A Modular Testing Environment for Implementation Attacks

Lyndon Judge, Michael Cantrell, Cagil Kendir, and Patrick Schaumont

Center for Embedded Systems for Critical Applications (CESCA)
Bradley Department of Electrical and Computer Engineering
Virginia Tech, Blacksburg, VA 24061, USA
Email: {lvjudge1, mcantrell, ckendir, schaum}@vt.edu

***Abstract*—Implementation attacks, including side-channel, fault, and probing attacks, have received significant attention in both research and commercial communities. Successful attacks have been demonstrated against standard cryptographic algorithms implemented on a wide variety of common platforms. In order to protect against these attacks, designers must incorporate complex countermeasures into the implementation of sensitive operations. Validating the effectiveness of implementation attack countermeasures requires specialized expertise and techniques not commonly used in other types of security and functional testing. We propose a modular testing environment for use in verifying the implementation attack resistance of secure systems. The proposed environment is an open-source solution that allows implementation attack testing to be independent of the system platform, implementation details, and type of attack under evaluation. These key features make the environment suitable for use with an implementation attack security standard in which standard test procedures are published openly and used to evaluate cryptographic systems. We use the proposed test environment to demonstrate a successful side-channel attack on AES, which illustrates the practical usefulness of our design for analyzing implementation attack security. Our open-source design is available at http://rijndael.ece.vt.edu/iameter.**

***Index Terms*—Side-channel Analysis, Differential Power Analysis (DPA), Timing Analysis, Differential Fault Analysis (DFA), Security Testing**

I. Introduction

Implementation attacks are one of the most serious security threats facing cryptographic systems. Unlike classical cryptanalytic attacks, which target the underlying algorithm or protocol, these attacks exploit weaknesses in the implementation of a cryptographic module that reveal information about the internal state of the module. Invasiveness of implementation attacks vary from noninvasive, which requires no physical manipulation of the target device, to invasive, which requires permanent modification of electronic circuits within the device. Attack cost and complexity is heavily influenced by the invasiveness of the attack method.

Implementation attacks are classified based on the way information is gathered from the target device. The most common types of attacks are side-channel, fault, and probing. Side-channel attacks are passive attacks in which the attacker observes the device under normal operation and analyzes characteristics such as execution time, power consumption, and electromagnetic radiation to determine a secret value. Fault attacks are active attacks that manipulate the device to cause errors in its output that can be used to derive a secret value. Probing attacks use probes placed within a chip to spy on its internal values and derive, or in some cases directly read, a secret value. Due to their low cost and noninvasiveness, side-channel attacks are the most common implementation attacks, although noninvasive fault attacks are growing in popularity.

Mitigating the risk of implementation attacks requires use of countermeasures to ensure that sensitive information cannot be observed by an attacker, operational faults cannot be induced, and tampering cannot be done without destroying the device. The specific countermeasures required to secure a certain device depend on a variety of factors including the level of access available to an attacker and the properties of the implementation platform. Implementation attack vulnerability results from design decisions made when mapping a cryptographic algorithm or protocol from its specification to the desired device platform and countermeasures typically require significant changes to the system architecture. This suggests that implementation attack risk mitigation requires a secure by design approach that includes these risks in system requirements and ensures that countermeasures are included in the initial design phase. All countermeasures must be properly tested to ensure mitigation of implementation attack risks without introducing additional risks.

We propose a modular testing environment that can be used to evaluate the vulnerability of secure devices to implementation attacks. We focus our design on providing separation between the device being tested and the test script, portability, and openness. This results in a test environment that can evaluate resistance against implementation attacks regardless of the device platform or the type of attack performed. The proposed test environment is suitable for use in standardized security analysis and allows fair comparison of the implementation attack resistance of different designs.

