High-Level Approaches to Hardware Security: A Tutorial

HAMMOND PEARCE*, New York University, USA
RAMESH KARRI*, New York University, USA
BENJAMIN TAN*, University of Calgary, Canada

Designers use third-party intellectual property (IP) cores and outsource various steps in the integrated circuit (IC) design and manufacturing flow. As a result, security vulnerabilities have been rising. This is forcing IC designers and end users to re-evaluate their trust in ICs. If attackers get hold of an unprotected IC, they can reverse engineer the IC and pirate the IP. Similarly, if attackers get hold of a design, they can insert malicious circuits or take advantage of "backdoors" in a design. Unintended design bugs can also result in security weaknesses. This tutorial paper provides an introduction to the domain of hardware security through two pedagogical examples of hardware security problems. The first is a walk-through of the scan chain-based side channel attack. The second is a walk-through of logic locking of digital designs. The tutorial material is accompanied by open access digital resources that are linked in this article.

CCS Concepts: • **Hardware**; • **Security and privacy → Hardware attacks and countermeasures**;

Additional Key Words and Phrases: hardware, cybersecurity, scan chain, logic locking

ACM Reference Format:
Hammond Pearce, Ramesh Karri, and Benjamin Tan. 2023. High-Level Approaches to Hardware Security: A Tutorial. *ACM Trans. Embedd. Comput. Syst.* 1, 1, Article 1 (January 2023), 41 pages. https://doi.org/10.1145/3577200

1 INTRODUCTION

Cybersecurity is a system-level challenge with various assets and threats at all levels of abstraction. Throughout the entire lifecycle of embedded computer systems, designers and end-users need to be aware of risks arising from untrusted parties [58]. In digital system design, we are usually concerned with meeting power, performance, and area objectives, where we might also try to improve testability, reliability, and other non-functional properties.

In this tutorial, we present an introduction to basic concepts in the domain of **hardware security**. So that non-experts may gain insights into the mindset and challenges when working in this domain, we will also take you through a hands-on journey using two case studies[1] from domains which currently have active research communities.

Why become more familiar with hardware security? The semiconductor industry continues to become more valuable (possibly reaching US$600 billion in 2022 [29]), with chips used in multiple

*The authors contributed equally to this work.
[1]This tutorial is accompanied by a set of online materials, available at https://github.com/learn-hardware-security, which you can use to follow the case studies.

Authors' addresses: Hammond Pearce, hammond.pearce@nyu.edu, New York University, 6 MetroTech Center, Brooklyn, New York, USA, 11201; Ramesh Karri, rkarri@nyu.edu, New York University, 6 MetroTech Center, Brooklyn, New York, USA, 11201; Benjamin Tan, benjamin.tan1@ucalgary.ca, University of Calgary, 2500 University Drive NW, Calgary, Alberta, Canada, T2N 1N4.

1539-9087/2023/1-ART1 $15.00
https://doi.org/10.1145/3577200

end markets. Researchers and practitioners in digital design need to be well-equipped to deal with threats that arise whenever one produces something of value. Industry and governments continue to invest heavily in tools and techniques for improving hardware security. For example, at the time of writing, the United States' Defense Advanced Research Projects Agency (DARPA) funds the Automatic Implementation of Secure Silicon (AISS) program, which tries to "ease the burden of developing secure chips... [and] provide a means of rapidly evaluating architectural alternatives that best address the required design and security metrics, as well as varying cost models to optimize the economics versus security trade-off" [28]. The Semiconductor Research Corporation (SRC), an industry consortium, features hardware security as one of its priority areas [25], launched following a National Science Foundation (NSF) joint workshop [23]. The European Union is pushing for stronger standards in cybersecurity in both hardware and software [22], which will require more expertise in security-aware design. These are only a few examples of the strong interest in this fast-moving domain, and we hope readers of this tutorial will become inspired to make new contributions to hardware security. Through the case studies in this tutorial, we hope that you will gain an appreciation of how hardware mechanisms need to be scrutinized for their impact on security, as well as gain an understanding of how hardware intellectual property could be protected. In carefully considering the motivations and processes in these case studies, you should become better equipped to explore other areas of hardware security.

The tutorial is laid out as follows. In Section 2, we will talk about hardware security in the broadest sense. The first case study is presented in Section 3, where we will take you through an end-to-end attack on a commonly utilized technique for post-fabrication testing of digital circuits: i.e., an attack on *scan chains*. We will show how an embedded cryptographic key is vulnerable to leaks, even when the key is not itself exposed to the scan chain. Then, in Section 4, we will take you through the basics of logic locking from the ground up. Logic locking is a set of evolving techniques focused on the protection of hardware intellectual property from reverse engineering and piracy, particularly in the context of an untrusted foundry. This case study will work through the motivations and foundations of logic locking and then provide a step-by-step walk-through of one of the pivotal attacks in the literature: the SAT attack [74].

We assume that you, the reader, have a basic understanding of the fundamental topics related to digital system design (e.g., digital logic gates, Boolean algebra) but **no prior knowledge about scan chain security or logic locking**; thus we will start by making you more familiar with security generally before a hands-on dive into those topics. Readers with a bit more background will find the material a good refresher and we encourage you to sample some of the more recent work mentioned throughout the tutorial and at the end of each case study.

After working through this tutorial, you should:

- be able to apply a **security mindset** in thinking about a system;
- understand the role of the scan chain, its benefits, and risks to the system;
- be able to work through a scan-based exploit of a cryptographic design;
- understand the motivations for logic locking, its overall objectives, and the basic principles of the area; and
- be able to analyze and think critically about simple locking.

2 PRINCIPLES OF HARDWARE SECURITY

2.1 The Broader Context

All trustworthy software functionality fundamentally relies on the correct operation of trustworthy underlying hardware. For instance, when we load our banking data via our web browsers, we assume that the computer powering the web browser does not surreptitiously save our details or

leak them to third parties. When we use our bank card to make a purchase, we assume that the hardware doesn't carelessly leave our details in memory. Yet, hardware is often designed with only the desired functionality in mind, not essential security properties, and there are numerous examples in the literature of hardware being broken (e.g. smart cards [47], microcontroller memory protection fuses [73], and on DRAM memory systems such that they leak sensitive data [50]).

While functionality is of course important, as are other design metrics (cost, power consumption, performance, size, and reliability among them), leaving security as an afterthought is dangerous, and the increasing number of hardware-based attacks is highlighting this [11, 58]. That said, industry has recently started acknowledging these risks, and initiatives such as the MITRE Hardware CWEs [26, 48] have begun to classify known design weaknesses.

One major driver of industry adoption comes from the risks inherent in integrated circuit (IC) and printed circuit board (PCB) counterfeiting and reverse engineering. Electronic supply chains are distributed worldwide [58], introducing many possible points of attack. Designing an IC or PCB involves the creation of intellectual property (IP) that may come from third-party organizations or in-house or both, then integrating those components, and generating an IC/PCB layout; a blueprint of the design will then be sent to the manufacturer. Post manufacturing, the designs will be tested, which may be at yet another organization, before being packaged, distributed, and sold, again, with other parties in the loop. Given the high value represented by IP, there is considerable motivation to prevent reverse-engineering or piracy; for example, reverse-engineered ARM microcontrollers available on the grey market with reduced security [53] represent supply-chain risks.

Further compounding the security challenge are the risks of faults entering into the design, either accidentally or maliciously added. Hardware Trojans [81] refer to the category of maliciously introduced fault-inducing artifacts, and they can be caused by adding, altering, or removing components or connections. Unfortunately, designing a product such that it can be easily tested can also provide attack vectors for malicious third parties.

So how do we begin to get a handle on hardware security? In this tutorial, we will use as pedagogical examples two case studies to introduce you to hardware security concepts.

The first is on attacking systems via exploiting scan chains (Section 3), which illustrates how "helpful" hardware components can be manipulated for nefarious purposes. This example serves to demonstrate hardware-based exploitation, thus motivating research work on secure scan and other hardware-assisted security approaches. The second provides an intuitive introduction to logic locking (Section 4), culminating in a walk-through example application of the pivotal "SAT attack" [74]. This example serves as an on-boarding into the domain of intellectual property protection, providing insights into the motivations that drive the "cat-and-mouse" of protecting hardware *itself* from reverse-engineering analysis, theft, and so on. Both case studies focus on the *confidentiality* property of security, being the protection of cryptographic keys (in the scan chain example) and the functionality of a design (in the logic locking example). Readers who are already familiar with security properties should feel welcome to proceed to Section 3; otherwise, the next section provides a quick introduction to security as framed by "the CIA triad."

2.2 Confidentiality, Integrity, and Availability - the CIA Triad

The CIA Triad of confidentiality, integrity, and availability defines the central tenants of all cybersecurity [19, 60]. All attacks and defenses can be viewed within the context of the Triad, and for any device or product to be considered secure, it must address all three facets. These three properties can apply at various levels of abstraction, from the hardware design level through to the system level (potentially involving distributed systems).

Confidentiality refers to the protection of data and resources from unauthorized access. For example, your credit card stores important banking details in its embedded ICs. These details need

to be well-protected to prevent third parties from being able to copy them such that they may use them fraudulently to create new purchases of their own. There are many countermeasures to protect confidentiality, including password access and other access controls, encryption, and physical defenses.

Integrity covers the defense of information from unauthorized alteration. For instance, it should not be possible to alter your transaction (e.g. change the dollar value), once confirmed by the electronics in your bank card. Integrity-based measures provide assurance that data is both accurate and complete. In electronic systems, it is not only necessary to control access to the components in the digital realm, but also in the physical realm, and ensure that authorized users can only alter the information that they are legitimately authorized to alter. Examples of countermeasures in this space can also be based on encryption, such as taking digital hashes or signatures of files; or by duplicating data and storing it in multiple ways.

Availability refers to the need for a given system to be accessible by authorized users. Here, malicious attacks seek to prevent this access and so are often referred to as Denial of Service (DoS). Examples include cryptolockers and ransomware - for instance, the 2021 Colonial Pipeline attack which impacted the availability of gas on the east coast of the United States [61]. Within the context of a hardware-based system, an availability-based attack example could be as simple as stealing the aforementioned banking card or damaging the reader. Availability countermeasures can involve hardware and software duplication and redundancy (i.e., backup systems), and special types of hardware and software to detect if an attack is occurring.

Insight: Given your own experience, reflect on things you consider valuable in a given system. These things are *assets*. Imagine what could go wrong if any of the three properties of Confidentiality, Integrity, or Availability are violated.

2.3 Hardware Security Domains

There are many areas that fall under the umbrella of *hardware security*. For instance, Rostami *et al.* [58][2] pose several hardware security problems, such as IP piracy, reverse engineering, side-channel analysis, counterfeiting, and malicious implants (like backdoors or hardware Trojans) each with a vibrant community of researchers and practitioners exploring new attacks and defenses. Hardware security research also considers how the security of an overall system can be enhanced or mitigated by hardware, such as by the addition of security mechanisms (e.g., dedicated security chips [42]) or by attacks enabled by the oversights in a hardware design (e.g., Hermes (PCIe) [87], Spectre [41], Meltdown [46], RowHammer [40], Scan-chains [82]). From the embedded systems domain [57] through to the cloud [24], hardware security is crucial.

How can one digest such a wide field? One strategy is to consider hardware security as comprising *hardware for security*, i.e., hardware-assisted security (where surveys such as that by Tan [76], Jin [37], Coppolino *et al.* [24], and Brasser *et al.* [14] might be useful starting points), and *security for hardware*, i.e., the area of protecting hardware designs themselves (for this area, works such as that by Rostami *et al.* [58], Chakraborty *et al.* [15], and Xiao *et al.* [81] could provide good starting points for further reading). In examining these works, one should appreciate that security is rarely considered the sole design objective and that there is a tension between security and traditional goals to reduce power consumption/area while increasing performance. It remains an open problem to characterize the trade-offs between security and other design metrics, partly

[2]We reference a number of papers in this section as potential starting points for further reading, but please note that there are many, many works out there—our list is not exhaustive. Any apparent omissions are unintentional, and in fact, we challenge the reader to find the literature that is relevant to your interests, beyond what we've mentioned here. Happy hunting!

because notions and metrics for security vary (e.g., in logic locking, different works evaluate security success and impacts differently [77]).

Insight: When you make your own way through the literature, keep an eye out for different elements, including, but not limited to: the motivation for the work (often, the threat model), the metrics used for evaluation (both in terms of security and its impacts on other design factors), and the experimental platforms used. Do you notice any commonalities in each area of hardware security?

3 CASE STUDY: A TUTORIAL ON ATTACKING SCAN CHAINS

3.1 Overview

Scan chains, a Design for Test (DFT) technique, are implemented in integrated circuits (ICs) in order to test their correct functionality [18, 35]. They provide high fault coverage and do not need complex hardware for test pattern generation or signature analysis.

Fundamentally, a scan chain is a sequential combination of internal registers / flip-flops. They are constructed during synthesis by modifying normal D flip-flops to also include scan logic. This scan logic consists of multiplexers which are daisy-chained so that the D flip-flops can be disconnected from the main combinational circuit to instead sequentially feed data between themselves.

The generic architecture of a scan chain is depicted in Fig. 1. As shown, when the circuit is operating normally, the D flip-flops function normally, taking and returning state to the combinational circuit. However, when the circuit is put into test mode, each D flip-flop is joined using the multiplexers such that their contents may be serially 'shifted' (as if they were a shift register) between one another and out of the circuit. In this context the shifting is referred to as scanning.

In typical implementations of scan chains, the hardware is connected to a five-pin serial JTAG [36] boundary scan interface, where TCK is the test clock signal, TMS selects either normal mode or scan (test) mode, TRST is the reset control signal, TDI is connected to the input of the scan chain and is used to scan in new values, and TDO is connected to the output of the scan chain and is where the internal register values will appear.

However, scan chains represent a significant source of information about the internal functionality and implementation of a given device. As such, designers will typically seek to protect access to the scan architecture. The simplest method for this is to leave access to the scan chain unbound when physically packaging the IC. However, as packages can be broken open for reverse engineering purposes, this is surmountable. Alternatively, or in addition to, the access to the scan chain or JTAG

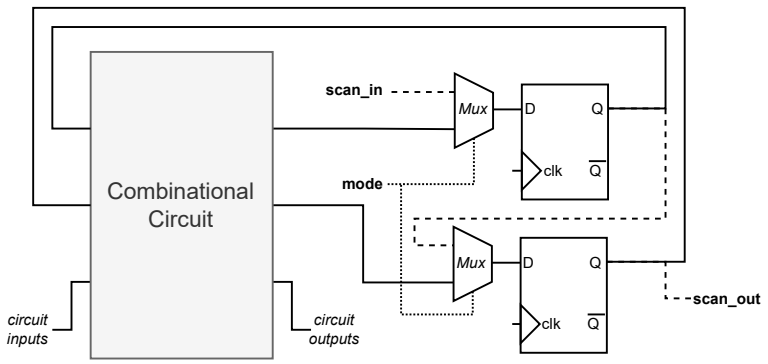


Fig. 1. Generalized scan chain architecture

interface can be controlled via other methods, including setting protection bits, fuses, or using access control passwords. However, given enough time with physical access to the component, protections such as these can still be compromised.

In this tutorial case study, we will examine how a compromised scan chain can be utilized to leak secrets about a given electronic circuit. We consider the case where a cryptographic algorithm is implemented as an application-specific integrated circuit (ASIC). Specifically, we consider the symmetric-key Data Encryption Standard (DES) [51]. We choose DES in this tutorial for three reasons: (1) It is a relatively straightforward algorithm that we can comprehensively describe within this tutorial, ensuring a self-contained case study; (2) Although retired (no longer recommended for new applications), DES was a widely-adopted algorithm used for decades to protect digital secrets (and still sees use within legacy systems); and (3) the process that will be illustrated in this tutorial is also suitable for attacking other encryption standards, including the more advanced and current best-practice Advanced Encryption Standard (AES) [52] algorithm. Scan chain attacks on both DES [82] and AES [4, 83] have both been previously demonstrated in the literature.

Insight: How does this example fit into the CIA Triad (Section 2.2)? A digital circuit with an embedded encryption key would be expected to keep this key *confidential*. For example, consider how an encrypted digital media stream (e.g. satellite TV) should only be broadcast to authorized users (e.g. those with the decoding hardware). In order to maintain this ideal, the decoding hardware should ensure that the embedded cryptographic keys are well-protected.

3.2 Data Encryption Standard (DES)

The Data Encryption Standard (DES) is a symmetric-key block cipher published by the National Institute of Standards and Technology (NIST) in the year 1977, and most recently updated in 1999 [51]. It encrypts 64 bits of data at a time, using a 56-bit key (usually stored as a 64-bit value with checksum bits). DES is a Feistel Cipher implementation, meaning it utilizes a repeating block structure and both encryption and decryption utilize the same algorithm. DES is based on 16 rounds of 64-bit blocks. The high-level structure of the algorithm is presented in Fig. 2. It is separated into two parts, *round key derivation* and *encryption/decryption*.

Round Key Derivation: This is depicted in the right-hand side of Fig. 2. Here, we input the 64-bit key and perform a permutation using table PC1 which re-orders the bits (see Table 3 in Appendix A). PC1 both removes the superfluous 8 parity bits and splits and reorders the remaining 56-bits of key value into two 28-bit halves. Within each round, these two halves are left-rotated independently, either 1 or 2 bits, depending on the specific round of operation (see Table 10 in Appendix A). These two halves are then concatenated and *compressed* to 48-bits by using the PC2 permutation table (see Table 9 in Appendix A). This output is termed as a *round key*. Because the PC2 inputs are rotated between each round, the output of PC2 will change each round—each round will have a different round key. Given that round key derivation is entirely separate from data encryption and decryption, it is possible to precompute and store the round keys if the intended encryption/decryption key is static.

