit. Therefore, both sets of keys will be eliminated when the SAT attack adds the clauses that prevent keys from being inconsistent with previous DIPs from being considered in future iterations. We will call these keys "eliminated keys." However, a key from the set $K(I_a, I_b)$ would cause both key gates to corrupt the signal and offset each other, preventing the output from changing. Even though the keys from this set are ultimately incorrect, they will not be eliminated in this iteration of the SAT attack. These keys are called "hidden keys" because they are hidden from elimination for this iteration and must be eliminated in the future. This analysis shows that directly interfering key gates cannot be solved simultaneously using only the DIPs that can solve each gate independently.

In order to eliminate the hidden keys in $K(I_a, I_b)$, either key gate A or key gate B needs to corrupt the signal, but not both. Therefore, an input of $(I_a, \ \ I_b)$ or $(\ \ I_a, I_b)$ must be applied on a subsequent iteration. In the first case, $(I_a, \ \ I_b)$ is guaranteed to flip key gate A and not key gate B for keys in the hidden set $K(I_a, I_b)$ because of the mutually exclusive mapping from input to key set. Likewise, $(\ \ I_a, I_b)$ is guaranteed to flip key gate B and not key gate A. In general, to eliminate a hidden key caused by a pair of gates, only one input from either key gate must be repeated in an iteration.

A simple way to estimate the SAT attack lower bound for directly interfering gates is to group the DIPs needed to solve the circuit by the type of key they eliminate. The initial group, which we will call DIP group 1, will unavoidably create hidden keys. Therefore, DIP group 2 is created to eliminate all hidden keys from DIP group 1. Likewise, DIP group 3 eliminates all hidden keys from DIP group 2, and so on. As the number of gates grows, so does the number of DIP groups needed to completely eliminate all keys. In this work, we'll analyze DIP group 1 and 2 and determine how many iterations are required by each and present the remaining hidden keys that need to be eliminated.

4.2.1 DIP Group 1

DIP group 1 is the starting point of the SAT attack. Any combination of inputs used for the DIPs in this group will result in hidden keys that will be need to be eliminated in future groups. A simple starting point is to use the inputs that solve each gate individually, similar to no interference. We denote the inputs using the following nomenclature: I_{a1} represents the first input to key gate A that would solve it independently, I_{a2} represents the second input, etc. Therefore, an input such as

I_{d6} represents the 6th input to key gate D (the 4th gate in the directly interfering sequence) that is used to solve that key gate without the presence of the other gates. This notation is used because the specific value of the input does not matter, but the iteration it was applied does. Therefore, when looking at which iteration an input was applied, one only needs to consult the numerical subscript.

At a minimum, all of the inputs that solve each gate individually must be iterated through during the SAT attack. Table 4.3 shows an example DIP group 1 for 4 key gates. As can be seen, DIP group 1 is identical to the DIPs used to solve non-interfering key gates.

Iteration	Gate A	Gate B	Gate C	Gate D
1	I_{a1}	I_{b1}	I_{c1}	I_{d1}
2	I_{a2}	I_{b2}	I_{c2}	I_{d2}
3	I_{a3}	I_{b3}	I_{c3}	I_{d3}
4	I_{a4}	I_{b4}	I_{c4}	I_{d4}
5	I_{a5}	I_{b5}	I_{c5}	I_{d5}

Table 4.3: An example DIP group 1. For iteration 1, the total DIP would be $(I_{a1}, I_{b1}, I_{c1}, I_{d1})$

The minimum number of iterations for each DIP group needs to be determined to arrive at a lower bound for the total SAT attack. Since DIP group 1 is identical to the inputs needed by non-interfering key gates, the lower bound is set by the gate with the largest number of iterations. Table 4.4 shows an example where key gate C is the dominant gate in the sequence. The other gates can complete their DIPs in the time that key gate C needs to iterate through all of its DIPs.

Iteration	Gate A	Gate B	Gate C	Gate D
1	I_{a1}	I_{b1}	I_{c1}	I_{d1}
2	I_{a2}	I_{b2}	I_{c2}	I_{d2}
3	I_{a3}	I_{b3}	I_{c3}	$\cdots$
4	$\cdots$	I_{b4}	I_{c4}	$\cdots$
5	$\cdots$	$\cdots$	I_{c5}	$\cdots$

Table 4.4: Key gate C is the limiting factor with 5 DIPs needed. Therefore, this DIP group 1 needs to take 5 iterations.

In addition to the number of iterations required for each DIP group, the hidden keys that are produced also need to be known so that future DIP groups can be created to eliminate those keys. When analyzing these hidden keys, we must also take into account the presence of correct keys, which will not corrupt the signal in the circuit for any input. The symbol used for a correct key will be C. For example, the key set $K(I_{a1}, I_{b1}, C, C)$ represents all keys that flip the first two gates for inputs I_{a1} and I_{b1} and never flip the last two gates.

As stated previously, a hidden key will occur on an iteration when every key gate that flips the signal is offset by another key gate. For example, in the DIP group shown in Table 4.4, the key sets $K(I_{a1}, I_{b1}, I_{c3}, I_{d3})$ and $K(I_{a1}, I_{b1}, C, C)$ will both remain hidden because each iteration has an even number of key gates corrupting the signal. Meanwhile, the key set $K(I_{a1}, I_{b1}, C, I_{d2})$ will be eliminated because the last key gate will be the only gate corrupting the signal in iteration 2, causing these keys to be eliminated.

In DIP group 1, the requirement that all gates are offset by another gate is equivalent to having an even number of each numerical subscript in a key set. For example,

the above key set $K(I_{a1}, I_{b1}, I_{c3}, I_{d3})$ has two 1s and two 3s, meaning the two gates will flip the signal on iterations 1 and 3. However, the key set $K(I_{a1}, I_{b1}, C, I_{d2})$ has two 1s and one 2. So, since there is an odd number of 2s, this key will be eliminated in iteration 2.

4.2.2 Hidden Key Categories

While determining which keys remain hidden, the relationship between when inputs were applied in DIP group 1 is more important than the actual iteration in which they were applied. In other words, two keys, such as $K(I_{a1}, I_{b1})$ and $K(I_{a2}, I_{b2})$ can be considered as the same since they both have two inputs from the same iteration, even though the iteration is different. It will be useful to categorize all types of keys like this. The notation we will use is as follows: the letters W-Z will indicate inputs used in different iterations in DIP group 1 without pointing to a specific iteration. For example, $K(W)$ represents key sets $K(I_1)$, $K(I_2)$, etc. Inputs that have the same letter indicate those inputs were applied in the same iteration during DIP group 1. For example, $K(WWC)$ describes the key sets $K(I_{a1}, I_{b1}, C)$ and $K(I_{a2}, I_{b2}, C)$, etc. However, when inputs have different letters, that indicates those inputs were not from the same iteration. So, $K(WWX)$ can describe the key sets $K(I_{a1}, I_{b1}, I_{c2})$ but not $K(I_{a1}, I_{b1}, I_{c1})$.

Additionally, which input was applied to which gate during DIP group 1 is not important. For example, two key sets such as $K(I_{a1}, I_{b1}, C, C)$ and $K(C, I_{b2}, I_{c2}, C)$ can be treated as functionally the same. They both have a single pair of offsetting gates and two correct keys applied. They can be considered as the same "category" even though the exact placement of the inputs is different between the two. So, using

the notation described above, the location of the letters W-Z and C does not imply a specific key gate. Instead, a key set such as $K(WWC)$ describes *all* keys that have two gates offsetting each other and the third key correct. This includes both the key set $K(I_{a1}, I_{b1}, C)$ and $K(C, K_{b3}, K_{c3})$, among many others.