The remainder of this paper is structured as follows. In the next section, we define the security problem of validating implementation attack resistance, review the related work, and present our solution. In Section III, we introduce our proposed test environment and describe key features of its design. We demonstrate the capabilities of the proposed test environment to perform side-channel analysis in Section IV. We discuss

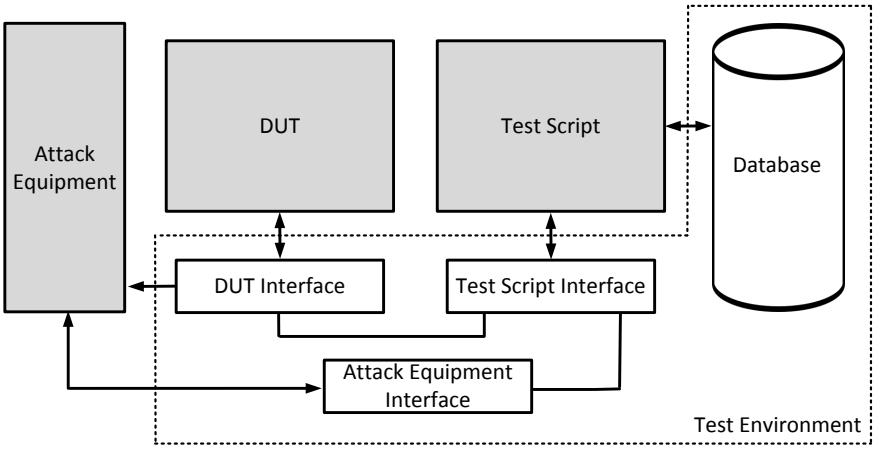

Fig. 1. Modular Test Environment for Implementation Attacks: Block Diagram

application of the proposed test environment to additional types of implementation attacks in Section V and conclude the paper in Section VI.

II. Problem Statement

Any cryptographic device that can be physically possessed or observed by an attacker is potentially vulnerable to implementation attacks, regardless of the security of the underlying algorithm or protocol employed by the device. Embedded systems are particularly vulnerable to implementation attacks because they often operate in such conditions, but servers and desktop computers can also be at risk. Despite the use of cryptography to protect sensitive data, devices are vulnerable to implementation attacks and successful attacks have been demonstrated for standard security algorithms including AES and RSA. Furthermore, results in [5] show that an unknown cryptographic algorithm is not a significant barrier to implementation attacks. This creates a significant security risk for these devices. In addition, the sophistication of implementation attack techniques continues to increase rapidly, meaning that systems thought to be secure may soon become vulnerable.

Due to the growing threat of implementation attacks, secure devices must implement countermeasures to protect sensitive data. Effective countermeasures require precise modifications to system design and can be difficult to implement properly. In addition, there is a large knowledge gap between system designers, who are not typically knowledgeable about implementation attacks and countermeasures, and attackers, who possess significant expertise. This can lead designers to underestimate a system's vulnerability to implementation attacks and put sensitive data at risk.

Implementation attack vulnerability testing is fundamentally different from other forms of security testing. A system's vulnerability to implementation attacks is heavily influenced by system design and architecture, as well as the characteristics of the device platform. The exact impact of these factors is difficult to predict outside of an attack scenario, which makes it impossible to evaluate implementation attack risk using common white-box testing techniques such as static source code analysis and data-flow analysis. Instead, implementation attack resistance testing requires a dynamic approach that tests the complete system implementation together with the platform.

At present, there is no standard technique or open source platform for evaluation of vulnerability to implementation attacks. This creates a significant disadvantage for both designers and users in terms of validating correct application of countermeasures and establishing a minimum acceptable level of implementation attack resistance. In response to this concern, the United States National Institute of Standards and Technology (NIST) has decided to establish standards for implementation attack resistance in its forthcoming FIPS 140-3 computer security standard [12]. The FIPS 140-3 standard is currently undergoing a public comment period in which NIST has specifically requested comments on the appropriate role of implementation attack resistance in security level certification [10]. NIST has also sought input on development of standard techniques and tools for noninvasive implementation attack resistance validation through the Non-Invasive Attack Testing Workshop held in 2011. A standardized test environment and procedure is necessary to provide the quality and repeatability of implementation attack resistance evaluation required by the FIPS 140 standard.