Insight: There are 16 round keys, each 48 bits in size. However, we do not need to break $2^{(16*48)}$ bits of encryption. The round keys were originally derived from the same key, and although this key was 64 bits originally, 8 of the bits are parity bits, and so there is effectively only 56 bits of entropy protecting the encryption.
The 56-bit key length is what makes DES unsuitable for protecting secrets in the modern era—it is short enough that it can be brute forced, and has been vulnerable to this for many years (for example, see the EFF DES Cracker from 1998 [80]).

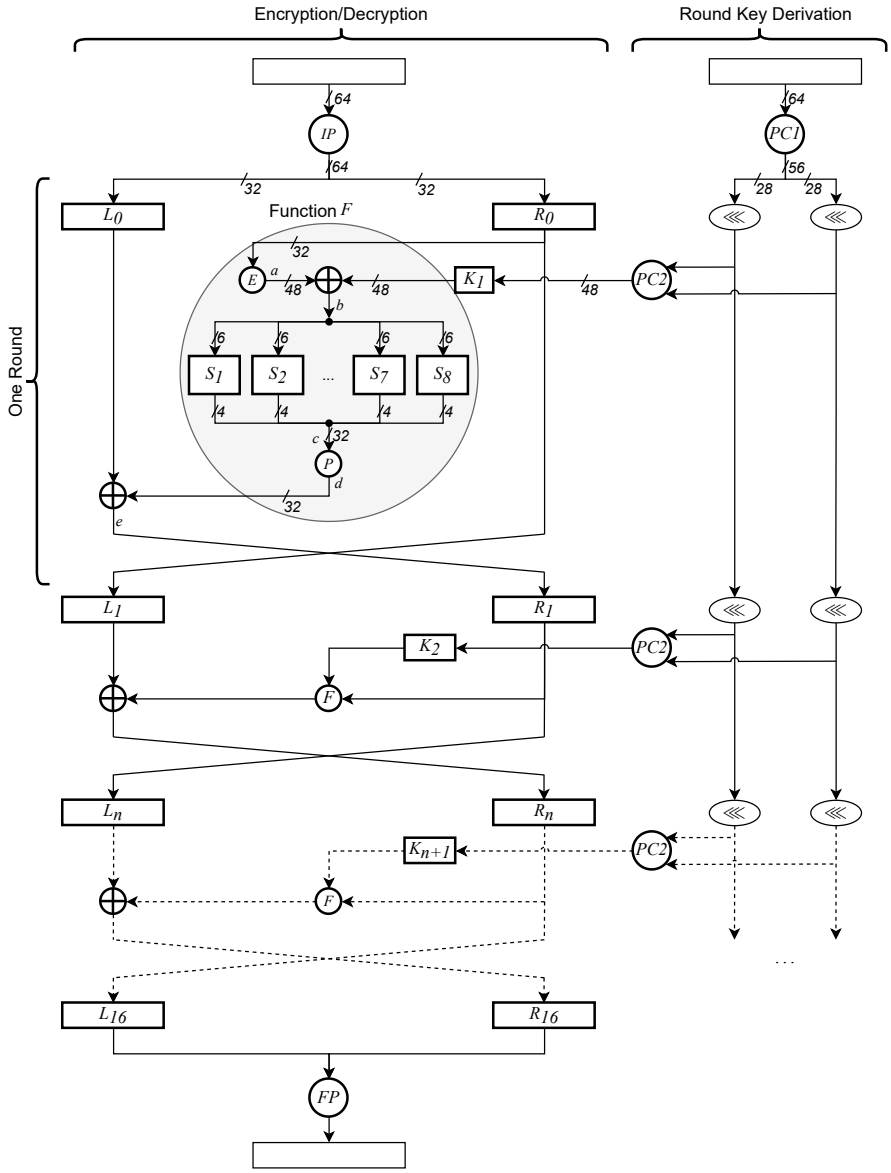


Fig. 2. High level structure of DES.

Encryption/decryption: This is depicted in the left-hand side of Fig. 2. Here, we input the 64-bit data. This is first permuted using the Initial Permutation (IP) table (see Table 3 in Appendix A) before the left-hand and right-hand 32-bits are separated into L_0 and R_0.

Then, in each round n of the encryption / decryption process, the 32-bit value R_n is taken and combined in function F with the appropriate round key k_{n+1}. When encrypting, the round keys are used in the forward order (as is shown in Fig. 2). When decrypting, the round key order is reversed, but no other changes need to be made to the process.

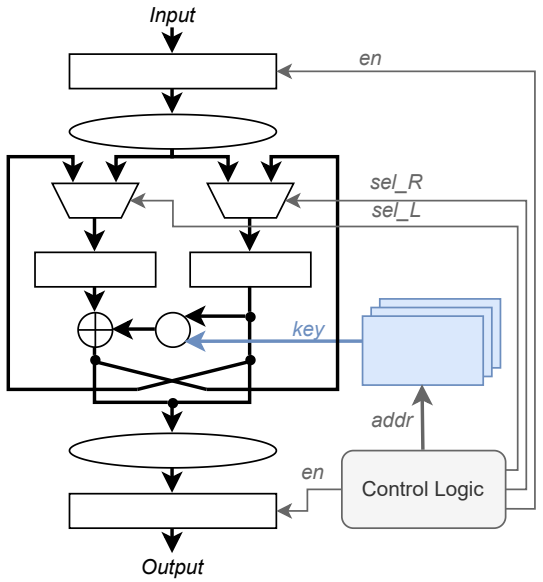


Fig. 3. DES hardware block diagram

Function F in Fig. 2 is detailed as follows. Firstly, the input from the R register is *expanded* using expansion permutation function E (see Table 5 in Appendix A) to 48-bits, making value a. This is combined with the appropriate round key k_{n+1} using an exclusive-or operation to make value b. Each consecutive six-bit block of b is then passed through the appropriate substitution box S (see Table 7 in Appendix A) which makes 4-bit blocks then concatenated to make value c. Value c is then passed through permutation P (see Table 6 in Appendix A) to make value d, which is finally then combined using exclusive-or with the the L_n to make value e. In the next round, R_{n+1} takes the value e, and L_{n+1} takes the previous R_n, unless it is the last round. In this case ($n = 16$), the swap does not occur; and instead, the resultant value is concatenated, before being passed into the Final Permutation (FP) (see Table 4 in Appendix A). The final permutation is derived by using the inverse of the IP. This results in the algorithm's final encrypted output.

3.3 Attacking a practical DES implementation

The DES implementation can be realized in hardware in a number of different ways. One resource-efficient approach is to utilize a single set of L and R registers, and use these iteratively to perform each round of an encryption or decryption. These can be combined with pre-computed round-keys stored in an on-chip read-only memory (ROM). This approach is presented in Fig. 3. Internally, the initial permutation, final permutation, expansion E and permutation P functions are implemented as fixed one-to-one mappings. S-boxes are implemented either as gates or as ROMs. The DES controller primarily consists of a 4-bit counter which will index the appropriate round key for each round, as well as decode logic that will control the enable and select lines.

An implementation of this algorithm featuring a scan chain is what we focus on attacking, although the scan chain will not include the round keys ROM (otherwise the attack becomes trivial) or any registers from the control logic (we discuss this potential later, in subsubsection 3.8.1).

What does an attacker know? Firstly, we assume the attacker knows that the circuit they wish to attack is implementing DES. In addition, as it is public, the attacker knows the details of the DES

algorithm. Secondly, the attacker will know certain details of the implementation. For instance, the attacker would likely be able to obtain vendor-provided timing diagrams of the system. This is important, as from these, the attacker can infer the structure of the given DES circuit - for instance, whether or not it is iterative, pipelined, or purely combinational. In this case study, our architecture is iterative, meaning it operates one round per clock cycle. In the first clock cycle, the permuted input will be loaded to the L and R registers. In the next 15 clock cycles, the L and R registers will be iteratively updated by the result of the round key computations. The output register will be loaded after the final L and R values are permuted. Likewise, the attacker can determine that the round keys are stored internally in the design (in a ROM).

It is reasonable to assume that the attacker will gain access to the scan chains, either via a JTAG port or via breaking open the IC package and directly probing the scan chain ports. However, the attacker will not know the structure of the scan chain. These are typically determined according to the position of the register in the physical layout of the circuit, and would not be known unless the digital design files of the IC (a more protected form of IP than the typically vendor-provided architecture and timing diagrams) were obtained.

Based on the above, we thus need to break our attack into two phases. The first will determine the structure of the scan chain. The second will retrieve a DES round key, and from this, obtain the original DES key. Note that while we follow similar steps to that presented in Yang et. al's work [82], we use an alternative, more consistent nomenclature (where the 'first bit' bit 1 will always refer to the left-most or most significant bit of a value or register) with alternative constants, present the attack steps in more detail (including accompanying code and code discussion), and use an alternative methodology for determining the final key.

Insight: In this subsection we detailed what the hypothetical attacker does and does not know. This process is more formally known as *attacker* or *threat modeling* [70], and it is an important step when considering attacks and defenses for given cybersecurity threats.

3.3.1 Following the case study. In the rest of this section, we proceed with examples and code written in the Python programming language. We use Python version 3.8.10. The listings are designed to be run sequentially and in-order, and are provided with example outputs in immediately-following listings. E.g., Copying and running Listing 1 will give the output of Listing 2, then copy and run Listing 3 to get Listing 4, and so on. Most listings terminate with an `exit()` command (and possibly some `print` statements before these). You can comment these out safely, they are presented only for pedagogical purposes. **The complete code for this case study, as well as the accompanying DES implementation, are available in our open repository at https://github.com/learn-hardware-security/py-des-scan**. This also includes links to a Google Colab/Jupyter notebook.

3.4 DES implementation (Python)

We can emulate the DES implementation from Fig. 3 according to the Python class illustrated in Fig. 4. We define it as having only three public methods (assume all attributes are private).

- `__init__(seed=None, force_key=None)`, which is used to instantiate the DES implementation. A random seed should be provided to derive the random structure of the scan chain, and if no key is provided in `force_key`, will also randomly generate a key. If no seed is provided, the program will set it to numeric 1. In this tutorial, we will focus on determining the value of the key when one was not provided.
- `ReInit()` is used to reset the hardware between runs. It does not recompute keys.
- `RunEncryptOrDecrypt(input, do_encrypt, num_rounds) -> Tuple[str, list]` is the all-in-one function used to run the emulated hardware. It takes a 16-character hexadecimal

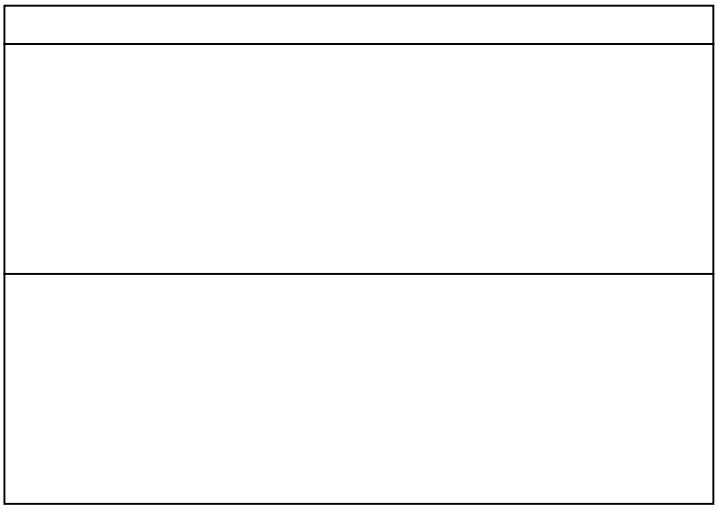


Fig. 4. Simplified UML diagram for `DESWithScanChain` Python Class (attributes and public/private methods).

string as `input`, as well as a boolean `do_encrypt` that determines if the hardware should encrypt (`True`) or decrypt (`False`). It also takes a number of execution rounds `num_rounds`, which refers to the number of clock cycles the hardware should be provided when executing. The function returns two values as a Tuple. The first of these is the value of the output register, encoded as a 16-character hexadecimal string. The second of these is a list of scan chain outputs, as observed between every *round* of execution (i.e. every tick of the encryption hardware). The length of this list will thus be the same as the provided `num_rounds` argument.

We now instantiate the design according to the code presented in Listing 1.

Running example. In this case study we will execute the code at each stage of the analysis. For the example DES instantiation provided, the output is presented in Listing 2. For a more interactive presentation, consider obtaining the code from the associated GitHub repository (see subsubsection 3.3.1).

```
#Useful modules for this tutorial
from typing import *
import itertools

#Import the DESWithScanChain module
# assuming the py-des-scan GitHub repository is downloaded to a folder called 'py_des_scan'
import py_des_scan.des_scan as des

#Define a random seed for the emulated hardware's key and scan chain
seed = 6

#Instantiate the DES module that we will test/attack
dut = des.DESWithScanChain(seed)

#Do a test run of the DES with a given input
test_code = "0BADC0DEDEADC0DE"
print("Input: " + test_code)
(check_ciphertext, _) = dut.RunEncryptOrDecrypt(test_code)
print("Ciphertext: " + check_ciphertext)
(plaintext, _) = dut.RunEncryptOrDecrypt(check_ciphertext, do_encrypt=False)
print("Plaintext: " + plaintext)
```

Listing 1. Tutorial Setup and Instantiating the DES Design Under Test (DUT)

```
Input: 0BADC0DEDEADC0DE
Ciphertext: 5FB5CD14D3136003
Plaintext: 0BADC0DEDEADC0DE
```

Listing 2. Example Output for Listing 1

3.5 Attack Phase 1: Determining the Scan Chain Structure

The first part of the attack deals with reverse-engineering the locations of the flip-flops for the Input, L, and R registers in the scan chain. This is necessary as the scan chain output does not intrinsically reveal the correspondence between the data values it provides and the registers internal to the design. The general process for this is follows.

(1) Reset the DES hardware to clear all registers.
(2) Present DES hardware with input $(8000000000000000)_{16}$ (i.e. 64-bits with the left-most bit (i.e. MSB) set).
(3) Run it for one clock cycle such that the input register is loaded.
(4) Scan out the bit stream pattern (Pattern 1).
(5) Pattern 1 will have one bit active. This position corresponds with the position of the input's currently active bit (i.e., the MSB).
(6) Run it for one additional clock cycle so that the input is loaded into the L and R registers (after they pass through the initial permutation step).
(7) Scan out the bit stream pattern (Pattern 2).
(8) Pattern 2 will have two bits active. As the input is not cleared after loading L and R registers, one of these will match the bit in Pattern 1 and can be disregarded. The other bit represents the bit position in the L or R register after passing through the initial permutation (for example, the first bit of the input will pass through to the 8th bit of the R register).
(9) Repeat steps 1-8, shifting the input by 1 bit each time, 63 more times to determine the position of the remaining input and L/R register bits.