The requirement that all hidden keys must have offsetting gates can be easily transferred into this new key category notation. A key set with input W applied has an offsetting gate if there is another W present in that key. For a key to remain hidden in DIP group 1, there must be an even number of every letter W-Z. For example, $K(WWCC)$ has a even number of Ws, and thus will remain hidden because the gates with the W inputs applied will always offset. However, $K(WWXC)$ will be eliminated because the X input will flip a single key gate during whatever iteration X corresponds to in DIP group 1. This means that all key sets that fit into the $K(WWXC)$ do not need to be considered in DIP group 2 because they will all have been eliminated.

Finally, the number of correct keys being applied to key gates does not affect a key set being hidden or eliminated. Since the gates that have these correct keys applied can never flip the signal, they can essentially be ignored. For example, the keys from key sets $K(WWCC)$ and $K(WWCCCC)$ are functionally the same because they both have 2 inputs applied from the same iteration. In our notation, we will designate an arbitrary amount of correct keys by using $C+$. For example, the two key categories shown above can both represented by $K(WWC+)$. This is a shorthand notation that is useful because it is able to describe an arbitrary amount of keys.

An important note is that our notation using X-Z as a placeholder for the iteration number only supports up to 4 different iterations in a single key. This can support

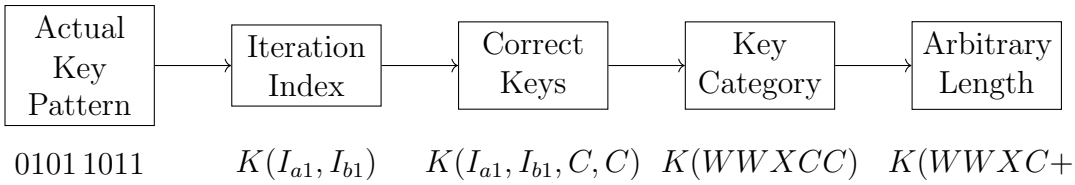

Figure 4.3: The notation described abstracts individual keys into groups of keys behaving the same.

up to 8 directly interfering key gates because all gates must be at a minimum offset by a single other key gate to be considered beyond DIP group 1. So, the largest key category that can be fully supported in this notation is $K(WWXXYYZZC+)$. Keys that are larger than this key will be unable to be described by this notation. However, this work will not analyze such keys. A fully extendable notation could be created with numbers or the entire alphabet to denote the variable iteration number. However, it could cause confusion with the actual iteration number (if 1, 2, 3, etc. were used), or the key gate label (if A, B, C, etc. were used) when compared with an input such as I_{a1}, so the letters W-Z were chosen for minimum ambiguity.

Figure 4.3 shows a summary of the key abstraction techniques described in this section. This notation allows us to talk about many different types of keys at the same time without having to refer to the exact value of the key. As we examine more DIP groups, it is easier to discuss which iteration a key was applied without referring to any specific iteration.

4.2.3 DIP Group 2

DIP group 2 can be crafted to eliminate the simplest keys hidden during DIP group 1, which take the form $K(WWC+)$. That is, two key gates have inputs corresponding

to the same iteration during DIP group 1, and all other gates have correct keys applied. As stated previously, a single input from the WW pair must be repeated to eliminate that specific key set. Similarly, a key set remains hidden any time two Ws are repeated. For example, the input $(I_{a1}, I_{b2}, I_{c1}, I_{d3})$ will not eliminate keys of the form $K(I_{a1}, C, I_{c1}, C)$ because key gates A and C will collide. Therefore, it is required for DIP group 2 that every iteration does not repeat an input twice.

It can be useful to look at all possible pairs for one instance of a $K(WWC+)$ key to determine how DIP group 2 should be created. An example shown for the first iteration of DIP group 1 is shown in Table 4.5. For the four key gates shown, there are 6 total pairs of WW inputs. Three inputs must be repeated in order to repeat a single input from every pair. This is visualized as choosing any three columns from the table below will include an entry from every row. In general, for n key gates, $n-1$ inputs must be repeated to eliminate all hidden pairs created by the iterations in DIP group 1.

Gate A	Gate B	Gate C	Gate D
I_{a1}	I_{b1}	$\cdots$	$\cdots$
I_{a1}	$\cdots$	I_{c1}	$\cdots$
I_{a1}	$\cdots$	$\cdots$	I_{d1}
$\cdots$	I_{b1}	I_{c1}	$\cdots$
$\cdots$	I_{b1}	$\cdots$	I_{d1}
$\cdots$	$\cdots$	I_{c1}	I_{d1}

Table 4.5: All pairs of inputs that result in hidden keys for iteration 1 of DIP group 1.

The general format of DIP group 2 is now fully defined - each iteration must not repeat an input pattern twice, and each input pattern must be applied to $n-1$ gates. Thus, each iteration in DIP group 2 can be created by using an input from a unique DIP group 1 iteration for each key gate. These inputs can then be shifted left or right on subsequent iterations, as seen in Table 4.6. In this table, the numerical subscripts, which represent the iteration that specific input was applied in DIP group 1, shift to the left every row. All numbers are used at least 3 times, and no number is repeated twice in the same row. Thus, the set of DIPs shown below will eliminate all $K(WWC+)$ keys for this specific circuit.

Iteration	Gate A	Gate B	Gate C	Gate D
5	I_{a1}	I_{b2}	I_{c3}	I_{d4}
6	I_{a2}	I_{b3}	I_{c4}	I_{d1}
7	I_{a3}	I_{b4}	I_{c1}	I_{d2}
8	I_{a4}	I_{b1}	I_{c2}	I_{d3}

Table 4.6: An example DIP group 2. For iteration 1, the total DIP would be $(I_{a1}, I_{b2}, I_{c3}, I_{d4})$.

As the number of inputs needed to be repeated grows beyond the number of key gates in the circuit, shifting the inputs every iteration will create redundancy. This can be seen in Table 4.7. The highlighted inputs all come from iteration 3 during DIP group 1. As was discussed above, only 2 inputs need to be repeated for 3 key gates. Thus, the input I_{a3} that is applied on iteration 9 to key gate A below will not eliminate any keys. In this example, the inputs with subscripts 2, 3, 4, and 5 are all repeated 3 times, causing redundancy and additional iterations that are not needed.

Iteration	Gate A	Gate B	Gate C
7	I_{a1}	I_{b2}	I_{c3}
8	I_{a2}	I_{b3}	I_{c4}
9	I_{a3}	I_{b4}	I_{c5}
10	I_{a4}	I_{b5}	I_{c6}
11	I_{a5}	I_{b6}	I_{c1}

Table 4.7: An example DIP group 2 showing redundancy. There are 3 key gates, but each key gate has 6 iterations needed to be solved independently. The highlighted inputs show that every iteration from DIP Group 1 has 3 inputs repeated.

The above DIP group 2 can be changed into the inputs shown in Table 4.8 to eliminate redundancy. Here, each subscript number is repeated only twice, which still eliminates all hidden keys of the form $K(WWC+)$ without using extra iterations. The difference between this group and the previous group is that the shifting of inputs only occurs locally in a chunk of iterations. Iterations 7 and 8 are chunked together and shifted to the left from each other. After iteration 8, no more inputs with subscripts 1, 2, or 3 are needed since those subscripts have all been repeated twice. Therefore, iterations 9 and 10 can be chunked and shift the inputs with subscript 4, 5, and 6. In general, for n key gates, a chunk can be made with $n-1$ iterations that shift inputs locally within the chunk and apply each input the ideal number of times. This results in minimized redundancy and the least number of iterations for DIP group 2.

Now that the form for DIP group 2 is fully described, we must determine the lower bound on its size. Each iteration in DIP group 1 creates $n-1$ pairs of hidden inputs that must be repeated. Since each pair only needs a single input to be repeated, $n-1$ inputs must be repeated in DIP group 2 per iteration in DIP group 1. For m iterations in DIP group 1, this results in $(n-1)m$ inputs needing to be repeated in

Iteration	Gate A	Gate B	Gate C
7	I_{a1}	I_{b2}	I_{c3}
8	I_{a2}	I_{b3}	I_{c1}
9	I_{a4}	I_{b5}	I_{c6}
10	I_{a5}	I_{b6}	I_{c5}

Table 4.8: An example DIP group 2 without redundancy. The inputs are shifted inside a "group," yielding one less iteration than the previous DIP group.

