Software Implementation of Sensitization Attacks on Obfuscated Circuits

John McLaughlin[1]

Abstract— **Logic obfuscation is one of the many techniques used to protect digital hardware from being counterfeited or reverse engineered. The process involves randomly inserting logic gates, known as "key gates" to an existing digital circuit design, with the extra inputs, known as "key bits", forming a secret key that is necessary for the circuit to function. The Sensitization Attack is a proposed method to break logic obfuscation. Using recursion, a software script is written to simulate a sensitization attack, providing a useful tool to study the efficacy of obfuscation against such attacks.**

I. INTRODUCTION

This paper was written as part of a semester long assignment for ECE 646, Cryptography and Computer Network Security, under Dr. Kris Gaj. The project this paper concerns was commissioned by a PhD research group studying hardware logic obfuscation, led by Dr. Avesta Sasan and consisting of Shervin Roshanisefat, Hadi Mardani Kamali, and Kimia Zamiri Azar.

A growing field of study in the world of cryptography are attacks on the intellectual property of hardware, particularly digital circuits, as well as techniques to protect hardware from those attacks [1]. Bad actors within chip manufacturing facilities have been known to steal chips from the production site and reverse engineer them [2]. Their criminal operation then develops counterfeit chips based on the original design, or more maliciously, tampers with the design and reintroduces the altered chip into circulation. While this is a problem for all industries that rely on these chips, it in particular presents a national security risk, as the defense industry relies on digital hardware for nearly all of its systems [3].

There are many ways to counter these operations. There are several effective methods for detecting counterfeit chips, ranging from physical inspection up to and including X-Ray microscopy [4]. However, even more desirable are techniques to protect against reverse engineering in the first place.

This paper centers around one method of protecting digital hardware: Key-based logic obfuscation. An obfuscated circuit is harder or even impossible for an attacker to determine its design without extra information. Consider logic obfuscation to be a hardware version of message encryption. However, just like message encryption, logic obfuscation can be broken. This paper, and the project it describes, will implement a sensitization attack, an effective method of breaking key-based logic obfuscation.

*This work was not supported by any organization

[1]J. McLaughlin is studying for a M.S. in Computer Engineering, George Mason University, 4400 University Drive, Fairfax, VA

II. BACKGROUND

A. Logic Obfuscation

Hardware logic obfuscation is the technique of protecting the intellectual property of a digital circuit design by inserting extra gates into the circuit [5]. These new gates are referred to throughout this paper as key gates. While one input into these new key gates are the wires of the original circuit design where the key gate was inserted, the other input to the key gate is an extra input into the circuit, referred to throughout this paper as a key bit. All of the key bits together form a key string, the length of which is the same as the number of key gates in the circuit. The purpose of logic obfuscation then is to prevent the circuit from functioning properly without the correct key. The process is similar in theory to key encryption of messages seen in standard cryptography. The difference, of course, is that you are encrypting hardware functionality.

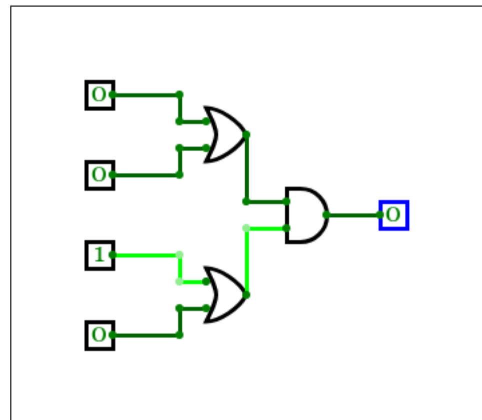

Fig. 1. Standard Digital Circuit

Figures 1 and 2 show an example of logic obfuscation. Figure 1 shows a simple digital circuit, with four inputs and one output. Two inputs each are fed into an OR gate, and the outputs of these gates are fed into an AND gate. If this were a manufactured chip by itself, an attacker could steal the design and easily reverse engineer it. Figure 2 shows the same circuit, but obfuscated: Two XOR gates are randomly placed in the circuit as key gates, with an unknown key bit being fed into them. After the fabrication of the circuit, these key bits are placed in inaccessible memory, preventing an attacker from knowing the key bit values and thus preventing the attacker from knowing the full functionality of the chip.