Python code to complete this is presented in Listing 3.

```
#Define arrays for storing the determined indices in the random scan chain
input_scan_indices = [None] * 64
left_r_scan_indices = [None] * 32
right_r_scan_indices = [None] * 32

#Input a single bit in each of the 64 possible positions and run two rounds,
# capturing the scan chains of each cycle.
# In the first cycle, we can determine that bit of the input register
# In the second cycle, we can determine the bit of the L/R register
for i in range(64):
    dut.ReInit() #Reset the hardware

    #Determine the input hex string
    input_num = 1 << (63 - i)
    input_hexstr = '%016X' % (input_num)

    #Get scans for the input (we run in Encrypt mode)
    (_, scans) = dut.RunEncryptOrDecrypt(input_hexstr, True, 2)

    #The scans are 192 bits (represented as ASCII 0/1 characters) long.
    # In scans[0], only one bit will be True;
    # this represents the i-th bit in the Input register.
    for j in range(192):
        if scans[0][j] == "1":
            input_index = j
            break
    #We can store this immediately, using "i" as the position
    input_scan_indices[i] = input_index;

```

```
    # In scans[1], two bits will be True;
    # the one not present in the first scan represents the i-th bit in the L/R registers
    # after Initial Permutation.
    for j in range(192):
        if scans[1][j] == "1" and j != input_index:
            lr_index = j
            break
    #We need to invert the initial permuatation before we can store this
    # For this we can use the Final Permuation table as this is the pre-computed inverse
    lr_pos = des.FINAL_PERM[i] - 1; #table values are 1-indexed
    #The low 32 lr_pos values refer to the L register, high values to R register
    if lr_pos < 32:
        left_r_scan_indices[lr_pos] = lr_index
    else:
        right_r_scan_indices[lr_pos - 32] = lr_index

# Example output
print("input_scan_indices:", input_scan_indices)
print()
print("left_r_scan_indices:", left_r_scan_indices)
print()
print("right_r_scan_indices:", right_r_scan_indices)
```

Listing 3. Code to determine Input, L, and R positions in scan chain

```
input_scan_indices: [20, 152, 61, 26, 71, 78, 145, 110, 60, 36, 11, 2, 176, 140, 85, 130, 55,
     32, 111, 74, 179, 106, 27, 17, 83, 129, 79, 92, 182, 43, 125, 180, 141, 42, 49, 103, 167,
      191, 40, 95, 12, 155, 146, 21, 188, 168, 153, 7, 166, 147, 3, 75, 173, 161, 33, 119, 70,
      0, 171, 35, 144, 73, 25, 139]

left_r_scan_indices: [156, 80, 118, 142, 186, 151, 131, 160, 162, 123, 45, 58, 124, 23, 184,
     127, 10, 117, 143, 63, 189, 190, 159, 38, 99, 102, 51, 132, 4, 47, 112, 15]

right_r_scan_indices: [13, 116, 62, 177, 66, 72, 157, 54, 90, 50, 137, 31, 128, 28, 57, 170,
     59, 1, 29, 91, 67, 172, 136, 134, 165, 185, 121, 120, 133, 37, 164, 34]
```

Listing 4. Example Output for Listing 3

3.6 Attack Phase 2: Determining Round Key 1

Now that we know the location of the L and R values in the scan chain, we can load out their contents. A function to do this process is presented in Listing 5.

```
#Given the set of indices for the L and R registers and the given scan chain output,
# return the contents of the L and R registers.
def read_scan_l_r(left_scan_indices, right_scan_indices, scan) -> Tuple[list, list]:
    l_reg = [None]*32
    r_reg = [None]*32
    for i in range(32):
        l_reg[i] = scan[left_scan_indices[i]]
        r_reg[i] = scan[right_scan_indices[i]]
    return (l_reg, r_reg)

#Example read from l/r regs
test_code = "0BADC0DEDEADC0DE"
(_, scans) = dut.RunEncryptOrDecrypt(test_code, True, 2)
(l_reg, r_reg) = read_scan_l_r(left_r_scan_indices, right_r_scan_indices, scans[1])
print("l_reg contains:", l_reg)
print()
print("r_reg contains:", r_reg)
print()
```

Listing 5. Code to read contents of L and R register from scan chain

```
l_reg contains: ['1', '1', '0', '1', '1', '1', '0', '0', '1', '0', '0', '1', '1', '0', '0',
     '0', '1', '0', '1', '1', '1', '0', '1', '0', '0', '0', '1', '0', '0', '0', '1', '1']
```

```
r_reg contains: ['1', '1', '1', '1', '1', '1', '1', '0', '0', '0', '1', '0', '0', '0', '1',
    '0', '1', '0', '1', '1', '1', '0', '1', '1', '1', '0', '0', '1', '1', '0', '0', '1']
```

Listing 6. Example Output for Listing 5

We wish to observe the intermediate (less protected) encryption data. We are specifically interested in the result after the first round of encryption. Here, only the first round key K_1 has been applied to the data. Consider Fig. 2. Here, the first round of DES can be described using Equations 1-6.

$$a = E(R_0) \tag{1}$$

$$b = a \oplus K_1 \tag{2}$$

$$c = SBoxes(b) \tag{3}$$

$$d = P(c) \tag{4}$$

$$R_1 = e = L_0 \oplus d \tag{5}$$

$$L_1 = R_0 \tag{6}$$

Consider the scenario where a known input is loaded into L_0 and R_0. An encryption round is run, generating L_1 and R_1, which are then scanned out using the scan chain. Given that we know R_0, and E is a simple permutation, we can derive a using the inverse permutation E^{-1} (Equation 1). As we know R_1, we know e; and given that we also know L_0 we can compute d (Equation 5). From this we can compute c by taking the inverse permutation P^{-1} (Equation 4). All that remains is b, which we can then use to calculate K_1.

Let us explore the s-boxes further. In each round of DES, eight different s-boxes $\{S_1, S_2, \ldots, S_8\}$ are used. The first, S_1, is presented in Table 1. Six bits are used to determine which 4-bit value will be taken from an s-box (in other words, each s-box compresses 6 bits to 4 bits). For S_1, it is the six left-most bits (most significant bits) $\{b_1, b_2, b_3, b_4, b_5, b_6\}$. These are separated into row and column indices as indicated in the table. The first and last bits b_1 and b_6 are used to determine the s-box row. The middle four bits, $b_{2\ldots5}$, will be used to determine the s-box column. For example, input $b_{1..6} = (100100)_2$ uniquely identifies value 14 from the table (row 3, column 3).

Table 1. DES s-box S_1

S1	x0000x	x0001x	x0010x	x0011x	x0100x	x0101x	x0110x	x0111x	x1000x	x1001x	x1010x	x1011x	x1100x	x1101x	x1110x	x1111x
0yyyy0	14	4	13	1	2	15	11	8	3	10	6	12	5	9	0	7
0yyyy1	0	15	7	4	14	2	13	1	10	6	12	11	9	5	3	8
1yyyy0	4	1	14	8	13	6	2	11	15	12	9	7	3	10	5	0
1yyyy1	15	12	8	2	4	9	1	7	5	11	3	14	10	0	6	13

Each s-box outputs each possible value exactly four times. For instance, in s-box 1, the value 1 is emitted for input $(000110)_2$, $(001111)_2$, $(100010)_2$, and $(101101)_2$. Because of this, it is not possible to determine the input to an s-box by observing just one output. However, each input to the s-boxes is the exclusive-or combination of the round-key bits as well as value a, which is computed from the expansion E of the register R. This means if we provide multiple values to be combined with the constant round key K_1 we can observe multiple different outputs based on the same key. Given four possible outputs for each value, we thus need to use three different inputs.

Given we can set value a arbitrarily, let us see how to reverse the key bits for the first s-box S_1. Firstly, we apply our first input $a^1_{1..6}(000000)_2$. Given Equation 2 and Equation 3, that means we can derive Equation 7.

$$c^1_{1..6} = S_1(K_{1,1}, K_{1,2}, K_{1,3}, K_{1,4}, K_{1,5}, K_{1,6}) \tag{7}$$

Given that each output of each s-box appears only once in each row, we shall switch one-bit in the column space (i.e. in the middle four bits of b). Thus, we apply our second input $a^2_{1..6} = (001000)_2$ to give our second output in Equation 8.

$$c^2_{1..6} = S_1(K_{1,1}, K_{1,2}, \overline{K_{1,3}}, K_{1,4}, K_{1,5}, K_{1,6}) \tag{8}$$

We then switch two-bits in the input of the s-box S_1, changing both a row and a column. This is done through applying the third input $a^3_{1..6} = (010001)_2$, giving our third value Equation 9.

$$c^3_{1..6} = S_1(K_{1,1}, \overline{K_{1,2}}, K_{1,3}, K_{1,4}, K_{1,5}, \overline{K_{1,6}}) \tag{9}$$

For each of Equations 7-9 we will get four possible values for the key bits input to the s-box. However, only one of these values will be common to all three equations. This will be the final value of $K_{1,1...6}$.

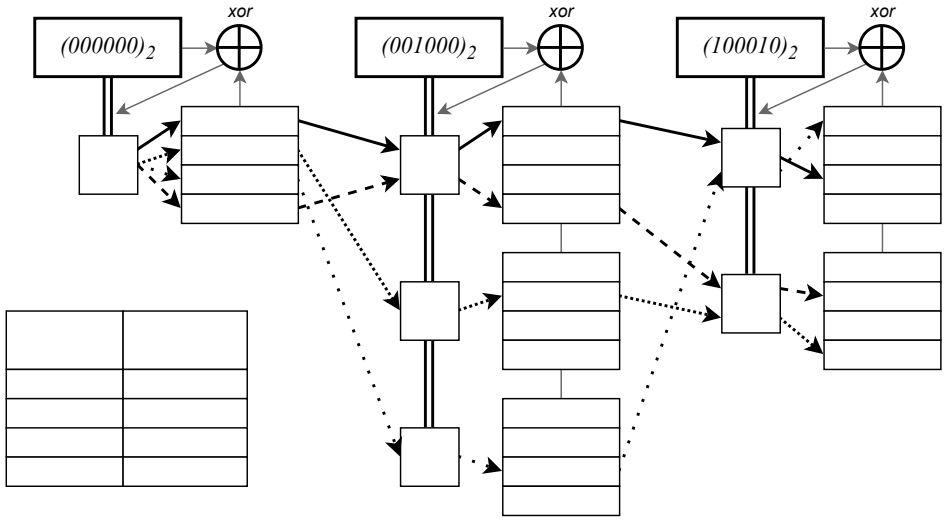


Fig. 5. Reversing Round Key bits $K_{1,1..6}$.

Let us consider an example. We provide input $(000000)_2$ to point $a_{1..6}$, and observe that the s-box returns output 1. Given that the input was all zeros, this means that the round key bits $K_{1,1..6}$ are one of $(000110)_2$, $(001111)_2$, $(100010)_2$, or $(101101)_2$. We then provide input $(001000)_2$. If this returns value 8, then the only matching key bits that could emit this are $(000110)_2$ or $(101101)_2$. If it returns value 4, then the only key bits that could do this are $(001111)_2$. Likewise, if it returns 6, the only possibility is that the key bits are $(100010)_2$. If it did return 8, then we need to provide the third input, $(010001)_2$. If this returns 14, then the only possibility is that the key bits are $(000110)_2$. Alternatively, if it returns 1, then the key bits must be $(101101)_2$. This example is depicted pictorially in Fig. 5, and we present the code for performing the s-box reversal in Listing 7, the code also showing an example of how it can be used to obtain the values for s-box S_1.

```
#Given the concatenated output of the s-boxes (i.e. point 'c' in Fig. 2),
# (as a list of bits)
# and the concatenated input to the xor function (i.e. point 'a' in Fig. 2),
# (also as a list of bits)
# return the list of possible values for each s-box (i.e. list of lists).
def sboxes_output_to_possible_inputs(sboxes_output, sboxes_xor_input) -> List[list]:
    sboxes = []
```

```
    for i in range(8): #for each s-box
        #Get the output of _this_ s-box
        sbox_output = sboxes_output[i*4:(i+1)*4]
        #Get the input to the xor for _this_ s-box
        sbox_xor_input = sboxes_xor_input[i*6:(i+1)*6]

        #Convert the output of the s-box to an integer
        # (the des library stores s-box outputs as integers)
        sbox_value = 0
        for j in range(4):
            sbox_value |= (int(sbox_output[j]) << (3 - j))

        #Find the 4 s-box inputs that produce the given output
        possible_sbox_inputs = []
        for row in range(4): #Every s-box value appears at least once in every row
            col = des.SBOXES[i][row].index(sbox_value)
            #print("Value %i Sbox %i row %i column %i" % (sbox_value, i, row, col))
            possible_input = [
                (row & 0b10) >> 1,
                (col & 0b1000) >> 3,
                (col & 0b100) >> 2,
                (col & 0b10) >> 1,
                col & 0b1,
                row & 0b1
            ]
            #for each bit, undo the XOR operation
            for k in range(len(possible_input)):
                possible_input[k] = possible_input[k] ^ sbox_xor_input[k]

            possible_sbox_inputs.append(possible_input)
        sboxes.append(possible_sbox_inputs)

    return sboxes

#Example s-box reversal for S1
print(sboxes_output_to_possible_inputs([0,0,0,1],[0,0,0,0,0,0]), end="\n\n")
print(sboxes_output_to_possible_inputs([1,0,0,0],[0,0,1,0,0,0]), end="\n\n")
print(sboxes_output_to_possible_inputs([1,1,1,0],[1,0,0,0,1,0]))
```

Listing 7. Code that reverses s-box outputs to a list of possible inputs

```
[[[0, 0, 0, 1, 1, 0], [0, 0, 1, 1, 1, 1], [1, 0, 0, 0, 1, 0], [1, 0, 1, 1, 0, 1]]]

[[[0, 0, 0, 1, 1, 0], [0, 1, 0, 1, 1, 1], [1, 0, 1, 1, 1, 0], [1, 0, 1, 1, 0, 1]]]

[[[1, 0, 0, 0, 1, 0], [1, 0, 1, 0, 1, 1], [0, 0, 0, 1, 1, 0], [0, 1, 0, 1, 0, 1]]]
```

Listing 8. Example Output for Listing 7

If $c^1_{1..6}$ is a value other than 1, we can still use this method to identify the bits of $K_{1,1..6}$. Likewise, we can construct similar patterns for the other s-boxes $S_2 \ldots S_8$. These patterns can then be combined to discover the round key K_1. We derive these patterns here.

We first discuss how to prepare the input a. Recall from Fig. 2 and Equation 1 that it is constructed using expansion E from the R register. The formula for this is presented as Equation 10. Note the constraints within the construction of a, that is, some bits need to match (e.g. a_1 and a_{47} are both

taken from r_{32}, so they must be the same—both either '1' or '0').

$$\begin{aligned} a_{1..48} = \{ & \\ & r_{32}, r_1, r_2, r_3, r_4, r_5, \\ & r_4, r_5, r_6, r_7, r_8, r_9, \\ & r_8, r_9, r_{10}, r_{11}, r_{12}, r_{13}, \\ & r_{12}, r_{13}, r_{14}, r_{15}, r_{16}, r_{17}, \\ & r_{16}, r_{17}, r_{18}, r_{19}, r_{20}, r_{21}, \\ & r_{20}, r_{21}, r_{22}, r_{23}, r_{24}, r_{25}, \\ & r_{24}, r_{25}, r_{26}, r_{27}, r_{28}, r_{29}, \\ & r_{28}, r_{29}, r_{30}, r_{31}, r_{32}, r_1 \\ \} & \end{aligned} \tag{10}$$

Given the constraint from Equation 10, we thus present the desired trio of inputs for reverse engineering all bits of round key K_1 as Equations 11-13.

$$a^1_{1..48} = (000000\ 000000\ 000000\ 000000\ 000000\ 000000\ 000000\ 0000000)_2 \tag{11}$$

$$a^2_{1..48} = (001000\ 001000\ 001000\ 001000\ 001000\ 001000\ 001000\ 001000)_2 \tag{12}$$

$$a^3_{1..48} = (100010\ 100010\ 101000\ 000101\ 010001\ 010001\ 010101\ 010010)_2 \tag{13}$$

However, we cannot just present the desired inputs at location a. One option is to *scan in* the value to the R register using the scan chain and Equation 10 to determine the R bits. However, depending on the circuit, it may have additional barriers to scanning in values as opposed to scanning them out. Another option is to determine the value from the perspective of the initial input. Using the values within the Initial Permutation table, we can derive equations for L_0 and R_0, as presented in Equation 14 and Equation 15.

$$\begin{aligned} L_0 = l_{1..32} = \{ & \\ & i_{58}, i_{50}, i_{42}, i_{34}, i_{26}, i_{18}, i_{10}, i_2, \\ & i_{60}, i_{52}, i_{44}, i_{36}, i_{28}, i_{20}, i_{12}, i_4, \\ & i_{62}, i_{54}, i_{46}, i_{38}, i_{30}, i_{22}, i_{14}, i_6, \\ & i_{64}, i_{56}, i_{48}, i_{40}, i_{32}, i_{24}, i_{16}, i_8, \\ \} & \end{aligned} \tag{14}$$

$$\begin{aligned} R_0 = r_{1..32} = \{ & \\ & i_{57}, i_{49}, i_{41}, i_{33}, i_{25}, i_{17}, i_9, i_1, \\ & i_{59}, i_{51}, i_{43}, i_{35}, i_{27}, i_{19}, i_{11}, i_3, \\ & i_{61}, i_{53}, i_{45}, i_{37}, i_{29}, i_{21}, i_{13}, i_5, \\ & i_{63}, i_{55}, i_{47}, i_{39}, i_{31}, i_{23}, i_{15}, i_7 \\ \} & \end{aligned} \tag{15}$$

Given this, we can substitute Equation 15 into Equation 10 to produce an equation for a based only upon original input i, Equation 16:

$$
\begin{aligned}
a_{1..48} = \{ & \\
& i_7, i_{57}, i_{49}, i_{41}, i_{33}, i_{25}, \\
& i_{33}, i_{25}, i_{17}, i_9, i_1, i_{59}, \\
& i_1, i_{59}, i_{51}, i_{43}, i_{35}, i_{27}, \\
& i_{35}, i_{27}, i_{19}, i_{11}, i_3, i_{61}, \\
& i_3, i_{61}, i_{53}, i_{45}, i_{37}, i_{29}, \\
& i_{37}, i_{29}, i_{21}, i_{13}, i_5, i_{63}, \\
& i_5, i_{63}, i_{55}, i_{47}, i_{39}, i_{31}, \\
& i_{39}, i_{31}, i_{23}, i_{15}, i_7, i_{57} \\
\} &
\end{aligned}
\tag{16}
$$

Thus, from Equation 14 and Equation 16 we can produce the desired input i for each of our inputs $a^{1..3}$ (Equations 11-13). These are presented in Equation 17.

$$
\begin{aligned}
i^1_{1..64} &= (0000000000000000)_{16} \\
i^2_{1..64} &= (0000AA000000AA00)_{16} \\
i^3_{1..64} &= (8220000A8002200A)_{16}
\end{aligned}
\tag{17}
$$

We can verify this answer in Python by presenting these inputs and then computing the value at point a and checking that it matches the desired a-values from Equations 11-13.

```
#Use three specially crafted inputs to determine the unique round key R1
# they ensure L1 is 0, and R1 has a special value in it
special_inputs = ["0000000000000000",
                  "0000AA000000AA00",
                  "8220000A8002200A"]

#Permute the special_inputs to compute the values that will be XORed with the
# round-key R1 before the s-boxes (i.e. compute the value 'a' in Fig. 2)
special_inputs_at_pt_a = []
for i in range(len(special_inputs)):
    after_ip = des.permute(des.hex2bin(special_inputs[i]), des.INITIAL_PERM, 64)
    l0 = after_ip[:32]
    r0 = after_ip[32:]
    r0_expanded = des.permute(r0, des.EXPANSION_FUNC, 48)
    r0_expanded_list = []
    for i in range(48):
        r0_expanded_list.append(int(r0_expanded[i],2))
    special_inputs_at_pt_a.append(r0_expanded_list)

#Example output
#For each value in special_inputs_at_pt_a,
# print it in blocks of 6 bits
for i in range(len(special_inputs_at_pt_a)):
    print("Special input %i (%s): " % (i, special_inputs[i]))
    print("Value at pt. a: ", end="")
    for j in range(len(special_inputs_at_pt_a[i])):
        if(j % 6 == 0):
            print(" ", end="")
        print("%d" % special_inputs_at_pt_a[i][j], end="")
    print()
exit()
```