DIP group 2. Each iteration in DIP group 2 is able to repeat n inputs since each of the n gates can have an input applied to it. Therefore, the total number of iterations λ_2 needed by DIP group 2 is given by equation 4.1, below. In the special case that $m = n$, this reduces to $\lambda_2 = n - 1$.

$$\lambda_2 = \frac{(n-1)m}{n} \tag{4.1}$$

After DIP group 2, all keys of the form $K(WWC+)$ are guaranteed to be eliminated. Additionally, all keys of form $K(W + C+)$, where an arbitrary number of Ws are applied, are also eliminated. This is because each DIP in DIP group 2 can only have a single input from each iteration in DIP group 1. Thus, any key set corresponding to all inputs from a single iteration, like $K(WWWW)$ or $K(WWWWWWWW)$, will only flip a single gate in DIP group 2.

The remaining hidden keys after DIP group 2 must have two letters from W-Z. The simplest form of these is $K(WWXXC+)$. One important note is that, when there are two letters W-Z in the key, all keys where the number of Ws does not equal the number of Xs are eliminated by DIP group 2. For example, $K(WWWWXX)$ is eliminated because there are 4 Ws and 2 Xs. Therefore, the remaining keys after DIP

group 2 are those that have an equal number of inputs from each iteration in DIP group 1, such as $K(WWXX)$ and $K(WWWWXXXX)$. Note that this does not apply to keys where there are 3 letters W-Z, such as $K(WWWXXX)$, because those would have flipped an odd number of gates in DIP group 1 and been eliminated.

4.2.4 Early Repeat Optimization

When the gates in a directly interfering chain have a different number of iterations required to solve each gate independently, the gate with the largest number of inputs, which we will call the dominant gate, will still need to iterate through all inputs in DIP group 1. This will cause several iterations to have empty input "spaces" where no input is defined for the other gates, as shown in Table 4.9. In this example, key gate A is the dominant gate, and the other key gates iterate through all their inputs before the end of DIP group 1. The spaces are indicated by dashes. While these spaces will need some sort of input value applied, DIP group 1 does not define what they need to be. In fact, they can take any input and still eliminate every key DIP group 1 is crafted to eliminate.

Iteration	Gate A	Gate B	Gate C	Gate D
1	I_{a1}	I_{b1}	I_{c1}	I_{d1}
2	I_{a2}	I_{b2}	I_{c2}	I_{d2}
3	I_{a3}	I_{b3}	I_{c3}	I_{d3}
4	I_{a4}	I_{b4}	I_{c4}	$--$
5	I_{a5}	I_{b5}	$--$	$--$
6	I_{a6}	$--$	$--$	$--$

Table 4.9: When gates have a different number of inputs needed to be solved, empty "spaces" occur in DIP group 1.

The way DIP group 2 eliminates hidden keys created from DIP group 1 is by repeating a single input from each pair of WW inputs. Since any input is able to go in the empty spaces in DIP group 1, those spaces can be used to start repeating the inputs that DIP group 2 needs to repeat before DIP group 2 even starts. This reduces the number of repeats that DIP group 2 needs to make, lowering the minimum number of iterations it requires. Table 4.10 shows such an example DIP group 1, where the highlighted inputs are being repeated. Each of these inputs no longer need to be repeated in DIP group 2.

Iteration	Gate A	Gate B	Gate C	Gate D
1	I_{a1}	I_{b1}	I_{c1}	I_{d1}
2	I_{a2}	I_{b2}	I_{c2}	I_{d2}
3	I_{a3}	I_{b3}	I_{c3}	I_{d3}
4	I_{a4}	I_{b4}	I_{c4}	I_{d1}
5	I_{a5}	I_{b5}	I_{c1}	I_{d2}
6	I_{a6}	I_{b1}	I_{c2}	I_{d3}

Table 4.10: The empty spots in DIP group 1 can be filled with inputs from previous iterations to start to repeat the inputs that cause hidden keys.

Take the number of inputs a specific gate requires to be solved as m_i, where i is the index of the key gate. The number of inputs that can be repeated in DIP group 1 for that key gate is given by $\lambda_1 - m_i$, where λ_1 is the number of iterations in DIP group 1, which is determined by the dominant gate. Since m_i inputs must be repeated, there are $m_i - (\lambda_1 - m_i) = 2m_i - \lambda_1$ inputs left that DIP group 2 needs to repeat for each key gate. If it so happens that $2m_i < \lambda_1$, then all inputs can be repeated in DIP group 1 for that key gate, and DIP group 2 is no longer necessary.

The number of iterations needed for DIP group 2 is then given by the maximum number of inputs left to be repeated for each key gate, given by the equation $max(2m_i - \lambda_1)$. Table 4.11 shows the DIP group 2 for the example key gates that were used in Table 4.10. Key gate D was able to repeat all 3 inputs in DIP group 1, so it does not have any more inputs to repeat. Key gate C was able to repeat 2 of 4 inputs in DIP group 1, so it must repeat only 2 more in DIP group 2. Lastly, key gate B repeated a single input in DIP group 1, meaning it must repeat 4 inputs in DIP group 2. Thus, this DIP group 2 would take 4 iterations instead of the 5 iterations had inputs not been repeated in DIP group 1.

Iteration	Gate A	Gate B	Gate C	Gate D
7	$--$	I_{b2}	I_{c3}	$--$
8	$--$	I_{b3}	I_{c4}	$--$
9	$--$	I_{b4}	$--$	$--$
10	$--$	I_{b5}	$--$	$--$

Table 4.11: When inputs have been repeated in DIP group 1, the number of repeats needed for DIP group 2 is reduced.

The final optimization that can be made is using the dominant key gate to also repeat inputs during DIP group 2. In the previous example, shown by Table 4.11, key gate B takes 4 iterations to repeat all remaining input patterns while key gate A, the dominant key gate, has no useful inputs being applied. Additional input patterns can be offloaded onto the dominant key gate to further reduce the number of iterations necessary. Table 4.12 shows an example. Here, the inputs I_{b4} and I_{b5} have been offloaded onto key gate A. This can be done because it does not matter which key

gate repeats inputs, all that matters is that $n-1$ inputs from each iteration are repeated. So, key gate A repeating inputs from iterations 4 and 5 removes the need for key gate B to repeat those inputs. Thus, this DIP group 2 only takes 2 iterations instead of the original 5.

Iteration	Gate A	Gate B	Gate C	Gate D
7	I_{a4}	I_{b2}	I_{c3}	$--$
8	I_{a5}	I_{b3}	I_{c4}	$--$

Table 4.12: Using the dominant key gate to help repeat inputs can further reduce the number of iterations in DIP group 2.

To find the number of iterations required for DIP group 2, the number of inputs needed to be repeated for each key gate must be tracked. During each iteration, every gate can repeat a single input. However, the dominant gate can be used to repeat an extra input for the gate requiring the most iterations in DIP group 2. The algorithm for this can be seen in Algorithm 2. The input to this algorithm is an array of the number of inputs remaining to be repeated for each key gate.

Algorithm 2 DIP Group 2 Iterations

Input: *inputs_remaining*
Output: *min_iterations*
  $iterations \leftarrow 0$
  while $any(inputs\_remaining > 0)$ **do**
    $largest\_index \leftarrow indexofmaxinputs\_remaining$
    $inputs\_remaining[largest\_index] \leftarrow inputs\_remaining[largest\_index] - 2$
    $inputs\_remaining[!largest\_index] \leftarrow inputs\_remaining[!largest\_index] - 1$
    $iterations \leftarrow iterations + 1$
  end while
  $min\_iterations \leftarrow iterations$

4.2.5 Input-Independent Gates

As was seen in the previous chapter, an input-independent gate can be solved in a single iteration because every incorrect key will flip the signal. To see how this can be extended to directly interfering gates, consider two directly interfering, input-independent key gates. When an incorrect key is applied to both key gates, the original signal of the circuit will be restored. However, unlike input-dependent gates, this effect will happen for every input on the circuit. Thus, this set of two keys has become a correct key since there is no combination in which it will cause the output to be corrupted. Even though both parts of the key are incorrect with respect to their own key gate, their combination has become correct.