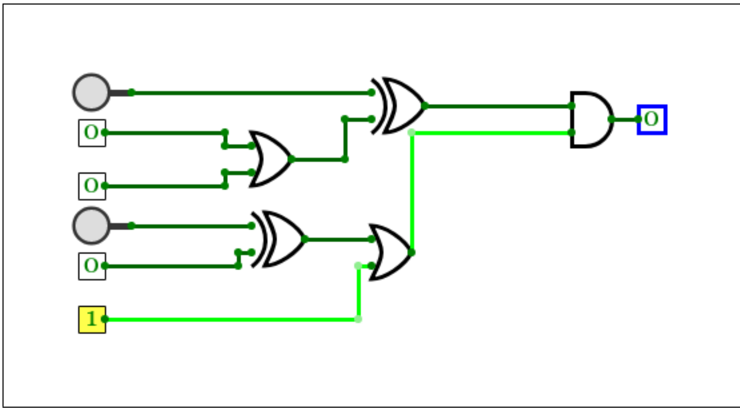

Fig. 2. Figure 1 After Obfuscation

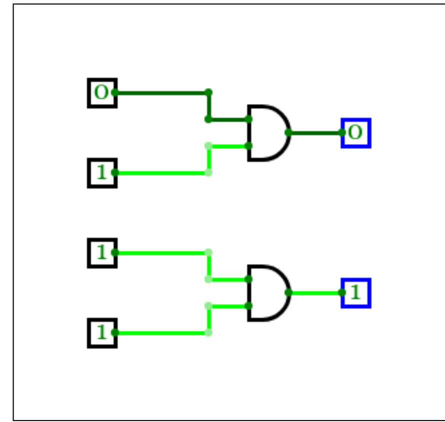

Fig. 3. Propagating Value of an AND Gate

B. Sensitization Attacks

There is, however, a way for an attacker to determine the key bit values. This process is known as a sensitization attack [5][6]. A sensitization attack on an obfuscated circuit is the process in which an attacker tries to determine the value of a key bit by sensitizing it to an output of the circuit. This means that the output of the circuit would be equal to that key bits value, and would change when the key bits value changes. In detail, the sensitization attack would go like this [6]:

- An attacker gains access to a functional chip with valid key bits loaded into memory.
- The attacker gains access to the obfuscated netlist of the circuit.
- The attacker attempts to find the input values that causes the first key bit to propagate to the output.
- If the attacker successfully finds an input sequence that propagates the key bit to the output, they set the inputs of the working chip to that sequence. The resulting output is the value of the first key bit.
- Repeat steps 3 and 4 for all other key bits.

The most important step here, and the one this project is most concerned about, is step 3. Finding the input values that cause a key bit to propagate to an output involves finding specific inputs to the gates in the key bits path that cause the output of these gates to be the same as the key bits value. These values are called propagating values.

Figure 3 shows us that an AND gates propagating value is 1. When you set one input of an AND gate to 1, the output of the AND gate will always be equal to the other inputs value. If the attacker can successfully set the other inputs of each gate in the key bits path to the output to their propagating values, they can propagate the key bit to the output.

However, the real trick is ensuring those inputs can, in fact, be set to the gates propagating values. The output of each gate leading up to that input must be determinable. Complicating this is the fact that there can be key gates along the path to the input you need to set to a propagating value. Since the attacker may not know the value of that particular key bit, this prevents the attacker from generating the propagating value.

Fortunately (or depending on your point of view, unfortunately), the attacker may be able to mute the key bit, thus making whatever value it is irrelevant and provide another chance to generate the propagating value. This is done by setting the other input to the gates muting value. When you set an input to the gates muting value, the output will always be the same, regardless of what the other input is.

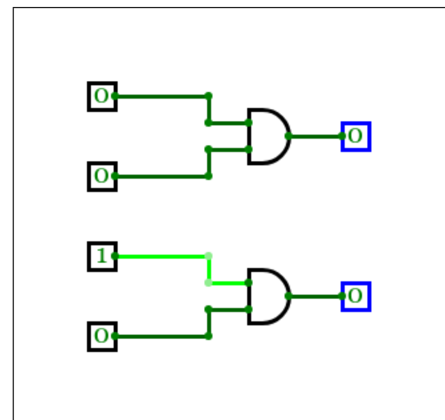

Fig. 4. Muting Value of an AND Gate

Lets take a look at our AND gate, shown in Figure 4. The muting value of an AND gate is 0. No matter what the value of the other input is, the output of the AND gate will be 0 when one of the inputs is 0.

If all key gates leading to a gate that requires a propagating value can be muted, and still maintain the propagating value, the key bit in question can be propagated through the gate.