Listing 9. Code to prepare the trio of special inputs and determine the a values for those inputs

```
Special input 0 (0000000000000000):
Value at pt. a:  000000 000000 000000 000000 000000 000000 000000 000000
Special input 1 (0000AA000000AA00):
```

```
Value at pt. a:  001000 001000 001000 001000 001000 001000 001000 001000
Special input 2 (8220000A8002200A):
Value at pt. a:  100010 100010 101000 000101 010001 010001 010101 010010
```

Listing 10. Example Output for Listing 9

In practice, we cannot observe the values at point c (i.e. immediately after the scan-chains) directly. Instead, we must scan out value e from the R register after the first round is completed (i.e. R is R_1) and perform the inverse operations to return it to value c. Firstly, we must undo the exclusive-or operation against L_0 to compute d (Equation 4). To simplify this, we set L_0 to be all zeros using Equation 14 when setting the input i. This makes $d = e$. We then need to perform the inverse of permutation P. This is a straight-through permutation (i.e. every bit in input appears in different location in output), and so inverting it is straightforward. We present this as Equation 18.

$$\begin{aligned} d_{1..32} = \{ & \\ & c_{16}, c_7, c_{20}, c_{21}, \\ & c_{29}, c_{12}, c_{28}, c_{17}, \\ & c_1, c_{15}, c_{23}, c_{26}, \\ & c_5, c_{18}, c_{31}, c_{10}, \\ & c_2, c_8, c_{24}, c_{14}, \\ & c_{32}, c_{27}, c_3, c_9, \\ & c_{19}, c_{13}, c_{30}, c_6, \\ & c_{22}, c_{11}, c_4, c_{25} \\ \} & \end{aligned} \tag{18}$$

This is presented in code in Listing 11.

```
special_results_after_sbox_pt_c = []
#For each of the 3 special inputs
for special_input in special_inputs:
    #Run 3 rounds of the encryption over the special input (i.e. determine L1, R1)
    (_, scans) = dut.RunEncryptOrDecrypt(special_input, True, 3)

    #Using the scan chain layout we computed earlier, extract the values of L1 and R1
     registers
    (l_reg, r_reg) = read_scan_l_r(left_r_scan_indices, right_r_scan_indices, scans[2])

    #Undo the P permutation to get the values directly emitted from the SBox
    # (i.e. the values at point 'c' in Fig. 2)
    special_result = [None]*32
    for i in range(32):
        special_result[des.P_PERM[i]-1] = r_reg[i]
    special_results_after_sbox_pt_c.append(special_result)

#(For testing only)
#For each value in special_results_after_sbox_pt_c,
# print it in blocks of 4 bits
for i in range(len(special_results_after_sbox_pt_c)):
    print("Special input %i (%s): " % (i, special_inputs[i]))
    print("Value at pt. c: ", end="")
    for j in range(len(special_results_after_sbox_pt_c[i])):
        if(j % 4 == 0):
            print(" ", end="")
        print("%d" % int(special_results_after_sbox_pt_c[i][j]), end="")
    print()
exit()
```

Listing 11. Code to compute the value at point c for each of the special inputs

```
Special input 0 (0000000000000000):
Value at pt. c:  1111 1000 1110 0011 0001 1011 1010 0000
Special input 1 (0000AA000000AA00):
Value at pt. c:  0011 0011 0101 1010 0110 0100 1100 1100
Special input 2 (8220000A8002200A):
Value at pt. c:  1010 1001 0000 1001 1010 0101 1010 0011
```

Listing 12. Example Output for Listing 11

We can now use the values at point c along with the values at point a to determine the possible s-box values according to the code in Listing 7. This is presented in Listing 13.

```
#For each of the s-box special results at point c,
# use the function sboxes_output_to_possible_inputs()
# to determine the possible key inputs given the input
# to the xor at point 'a' in Fig. 2.
sbox_possible_key_values = []
for i in range(len(special_results_after_sbox_pt_c)):
    sbox_possible_key_values.append(
        sboxes_output_to_possible_inputs(
            special_results_after_sbox_pt_c[i], special_inputs_at_pt_a[i]
        )
    )

#Example output
#Print the possible key values for each of the s-boxes for each of the 3 special inputs
for i in range(len(sbox_possible_key_values)):
    print("Special input %i (%s): " % (i+1, special_inputs[i]))
    print("Possible key values...")
    for j in range(len(sbox_possible_key_values[i])):
        print(" for s-box %i: " % (j+1), end="")
        for k in range(len(sbox_possible_key_values[i][j])):
            for l in range(len(sbox_possible_key_values[i][j][k])):
                print("%d" % sbox_possible_key_values[i][j][k][l], end="")
            print(' ', end="")
        print()
exit()
```

Listing 13. Code to determine possible round-key values for each s-box given each of the three special inputs

```
Example output:
Special input 1 (0000000000000000):
Possible key values...
 for s-box 1: 001010 000011 110000 100001
 for s-box 2: 000100 001101 110010 100011
 for s-box 3: 000110 010111 111100 110101
 for s-box 4: 000110 001111 110100 100001
 for s-box 5: 000110 001111 100100 101001
 for s-box 6: 011110 011011 111100 110001
 for s-box 7: 011010 001111 110000 101101
 for s-box 8: 011010 011001 110000 110111
Special input 2 (0000AA000000AA00):
Possible key values...
 for s-box 1: 011000 010101 110000 111101
 for s-box 2: 000100 001001 110010 100001
 for s-box 3: 000110 011101 110000 110011
 for s-box 4: 000110 010011 101000 100001
 for s-box 5: 000110 010111 110000 111001
 for s-box 6: 011110 001101 111100 101001
 for s-box 7: 011010 011111 100000 110111
 for s-box 8: 010100 011001 100010 111011
Special input 3 (8220000A8002200A):
Possible key values...
 for s-box 1: 110000 110011 011000 011011
 for s-box 2: 110010 111001 011010 011101
 for s-box 3: 101010 101101 000110 001111
 for s-box 4: 001001 011010 100001 110100
 for s-box 5: 011011 000110 111001 101000
 for s-box 6: 001101 011110 110111 111010
```

```
for s-box 7: 001111 011010 100101 111000
for s-box 8: 000110 011001 101000 101011
```

Listing 14. Example Output for Listing 13

For each s-box, there is only one possible key which will be present in the outputs for each of the three special inputs. For example, for s-box S_1, the only possible key value which is present for all inputs 1-3 is value $(110000)_2$ (see Lines 31, 41, 51). That means that the round key K_1 must begin with these six bits. We can determine each of these using the code in Listing 15.

```
#Each of the sbox_possible_key_values is a list of lists of possible
# key inputs for that sbox.
# Starting from the first input, remove any possibility that is not present
# in the other inputs.
# (i.e. find the only input that is in all three sets of sbox possibilities)

possible_roundkey_bits_after_expansion = []
for sbox_index in range(8):
    possible_values = sbox_possible_key_values[0][sbox_index]
    for i in range(1,len(sbox_possible_key_values),1): #start at 1
        other_values = sbox_possible_key_values[i][sbox_index]

        #remove any elements from possible_values that is not present in other_values
        possible_values = [x for x in possible_values if x in other_values]

    possible_roundkey_bits_after_expansion.append(possible_values)

#Example output
#Print the possible key values for each of the s-boxes after this removal step
# There should only be 1 possible set of bits per section of the key
print("Possible roundkeys")
for i in range(len(possible_roundkey_bits_after_expansion)):
    print("Bits %i-%i have %i possible value: " % (i*8+1,i*8+8,len(
     possible_roundkey_bits_after_expansion[i])), end="")
    for j in range(len(possible_roundkey_bits_after_expansion[i])):
        for k in range(len(possible_roundkey_bits_after_expansion[i][j])):
            print("%d" % possible_roundkey_bits_after_expansion[i][j][k], end="")
        print(" ", end="")
    print()
exit()
```

Listing 15. Code to reduce number of possible key inputs by removing non-duplicates

```
Possible roundkeys
Bits 1-8 have 1 possible value: 110000
Bits 9-16 have 1 possible value: 110010
Bits 17-24 have 1 possible value: 000110
Bits 25-32 have 1 possible value: 100001
Bits 33-40 have 1 possible value: 000110
Bits 41-48 have 1 possible value: 011110
Bits 49-56 have 1 possible value: 011010
Bits 57-64 have 1 possible value: 011001
```

Listing 16. Example Output for Listing 15

At this point, there should be only one possible round key. However, to enable experimental modifications to these 'special values', we will proceed as if we have not identified the round-key uniquely. If there are more than one possible value for a given section of the round-key, we would need to take the cartesian product of the possible key bits in order to produce a set of possible keys. This is presented in Listing 17. As noted in Listing 18, the round key is uniquely determined.

```
#Convert the set of possible bits per round-key section
# to a set of possible round keys for the round1 key
# by taking the cartesian product of the possibilities
# Note: there should only be one possible sbox input per sbox at this point,
# so the cartesian product should only return one element.
possible_roundkeys_round1 = []
```

```
for components in itertools.product(*possible_roundkey_bits_after_expansion):
    possible_roundkey_round1 = []
    for component in components:
        possible_roundkey_round1.extend(component)
    possible_roundkeys_round1.append(possible_roundkey_round1)

#Example output
#Print the possible roundkeys for round1
for possible_roundkey_round1 in possible_roundkeys_round1:
    possible_roundkey_val = 0
    for i in range(48):
        possible_roundkey_val |= (possible_roundkey_round1[i] << (47-i))
    print("Possible roundkey 1: %012X" % possible_roundkey_val)
exit()
```

Listing 17. List possible round keys K_1 (should only be 1 key).

```
Possible roundkey 1: C321A119E699
```

Listing 18. Example Output for Listing 17

Insight: The three special inputs were well-designed to reverse the s-boxes uniquely and produce only one possible round key K_1. However, what happens if the special inputs are not well-designed, or if we used only one or two special inputs? Consider experimenting with this.

3.7 Attack Phase 3: Determine the original key

Each round key contains 48-bits of the original 56-bit key (8 of the 64-bits in the 64-bit key are parity bits not used for encryption purposes). After determining round key K_1, we now have 48 bits of the key (positions derivable by undoing the PC2, shifts, and PC1 permutations). The attack can now proceed in two different ways. The first method is to perform a similar attack on R_2 and R_3 as depicted above, by using the scan chain to *scan in* replacement values to the L and R registers after each round of encryption is completed. However, this requires additional test infrastructure.

The second method notes that there are only 8-bits remaining of the key, giving only $2^8 = 256$ different key bit possibilities. As such, we should be able to brute-force the values relatively easily.

In this section we will proceed with the second method, and brute force the remaining bits of the key. We present this in code in Listing 19. Note the possible-key list structure printed by Line 40 (example output on Line 70). This has `[0]` or `[1]` in the position of 'known' bits, `[0, 1]` in position of 'unknown' bits, and `[None]` in position of the parity bits (every 8th bit). As there are 8 'unknown' bits, there should be 256 possible keys generated from the cartesian product, which is confirmed for the example on Line 71.

```
possible_keys = []
#For each possible round1 key, derive every possible key that could have generated it
for possible_roundkey_round1 in possible_roundkeys_round1:
    #First, undo the PC2 permutation
    key1 = [None]*56
    for i in range(48):
        key1[des.KEY_PC2[i]-1] = possible_roundkey_round1[i]

    #Now undo the two half-key rotations by
    # right rotating each half of the key
    key1_left = key1[:28]
    key1_right = key1[28:]
    key1_left = key1_left[-1:] + key1_left[:-1]
    key1_right = key1_right[-1:] + key1_right[:-1]
    key1 = key1_left + key1_right

    #Now undo the PC1 permutation
    key = [None]*64
    for i in range(56):
        key[des.KEY_PC1[i]-1] = key1[i]
```

```
21
22      #The format of the key is such that it has 64 bits,
23      # but only 48 of them are currently filled with values (the others are 'None')
24      # We will create all possible keys by taking the cartesian product
25      # of all possible values for the 8 unfilled key bits (ignoring parity bits).
26
27      #Prepare the cartesian product by creating a list of lists of
28      # known and possible values for the unfilled key bits.
29      combined_key_possibilities = []
30      for i in range(64):
31          if key[i] == None:
32              if (i+1) % 8 != 0:
33                  combined_key_possibilities.append([0,1]) #Unknown key bit
34              else:
35                  combined_key_possibilities.append([None]) #Parity bit
36          else:
37              combined_key_possibilities.append([key[i]]) #Known key bit
38
39      print("Key possibilities that would generate round key 1:")
40      print(combined_key_possibilities)
41
42      for components in itertools.product(*combined_key_possibilities):
43          #Combine all key bits into a single key
44          possible_key_bits = []
45          for component in components:
46              possible_key_bits.append(component)
47
48          #Calculate the parity bits
49          for i in range(8):
50              val_bits = possible_key_bits[i*8:(i+1)*8-1]
51              parity_bit = 0
52              for bit in val_bits:
53                  parity_bit ^= bit
54              possible_key_bits[i*8+7] = parity_bit
55
56          #Convert the binary list into a hex string
57          key_val = 0
58          for i in range(64):
59              key_val |= (possible_key_bits[i] << (63-i))
60          possible_key_val = '%016X' % key_val
61
62          #Store the possible key
63          possible_keys.append(possible_key_val)
64
65  #Example output
66  print("There are %d possible keys." % len(possible_keys))
```

Listing 19. Determining the list of possible keys that could generate round key K_1.

```
Key possibilities that would generate round key 1:
[[0], [0], [0], [0], [1], [0, 1], [0, 1], [None], [0], [1], [0, 1], [0, 1], [1], [1], [1], [
    None], [0], [0], [1], [0], [1], [None], [1], [0], [0], [0], [0], [1], [1], [
    None], [1], [0], [0], [0], [1], [1], [0], [None], [1], [0], [0, 1], [1], [0], [0, 1],
    [0], [None], [0], [0, 1], [1], [0, 1], [1], [1], [0], [None], [1], [0], [1], [0], [0],
    [0], [0], [None]]
There are 256 possible keys.
```

Listing 20. Example Output for Listing 19

The remaining step is to now perform the brute-force operation to check all the possible keys until the correct key is found. This is straightforward by creating new DES instances with specified keys (refer to Fig. 4). We will compare the ciphertexts that we generate to that generated in Listing 1. The brute-forcing code is presented in Listing 21.

```
1  print("Brute-force checking %d possible keys." % len(possible_keys))
2  for possible_key in possible_keys:
3      pos_des = des.DESWithScanChain(force_key=possible_key)
4      (test_ciphertext, _) = pos_des.RunEncryptOrDecrypt(test_code)
5      if(test_ciphertext == check_ciphertext):
```

```
        print("Found the key. It is %s" % possible_key)
        break

print("Checking the answer. The embedded secret key was " + dut.key_hex)
if(possible_key == dut.key_hex):
    print("The two keys match, the attack is successful.")
```

Listing 21. Brute forcing the key from the list of possibilities

```
Example output:
Brute-force checking 256 possible keys.
Found the key. It is 096F2B878D906CA0
Checking the answer. The embedded secret key was 096F2B878D906CA0
The two keys match, the attack is successful.
```

Listing 22. Example Output for Listing 21

The attack successfully recovered the key used to generate the round keys in the hardware.

3.8 Discussion

3.8.1 Design exploration. In this case study we explored how to perform a scan chain attack on (emulated) hardware. We presented relatively simple iterative hardware, but this attack would also work on more complex implementations. For instance, in a fully pipelined architecture, each of the 16 DES rounds would be instantiated separately, meaning there would be 17 pairs of L and R registers (from $\{L_0, R_0\}$ to $\{L_{16}, R_{16}\}$). While this would significantly increase the size of the scan chain, identifying the correspondence between each bit and its position in the scan chain is still possible, especially for the low registers [82]. L_0 and R_0 can be located first, by using the same methodology as presented earlier. As $L_1 = R_0$, identifying L_1 is also straightforward. Given $R_1 = L_0 \oplus f(R_0, K_1)$ we can determine R_1 as follows. We set R_0 to be a constant value; K_1 is already constant. We then iterate through L_0, setting each bit to be 1 in turn. As $f(R_0, K_1)$ is constant, there will only be a 1-bit difference each time we stream out R_0, and that difference will be based on the position of the 1 in L_0. Now we have the positions of L_0, R_0, L_1, and R_1 and the key extraction can proceed as outlined in the previous subsection (using the brute force method).

Our emulated hardware assumed that it took one clock cycle to load values into the input register. For some DES implementations this might take a different number of cycles, or no cycles at all if there is no input register present. However, using the scan chain it also possible to determine this. Firstly, the length of the scan chain reveals how many register bits are present. Secondly, by observing the scan chain after each clock tick, it is possible to determine when encryption has commenced. This is because cryptographic algorithms such as DES display an 'avalanche' effect [82], where small differences (i.e. 1-bit) will translate into larger changes in the subsequent rounds. By observing for this, it is possible to determine when the encryption has started [82].