Therefore, whenever a chain of directly interfering, input-independent gates flips the signal an odd number of times, that key is incorrect. When the chain flips the signal an even number of times, that key is now correct. As these keys have no input dependence, any input chosen by the SAT attack will eliminate all incorrect keys for this chain. Thus, only a single iteration is needed for the entire directly interfering set of gates.

Input-independent gates can also be combined in direct interference with input-dependent gates. When this happens, the chain of input-independent gates can be modelled as a single gate with some number of incorrect keys and some number of correct keys. This model is valid since both a single input-independent gate and chain of input-independent gates take a single iteration to solve, and it also helps decrease the complexity of the total chain by reducing all input-independent gates into a single unit.

To see how the addition of an input-independent gate affects the minimum number of iterations for directly interfering key gates, consider the simplest case: a single input-dependent gate. On the first iteration of the SAT attack, the SAT solver chooses input I_{a1} for the input-dependent gate. There is no input chosen for the input-independent gate, but it is still useful to list it in the gate notation used in this section, so the pseudo-input I_0 will represent no input. Thus, the DIP provided to the circuit on iteration 1 is (I_{a1}, I_0).

During this iteration, the keys that remain hidden are the ones that flip neither gate or flip both gates. The first category contains keys of the form $K(\ \ I_{a1}, F)$, where $K(F)$ represents keys that do not flip the input-independent gate. The keys that fit into the second category are $K(I_{a1}, T)$ where $K(T)$ represents keys that do flip the input-independent gate. A key will be eliminated when it flips only a single gate, and these keys take the form $K(I_{a1}, F)$ or $K(\ \ I_{a1}, T)$.

In the second iteration, I_{a2} is chosen as the input. Like the first iteration, the keys that are hidden in the second iteration take the form $K(\ \ I_{a2}, F)$ or $K(I_{a2}, T)$. The keys that are eliminated take the form $K(I_{a2}, F)$ or $K(\ \ I_{a2}, T)$. The insight needed is that the key set $K(\ \ I_{a2}, T)$ contains all of the keys in $K(I_{a1}, T)$, which were hidden in iteration 1. Likewise, the keys eliminated in iteration 1 contain all the keys hidden in iteration 2of the form $K(I_{a2}, T)$.

Therefore, after 2 iterations, the only keys that are remaining to be eliminated are of the form $K(\ \ I_{a1} \wedge \ \ I_{a2}, F)$. Since the key applied to the input-independent gate $K(F)$ never flips the signal, it can be essentially ignored. The keys that remain are the ones that would have been hidden had the input-independent gate not been present in the circuit, given by $K(\ \ I_{a1} \wedge \ \ I_{a2})$. In other words, the SAT attack proceeds as

normal without interference from the input-independent gate after iteration 2. The presence of this gate only required two iterations to eliminate all additional hidden keys that it added. Any directly interfering gates that require more than a single iteration to solve will not be affected by the presence of input-independent gates.

4.3 Indirect Interference

To understand how to analyze indirect interference, again consider the simplest possible case: the signal output of two key gates converge at some later gate. If the two gates never mask the effect of the other key gate, they are able be solved independently in the same manner as non-interfering gates. However, if there is a case where a flipped signal from one key gate is stopped from propagating to the output because of the other key gate, more analysis must be done.

Since the gates are not directly in each other's signal path, both gates flipping does not guarantee a collision. Instead, the chance of a collision depends on the type of convergent gate. The truth tables for 3 gate types are shown below: AND, OR, and XOR gates. In each truth table, the traditional inputs and outputs are shown on the left. On the right, the truth table where each input has been corrupted by a key gate is shown. In every output that is highlighted, the flipped output (Out') is the same as the original output, even though both inputs have been modified. This indicates that the two key gates have interfered and produced hidden keys because the output has not been changed.

Therefore, like direct interference, indirect interference can produce hidden keys and require additional iterations to solve. However, unlike direct interference, this is input dependent as it depends on the original signal to the converging gate in

A	B	Out	A'	B'	Out'
0	0	0	1	1	1
0	1	0	1	0	0
1	0	0	0	1	0
1	1	1	0	0	0

Table 4.13: The truth table for a normal AND gate and one where the inputs have been flipped.

A	B	Out	A'	B'	Out'
0	0	0	1	1	1
0	1	1	1	0	1
1	0	1	0	1	1
1	1	1	0	0	0

Table 4.14: The truth table for a normal OR gate and one where the inputs have been flipped.

addition to its type. On one extreme, the unmodified input to a converging AND gate can always be $(0, 0)$, producing zero hidden keys. In this example, the two key gates would be able to be solved simultaneously. On the other extreme, the inputs could be such that a hidden key is always produced, either through a specific input to an AND or OR gate or any input pattern to an XOR gate. In this example, hidden keys would be produced in the same way directly interfering gates, and the same DIP groups would be needed to eliminate all hidden keys.

Because the SAT solver has the freedom to choose the values applied to each primary input, it is likely that an optimized SAT attack can create the conditions necessary to not produce hidden keys, assuming the convergent gate is an AND or

A	B	Out	A'	B'	Out'
0	0	0	1	1	0
0	1	1	1	0	1
1	0	1	0	1	1
1	1	0	0	0	0

Table 4.15: The truth table for a normal XOR gate and one where the inputs have been flipped.

OR gate. However, there may be edge cases where the structure of the circuit makes it impossible to avoid hidden keys. Thus, we can approximate the minimum number of iterations required to complete an attack by assuming the two gates can be solved independently for AND and OR converging gates. When the convergent gate is an XOR gate, there will always be hidden keys, and the analysis from direct interference can be used.

Chapter 5: Results

5.1 Single Key

To control the number of incorrect keys, total keys, and probability of flipping the signal for a single key gate, a modified SARLock block was used. This key block can be seen in Figure 5.1. To change the total number of keys, the number of key bits can be increased or decreased. In the SARLock block, a comparator is used to compare the applied key to a single correct value. This can be changed to compare to a set of values, allowing the number of incorrect keys to be controlled by increasing or decreasing the size of the set. Lastly, the probability the signal is corrupted can be varied by using the Hamming distance comparator from SFLL. A larger Hamming distance will increase the number of keys that flip the signal for a given input, while a smaller Hamming distance will decrease the number of keys.