The key gates in an obfuscated circuit are classified into different groups:

- Isolated key gates: These key gates can be propagated to an output without needing to mute any other key bits. It has no key gates in its direct path, and it is not in the path of any other key gates.
- Dominating key gates: A dominating key gate is in every direct path of at least one other key gate. In order to propagate a dominating key gate, the key bits it is dominating will need to be muted. Note that a key bit

that is dominated by another key gate with an unknown key bit cannot be propagated.

- Convergent key gates: A convergent key gate is not necessary in the direct path of a key bit the attacker is attempting to propagate. Rather, its path shares a gate with the path of that key bit. As such, it needs to be muted as well in order to perform the propagation.

The algorithm initially proposed [5] to perform a sensitization attack involves grouping the isolated, dominating, and convergent key gates and dealing with them in separate loops. Figure 5 shows this algorithm.

```
Input  : Obfuscated netlist, Functional IC, Key Inputs
Output: Original netlist
Determine Runs of Keys;
Replace them with XOR gates;
Update Netlist;
for the remaining keys do
    For each  Isolated Key do
        Compute and apply propagation pattern;
        Determine KeyBits and update Netlist;
    end
    For each Consecutive||Concurrent||Sequential key do
        if there exists a golden pattern then
            Apply the golden pattern;
            Determine KeyBits, Update Netlist, Break;
        else
            ApplyBruteForce(), Break;
        end
    end
    For each  Non-mutable Key do
        ApplyBruteForce(), Break;
    end
end

ApplyBruteForce():
For each possible key combination do
    Generate random input patterns;
    Simulate the patterns and obtain the outputs OP_sim;
    Apply the patterns on IC and obtain the outputs OP_exe;
    if OP_sim==OP_exe then
        Valid Key = current key combination;
        Update netlist;
    end
end
```

Fig. 5. Initial Sensitization Attack Algorithm

III. PROJECT DESCRIPTION

The purpose of this project is to provide a means to simulate a sensitization attack through software. To this end, I wrote a Python script that performs the sensitization attack simulation. The figure below is a block diagram of the scripts inputs and outputs:

The script, whose file name is sa.py, takes in a file as an input, with a .bench format. The .bench file is a description of an obfuscated digital circuit at a combinational gate level. It defines the input wires into the circuit (including the key bit inputs), the output wires coming out of the circuit, as well as all the internal connections and gates in the circuit. In addition, the first line of the .bench file is the valid key bits values for the obfuscated circuit. The reasoning for this will be explained in the next section. The figure below shows an example of a .bench file, with two regular inputs, three key bit inputs, and one output.

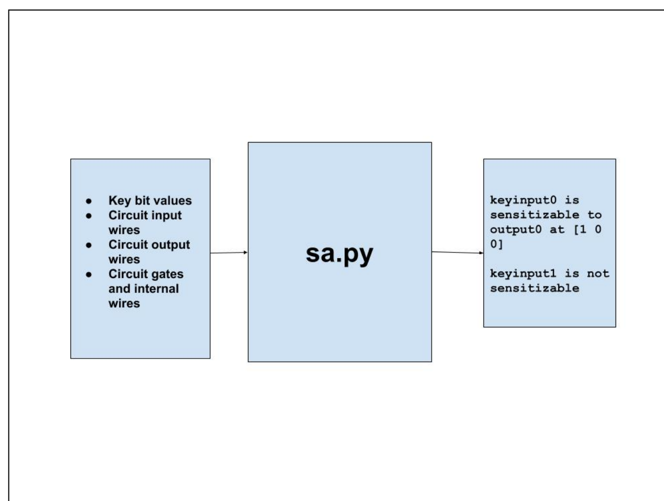

Fig. 6. High Level Block Diagram of Script

```
key=100

INPUT(G11gat)
INPUT(G12gat)
INPUT(keyinput0)
INPUT(keyinput1)
INPUT(keyinput2)

OUTPUT(G39gatenc)

G31gat = xor(G12gat, G11gat)
G32gat = xor(G31gat, G11gat)
G33gat = xor(G32gat, G11gat)
G34gat = xor(G33gat, G11gat)
G35gat = xor(G34gat, G11gat)
G36gat = xor(G35gat, G11gat)
G37gat = xnor(G36gat, G11gat)
G38gat = xnor(G37gatenc, G11gat)
G39gat = xor(G38gatenc, G11gat)
G37gatenc = xor(keyinput0, G37gat)
G38gatenc = xnor(keyinput1, G38gat)
G39gatenc = xor(keyinput2, G39gat)
```

Fig. 7. Script Input File Example

The script outputs text to the console that displays the following information for each key bit:

- Whether the key bit is sensitizable,
- What output the key bit is sensitized to,
- What inputs sensitizes the key bit to the output.