Our emulated hardware simplified some aspects of the reverse engineering by excluding control registers from the scan chain. If these were present, there would be additional and varying data bits in each cycle. However, these are control-driven rather than data-driven. As such, we can identify them by loading multiple different inputs to the hardware module and running the complete encryption/decryption cycle. Any flip-flops in the control unit would have the same pattern for each input trace, no matter the input, and could thus be identified and discarded. Further, we assume that a reset actually sets all internal register bits to zero. This may not be the case, as some circuits have implementations that reset to unknown or random garbage values. If this is the case, then instead of resetting the circuit using the random reset, the attacker should 'reset' the circuit by loading a constant input and running the complete encryption procedure. This would set all internal registers to a constant value, which can be the new fixed state of the chip rather than all bits being zero.

3.8.2 Potential pedagogical exercises. This case study presents a walk-through of an attack on a DES implementation, providing a sound basis for academic exercises. Consider modifying the design (e.g. using the suggestions in Section 3.8.1) prior to attempting the attack. Also consider attacking a different encryption algorithm that is vulnerable to scan attacks, such as AES [4, 83].

3.9 What's next in Scan Attacks?

Given the ongoing need for correctness in IC manufacturing alongside the pressing demands for new technologies, processes, and increased complexity, the presence of scan chains in integrated circuits is a relative certainty. Preventing abuse of the scan chain is thus paramount. Two broad categories of approaches are proposed [8]: (1) preventing unauthorized access to the scan chains, (2) obfuscating the scan chain I/O. These are discussed further here.

3.9.1 Preventing Unauthorized Access. Blocking access to the scan chain seems an intuitive method for protecting it, for instance using blocking logic / gates. However, blocks may be able to be overcome, especially if they exist only at the physical layer. Visual analysis (using reverse engineering techniques) will be able to identify the scan chain in the embedded IC die. Digital locking logic, meanwhile, may also be able to be bypassed: for instance, the technique R-DFS [33] which controls the scan chain via so-called secure cells is defeated via an attack termed 'shift-and-leak' [45]. Variants of this have all also shown weaknesses.

An alternative method for protecting unauthorized viewing of secret data, introduced in [83], uses an approach termed 'mirror key registers' (MKR). Here, the general approach is to prevent sensitive data (e.g. embedded keys) from entering the scan chain during the test mode (scanning). In the presented case study, this could be realized by using two sets of embedded keys in the DES hardware: one only for use during scan chain tests, and one for use during normal operation.

3.9.2 Obfuscating the Scan Chain I/O. A number of techniques have been proposed (and broken) for the protection of scan chain logic. An early attempt, termed a 'flipped scan' [62], statically inserts a number of inverters (NOT gates) between randomly selected scan chain flip-flops. However, this can be defeated by carefully analyzing the pattern of 1's and 0's after the reset condition [2]. Replacing the NOT gates with XOR gates and introducing feedback loops is another suggestion which has been broken [9, 10]. As such, adding multiplexers with each XOR gate, to enable dynamic control of the scan chain, is also proposed: here, keys (known by the tester) [7] or randomness governed by a physically unclonable function [9] or linear feedback shift registers [85] are required for the scan chain to enter the correct configuration. However, these may still be broken using approaches such as ScanSAT [6], which has also shown that it can break scan chain compression techniques. Keys may be also used to fully scramble the order of the scan chain [34], although this introduces a relatively large layout overhead in the IC. However, any system which just reorders the bits of the scan chain will be crackable eventually [8, 27].

3.9.3 Future work on protecting scan chains. Additional hardware used to protect scan chains may be a path forwards, for instance using a secondary encryption system to encrypt data before it is emitted from the hardware or during the scanning process. One such technique, using so-called 'Encrypt Flip Flops', has shown promise in this area [38, 39, 54], but may require the tester to have access to the encryption key, thus rendering the approach vulnerable to key losses. In addition, ScanSAT attacks have also shown they may break these approaches [5]. Future work will no doubt continue this cat-and-mouse game exploring new protections for scan chains, as well as determine the weaknesses of those protections.

4 CASE STUDY: DIGITAL HARDWARE INTELLECTUAL PROPERTY PROTECTION

4.1 Overview

In this section, we will work towards building an intuition about digital hardware intellectual property (IP) protection, specifically through logic locking,[3] a family of techniques that continues to evolve since first appearing in 2008 [59].

4.1.1 Setting. As one can intuit, hardware IP is extremely valuable. Creators of IP (or IP vendors) want to manage access of the IP within the context of licensing agreements and to avoid threats such as reverse-engineering, where malicious parties want to understand an IP to the level where they might be able to illicitly modify the design (say, to insert a Hardware Trojan) or piracy (where designs are stolen, cloned, and passed off as genuine or as a competing product). Ultimately, the designer (acting as a **defender** of the IP) wants to ensure that only legitimate users (or licensees) can use the IP as intended while other entities cannot.

4.1.2 Threat Models . In general terms, we can consider the following entities (and their employees) as being involved in the chip design flow: the **IP vendor**, the **system-on-chip integrator**, the **foundry**, the **test facility**, and the **end user**. Usually, we want to protect IP from potential adversaries: untrusted foundries, test facilities, or compromised/illegitimate end users, assuming that they have been compromised in some way (see survey papers like [12, 15] for more discussion about the threat model). Their aim is to do whatever they can to recover the IP; usually defined as gaining enough knowledge of the design so that they can get the IP to produce functionally correct outputs. We will see more precisely what this means in the context of logic locking in the next section. We usually **assume that the adversary can recover the IP's gate-level netlist**. Without any protection, an adversary can work out what different parts of the gate-level netlist do, work out what parts to modify if they desire, and re-use the IP.

Given this setting, there are two typical variations of the threat model. The **Oracle**-based model assumes that the adversary has in their possession an instance of the IP that produces **functionally correct outputs**—perhaps (legitimately) acquired on the open market. Within the **Oracle**-based approach, there are additional possible assumptions, including whether scan-chain access is available or if the Oracle's internal behavior can be observed (e.g., tamper-proof memory used or packaging-level protections to mitigate imaging). The **Oracle-less** model assumes that the adversary only has the gate-level netlist and is unable to gain information regarding correct functional input/output behavior.

Insight: As with any security problem, it is up to each practitioner to decide for themselves what assumptions are reasonable in terms of threat modeling. Take a moment to consider what adversary capabilities you think are realistic. In this tutorial, we will focus our attention on the basic locking of **combinational logic circuits** and **Oracle**-based analysis. From this foundation you should be well-equipped to engage with the recent advances in logic locking research.

4.2 Logic Locking: A "family" of approaches

Given the aforementioned threat model, **logic locking** seeks to transform a design in such a way that an adversary cannot simply copy or reverse-engineer a design[4]. In other words, we want to **change the design by inserting or replacing some of its logic** so that the IP is associated with some *secret* that can be shared between the IP vendor and trusted legitimate end-user that will "unlock" the IP's true functionality. The secret is typically in the form of some sort of "key" input.

[3]This is also seen in literature as logic obfuscation, logic encryption, redaction, and so on.

[4]Recent work by Beerel et al. [12] attempts to formalize the aims of logic locking, but we will take the intuitive understanding as sufficient for the purposes of an introduction to the topic in this tutorial paper.

(a) A simple circuit (b) A locked circuit

Fig. 6. An example of logic locking

Table 2. Truth Table for the circuit in Fig. 6. Notice the differences in the output 'o' when the key is set to 0 or 1.

Original				key = 0				key = 1			
a	b	c	o	a	b	c	o	a	b	c	o
0	0	0	0	0	0	0	0	**0**	**0**	**0**	**1**
0	0	1	1	0	0	1	1	0	0	1	1
0	1	0	0	0	1	0	0	**0**	**1**	**0**	**1**
0	1	1	1	0	1	1	1	0	1	1	1
1	0	0	0	1	0	0	0	**1**	**0**	**0**	**1**
1	0	1	1	1	0	1	1	1	0	1	1
1	1	0	1	1	1	0	1	**1**	**1**	**0**	**0**
1	1	1	1	1	1	1	1	1	1	1	1

For example, consider the circuit in Fig. 6 and its associated truth table in Table 2. Imagine that this is a valuable IP and we want to transform the design (i.e., "lock" it) such that the adversary cannot get the benefit of correct input/output behavior. The simplest modification we could do the circuit is to insert an XOR gate somewhere in the design and connecting it to a "key input" as in Fig. 6(b). Consider the truth table in Table 2 when key is set to 0 and when it is set to 1—notice that some of the rows are different to the original when $key = 1$. If the correct value of the key input is kept secret, an adversary that makes an incorrect guess of the key input will end up with a circuit that does not behave correctly for all input combinations. This is the basic premise of the first logic locking approach proposed by Roy et al. in 2008 [59]: randomly add XOR and XNOR gates to various points of the design attached to new key inputs. *As an exercise, you can create your own 3-input, 1-output circuit with 4-5 logic gates, construct its truth table, and then examine what happens when you "lock" the design with* XOR *and* XNOR *insertion.*

Insight: One way to think about the general premise of logic locking is to add "additional" or "surplus" functionality to the design, from which only a subset is actually useful. In our basic example, we added a new input and a gate, which actually expands the Boolean function from a 3-input to 4-input function. Added parts used in logic locking include new (key) inputs (and thus, new parts of the truth table) or new states and transitions (such as by adding memory elements like flip-flops). The useful (protected) part of the design should be hard to discover or use, unless, of course, you are authorized to by receiving the requisite secret material.

Since 2008, there has been several back-and-forth exchanges in the academic community with proposed attacks and countermeasures [15, 77]. Check out the references for a selection of related work. For this tutorial, we will keep ourselves occupied with random logic locking (RLL) (basic XOR/XNOR locking).

4.3 Early analyses

Early analyses of logic locking has had a close link with VLSI testing ideas, so to start to get a flavor of the type of analyses that is possible, let us consider the idea of the **sensitization attack** [56], where an adversary has access to an activated chip (i.e., a locked design which has had the secret material loaded to unlock the functionality).

Consider again the simple design of Fig. 6(b). We know the structure of the design, but we do not know the key input's intended value—the idea here is to sensitize the path so the key input value in the activated chip can be observed directly on the output, which we can do by choosing appropriate test input vectors.

For example, as we can see in Fig. 7, one can set the inputs "abc" to "000" so that the actual value of key can be seen on the output. By setting $c = 0$, we ensure the output of gate g3 reflects the output of gate g2. By setting either a or b to 0, we can guarantee that the output of gate g1 is 0, thus making the output of gate g2 the value of the key input. Typical test generation algorithms can help adversaries find the test vectors needed to sensitize the design [31, 56].

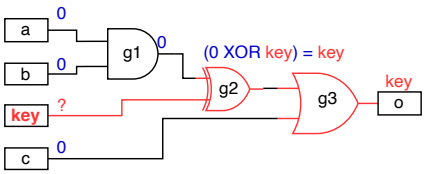


Fig. 7. Sensitization attack

As one might expect, a defender that knows that this attack is possible can try to respond by adding multiple key inputs and gates such that it is not possible to sensitize an individual key input [56], i.e., the choice of where to insert locking logic needs to be carefully considered. Recent work has shown that the sensitization and other attacks that use automatic test pattern generation (ATPG) have shown success on various logic locking flavors [30, 31]. This kind of adversarial back-and-forth or "cat-and-mouse" perspective has driven a lot of progress in logic locking [12, 15, 77].

Next, we will take a closer look at one of the most influential attacks in the logic locking literature: the "SAT attack" [74]. Understanding the SAT attack provides a good foundation for understanding the motivations for and intuitions around many proposed logic locking approaches, such as cyclic locking [86], stripped functionality logic locking [84], its descendants [63], and attacks (e.g., [64, 66–68]).

4.4 The SAT attack

In this section, we will do a step-by-step walk-through of the concepts and application of the attack by Subramanyan et al. [74].

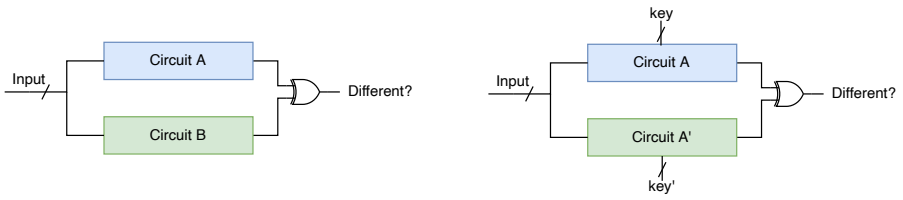
4.4.1 The building blocks. The SAT attack builds on a few key ideas, including the Boolean satisfiability problem, the construction of miter circuits, and the identification of distinguishing input patterns. The first idea is the **Boolean satisfiability problem**, often referred to as SAT. Given a Boolean formula, SAT is the problem of determining if variables in the formula can be set to TRUE and FALSE such that the formula as a whole evaluates to TRUE. If so, the formula is *satisfiable* (or SAT), otherwise, it is *unsatisfiable* (or UNSAT). While SAT is an NP-complete problem, several heuristic algorithms exist that can be used to solve SAT instances [32]. So, if we can produce a Boolean formula, we can give it to a SAT solver to try to find the variable assignments that make the

formula evaluate to TRUE, if the formula is satisifiable. For example, the formula $a \wedge b$[5] is satisfiable, as the formula evaluates to TRUE when $a = 1$ and $b = 1$.

The next thing we need to understand is the idea of a **miter circuit**. A miter circuit is a circuit comprising two circuits that are fed the same input, as shown in Fig. 8(a); we can check to see if the output of the circuits match for any given input. Intuitively, the two circuits are equivalent if they produce the same output as each for any input. The miter circuit is especially useful if we use a SAT solver because we can write a Boolean formula representing the miter circuit (including the output comparison) to feed into the SAT solver. The SAT solver then tries to answer the question: "Is there any input where the circuit outputs are different"? A SAT solver that returns SAT has found an input where the output of the two circuits disagrees.

We can use the miter circuit to perform formal equivalence checking of two combinational circuits: say we have two circuits that we *think* have the same behavior, we can connect them together as a miter (same inputs, XOR'd output) and use a SAT solver to see if the miter is satisfiable. The circuits are equivalent if the solver returns UNSAT; i.e., the solver cannot find any input that causes the two circuits to produce different outputs (i.e., XOR at the end will always be 0 (FALSE)).

This leads us to the third idea: the idea of a distinguishing input pattern. Let us now consider a miter circuit that is built with two copies of a logic locked design, as in Fig. 8(b). As before, the same input is fed into each circuit copy. When we feed this miter circuit into a SAT solver, we are essentially asking the same question as before: "*Is there any input where the circuit outputs are different?*" – except, in this case, we have separate key and key' inputs to each copy of the locked circuit. Given that the two circuits in the miter are exactly the same, the only way in which a SAT solver will be able to make the miter circuit formula satisfiable (if it is satisifiable) is to find an input and different values of key and key' that will make the different copies present different outputs. In other words, the solver will find a **distinguishing input pattern** (DIP) which means that the value (variable assignments) of key and key' that are found belong to two different classes of keys that make the locked circuit behave differently for that DIP.



(a) (b)

Fig. 8. (a) A general miter circuit and (b) the starting miter for the SAT attack.

4.4.2 The algorithm. We can now put together the aforementioned ideas into an an attack on logic locked designs when the adversary has access to an Oracle. Let us assume that we created the miter circuit using two copies of the locked design and that we have fed this to a SAT solver that returns SAT. The solver has given as a DIP and two possible outputs for that DIP. *Which output is correct?* With access to an Oracle, we can simply ask it which is the correct output for that DIP. Crucially, because we know which output is correct for that DIP, we can revise the miter circuit Boolean formula that we give to the SAT solver to include the new information – i.e., that for the DIP, the output must be so. The formula can be fed into the SAT solver again, which will attempt

[5]The notation $\wedge$ means logical conjuction (AND). $\vee$ means logical disjunction (OR).

to find another DIP. Each time we find a DIP, we can query the Oracle, modify the formula to take into account what the correct behavior should be for that DIP, and repeat; the solver prunes the key space iteratively. In each iteration, we effectively ask the solver to answer the question: *"Is there any input where the circuit outputs are different, given that when the inputs are {previous DIPs} the corresponding outputs must be {the correct outputs from the Oracle}"*? Eventually, the solver will return UNSAT when it can no longer find any DIPs; what is left will let us identify a key in the equivalence class of correct keys.

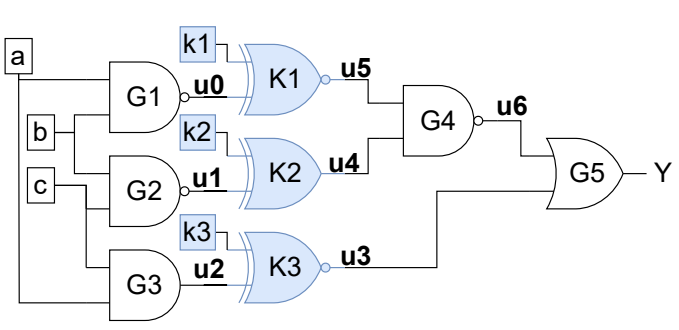
Taken all together, Algorithm 1 presents Subramanyan et al.'s "Logic Decryption Algorithm" [74], where $C(I, K, O)$ is the Boolean input-key-output relation (i.e., logic locked circuit represented as a Boolean formula), I is a vector of inputs, K is a vector of key inputs, O is a vector of outputs, and $oracle()$ is a function that represents querying the Oracle. Subscripts represent each iteration of the algorithm.