To test the number of iterations needed for a single key gate, the circuit described above was created using 8 key bits. First, the number of correct keys was varied while keeping the Hamming distance to be a constant. Figure 5.2 shows the number of SAT attack iterations needed for a Hamming distance of 0. When the Hamming distance is 0, the signal is flipped when the applied key equals the input. Thus, this is similar a SARLock block except that it contains more correct keys. The first thing

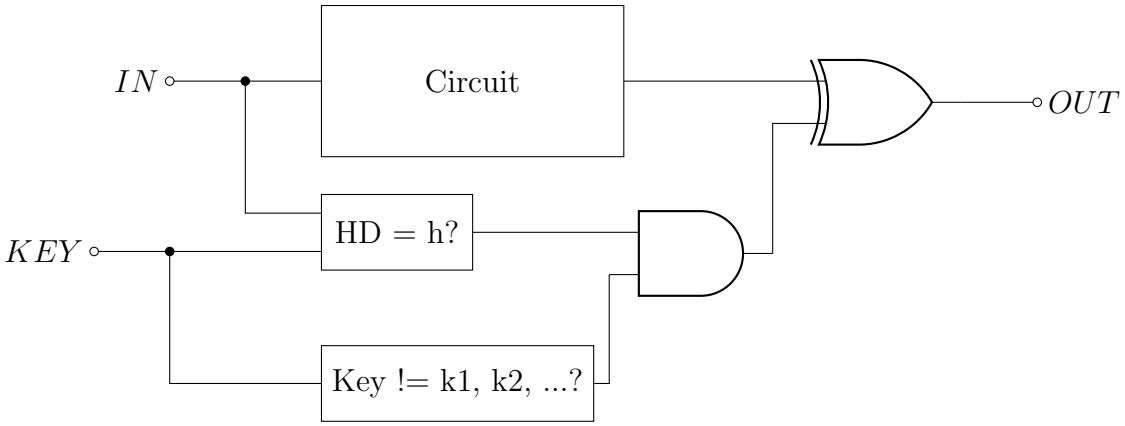

Figure 5.1: A circuit that has a variable number of total keys, incorrect keys, and flip probability.

that can be noticed is that the measured minimum number of iterations equals the actual number of iterations taken for every circuit. This is because the keys $K(I)$ that can be eliminated by each input I are mutually exclusive from each other. So, no iteration will eliminate a key that has already been eliminated. Thus, $UIK_i = IK_i$, and the number of iterations λ will exactly equal the expression in equation 3.4.

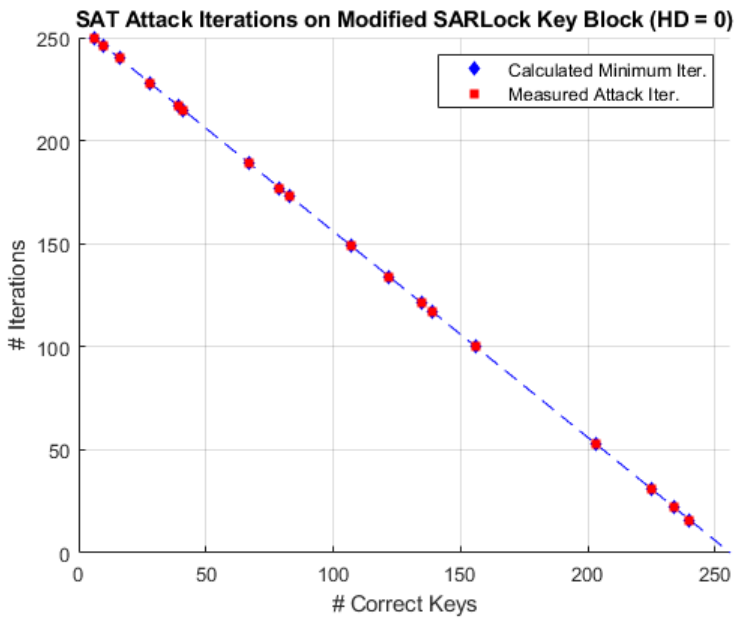

Figure 5.2: The number of iterations when the percentage of correct keys is varied showcases a linear relationship.

Figure 5.3 shows the same setup used on a circuit testing for a Hamming distance of 1. For 8 key bits, a Hamming distance of 1 means that there are 8 keys that will be flipped for every input. Thus, the chance of any key flipping for a given input is $8/2^8 = 1/32$. Dividing that by the total number of keys gives an expected slope of 1/8 for the minimum number of iterations. Here, the number of actual iterations used by the SAT attack is not equal to the minimum number because each input will eliminate redundant keys that have already been eliminated. The actual iterations also form a linear relationship, though with a different slope than the minimum.

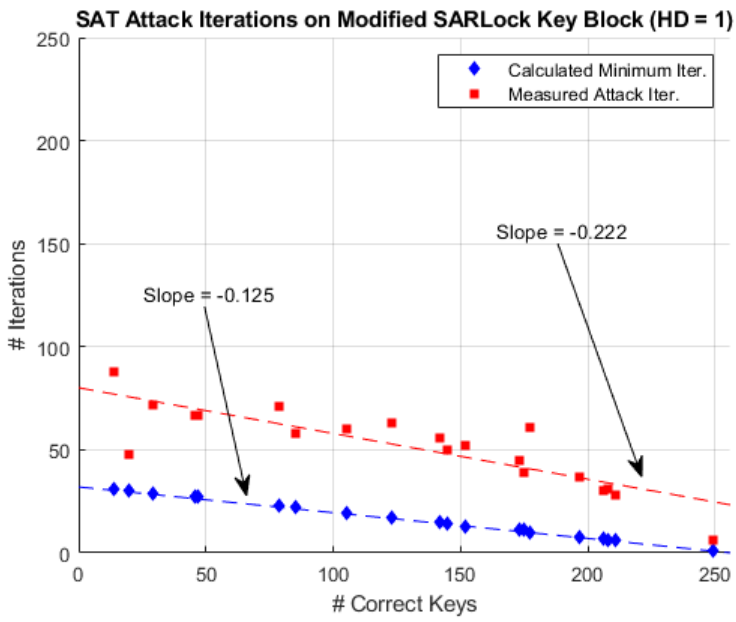

Figure 5.3: The number of iterations when the percentage of correct keys is varied showcases a linear relationship.

Next, the Hamming distance was varied while keeping the number of correct keys to be 20. Each Hamming distance maps to a certain $p(Change)$ value, and as such, the data measured cannot be extracted to arbitrary $p(Change)$ values. Additionally, multiple Hamming distances map to the same $p(Change)$ value. For example, a Hamming distance of 1 and 7 both have 8 keys that can be eliminated each iteration. Therefore, only Hamming distances of 1-4 were tested. Figure 5.4 shows the results for this test. As expected, all circuits with a Hamming distance of 0 required the exact minimum number of iterations to complete, while all other Hamming distances took more iterations than the minimum.

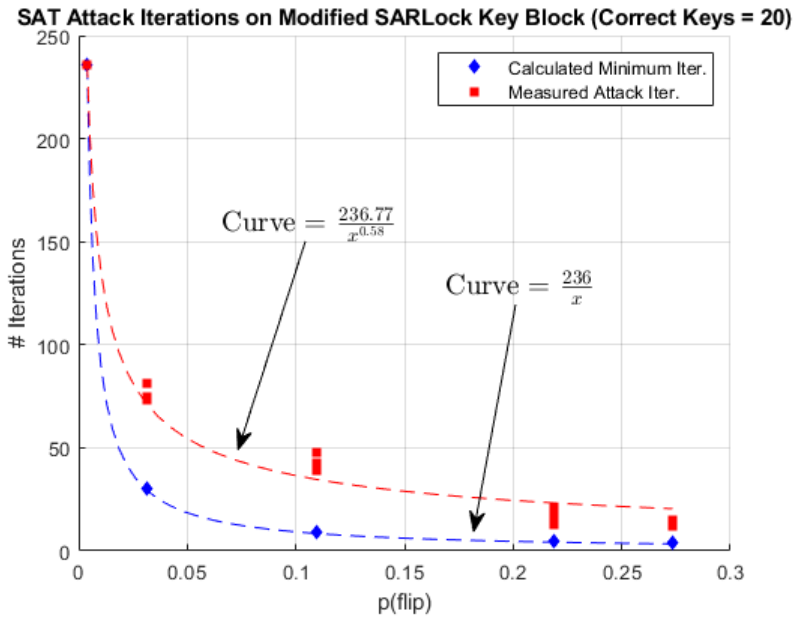

Figure 5.4: The number of iterations when the Hamming distance is varied showcases an inverse relationship.

5.2 No Interference

The minimum number of iterations for no interference was first tested by placing three key gates on random outputs of a circuit. Figure 5.5 shows the measured number of iterations performed by an actual SAT attack compared to the minimum number of iterations calculated. The dotted line has a slope of 1, where the measured number equals the minimum number for any data point on that line. Every circuit took at least the calculated minimum number of iterations to complete the SAT attack.