If the value of an input is not relevant to successfully sensitizing a key bit to an output, the value for that input will be displayed as an X.

IV. DETAILED DESIGN

A. File Parsing

[7] The first step the script takes is parsing the .bench file. Most of the code for parsing the .bench file was already

written by someone else, and provided to me by the PhD team. Parsing the .bench file involves two different classes: the Wire class and the Circuit class.

As its name suggests, the Wire class is a structural representation of a wire in the digital circuit. The initial version of this class contained the following fields which were relevant to this project:

- Wire name: The name of the wire given in the .bench file.
- Wire type: The type of gate this wire is outputting from, represented via string (e.g. and, or, xor). If the wire is an input to the circuit, its type is inp.
- Operands: A list of Wire objects, representing the inputs to the gate this wire is outputting from.
- Fanouts: A list of Wire objects, representing the outputs to the gates this particular wire is inputting into.

I modified this class slightly, adding one extra field:

- Value: An integer representing the logic value of this wire. It can take on three different values: 0, 1, and -1. A -1 denotes that a logic value hasnt been set for this wire.

Upon the creation of a Wire object, the value field is initially set to -1.

The Circuit class is a structural representation of the entire digital circuit. The initial version of this class contains the following fields that are relevant to this project:

- Inputs: A list of Wire objects that are inputs to the circuit.
- Outputs: A list of Wire objects that are outputs to the circuit.

I added an extra field to this class:

- Key Values: A list of integers, representing the valid key bits for the obfuscated circuit. This list is populated by parsing the first line of the .bench file, which contains the valid key bit values.

B. Main Algorithm

Once the entire .bench file is parsed and the Circuit class is completely populated, a list of Wire objects is created called keys. All input wires whose name contains key is placed in the keys list, as these are the key bit inputs. Another list called unsolved keys is created and initially set to the same values as the keys list.

An algorithm had already been proposed for finding different key bit values, as shown in Figure 4 earlier in this report. However, I did not find this algorithm to be very conducive for a software approach. Instead, I changed the algorithm, shown below:

```
Loop i through unsolvedKeys:
——propagateForward(unsolvedKeys[i])
——did unsolvedKey[i] propagate to the output?
————Yes:
————————Print that it is sensitizable
————————Set key value
————————Remove unsolvedKeys[i]
————————Set i = 0
————No:
————————Print that it is unsensitizable
```

Fig. 8. Main Algorithm Loop

Setting the key value uses the valid key parsed from the first line of the .bench file. This is necessary because the script may have previously attempted to sensitize a key bit and failed, but would have succeeded if the script knew the value of a dominating key bit or unmutable key bit that has just been sensitized. This is also why the index is reset to 0 when a key bit is successfully sensitized: It allows the script to re-try key bits it failed to sensitize earlier. The script will finish execution when either the list of unsolved keys is empty, or the list of unsolved keys consists entirely on unsensitizable key bits.

C. Propagating Forward

```
propagateForward(wire):
—Is wire an output?
——Yes: return true
—Loop i through wire.fanouts:
——wire.fanouts[i].value = wire.value
——Check gate type
——Get other operand to wire.fanouts[i]
———Is operand a key bit?
————Yes:
—————Is key bit value known?
——————Yes:
———————Is it the propagating value?
————————Yes:propagateForward(wire.fanouts[i])
————————No: return false
——————No: return false
————No:
—————propagateBackward(operand)
—————Did we successfully propagate
backward?
——————Yes: propagateFor-
ward(wire.fanouts[i])
——————No: return false
```

Fig. 9. propagateForward Function

The propagateForward method is what attempts to propagate the key bit to the output of the circuit. It takes in as an argument a Wire object. If the wire is an output to the circuit, the method returns true. The method loops through the fanouts of the wire. For each fanout, the method checks the gate type. Based on the gate type, it looks at the other operand. The method returns false if the operand is:

- an unknown key bit, or
- a key bit whose known value is not the gate's propagating value.

Otherwise, it sets the value of the operand to the propagating value. The gate type and their corresponding propagating values are shown in Table 1.