Algorithm 1: SAT Attack Algorithm from [74] (variable names changed)

Input : Locked circuit C, Oracle
Output: The values of K

i = 1 ;
$F_1 = C(I, K_1, O_1) \wedge C(I, K_2, O_2)$;
while $sat[F_i \wedge (O_1 \neq O_2)]$ **do**
  I_i^d = a DIP value that satisfy $[F_i \wedge (O_1 \neq O_2)]$;
  $O_i^d = oracle(I^d)$;
  $F_{i+1} = F_i \wedge C(I_i^d, O_i^d, K_1) \wedge C(I_i^d, O_i^d, K_2)$;
  $i = i + 1$;
end
K = the value of K in the sat assignment of $F_i \wedge (O_1 \equiv O_2)$;

4.4.3 Worked example. To see more clearly how the SAT attack algorithm process works, let us try a worked example on the locked design shown in Fig. 9.



oracle

a	b	c	y
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Fig. 9. Logic locked example design with its "Oracle" as a truth table

For this example, we will use a simple, Web browser-based SAT solver called logictools [75] with its corresponding syntax for representing propositional formulas[6]. We will use the `dpll:old` engine in this walk-through, as shown in Fig. 10. Typically, SAT solver tools ingest formulas in

[6] + is XOR, <=> is equivalence, & is AND, ~ or - is negation.

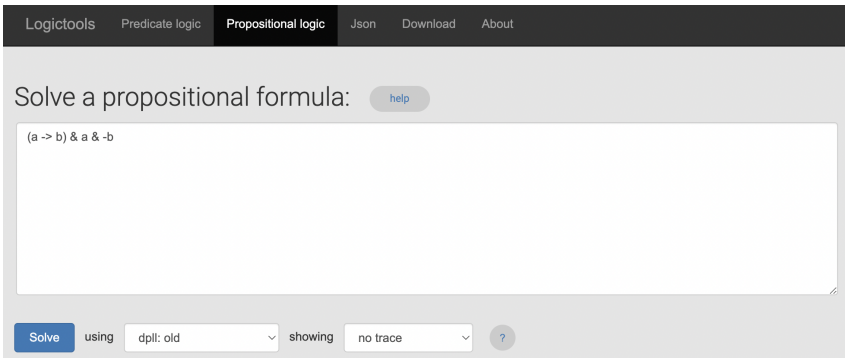


Fig. 10. The web browser-based logictools SAT solver from [75]

conjunctive normal form (CNF) which can be produced from propositional formulas using the Tseytin transformation [79]; logictools can ingest propositional formulas directly (and performs the transformation internally), so we will stick with propositional formulas for clarity in this example.

To begin, let us construct a miter circuit using two copies of the locked design. For readability, we will label the internal nets of the first copy as shown in Fig. 9 and represent the second copy's nets with "w". We distinguish between copy 1's keys and copy 2's keys by appending 'a' and 'b', respectively. The miter circuit is shown as a formula in Fig. 11.

circuit copy 1

```
((~(a&b) <=> u0) & (~(b&c) <=> u1) & ((c&a) <=> u2) & (~(k1a+u0) <=> u5)
& ((k2a + u1) <=> u4) & (~(k3a+u2) <=> u3) & (~(u5&u4) <=> u6) &
(y <=> (u6 | u3)))
&
```

circuit copy 2

```
((~(a&b) <=> w0) & (~(b&c) <=> w1) & ((c&a) <=> w2) & (~(k1b+w0) <=> w5)
& ((k2b + w1) <=> w4) & (~(k3b+w2) <=> w3) & (~(u5&w4) <=> w6) &
(yx <=> (w6 | w3)))
&
```

output compare

```
(y + yx)
```

Fig. 11. Miter circuit as a formula: SAT Attack Iteration #1

When we solve using `dpll:old`, we receive the following feedback:

```
Clause set is true if we assign values to variables as: u0 a -b u1 c
    u2 u5 k1a -u4 k2a u3 k3a u6 y w0 w1 w2 w5 k1b w4
-k2b -w3 -k3b -w6 -yx
```

In other words, the miter circuit formula is **satisfiable**! Of particular interest is the value of the inputs that the solver has found, being $a = 1$, $b = 0$ (see `-b` in the feedback), and $c = 1$. This is a distinguishing input pattern. In this case, circuit copy 1 produces an output of 1, while circuit copy 2 produces an output of 0. We can check the Oracle to see which is the intended output; from Fig. 9, the correct output should be 1. As we can see in line 6 of Algorithm 1, we should make a new formula by adding more copies of the circuit but with the constraint that the output produced with the DIP of $a = 1, b = 0, c = 1$ should be 1. The formula for the next iteration of the SAT attack thus looks like Fig. 12.

We add the miter circuit formula from iteration #1 to new copies of the circuit. Note the new internal node names "uuX" and "wwX" to represent the next copy, as well as "aa", "bb", "cc", and "yy" for the DIP and corresponding output from the oracle. Note also that the variables representing

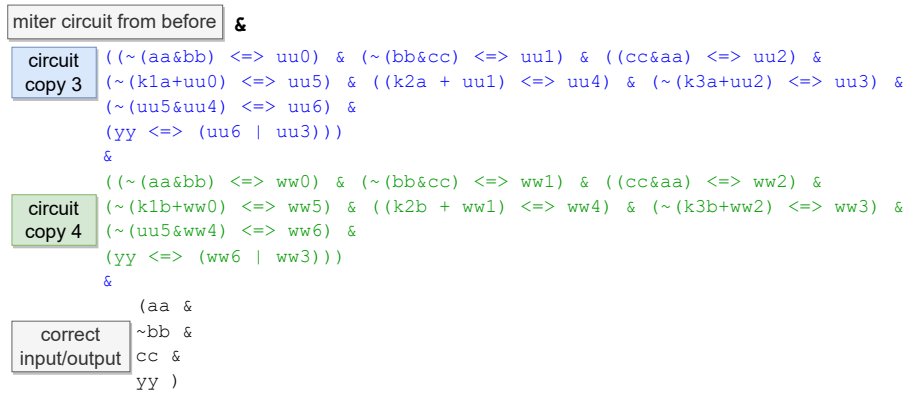


Fig. 12. SAT Attack Iteration #2

the key (e.g., k1a, k1b) are the same as those in the initial miter circuit formula; this forces keys found in future iterations to make the circuit produce the correct output for the DIP. Once again, using the SAT solver produces the following feedback:

```
Clause set is true if we assign values to variables as: u0 -a b -u1 c
    -u2 u5 k1a u4 k2a -u3 k3a -u6 -y w0 -w1 -w2 w5
k1b w4 k2b w3 -k3b -w6 yx uu0 aa -bb uu1 cc uu2 uu5 -uu4 uu3 uu6 yy
    ww0 ww1 ww2 ww5 -ww4 -ww3 ww6
```

This time, the DIP that is found is $a = 0, b = 1, c = 1$, which the Oracles tells us should produce the output 1. We can do yet another iteration of the SAT attack, adding Fig. 13 to our growing formula.

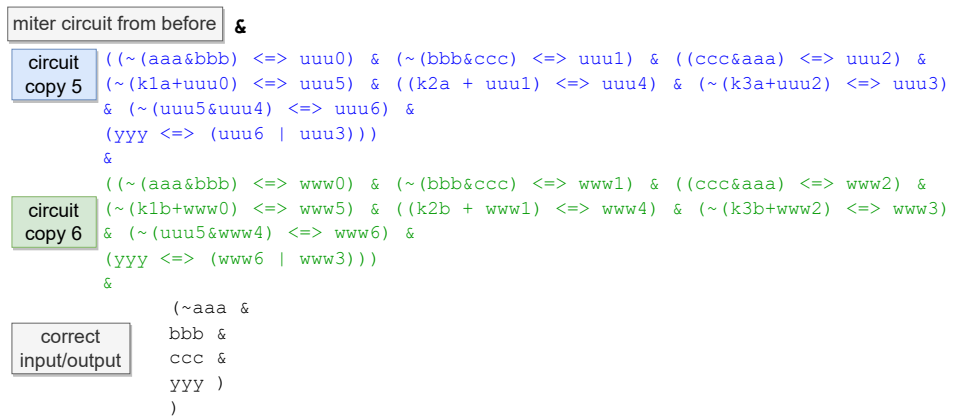


Fig. 13. SAT Attack Iteration #3

Running the SAT solver, we find that the formula is still satisifiable, with the following feedback:

```
Clause set is true if we assign values to variables as: u0 -a b u1 -c -u2
    u5 k1a -u4 k2a u3 -k3a u6 y w0 w1 -w2 w5 k1b w4 -k2b -w3 k3b -w6 -yx
    uu0 aa -bb uu1 cc uu2 uu5 -uu4 -uu3 uu6 yy ww0 ww1 ww2 ww5 ww4 ww3 -
    ww6 uuu0 -aaa bbb -uuu1 ccc -uuu2 uuu5 uuu4 uuu3 -uuu6 yyy www0 -www1
    -www2 www5 -www4 -www3 www6
```

If we construct the new formula for another iteration and run that through the solver (see the Appendix for the full formula), we finally receive the following feedback:

```
Clause set is false for all possible assignments to variables.
```

The formula is **unsatisfiable**! This means that the solver can no longer find any DIPs; in other words, all the remaining inputs and keys will make the two copies of the circuit behave equivalently. Because the formula also adds constraints that any remaining keys make the circuit produce the correct input/output behavior (from the previous DIPs), any key that satisfies the formula as a whole, without the `y + yx` constraint, should be a correct key. Thus, running the formula with that constraint missing, gives us:

```
Clause set is true if we assign values to variables as: -u0 a b
-u1 c u2 -u5 k1a -u4 -k2a u3 k3a u6 y -w0 -w1 w2 -w5 k1b -w4 -k2b w3 k3b
    w6 yx uu0 aa -bb uu1 cc uu2 uu5 uu4 uu3 -uu6 yy
ww0 ww1 ww2 ww5 ww4 ww3 -ww6 uuu0 -aaa bbb -uuu1 ccc -uuu2 uuu5
-uuu4 -uuu3 uuu6 yyy www0 -www1 -www2 www5 -www4 -www3 www6 uuuu0 -aaaa
    bbbb uuuu1 -cccc -uuuu2 uuuu5 uuuu4 -uuuu3 -uuuu6 -yyyy wwww0 wwww1 -
    wwww2 wwww5 wwww4 -wwww3 -wwww6
```

If we hone in on the variables representing the key bits, we can see that $k1a = k1b = 1, k2a = k2b = 0, k3a = k3b = 1$.

4.5 Discussion: What's next for logic locking?

The arrival of the "SAT attack" marked a turning point in the logic locking domain. Recent survey works such as that by Chakraborty et al. [15] provide a good overview of the various techniques which you should now be able to better appreciate. Several post-SAT techniques try to reduce the number of keys that are pruned with each iteration of the SAT attack (e.g., [84]), while others try to introduce circuit structures that cause issues for SAT solvers (e.g., [65]). New defense techniques are proposed and countered, even now, as the "cat-and-mouse" game continues. Other recent work includes the proposal of FPGA-based redaction [49] or universal circuits [12], among other strategies.

While this tutorial covered the first formulation of the SAT attack from Subramanyan et al. [74], there are a few more things we should note. As we mentioned in subsubsection 4.1.2, this tutorial focused on a combinational design, so you are probably interested to know what happens in more realistic systems, where we have memory elements and sequential logic. In the early days of logic locking, the notion of Oracle access is often paired with the idea of a fully-scanned design with an adversary-accessible scan chain (as you explored in the previous case study in Section 3). With design-for-test structures like the scan-chain, an adversary can treat the different parts of the design as combinational by scanning in input sequences and scanning out the register contents (assuming, of course, that the logic locking key is itself not trivially connected to the scan chain!). Thus, advances in scan chain protection, such as the recently proposed DisOrc [44], provide an orthogonal means to protect against the SAT attack and other Oracle-based attacks on logic locking. Other approaches, like the KC2 attack [66], incorporate the idea of *sequential unrolling*, where the adversary makes several copies of the circuit in the Boolean formula to represent the inputs and outputs of the design across several clock cycles. However, as one can see, even with our simple tutorial walk-through, the Boolean formula grows very quickly!

Logic locking continues to evolve, with new defenses and attacks being proposed regularly by academia. Readers that are interested to explore more about logic locking can consider reading some of the following works. Setting aside Oracle-based attacks, there are several Oracle-less attacks as

well. These include analyzing the circuit in terms of logic redundancy with incorrect keys [43] as well as efforts focused on structural analysis and machine learning [3, 16, 71]. We direct interested readers to a useful survey paper on machine learning and logic locking by Sisejkovic et al. [72]. Sequential obfuscation (e.g., [17, 55]), where the state space of a design is modified, is another emerging area.

Recently, reconfigurable fabrics, such as embedded field-programmable gate arrays (eFPGAs), (e.g., [1, 13, 21, 49, 69]) have been proposed to withhold elements of the design. Working out what to "redact" or lock remains challenging, with several recent approaches emerging (e.g., [20, 78]). For access to tools and benchmarks, we encourage the readers to check out https://trust-hub.org/ and https://cadforassurance.org/ – these websites collate the results of several research efforts.

Fundamentally, there is an ongoing need to formalize notions of security, trade-off security against the cost of each solution, and devise new means of attack and defense.

5 CONCLUSIONS

In this paper, we provided a tutorial introduction to issues in hardware security. In particular, we presented two detailed case studies of problems in hardware security: attacking cryptography via the scan chain side channel and attacks on logic locking for hardware intellectual property protection. Through these pedagogical examples, we provide a foundation which readers can use to engage with more recent research in these domains. The tutorial examples are supported by an open access online resource hosted at https://github.com/learn-hardware-security.

ACKNOWLEDGMENTS

This research was supported in part by NSF Grant #2039607. Any opinions, findings, and conclusions, or recommendations expressed are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

[1] Zain Ul Abideen, Tiago Diadami Perez, and Samuel Pagliarini. 2021. From FPGAs to Obfuscated eASICs: Design and Security Trade-offs. In *2021 Asian Hardware Oriented Security and Trust Symposium (AsianHOST)*. 1–4. https://doi.org/10.1109/AsianHOST53231.2021.9699758

[2] Mukesh Agrawal, Sandip Karmakar, Dhiman Saha, and Debdeep Mukhopadhyay. 2008. Scan Based Side Channel Attacks on Stream Ciphers and Their Counter-Measures. In *Progress in Cryptology - INDOCRYPT 2008 (Lecture Notes in Computer Science)*, Dipanwita Roy Chowdhury, Vincent Rijmen, and Abhijit Das (Eds.). Springer, Berlin, Heidelberg, 226–238. https://doi.org/10.1007/978-3-540-89754-5_18

[3] Abdulrahman Alaql, Md Moshiur Rahman, and Swarup Bhunia. 2021. SCOPE: Synthesis-Based Constant Propagation Attack on Logic Locking. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 29, 8 (Aug. 2021), 1529–1542. https://doi.org/10.1109/TVLSI.2021.3089555 Conference Name: IEEE Transactions on Very Large Scale Integration (VLSI) Systems.

[4] Sk Subidh Ali, Ozgur Sinanoglu, and Ramesh Karri. 2014. Test-mode-only scan attack using the boundary scan chain. In *2014 19th IEEE European Test Symposium (ETS)*. 1–6. https://doi.org/10.1109/ETS.2014.6847798 ISSN: 1558-1780.

[5] Lilas Alrahis, Muhammad Yasin, Nimisha Limaye, Hani Saleh, Baker Mohammad, Mahmoud Al-Qutayri, and Ozgur Sinanoglu. 2021. ScanSAT: Unlocking Static and Dynamic Scan Obfuscation. *IEEE Transactions on Emerging Topics in Computing* 9, 4 (Oct. 2021), 1867–1882. https://doi.org/10.1109/TETC.2019.2940750 Conference Name: IEEE Transactions on Emerging Topics in Computing.

[6] Lilas Alrahis, Muhammad Yasin, Hani Saleh, Baker Mohammad, Mahmoud Al-Qutayri, and Ozgur Sinanoglu. 2019. ScanSAT: unlocking obfuscated scan chains. In *Proceedings of the 24th Asia and South Pacific Design Automation Conference (ASPDAC '19)*. Association for Computing Machinery, New York, NY, USA, 352–357. https://doi.org/10.1145/3287624.3287693

[7] Yuta Atobe, Youhua Shi, Masao Yanagisawa, and Nozomu Togawa. 2012. Dynamically changeable secure scan architecture against scan-based side channel attack. In *2012 International SoC Design Conference (ISOCC)*. 155–158. https://doi.org/10.1109/ISOCC.2012.6407063

[8] Kimia Zamiri Azar, Hadi Mardani Kamali, Houman Homayoun, and Avesta Sasan. 2021. From Cryptography to Logic Locking: A Survey on the Architecture Evolution of Secure Scan Chains. *IEEE Access* 9 (2021), 73133–73151. https://doi.org/10.1109/ACCESS.2021.3080257 Conference Name: IEEE Access.