To see how the number of key gates affects the actual number of iterations, multiple Anti-SAT blocks were placed on the outputs of a circuit. The number of gates placed varied from 2-4 gates, while each gate individually had a minimum number of iterations of 16, 32, 64, 128, or 256. Figure 5.6 shows the results split by the

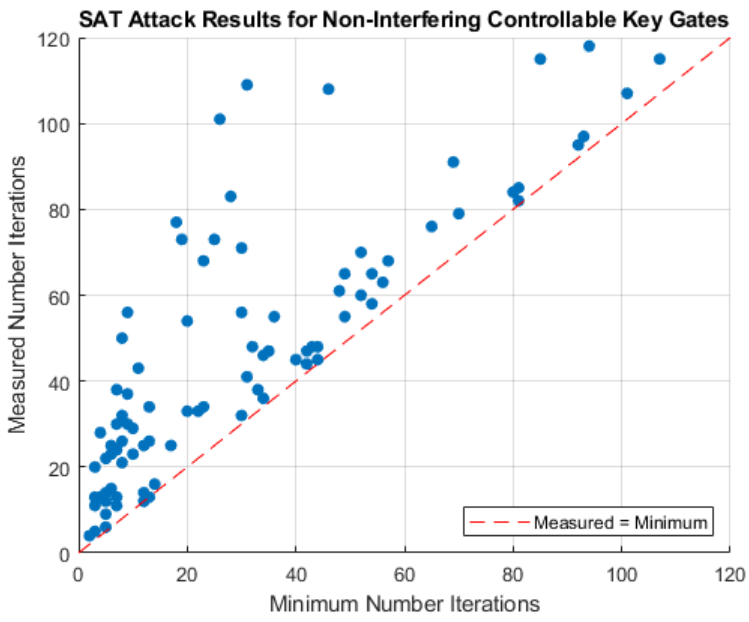

Figure 5.5: Circuits locked with 3 non-interfering gates showing the minimum number of iterations for the SAT attack.

total circuit's minimum number of iterations as well as the number of gates. In each group, the average number of iterations taken by the SAT attack increased with the number of locking gates added. Having more gates in the circuit increases the number of input patterns that eliminate redundant keys. When inputs eliminate mostly redundant keys, the number of iterations increases.

5.3 Direct Interference

Direct interference was first tested by verifying that DIP group 1 and 2 produce a set of inputs that minimizes the number of iterations needed to complete the SAT attack. A chain of 2-6 directly interfering Anti-SAT gates were placed on the output of a random circuit. Each gate had 2-8 input bits in its Anti-SAT block. Because of

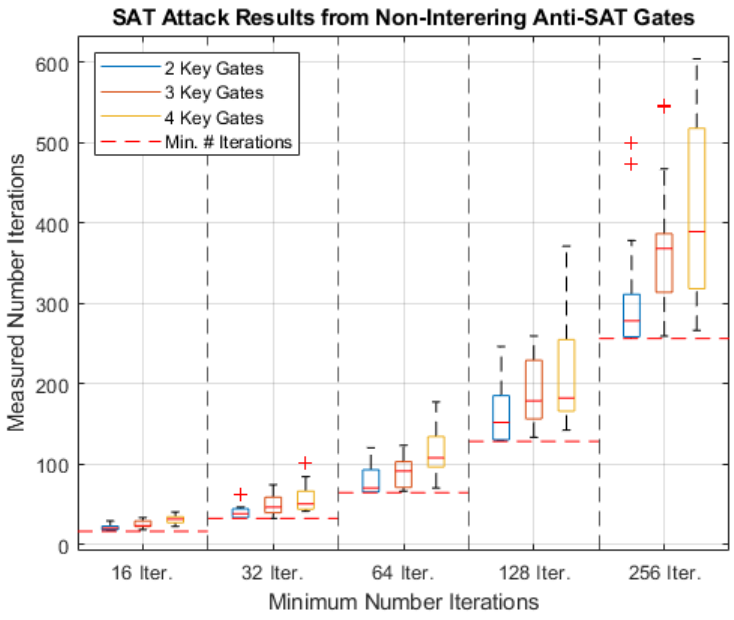

Figure 5.6: Circuits locked with 2-4 non-interfering Anti-SAT gates showing a dependency on the number of interfering key gates.

the nature of Anti-SAT, an increase in the number of input bits doubles the number of iterations needed to solve that key gate independently. Thus, every gate in the circuit can be completely solved in the time it takes the largest gate to be solved, as the extra space in DIP group 1 can be used to completely repeat inputs from DIP group 2. This assumes that two gates do not both share the largest number of inputs, and this constraint was enforced for this test.

The SAT attack on each circuit created was run twice. In the first run of the SAT attack, the SAT solver was free to choose input patterns to the circuit as normal. This gave an accurate picture for how many iterations each circuit took to solve. In the second run of the SAT attack, the inputs from DIP groups 1 and 2 were used, after which the SAT solver chose the inputs itself. The result of this is to show how the

iterations for direct interference can be minimized. Table 5.1 compares the iterations for each run of the SAT attack. For the majority of circuits, the SAT attack was minimized to the number of iterations needed by the dominant gate because the other gates were able to be completely solved simultaneously in DIP group 1. A few circuits with greater than 3 key gates took additional iterations because the $K(WWXXC+)$ keys were not completely eliminated, as DIP group 2 only ensures $K(WWC+)$ keys are eliminated.

Number Gates	Largest # Inputs	Normal Iterations	Chosen Iterations
4	5	134	64
5	5	86	45
2	8	378	256
4	7	198	128
5	8	329	265

Table 5.1: Choosen the inputs to be applied based on DIP groups 1 and 2 minimized the number of iterations needed for the SAT attac.

Another way to visualize that DIP groups 1 and 2 minimize the number of iterations in a SAT attack is shown in Figure 5.7. In this plot, chains of modified SARLock gates were added to an output of a circuit. Each gate in the chain had a randomized number of iterations needing to solve that key independently by changing the number of input bits and number of correct keys for that gate. The minimum line presented is the sum of the number of iterations needed by DIP group 1 and DIP group 2. As can be seen, the number of iterations needed to solve each circuit approaches the minimum value. As in the last set of data, DIP group 3 is not represented here, so

the minimum presented is underestimating the true minimum number of iterations for circuits with 4 or more key gates.

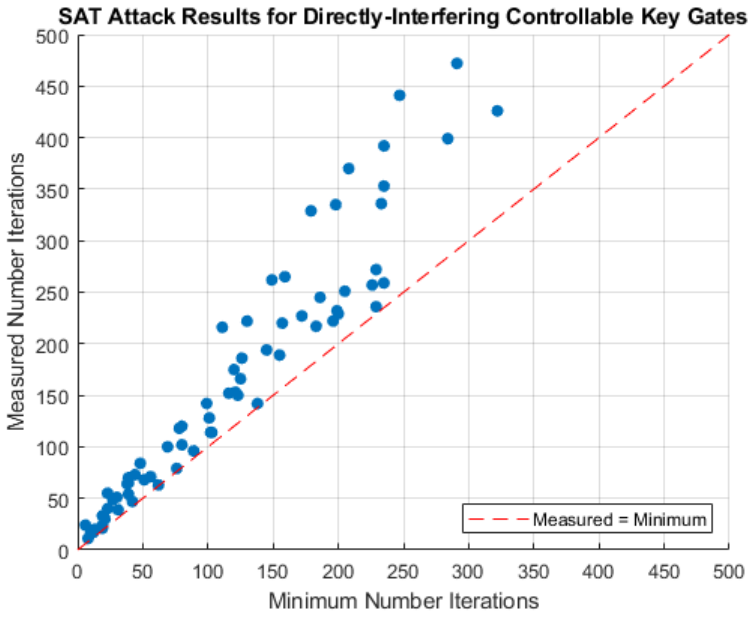

Figure 5.7: The number of iterations for directly interfering gates approaches the calculated minimum number of iterations shown by the dotted line.