TABLE I
PROPAGATING VALUES BY GATE TYPE

Gate	Value
AND	1
OR	0
XOR	0
XNOR	1

After the value of the other operand is set to the propagating value, the propagateBackward method is called, with the operand as an argument. The propagateBackward method is explained in the next subsection. The method returns a boolean value, true or false. If it returns true, propagateForward is recursively called again, with the fanout as its argument. This method returns the boolean result of its own recursive call, so if the method reaches the output of the circuit, the first recursive iteration of propagateForward will return true, and the main algorithm will know that the key bit is sensitizable.

D. *Propagating Backward*

```
propagateBackward(wire):
——Check gate type
——Is either wire operand an input?
———Yes: Do we know the input value?
————Yes: Set other operand to proper value
————No: Is it a key bit?
—————Yes: Set other operand to muting value
—————No: Set input to proper value
———No: Set both operands to proper values
——-return propagateBackward(each non-input operand)
```

Fig. 10. propagateBackward Function

The propagateBackward method attempts to recreate a desired output of a gate by setting its inputs and ensuring that the digital circuit can achieve those input values in a deterministic fashion. The method takes in a Wire object as an argument. It immediately checks the gate type this wire is the output for, then it checks the wire's current value.

Depending on the gate type and the wire's value, the method will set one operand to a possible value, and recursively call the propagateBackward method, with the operand as an input. If this returns true, it will set the other operand's value to one that will generate the wire's set output, and call propagateBackward with that operand. If that method returns true, then this recursive iteration of the method returns true as well.

However, there are other factors that are addressed. Before calling propagateBackward again on an operand, the operand is checked to see if it is a key bit. If so, it is then checked to see if its value is known or not. A known value will automatically drive what the other operand's value can be. However, if the key bit's value is not known, the method will set the other operand to the gate's muting value, and then call propagateBackward on the operand.

If this method recursively calls propagateBackward on the wire's operand and returns false, it will change the operand's value if possible, and try again. If it still returns false, the method will set the other operand's value to the gate's muting value and recursively call propagateBackward on that operand. If that returns false, this iteration of the method returns false as well.

If either of the wire's operands is an input to the digital circuit, it will check the operand's value. If it is undefined (i.e. the value is -1), it will set the value without calling propagateBackward on it. If the value is defined (i.e. the value is 0 or 1), it will drive what the other operand's value can be.

TABLE II
MUTING VALUES BY GATE TYPE

Gate	Value
AND	0
NAND	0
OR	1
NOR	1

V. RESULTS

A. *Code Complexity*

The sensitization attack simulation script file "sa.py", including comments, contains 741 lines of code. It consists of three major methods, excluding the main method:

- sa: This method performs a sensitization attack on a digital circuit described by an already parsed .bench file.
- propagateForward: This method recursively attempts to propagate a wire's value to the output of a digital circuit described by the .bench file.
- propagateBackward: This method recursively attempts to find inputs to the circuit that will produce a desired value.

The largest of these methods, and the majority of the code, is the propagateBackward method, with 564 lines of code. In contrast, the propagateForward method has 66 lines of code. The primary reason for this is because of the number of possible input values a gate can have to achieve a desired output. For instance, a 2 input AND gate, in order to get a desired value of 0, could potentially have the following inputs: [0, 0] [1, 0] [0, 1]. These all have to be checked to see which is possible. This also had to be done for each gate type.

Code complexity, particularly in the propagateBackward method (and to a lesser extent propagateForward) can be reduced by introducing indirect recursion [8], creating methods that reduce the amount of repeated code. Repeated code throughout these methods contributes to a lot of the code size.

B. *Testing*

The script was tested on a total of 12 different benchmark circuits, each obfuscated with four different key bit configurations, increasing the number of key bits each time. This

brings the number of total files the script was tested on to 48. The benchmark with the highest number of gates, i8, had 2466 gates, and the benchmark with the lowest number of gates was the c432 benchmark, with 162 gates. These gate counts do not include key gates. The number of inputs (not including key bits) also varied significantly, with the lowest number being 33, belonging to the c1908 benchmark, and the highest amount of inputs belonging to the c2670 benchmark again, with 233 regular inputs.

Success in sensitizing key bits varied with each benchmark, due to the randomness in which key gates were inserted into the design. However, a few patterns emerged. For nearly all benchmarks, an increase in the number of key bits in general decreased the proportion of key bits that were sensitized to an output. This makes sense when you realize that an increased number of key bits also increases the potential for dominating and convergent key gates, both of which make sensitization more difficult. This is most plainly seen in Figure 11, which shows the ratio of successful key bit sensitizations on the c432 benchmark for increasing numbers of key bits.