[9] Subhadeep Banik, Anupam Chattopadhyay, and Anusha Chowdhury. 2014. Cryptanalysis of the Double-Feedback XOR-Chain Scheme Proposed in Indocrypt 2013. In *Progress in Cryptology – INDOCRYPT 2014 (Lecture Notes in Computer Science)*, Willi Meier and Debdeep Mukhopadhyay (Eds.). Springer International Publishing, Cham, 179–196. https://doi.org/10.1007/978-3-319-13039-2_11

[10] Subhadeep Banik and Anusha Chowdhury. 2013. Improved Scan-Chain Based Attacks and Related Countermeasures. In *Progress in Cryptology – INDOCRYPT 2013 (Lecture Notes in Computer Science)*, Goutam Paul and Serge Vaudenay (Eds.). Springer International Publishing, Cham, 78–97. https://doi.org/10.1007/978-3-319-03515-4_6

[11] Kanad Basu, Samah Mohamed Saeed, Christian Pilato, Mohammed Ashraf, Mohammed Thari Nabeel, Krishnendu Chakrabarty, and Ramesh Karri. 2019. CAD-Base: An Attack Vector into the Electronics Supply Chain. *ACM Transactions on Design Automation of Electronic Systems* 24, 4 (April 2019), 38:1–38:30. https://doi.org/10.1145/3315574

[12] Peter Beerel, Marios Georgiou, Ben Hamlin, Alex J. Malozemoff, and Pierluigi Nuzzo. 2022. Towards a Formal Treatment of Logic Locking. *IACR Transactions on Cryptographic Hardware and Embedded Systems* (Feb. 2022), 92–114. https://doi.org/10.46586/tches.v2022.i2.92-114

[13] Jitendra Bhandari, Abdul Khader Thalakkattu Moosa, Benjamin Tan, Christian Pilato, Ganesh Gore, Xifan Tang, Scott Temple, Pierre-Emmanuel Gaillardon, and Ramesh Karri. 2021. Exploring eFPGA-based Redaction for IP Protection. In *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. 1–9. https://doi.org/10.1109/ICCAD51958.2021.9643548 ISSN: 1558-2434.

[14] Ferdinand Brasser, Lucas Davi, Abhijitt Dhavlle, Tommaso Frassetto, Sai Manoj Pudukotai Dinakarrao, Setareh Rafatirad, Ahmad-Reza Sadeghi, Avesta Sasan, Hossein Sayadi, Shaza Zeitouni, and Houman Homayoun. 2018. Special Session: Advances and Throwbacks in Hardware-Assisted Security. In *2018 International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*. IEEE, Turin, 1–10. https://doi.org/10.1109/CASES.2018.8516874

[15] Abhishek Chakraborty, Nithyashankari Gummidipoondi Jayasankaran, Yuntao Liu, Jeyavijayan Rajendran, Ozgur Sinanoglu, Ankur Srivastava, Yang Xie, Muhammad Yasin, and Michael Zuzak. 2020. Keynote: A Disquisition on Logic Locking. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 10 (Oct. 2020), 1952–1972. https://doi.org/10.1109/TCAD.2019.2944586 Conference Name: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems.

[16] Prabuddha Chakraborty, Jonathan Cruz, Abdulrahman Alaql, and Swarup Bhunia. 2021. SAIL: Analyzing Structural Artifacts of Logic Locking Using Machine Learning. *IEEE Transactions on Information Forensics and Security* 16 (2021), 3828–3842. https://doi.org/10.1109/TIFS.2021.3096028 Conference Name: IEEE Transactions on Information Forensics and Security.

[17] Rajat Subhra Chakraborty and Swarup Bhunia. 2009. HARPOON: An Obfuscation-Based SoC Design Methodology for Hardware Protection. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 28, 10 (Oct. 2009), 1493–1502. https://doi.org/10.1109/TCAD.2009.2028166

[18] Douglas Chang, Mike Tien-Chien Lee, Kwang-Ting Cheng, and Malgorzata Marek-Sadowska. 1998. Functional scan chain testing. In *Proceedings Design, Automation and Test in Europe*. IEEE Comput. Soc, Paris, France, 278–283. https://doi.org/10.1109/DATE.1998.655868

[19] Mike Chapple. 2018. Confidentiality, Integrity and Availability - The CIA Triad. https://www.certmike.com/confidentiality-integrity-and-availability-the-cia-triad/

[20] Jianqi Chen, Monir Zaman, Yiorgos Makris, R. D. Shawn Blanton, Subhasish Mitra, and Benjamin Carrion Schafer. 2020. DECOY: DEflection-Driven HLS-Based Computation Partitioning for Obfuscating Intellectual PropertY. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*. 1–6. https://doi.org/10.1109/DAC18072.2020.9218519 ISSN: 0738-100X.

[21] Prattay Chowdhury, Chaitali Sathe, and Benjamin Carrion Schaefer. 2022. Predictive Model Attack for Embedded FPGA Logic Locking. In *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design (ISLPED '22)*. Association for Computing Machinery, New York, NY, USA, 1–6. https://doi.org/10.1145/3531437.3539728

[22] European Commission. 2022. State of the Union: New EU cybersecurity rules. https://ec.europa.eu/commission/presscorner/detail/en/ip_22_5374

[23] Computing Community Consortium. 2013. *Research Needs for Trustworthy, and Reliable Semiconductors*. Technical Report. https://www.src.org/calendar/e004965/sa-ts-workshop-report-final.pdf

[24] Luigi Coppolino, Salvatore D'Antonio, Giovanni Mazzeo, and Luigi Romano. 2019. A comprehensive survey of hardware-assisted security: From the edge to the cloud. *Internet of Things* 6 (June 2019), 100055. https://doi.org/10.1016/j.iot.2019.100055

[25] Semiconductor Research Corporation. 2022. Semiconductor Research Corporation - SRC. https://www.src.org/program/grc/hws/

[26] The MITRE Corporation. 2022. CWE - CWE-1194: Hardware Design (4.1). https://cwe.mitre.org/data/definitions/1194.html

[27] Aijiao Cui, Yanhui Luo, Huawei Li, and Gang Qu. 2017. Why current secure scan designs fail and how to fix them? *Integration* 56 (Jan. 2017), 105–114. https://doi.org/10.1016/j.vlsi.2016.10.011

[28] DARPA. 2020. DARPA Selects Teams to Increase Security of Semiconductor Supply Chain. https://www.darpa.mil/news-events/2020-05-27

[29] Deloitte. 2022. *2022 semiconductor industry outlook*. Technical Report. https://www2.deloitte.com/us/en/pages/technology-media-and-telecommunications/articles/semiconductor-industry-outlook.html

[30] Danielle Duvalsaint, Xiaoxiao Jin, Benjamin Niewenhuis, and R. D. Blanton. 2019. Characterization of Locked Combinational Circuits via ATPG. In *2019 IEEE International Test Conference (ITC)*. 1–10. https://doi.org/10.1109/ITC44170.2019.9000130 ISSN: 2378-2250.

[31] Danielle Duvalsaint, Zeye Liu, Ananya Ravikumar, and Ronald D. Blanton. 2019. Characterization of Locked Sequential Circuits via ATPG. In *2019 IEEE International Test Conference in Asia (ITC-Asia)*. 97–102. https://doi.org/10.1109/ITC-Asia.2019.00030

[32] Jun Gu, Paul W. Purdom, John Franco, and Benjamin W. Wah. 1996. *Algorithms for the Satisfiability (SAT) Problem: A Survey,*. Technical Report. CINCINNATI UNIV OH DEPT OF ELECTRICAL AND COMPUTER ENGINEERING. https://apps.dtic.mil/sti/citations/ADA326042 Section: Technical Reports.

[33] Ujjwal Guin, Ziqi Zhou, and Adit Singh. 2018. Robust Design-for-Security Architecture for Enabling Trust in IC Manufacturing and Test. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 26, 5 (May 2018), 818–830. https://doi.org/10.1109/TVLSI.2018.2797019 Conference Name: IEEE Transactions on Very Large Scale Integration (VLSI) Systems.

[34] David Hely, Marie-Lise Flottes, Frédéric Bancel, Bruno Rouzeyre, Nicolas Berard, and Michel Renovell. 2004. Scan design and secure chip [secure IC testing]. In *IOLTS: International On-Line Testing Symposium*. IEEE, Madeira Island, Portugal, 219–224. https://doi.org/10.1109/OLT.2004.1319691

[35] Yu Huang, Ruifeng Guo, Wu-Tung Cheng, and James Chien-Mo Li. 2008. Survey of Scan Chain Diagnosis. *IEEE Design & Test of Computers* 25, 3 (May 2008), 240–248. https://doi.org/10.1109/MDT.2008.83

[36] IEEE. 2013. IEEE Standard for Test Access Port and Boundary-Scan Architecture. *IEEE Std 1149.1-2013 (Revision of IEEE Std 1149.1-2001)* (May 2013), 1–444. https://doi.org/10.1109/IEEESTD.2013.6515989

[37] Yier Jin. 2019. Towards Hardware-Assisted Security for IoT Systems. In *2019 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. 632–637. https://doi.org/10.1109/ISVLSI.2019.00118 ISSN: 2159-3477.

[38] Rajit Karmakar, Santanu Chatopadhyay, and Rohit Kapur. 2018. Encrypt Flip-Flop: A Novel Logic Encryption Technique For Sequential Circuits. https://doi.org/10.48550/arXiv.1801.04961 arXiv:1801.04961 [cs].

[39] Rajit Karmakar, Santanu Chattopadhyay, and Rohit Kapur. 2020. A Scan Obfuscation Guided Design-for-Security Approach for Sequential Circuits. *IEEE Transactions on Circuits and Systems II: Express Briefs* 67, 3 (March 2020), 546–550. https://doi.org/10.1109/TCSII.2019.2915606 Conference Name: IEEE Transactions on Circuits and Systems II: Express Briefs.

[40] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. 2014. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. 361–372. https://doi.org/10.1109/ISCA.2014.6853210 ISSN: 1063-6897.

[41] Paul Kocher, Jann Horn, Anders Fogh, and Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*.

[42] Kari Kostiainen, Aritra Dhar, and Srdjan Capkun. 2020. Dedicated Security Chips in the Age of Secure Enclaves. *IEEE Security Privacy* 18, 5 (Sept. 2020), 38–46. https://doi.org/10.1109/MSEC.2020.2990230 Conference Name: IEEE Security Privacy.

[43] Leon Li and Alex Orailoglu. 2019. Piercing Logic Locking Keys through Redundancy Identification. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 540–545. https://doi.org/10.23919/DATE.2019.8714955 ISSN: 1558-1101.

[44] Nimisha Limaye, Emmanouil Kalligeros, Nikolaos Karousos, Irene G. Karybali, and Ozgur Sinanoglu. 2021. Thwarting All Logic Locking Attacks: Dishonest Oracle With Truly Random Logic Locking. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 40, 9 (Sept. 2021), 1740–1753. https://doi.org/10.1109/TCAD.2020.3029133 Conference Name: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems.

[45] Nimisha Limaye, Abhrajit Sengupta, Mohammed Nabeel, and Ozgur Sinanoglu. 2019. Is Robust Design-for-Security Robust Enough? Attack on Locked Circuits with Restricted Scan Chain Access. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 1–8. https://doi.org/10.1109/ICCAD45719.2019.8942047 ISSN: 1558-2434.

[46] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *27th USENIX Security Symposium (USENIX Security 18)*.

[47] Konstantinos Markantonakis, Michael Tunstall, Gerhard Hancke, Ioannis Askoxylakis, and Keith Mayes. 2009. Attacking smart card systems: Theory and practice. *Information Security Technical Report* 14, 2 (May 2009), 46–56. https://doi.org/10.1016/j.istr.2009.06.001

[48] The MITRE Corporation (MITRE). 2022. CWE - CWE Most Important Hardware Weaknesses. https://cwe.mitre.org/scoring/lists/2021_CWE_MIHW.html

[49] Prashanth Mohan, Oguz Atli, Joseph Sweeney, Onur Kibar, Larry Pileggi, and Ken Mai. 2021. Hardware Redaction via Designer-Directed Fine-Grained eFPGA Insertion. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, Virtual, 6.

[50] Onur Mutlu and Jeremie S. Kim. 2020. RowHammer: A Retrospective. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 8 (Aug. 2020), 1555–1571. https://doi.org/10.1109/TCAD.2019.2915318

[51] NIST. 1999. *Data Encryption Standard (DES)*. Technical Report Federal Information Processing Standard (FIPS) PUB 46-3. U.S. Department of Commerce National Instutute of Standards and Technology. https://csrc.nist.gov/csrc/media/publications/fips/46/3/archive/1999-10-25/documents/fips46-3.pdf

[52] NIST. 2001. *Advanced Encryption Standard (AES)*. Technical Report Federal Information Processing Standard (FIPS) 197. U.S. Department of Commerce National Instutute of Standards and Technology. https://doi.org/10.6028/NIST.FIPS.197

[53] Johannes Obermaier, Marc Schink, and Kosma Moczek. 2020. One Exploit to Rule them All? On the Security of Drop-in Replacement and Counterfeit Microcontrollers. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association. https://www.usenix.org/conference/woot20/presentation/obermaier

[54] Anu Paul, Mohankumar N., and Nirmala Devi M. 2022. Obviating Multiple Attacks with Enhanced Logic Locking. In *Proceedings of the 2022 Fourteenth International Conference on Contemporary Computing (IC3-2022)*. Association for Computing Machinery, New York, NY, USA, 162–167. https://doi.org/10.1145/3549206.3549235

[55] Md Moshiur Rahman and Swarup Bhunia. 2022. Practical Implementation of Robust State Space Obfuscation for Hardware IP Protection. https://doi.org/10.36227/techrxiv.21405075.v1

[56] Jeyavijayan Rajendran, Youngok Pino, Ozgur Sinanoglu, and Ramesh Karri. 2012. Security analysis of logic obfuscation. In *DAC Design Automation Conference 2012*. 83–89. https://doi.org/10.1145/2228360.2228377 ISSN: 0738-100X.

[57] Srivaths Ravi, Paul Kocher, Ruby Lee, Gary McGraw, and Anand Raghunathan. 2004. Security as a new dimension in embedded system design. In *Proceedings of the 41st annual conference on Design automation - DAC '04*. ACM Press, San Diego, CA, USA, 753. https://doi.org/10.1145/996566.996771

[58] Masoud Rostami, Farinaz Koushanfar, and Ramesh Karri. 2014. A Primer on Hardware Security: Models, Methods, and Metrics. *Proc. IEEE* 102, 8 (Aug. 2014), 1283–1295. https://doi.org/10.1109/JPROC.2014.2335155

[59] Jarrod A. Roy, Farinaz Koushanfar, and Igor L. Markov. 2008. EPIC: Ending Piracy of Integrated Circuits. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '08)*. ACM, New York, NY, USA, 1069–1074. https://doi.org/10.1145/1403375.1403631

[60] Spyridon Samonas and David Coss. 2014. The CIA strikes back: Redefining confidentiality, integrity and availability in security. *Journal of Information System Security* 10, 3 (July 2014).

[61] David E. Sanger and Nicole Perlroth. 2021. Pipeline Attack Yields Urgent Lessons About U.S. Cybersecurity. *The New York Times* (May 2021). https://www.nytimes.com/2021/05/14/us/politics/pipeline-hack.html

[62] Gaurav Sengar, Debdeep Mukhopadhyay, and Dipanwita Roy Chowdhury. 2007. Secured Flipped Scan-Chain Model for Crypto-Architecture. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 26, 11 (Nov. 2007), 2080–2084. https://doi.org/10.1109/TCAD.2007.906483 Conference Name: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems.

[63] Abhrajit Sengupta, Mohammed Nabeel, Nimisha Limaye, Mohammed Ashraf, and Ozgur Sinanoglu. 2020. Truly Stripping Functionality for Logic Locking: A Fault-based Perspective. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2020), 1–1. https://doi.org/10.1109/TCAD.2020.2968898

[64] Kaveh Shamsi, Meng Li, Travis Meade, Zheng Zhao, David Z. Pan, and Yier Jin. 2017. AppSAT: Approximately deobfuscating integrated circuits. In *2017 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. 95–100. https://doi.org/10.1109/HST.2017.7951805 ISSN: null.

[65] Kaveh Shamsi, Meng Li, Travis Meade, Zheng Zhao, David Z. Pan, and Yier Jin. 2017. Cyclic Obfuscation for Creating SAT-Unresolvable Circuits. In *Proceedings of the on Great Lakes Symposium on VLSI 2017 (GLSVLSI '17)*. Association for Computing Machinery, Banff, Alberta, Canada, 173–178. https://doi.org/10.1145/3060403.3060458

[66] Kaveh Shamsi, Meng Li, David Z. Pan, and Yier Jin. 2019. KC2: Key-Condition Crunching for Fast Sequential Circuit Deobfuscation. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 534–539. https://doi.org/10.23919/DATE.2019.8715053 ISSN: 1558-1101.

[67] Kaveh Shamsi, David Z. Pan, and Yier Jin. 2019. IcySAT: Improved SAT-based Attacks on Cyclic Locked Circuits. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 1–7. https://doi.org/10.1109/ICCAD45719.2019.8942049 ISSN: 1558-2434.

[68] Yuanqi Shen, You Li, Amin Rezaei, Shuyu Kong, David Dlott, and Hai Zhou. 2019. BeSAT: behavioral SAT-based attack on cyclic logic encryption. In *Proceedings of the 24th Asia and South Pacific Design Automation Conference*. ACM, Tokyo Japan, 657–662. https://doi.org/10.1145/3287624.3287670

[69] Mustafa M. Shihab, Jingxiang Tian, Gaurav Rajavendra Reddy, Bo Hu, William Swartz, Benjamin Carrion Schaefer, Carl Sechen, and Yiorgos Makris. 2019. Design Obfuscation through Selective Post-Fabrication Transistor-Level Programming. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 528–533. https://doi.org/10.23919/DATE.2019.8714856 ISSN: 1558-1101.