Lastly, the effect on the number of key gates in a chain was tested for direct interference. On each circuit, 2-6 Anti-SAT gates with 6 inputs each was placed on the primary output of a circuit. Again, the SAT attack was run using randomized inputs and the chosen inputs from DIP groups 1 and 2. Figure 5.8 shows the results grouped by the number of gates in each chain. The blue boxes show the natural run of the SAT attack, while the orange boxes show when specific inputs were applied each iteration. For the group of 2 and 3 gates in the chain, every SAT attack with the supplied DIPs from DIP groups 1 and 2 took the minimum number of iterations, as shown by the single red line in those groups. This is because DIP group 2 eliminates

all possible keys when there are less than 4 key gates. Once there are 4 key gates, keys of type $K(WWXX)$ are present, and DIP group 2 does not eliminate those keys. In the future, with DIP group 3 accounted for, the minimum number of iterations for chains with 4 or more key gates would be greater than what is shown here, as this is truncating the minimum after DIP group 2.

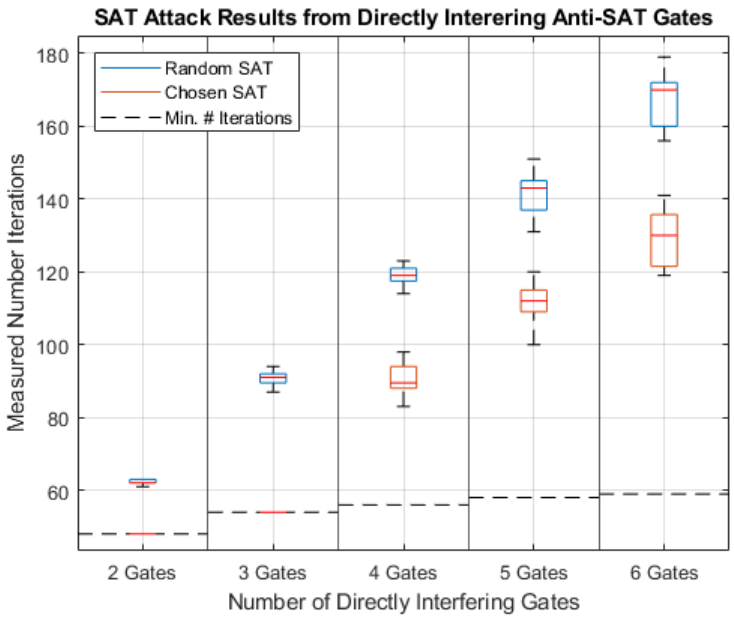

Figure 5.8: The number of iterations grows with the number of directly interfering key gates. Choosing the most efficient inputs each iteration lowers the iterations required.

5.4 Indirect Interference

Since the minimum number of iterations for indirectly interfering gates depends on the type of the convergent gate, the three primary types of convergent gates were tested (AND, OR, and XOR). For every circuit tested, a random number of outputs

to the circuit were converted to internal nets and applied to the input of a newly created convergent gate. The output of the convergent gate was added to the outputs of the circuit. Each original output, now the inputs to the convergent gate, had an Anti-SAT key gate placed on it, creating a variable number of indirectly interfering gates. Figure 5.9 shows an example of this being applied to a circuit that uses 2 key gates.

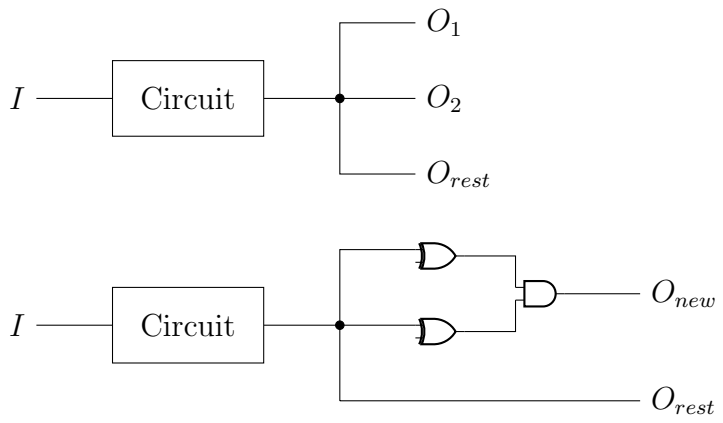

Figure 5.9: A circuit showing how outputs are tied together to create indirectly interfering gates. Here, outputs O_1 and O_2 are removed as outputs from the circuit and replaced with a new output, O_{new}.

First, the minimum number of iterations for all three different types of converging key gates was verified. Table 5.2 shows the results of the SAT attack for 5 trials of each type of gate. These circuits were locked with 2 Anti-SAT gates containing 5 input bits each. In each trial, two SAT attacks were ran against in the circuit. In the first run, the SAT attack was allowed to choose inputs naturally. In the second run, the inputs were provided for the SAT attack to use. For AND/OR gates, only DIP group 1 inputs were provided since those gates can behave like non-interfering

gates. Therefore, any inputs beyond the 32nd iteration were chosen naturally. For XOR gates, both DIP groups 1 and 2 were both applied since XOR gates act like directly interfering gates.

While the natural number of iterations varied, all three gates had some trials take the minimum number of iterations expected. For AND/OR gates, reaching the minimum number of iterations depends on the primary inputs chosen each iteration, so there is variation in the number of iterations when the DIPs were chosen. In contrast, XOR gates always behave like directly interfering gates, and so every trial takes the minimum number.

	AND Gate		OR Gate		XOR Gate	
Trial	Natural	Chosen	Natural	Chosen	Natural	Chosen
1	71	53	69	32	63	48
2	68	32	70	55	63	48
3	67	32	72	35	62	48
4	69	32	67	32	62	48
5	71	51	68	74	61	48

Table 5.2: SAT attack iterations for 2 indirectly interfering gates. Three different types of convergent gates were tested.

Next, the impact of the number of gates was analyzed by using a variable number of 5 bit Anti-SAT gates. The results for this method can be seen in Figure 5.10. The number of iterations taken by the SAT attack varied greatly between trials for AND/OR gates, while the number of iterations was more constant for XOR gates. This is due to the fact that the number of keys eliminated each iteration for AND/OR gates is dependent on the value of the inputs to the convergent gate. In one SAT

attack, the inputs might be chosen to have relatively few iterations, while in another SAT attack, the worst case inputs can be chosen. Meanwhile, there is no preferred input values for XOR gates, meaning the value of the inputs do not affect the number of iterations.

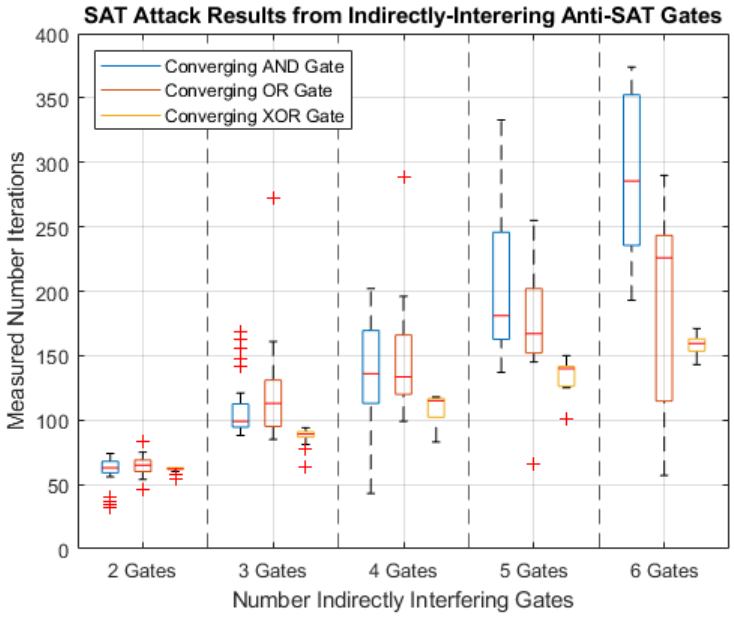

Figure 5.10: The circuits with XOR convergent gates have a more consistent number of iterations required by the SAT attack. The values are similar to those required by directly interfering gates.