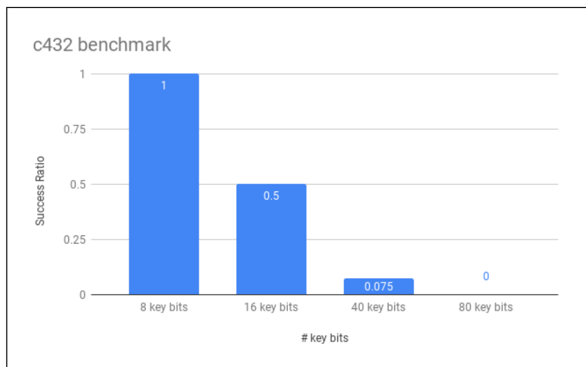

Fig. 11. c432 Testing Results. 162 gate circuit with 36 inputs.

In addition, there is a mild inverse relationship between gate count and success rate in key bit sensitization. Figures 12 and 13 show the gate count of each benchmark and the average success rate for sensitization of each benchmark, respectively. The relationship is not entirely 1:1, due to the randomness in key gate insertion, though fairly large benchmarks such as dalu and i8, have low success rate in sensitization, whereas smaller benchmarks such as c432 and i4 have much higher rates.

It is also worth noting that the success rates in sensitization for each benchmark was fairly low compared to the number of key bits, with the highest average success rate (the i4 benchmark) barely topping 40 percent of key bits. Several key bit configurations for benchmarks were completely resistant to the attack, with the dalu benchmark fending it off completely no matter the key bit size.

C. Discoveries

A number of interesting discoveries regarding digital circuit design and its resistance towards sensitization attacks were found while testing the script. The first of these involves

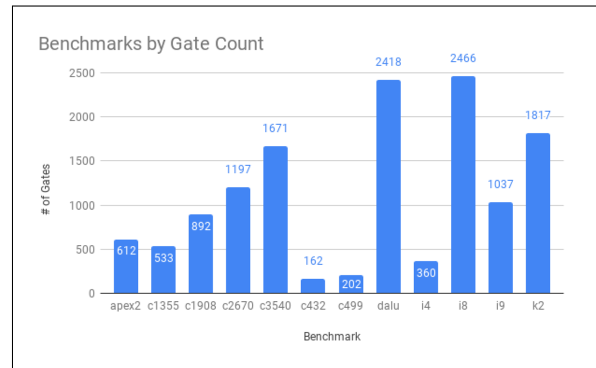

Fig. 12. Number of gates for each benchmark

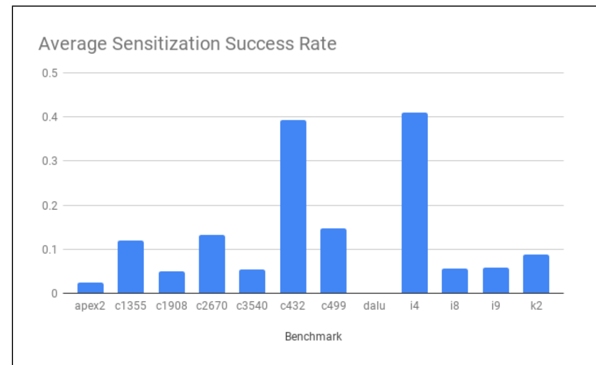

Fig. 13. Averages of sensitization success rates for each benchmark.

propagation. The simplicity of propagation heavily relies on what type of gate the attacker is focusing their attention on. NOR, NAND, and NOT gates do not have propagating values. Instead, they have specific input values that cause the output to be the inversion of the other input, as shown by Figure 14 below.

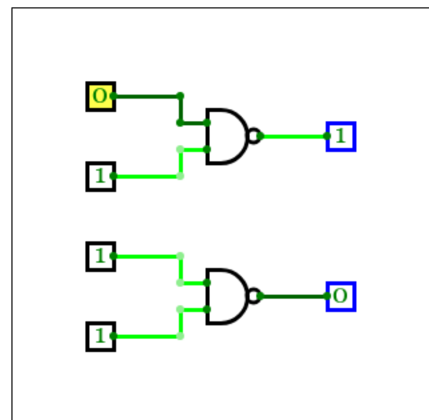

Fig. 14. Inverting Values of NAND Gates

However, this can be potentially remedied by looking at the next gate in the path. If the next gate is another NAND or an XOR gate, feeding a 1 to one input will cause their output to be the inversion of the other input. This causes the original input being propagated to be inverted twice, propagating the original input to the output of the second gate. The same

effect can be made if the next gate is a NOR or XNOR gate and you feed a 0 to its other input. Figure 15 demonstrates this, showing an attempt to propagate a 0, the first input to a NAND gate, to the output.