[70] Adam Shostack. 2014. *Threat modeling: Designing for security.* John Wiley & Sons.

[71] Dominik Sisejkovic, Farhad Merchant, Lennart M. Reimann, Harshit Srivastava, Ahmed Hallawa, and Rainer Leupers. 2021. Challenging the Security of Logic Locking Schemes in the Era of Deep Learning: A Neuroevolutionary Approach. *ACM Journal on Emerging Technologies in Computing Systems* 17, 3 (May 2021), 30:1–30:26. https://doi.org/10.1145/3431389

[72] Dominik Sisejkovic, Lennart M. Reimann, Elmira Moussavi, Farhad Merchant, and Rainer Leupers. 2021. Logic Locking at the Frontiers of Machine Learning: A Survey on Developments and Opportunities. In *2021 IFIP/IEEE 29th International Conference on Very Large Scale Integration (VLSI-SoC)*. 1–6. https://doi.org/10.1109/VLSI-SoC53125.2021.9606979 ISSN: 2324-8440.

[73] Daehyun Strobel, David Oswald, Bastian Richter, Falk Schellenberg, and Christof Paar. 2014. Microcontrollers as (In)Security Devices for Pervasive Computing Applications. *Proc. IEEE* 102, 8 (Aug. 2014), 1157–1173. https://doi.org/10.1109/JPROC.2014.2325397 Conference Name: Proceedings of the IEEE.

[74] Pramod Subramanyan, Sayak Ray, and Sharad Malik. 2015. Evaluating the security of logic encryption algorithms. In *2015 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. 137–143. https://doi.org/10.1109/HST.2015.7140252

[75] Tanel Tammet. 2022. Logictools. http://logictools.org

[76] Benjamin Tan. 2022. Challenges and Opportunities for Hardware-Assisted Security Improvements in the Field. In *2022 23rd International Symposium on Quality Electronic Design (ISQED)*. 90–95. https://doi.org/10.1109/ISQED54688.2022.9806254 ISSN: 1948-3295.

[77] Benjamin Tan, Ramesh Karri, Nimisha Limaye, Abhrajit Sengupta, Ozgur Sinanoglu, Md Moshiur Rahman, Swarup Bhunia, Danielle Duvalsaint, R. D. Shawn Blanton, Amin Rezaei, Yuanqi Shen, Hai Zhou, Leon Li, Alex Orailoglu, Zhaokun Han, Austin Benedetti, Luciano Brignone, Muhammad Yasin, Jeyavijayan Rajendran, Michael Zuzak, Ankur Srivastava, Ujjwal Guin, Chandan Karfa, Kanad Basu, Vivek V. Menon, Matthew French, Peilin Song, Franco Stellari, Gi-Joon Nam, Peter Gadfort, Alric Althoff, Joseph Tostenrude, Saverio Fazzari, Eric Breckenfeld, and Kenneth Plaks. 2020. Benchmarking at the Frontier of Hardware Security: Lessons from Logic Locking. http://arxiv.org/abs/2006.06806

[78] Chiara Muscari Tomajoli, Luca Collini, Jitendra Bhandari, Abdul Khader Thalakkattu Moosa, Benjamin Tan, Xifan Tang, Pierre-Emmanuel Gaillardon, Ramesh Karri, and Christian Pilato. 2022. ALICE: an automatic design flow for eFPGA redaction. In *Proceedings of the 59th ACM/IEEE Design Automation Conference (DAC '22)*. Association for Computing Machinery, New York, NY, USA, 781–786. https://doi.org/10.1145/3489517.3530543

[79] Grigori Samuilovitsch Tseitin. 1983. On the Complexity of Derivation in Propositional Calculus. In *Automation of Reasoning*, Jörg H. Siekmann and Graham Wrightson (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 466–483. https://doi.org/10.1007/978-3-642-81955-1_28

[80] Paul Van De Zande. 2001. *The day DES died.* Technical Report 22. SANS Institute.

[81] Ken Xiao, Domenic Forte, Yier Jin, Ramesh Karri, Swarup Bhunia, and Mark Tehranipoor. 2016. Hardware Trojans: Lessons Learned after One Decade of Research. *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 22, 1 (May 2016), 6:1–6:23. https://doi.org/10.1145/2906147

[82] Bo Yang, Kaijie Wu, and Ramesh Karri. 2004. Scan based side channel attack on dedicated hardware implementations of Data Encryption Standard. In *2004 International Conference on Test.* 339–344. https://doi.org/10.1109/TEST.2004.1386969

[83] Bo Yang, Kaijie Wu, and Ramesh Karri. 2006. Secure Scan: A Design-for-Test Architecture for Crypto Chips. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 25, 10 (Oct. 2006), 2287–2293. https://doi.org/10.1109/TCAD.2005.862745

[84] Muhammad Yasin, Abhrajit Sengupta, Mohammed Thari Nabeel, Mohammed Ashraf, Jeyavijayan (JV) Rajendran, and Ozgur Sinanoglu. 2017. Provably-Secure Logic Locking: From Theory To Practice. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*. Association for Computing Machinery, Dallas, Texas, USA, 1601–1618. https://doi.org/10.1145/3133956.3133985

[85] Dongrong Zhang, Miao He, Xiaoxiao Wang, and Mark Tehranipoor. 2017. Dynamically obfuscated scan for protecting IPs against scan-based attacks throughout supply chain. In *2017 IEEE 35th VLSI Test Symposium (VTS)*. 1–6. https:

//doi.org/10.1109/VTS.2017.7928947 ISSN: 2375-1053.

[86] Hai Zhou, Ruifeng Jiang, and Shuyu Kong. 2017. CycSAT: SAT-based attack on cyclic logic encryptions. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 49–56. https://doi.org/10.1109/ICCAD.2017.8203759 ISSN: 1558-2434.

[87] Yuankun Zhu, Yueqiang Cheng, Husheng Zhou, and Yantao Lu. 2021. Hermes Attack: Steal DNN Models with Lossless Inference Accuracy. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 1973–1988. https://www.usenix.org/conference/usenixsecurity21/presentation/zhu

A DES TABLES

This appendix provides the various permutation tables used within DES [51]. The general process for reading a permutation table is as follows. The output bits are generated in-order using the input, with the input indexed using the appropriate value from the table. For example, the first (left-most / MSB) bit of the output of *IP* will be the value of the 58th bit of the input (refer to Table 3). Then, the second bit will be the 50th bit of the input, and so on. The IP, FP, E, P, PC1, and PC2 tables are presented as tables only for ease of presentation. They are vectors, not tables.

A.1 Encryption / Decryption permutation tables

Table 3. Initial Permutation (IP)

58	50	42	34	26	18	10	2
60	52	44	36	28	20	12	4
62	54	46	38	30	22	14	6
64	56	48	40	32	24	16	8
57	49	41	33	25	17	9	1
59	51	43	35	27	19	11	3
61	53	45	37	29	21	13	5
63	55	47	39	31	23	15	7

Table 4. Final Permutation (FP) (is equal to the inverse of the IP table, IP^{-1}).

40	8	48	16	56	24	64	32
39	7	47	15	55	23	63	31
38	6	46	14	54	22	62	30
37	5	45	13	53	21	61	29
36	4	44	12	52	20	60	28
35	3	43	11	51	19	59	27
34	2	42	10	50	18	58	26
33	1	41	9	49	17	57	25

Table 5. Expansion Function (E). Note the repeating bits (e.g. 32, 1).

32	1	2	3	4	5
4	5	6	7	8	9
8	9	10	11	12	13
12	13	14	15	16	17
16	17	18	19	20	21
20	21	22	23	24	25
24	25	26	27	28	29
28	29	30	31	32	1

Table 6. Permutation (P).

16	7	20	21	29	12	28	17
1	15	23	26	5	18	31	10
2	8	24	14	32	27	3	9
19	13	30	6	22	11	4	25

Table 7. DES s-boxes

S1	**x0000x**	**x0001x**	**x0010x**	**x0011x**	**x0100x**	**x0101x**	**x0110x**	**x0111x**	**x1000x**	**x1001x**	**x1010x**	**x1011x**	**x1100x**	**x1101x**	**x1110x**	**x1111x**
0yyyy0	14	4	13	1	2	15	11	8	3	10	6	12	5	9	0	7
0yyyy1	0	15	7	4	14	2	13	1	10	6	12	11	9	5	3	8
1yyyy0	4	1	14	8	13	6	2	11	15	12	9	7	3	10	5	0
1yyyy1	15	12	8	2	4	9	1	7	5	11	3	14	10	0	6	13

S2	**x0000x**	**x0001x**	**x0010x**	**x0011x**	**x0100x**	**x0101x**	**x0110x**	**x0111x**	**x1000x**	**x1001x**	**x1010x**	**x1011x**	**x1100x**	**x1101x**	**x1110x**	**x1111x**
0yyyy0	15	1	8	14	6	11	3	4	9	7	2	13	12	0	5	10
0yyyy1	3	13	4	7	15	2	8	14	12	0	1	10	6	9	11	5
1yyyy0	0	14	7	11	10	4	13	1	5	8	12	6	9	3	2	15
1yyyy1	13	8	10	1	3	15	4	2	11	6	7	12	0	5	14	9

S3	**x0000x**	**x0001x**	**x0010x**	**x0011x**	**x0100x**	**x0101x**	**x0110x**	**x0111x**	**x1000x**	**x1001x**	**x1010x**	**x1011x**	**x1100x**	**x1101x**	**x1110x**	**x1111x**
0yyyy0	10	0	9	14	6	3	15	5	1	13	12	7	11	4	2	8
0yyyy1	13	7	0	9	3	4	6	10	2	8	5	14	12	11	15	1
1yyyy0	13	6	4	9	8	15	3	0	11	1	2	12	5	10	14	7
1yyyy1	1	10	13	0	6	9	8	7	4	15	14	3	11	5	2	12

S4	**x0000x**	**x0001x**	**x0010x**	**x0011x**	**x0100x**	**x0101x**	**x0110x**	**x0111x**	**x1000x**	**x1001x**	**x1010x**	**x1011x**	**x1100x**	**x1101x**	**x1110x**	**x1111x**
0yyyy0	7	13	14	3	0	6	9	10	1	2	8	5	11	12	4	15
0yyyy1	13	8	11	5	6	15	0	3	4	7	2	12	1	10	14	9
1yyyy0	10	6	9	0	12	11	7	13	15	1	3	14	5	2	8	4
1yyyy1	3	15	0	6	10	1	13	8	9	4	5	11	12	7	2	14

S5	**x0000x**	**x0001x**	**x0010x**	**x0011x**	**x0100x**	**x0101x**	**x0110x**	**x0111x**	**x1000x**	**x1001x**	**x1010x**	**x1011x**	**x1100x**	**x1101x**	**x1110x**	**x1111x**
0yyyy0	2	12	4	1	7	10	11	6	8	5	3	15	13	0	14	9
0yyyy1	14	11	2	12	4	7	13	1	5	0	15	10	3	9	8	6
1yyyy0	4	2	1	11	10	13	7	8	15	9	12	5	6	3	0	14
1yyyy1	11	8	12	7	1	14	2	13	6	15	0	9	10	4	5	3

S6	**x0000x**	**x0001x**	**x0010x**	**x0011x**	**x0100x**	**x0101x**	**x0110x**	**x0111x**	**x1000x**	**x1001x**	**x1010x**	**x1011x**	**x1100x**	**x1101x**	**x1110x**	**x1111x**
0yyyy0	12	1	10	15	9	2	6	8	0	13	3	4	14	7	5	11
0yyyy1	10	15	4	2	7	12	9	5	6	1	13	14	0	11	3	8
1yyyy0	9	14	15	5	2	8	12	3	7	0	4	10	1	13	11	6
1yyyy1	4	3	2	12	9	5	15	10	11	14	1	7	6	0	8	13

S7	**x0000x**	**x0001x**	**x0010x**	**x0011x**	**x0100x**	**x0101x**	**x0110x**	**x0111x**	**x1000x**	**x1001x**	**x1010x**	**x1011x**	**x1100x**	**x1101x**	**x1110x**	**x1111x**
0yyyy0	4	11	2	14	15	0	8	13	3	12	9	7	5	10	6	1
0yyyy1	13	0	11	7	4	9	1	10	14	3	5	12	2	15	8	6
1yyyy0	1	4	11	13	12	3	7	14	10	15	6	8	0	5	9	2
1yyyy1	6	11	13	8	1	4	10	7	9	5	0	15	14	2	3	12

S8	**x0000x**	**x0001x**	**x0010x**	**x0011x**	**x0100x**	**x0101x**	**x0110x**	**x0111x**	**x1000x**	**x1001x**	**x1010x**	**x1011x**	**x1100x**	**x1101x**	**x1110x**	**x1111x**
0yyyy0	13	2	8	4	6	15	11	1	10	9	3	14	5	0	12	7
0yyyy1	1	15	13	8	10	3	7	4	12	5	6	11	0	14	9	2
1yyyy0	7	11	4	1	9	12	14	2	0	6	10	13	15	3	5	8
1yyyy1	2	1	14	7	4	10	8	13	15	12	9	0	3	5	6	11

A.2 Key generation permutation tables

Table 8. Key Permutation (*PC*1). Note that only 56 of the 64 bits are used (the remaining are parity bits).

Left							Right						
57	49	41	33	25	17	9	63	55	47	39	31	23	15
1	58	50	42	34	26	18	7	62	54	46	38	30	22
10	2	59	51	43	35	27	14	6	61	53	45	37	29
19	11	3	60	52	44	36	21	13	5	28	20	12	4

Table 9. Key Permutation (*PC*2). Note that the input is the two 28-bit key halves concatenated.

14	17	11	24	1	5
3	28	15	6	21	10
23	19	12	4	26	8
16	7	27	20	13	2
41	52	31	37	47	55
30	40	51	45	33	48
44	49	39	56	34	53
46	42	50	36	29	32

Table 10. Key bit rotations.

Round Number	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Number of Left Rotations	1	1	2	2	2	2	2	2	1	2	2	2	2	2	2	1

B FORMULA FOR THE SAT ATTACK WORKED EXAMPLE

The formula below should return UNSAT when used with a SAT solver. Removing the `(y + yx)` part should make the formula SAT and reveal the key.

```
((~(a&b) <=> u0) & (~(b&c) <=> u1) & ((c&a) <=> u2) & (~(k1a+u0) <=> u5) & ((k2a + u1) <=> u4) & (~(k3a+u2)
      <=> u3) & (~(u5&u4) <=> u6) &
(y <=> (u6 | u3)))
&
((~(a&b) <=> w0) & (~(b&c) <=> w1) & ((c&a) <=> w2) & (~(k1b+w0) <=> w5) & ((k2b + w1) <=> w4) & (~(k3b+w2)
      <=> w3) & (~(u5&w4) <=> w6) &
(yx <=> (w6 | w3)))
&
(y + yx) &

(
((~(aa&bb) <=> uu0) & (~(bb&cc) <=> uu1) & ((cc&aa) <=> uu2) & (~(k1a+uu0) <=> uu5) & ((k2a + uu1) <=> uu4) &
       (~(k3a+uu2) <=> uu3) & (~(uu5&uu4) <=> uu6) &
(yy <=> (uu6 | uu3)))
&
((~(aa&bb) <=> ww0) & (~(bb&cc) <=> ww1) & ((cc&aa) <=> ww2) & (~(k1b+ww0) <=> ww5) & ((k2b + ww1) <=> ww4) &
       (~(k3b+ww2) <=> ww3) & (~(uu5&ww4) <=> ww6) &
(yy <=> (ww6 | ww3)))
&
(aa &
~bb &
cc &
yy )
```

```
)

&

(
((~(aaa&bbb) <=> uuu0) & (~(bbb&ccc) <=> uuu1) & ((ccc&aaa) <=> uuu2) & (~(k1a+uuu0) <=> uuu5) & ((k2a + uuu1
      ) <=> uuu4) & (~(k3a+uuu2) <=> uuu3) & (~(uuu5&uuu4) <=> uuu6) &
(yyy <=> (uuu6 | uuu3)))
&
((~(aaa&bbb) <=> www0) & (~(bbb&ccc) <=> www1) & ((ccc&aaa) <=> www2) & (~(k1b+www0) <=> www5) & ((k2b + www1
      ) <=> www4) & (~(k3b+www2) <=> www3) & (~(uuu5&www4) <=> www6) &
(yyy <=> (www6 | www3)))
&
(~aaa &
bbb &
ccc &
yyy )
)

&

(
((~(aaaa&bbbb) <=> uuuu0) & (~(bbbb&cccc) <=> uuuu1) & ((cccc&aaaa) <=> uuuu2) & (~(k1a+uuuu0) <=> uuuu5) &
      ((k2a + uuuu1) <=> uuuu4) & (~(k3a+uuuu2) <=> uuuu3) & (~(uuuu5&uuuu4) <=> uuuu6) &
(yyyy <=> (uuuu6 | uuuu3)))
&
((~(aaaa&bbbb) <=> wwww0) & (~(bbbb&cccc) <=> wwww1) & ((cccc&aaaa) <=> wwww2) & (~(k1b+wwww0) <=> wwww5) &
      ((k2b + wwww1) <=> wwww4) & (~(k3b+wwww2) <=> wwww3) & (~(uuuu5&wwww4) <=> wwww6) &
(yyyy <=> (wwww6 | wwww3)))
&
(~aaaa &
bbbb &
~cccc &
~yyyy )
)
```