5.5 Arbitrary Circuits

Lastly, all types of interference were combined to create arbitrarily locked circuits. For each circuit, a random amount of gates were added and locked with randomized locking techniques. In the first iteration of locking, gates were placed at randomized

locations. Then, subsequent gates were placed in specific locations to create create no interference, direct interference, and indirect interference with the initial key gates.

Figure 5.11 contains the results for arbitrarily locked circuits. The circuits were locked to target the minimum number of iterations to be less than 150 iterations. As can be seen, the actual number of iterations approaches the minimum number but never crosses it. No correlation can be made between the two. There are several groups of circuits with similar minimum number of iterations, most notably around 128 and 64 iterations. This was because these circuits were locked with an Anti-SAT gate that was the dominant factor is determining the minimum number of iterations.

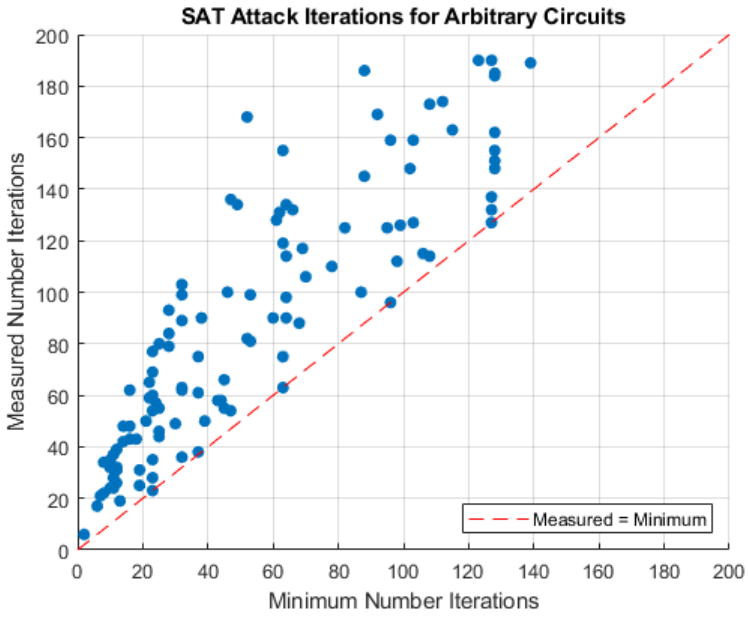

Figure 5.11: The SAT attack was run against circuits locked with arbitrary key gates.

Chapter 6: Contributions and Future Work

In this thesis, the minimum number of iterations required to complete the SAT attack was proposed as a metric for measuring circuit security. The contributions of this work and suggestions for future research will be summarized to conclude.

6.1 Contributions

This work showed that the number of iterations for a SAT attack can be lower bounded based on the structure of the circuit. The actual number of iterations, which is determined by running a SAT attack on the circuit, can be greater than the calculated minimum. For a single key gate, the iterations required to uncover the key is linearly proportional to the percentage of incorrect keys and inversely proportional to the probability a key corrupts the output. The number of iterations taken by an actual SAT attack followed these relationships.

When multiple key gates are placed together, their arrangement determines how many iterations are required to solve them. Gates that are not interfering can be solved simultaneously, and the iterations required to solve is determined by the largest key gate. When key gates are directly interfering, additional iterations are needed because hidden keys are always produced. The number of extra iterations needed depends on the number of gates in the chain and can be determined through the DIP

groups presented in this work. Key gates that are indirectly interfering can behave anywhere between non-interfering gates and directly interfering gates depending on the type of the convergent gate.

A designer can use these results to optimally place key gates to increase the minimum number of iterations required by a SAT attack to break a circuit. These results show that increasing the number of key bits for a single gate is more effective than adding additional key gates as it exponentially increases the number of iterations required. When additional key gates are added, direct interference provides the greatest minimum security, but it does not increases exponentially with the number of key bits.

6.2 Future Work

While this work establishes a minimum number of iterations in many cases, there are additional edge cases that need to be analyzed. One such of these we call hybrid interference, where two types of interference patterns collide. For example, two chains of directly interfering gates can converge and only share some of their gates. This example would be not only indirect interference or direct interference but a combination of the two. Additionally, a directly interfering chain of gates can split into two and have, say, 5 gates in one chain and 6 gates in the other that both share the 3 initial gates. The minimum number of iterations for these configurations is ambiguous and needs to be determined in order to allow this metric to be valid for all possible circuits.

Future work can also be conducted on DIP groups 3 and beyond for directly interfering gates. The analysis in this work only eliminated keys up to $K(WWC+)$, and so the minimum number of iterations calculated serves as an underestimate

to the true minimum number of iterations for larger circuits. Under the assumption that the amount of hidden keys not accounted for remains small, this estimate closely approximates the minimum. However, incorporating more key types, such as $K(WWXXC+)$, will give a more accurate estimate.

Another aspect that is not accounted for in this thesis is the prevention of propagation of corrupt signals in the circuit. This work only analyzes circuits where an incorrect signal is always propagated to the next key gate or output. However, there can be cases where the structure of the circuit blocks the propagation. For example, a corrupt signal can be an input to an AND gate where the other input is a 0. Thus, the output of the AND gate will always be a 0, and the corruption is not propagated. This can allow for an even number of gates in a directly interfering chain to flip the signal and still corrupt the output if some of the effects of the gates are stopped before they reach the other key gates.

Bibliography

[1] Jarrod A. Roy, Farinaz Koushanfar, and Igor L. Markov. Epic: Ending piracy of integrated circuits. In *2008 Design, Automation and Test in Europe*, pages 1069–1074, 2008.

[2] Jeyavijayan Rajendran, Huan Zhang, Chi Zhang, Garrett S. Rose, Youngok Pino, Ozgur Sinanoglu, and Ramesh Karri. Fault analysis-based logic encryption. *IEEE Transactions on Computers*, 64(2):410–424, 2015.

[3] Jeyavijayan Rajendran, Youngok Pino, Ozgur Sinanoglu, and Ramesh Karri. Security analysis of logic obfuscation. In *DAC Design Automation Conference 2012*, pages 83–89, 2012.

[4] Muhammad Yasin, Jeyavijayan JV Rajendran, Ozgur Sinanoglu, and Ramesh Karri. On improving the security of logic locking. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(9):1411–1424, 2016.

[5] Pramod Subramanyan, Sayak Ray, and Sharad Malik. Evaluating the security of logic encryption algorithms. In *2015 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 137–143, 2015.

[6] Yang Xie and Ankur Srivastava. Anti-sat: Mitigating sat attack on logic locking. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 38(2):199–207, 2019.

[7] Muhammad Yasin, Bodhisatwa Mazumdar, Jeyavijayan J V Rajendran, and Ozgur Sinanoglu. Sarlock: Sat attack resistant logic locking. In *2016 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 236–241, 2016.

[8] Muhammad Yasin, Bodhisatwa Mazumdar, Ozgur Sinanoglu, and Jeyavijayan Rajendran. Security analysis of anti-sat. In *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 342–347, 2017.

[9] Muhammad Yasin, Abhrajit Sengupta, Benjamin Carrion Schafer, Yiorgos Makris, Ozgur Sinanoglu, and Jeyavijayan (JV) Rajendran. What to lock? functional and parametric locking. *Proceedings of the on Great Lakes Symposium on VLSI 2017*, May 2017.

[10] Muhammad Yasin, Abhrajit Sengupta, Mohammed Thari Nabeel, Mohammed Ashraf, Jeyavijayan (JV) Rajendran, and Ozgur Sinanoglu. Provably-secure logic locking: From theory to practice. *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, Oct 2017.