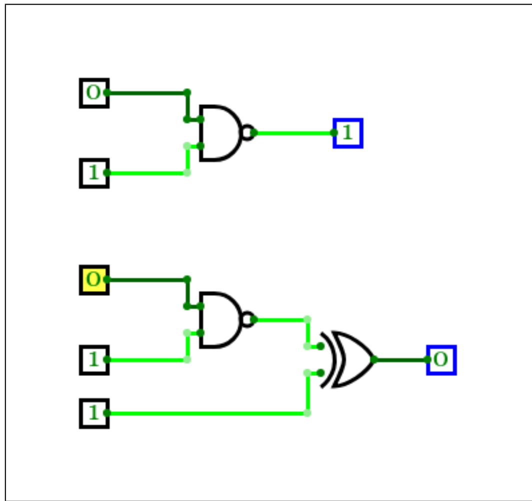

Fig. 15. Propagating Values of NAND Gates

What this means is that key bit propagation can be technically stopped if there is an odd number of NAND, NOR, and/or NOT gates in the key bit's path. "Technically" is the nominal word here, as there is a workaround for the attacker: If the attacker keeps track of how many NAND, NOR, and NOT gates are in a key bit's path to the output, they can also keep track of whether the key bit is inverted or not. If they have this information, this will not stop the attacker from learning the key bit's value if they can only achieve propagating its opposite value to the output.

Another interesting discovery was made, this time regarding muting values. Similar to the discovery explained above, certain gates make muting an input more difficult, or in some cases, impossible. Specifically, these gates are XOR, XNOR, and NOT gates. It is obvious why you cannot mute an input to a NOT gate; there is no other input that can perform the muting. XOR and XNOR gates, however, do not have a direct muting value. While they both have a propagating value, the other value you can enter into the gate will cause the inverse of the other input to be the output. Figure 16 demonstrates this.

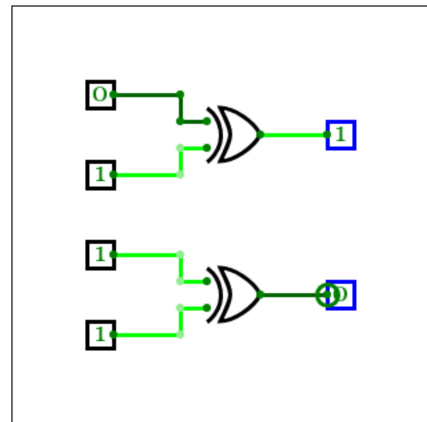

Fig. 16. Inverting Values of XOR Gates

The only way to possibly mute these inputs is to attempt to mute the outputs of these gates. If an XOR gate (or XNOR, or NOT) feeds into a gate with a muting value, such as an AND or an OR gate, you can mute the input(s) to the first gate by feeding the second gate its own muting value in its other input. Figure 17 shows this in action, successfully muting the output of an XOR gate with an AND gate, regardless of the inputs to the XOR gate.

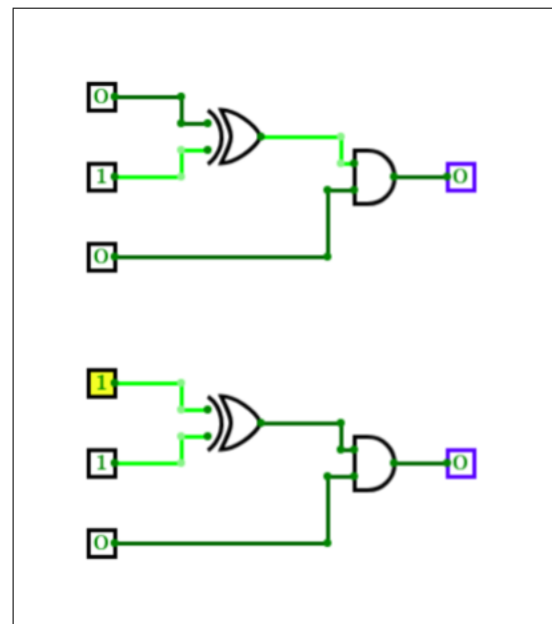

Fig. 17. Muting XOR Gate with AND Gate

What this implies (and shown via direct experimentation) is that an obfuscated digital circuit consisting entirely of XOR, XNOR, and/or NOT gates will completely resist a sensitization attack if there are dominating or convergent key gates in the circuit, as there are no gates to mute those interfering key bits.

This is further proven if you break down an XOR gate into NAND gates, which do have muting values. Figure 18 shows a digital circuit of NAND gates, forming an equivalent XOR gate.

One of the inputs is an unknown key bit. A NAND gate's muting value is 0, but due to the nature of the circuit, feeding a 0 to the other input will fully propagate the key bit to the output of the final NAND gate. But feeding a 1 to the other input will propagate the key bit's inversion, which makes sense since it was previously established that an odd number of NAND gates will achieve this (the key bit's path consists of at most 3 NAND gates). As such, even in this setup, it is impossible to mute an input to an XOR gate by itself.

VI. CONCLUSIONS

Hardware logic obfuscation is one of many techniques used to protect digital hardware intellectual property. What this project seeks to do is create a tool that can be used to test

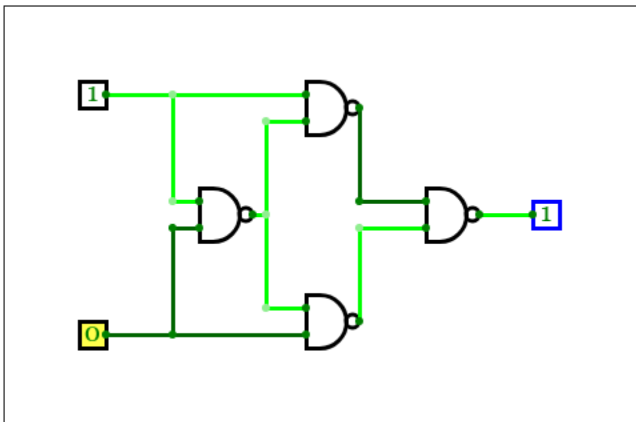

Fig. 18. XOR gate as NAND gates

the strength of a circuit's obfuscation in an academic setting. It has successfully set a foundation for a comprehensive script for testing obfuscated circuit designs, with room for improvement in the future.

Due to the increasing threat of integrated circuit counterfeiting and tampering with integrated circuit design, more research in counterfeit detection and prevention is needed. This script will aid in that endeavor, providing useful data when testing different key-based obfuscation designs and techniques.

ACKNOWLEDGMENT

Dr. Sasan and his team of PhD student researchers, Shervin Roshanisefat, Hadi Mardani Kamali, and Kimia Zamiri Azar were instrumental to the success of this project. In particular, communication with Shervin provided good insight into the background of this project.

Dr. Gaj was also helpful in providing helpful suggestions throughout the project.

REFERENCES

[1] M. Yasin, ”Towards Provably Secure Logic Locking for Hardening Hardware Security”. Diss. New York University Tandon School of Engineering, 2018.

[2] U. Guin, D. DiMase, M. Tehranipoor, ”Counterfeit Integrated Circuits: Detection, Avoidance, and the Challenges Ahead”, 2013.

[3] L. Greenemeier, ”The Pentagons Seek-and-Destroy Mission for Counterfeit Electronics”, Scientific American, April 28, 2017. [Online] Available: https://www.scientificamerican.com/article/the-pentagon-rsquo-s-seek-and-destroy-mission-for-counterfeit-electronics/

[4] K. Mahmood, P.L. Carmona, S. Shahbazmohamadi, F. Pla, B. Javidi, Real Time Automated Counterfeit Integrated Circuit Detection using X-Ray Microscopy. University of Connecticut. [Online]. Available: https://pdfs.semanticscholar.org/fbae/1a2fae29154185ee86953b61459becac3818.pdf [Accessed September 17 2018].

[5] J. Rajendran, Y. Pino, O. Sinanoglu, R. Karri, ”Security Analysis of Logic Obfuscation”. Proceedings of the 49th Annual Design Automation Conference. ACM, 2012.

[6] J. Rajendran, M. Sam, O. Sinanoglu, R. Karri, ”Security Analysis of Integrated Circuit Camouflaging”, Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security. ACM, 2013.

[7] ”Python Course”. [Online] Available: https://www.python-course.eu/

[8] ”Lecture 10: Recursion”, class notes for CMPT 126, School of Computer Science, Simon Fraser University. [Online] Available: https://www.cs.sfu.ca/ tamaras/recursion/