Implementation attack techniques and countermeasures are an active area of academic research. However, differences between device platforms, measurement equipment, and test setup make it difficult to compare results. Within the research community, there is great interest in performing implementation attacks in a way that allows fair comparison against one another. Comparison is useful for evaluating the relative strength of different attack methods, as well as determining the relative security level of different devices and implementation attack countermeasures. The DPA Contest [16] provides a framework for comparison of differential power analysis side-

channel attacks. The most popular iteration of the DPA contest provides a fixed number of power traces collected from an FPGA implementation of AES and requires entrants to submit attack scripts that can derive the secret key of the device. Since all attacks target the same implementation and rely on the same measurements, fair comparison is straightforward. However, the applicability of the DPA contest is limited due to its reliance on a trusted third party to provide the side-channel measurements needed to mount an attack.

A. Related Work

There are commercial systems available that can be used to assess vulnerability to implementation attacks. The DPA Workstation [6] developed by Cryptography Research Inc is a specialty tool that allows to perform side-channel analysis, specifically differential power or electromagnetic analysis, on embedded systems. The DPA Workstation includes an environment for collection of data from the device under attack and proprietary software that can perform side-channel analysis on all major standard ciphers. A similar system is Inspector [14] from Riscure, which supports both side-channel analysis and fault analysis. The Inspector platform includes custom measurement and fault injection hardware along with proprietary software to perform complete side-channel and fault attacks on standard ciphers. Brightsight also offers an implementation attack toolset that includes platforms for side-channel and fault analysis [4].

Side Channel Analysis Resistant Framework (SCARF) [9] is an academic tool developed by the Electronics and Telecommunications Research Institute and intended for use in research on countermeasures for side-channel and fault attacks. SCARF includes a number of custom evaluation boards that can be used to test attack resistance in devices such as smartcards, microprocessors, and FPGAs. However, SCARF only supports testing of the specific device models included in its custom evaluation boards. The Flexible Open-source Board for Side-channel analysis (FOBOS) [17] developed at George Mason University is an academic platform under development for implementation attack resistance testing. FOBOS aims to provide an open-source platform that can be used to evaluate effectiveness of side-channel analysis countermeasures on a variety of different FPGA platforms.

B. Proposed Solution

We propose a modular testing environment to relieve the current problems in implementation attack vulnerability analysis and standardization. A key feature of our design is the separation between the design under test (DUT) and the test script used to perform vulnerability analysis. This approach allows a clear separation between the tasks of system design and security testing, each of which requires separate expertise. We have designed the test environment to ensure that test scripts can be written without specific knowledge of the implementation details of the DUT or the platform on which the DUT is implemented. Due to the lack of dependency between the DUT and the test script, the proposed test environment allows a single test script to be used to evaluate multiple DUTs and provides a fair security comparison of each DUT.

A block diagram of the proposed test environment is shown in Figure 1. The unshaded blocks are implemented as part of the test environment itself, while the shaded blocks are user-designed modules. As shown, the user designs the test script and design under test. The attack equipment is also provided by the user and is used to observe or manipulate device behavior as required to perform the desired implementation attack. The test environment defines an interface for each user-designed module, which permits interaction within the test environment, while allowing each module to remain independent. The interface modules are designed to maximize portability and can be used to support virtually any DUT platform or attack equipment.

Advantages of the proposed test environment include portability and openness. Implementation attacks are complex and require specialized knowledge. Our test environment allows test scripts to be designed independently of the DUT, which accommodates integration of implementation attack expertise into test scripts. This approach makes it possible for implementation attack experts with no knowledge of the DUT to create highly effective test scripts. The portability inherent in our design enhances this feature by ensuring that test scripts, as well as the test environment itself, are independent of the platform on which the DUT is implemented. While hardware and software DUTs may have completely different implementation characteristics, both can be tested in the proposed environment using the same test script. In addition, the test environment is independent of the attack equipment used, as well as the type of implementation attack tested.

An additional feature of the proposed test environment is its openness; our design is fully open-source. The open source design has a number of practical advantages including extensibility and low-cost, however, the primary benefits are more far-reaching. The proposed test environment provides a uniform open platform that can be used in both industry and academic research and can facilitate cooperation and collaboration. For the past decade, the trend in security and cryptography has been to embrace openness, rather than obscurity, in the design and analysis of algorithms and protocols. The open selection process used by NIST to decide the AES [13] and SHA-3 [11] standard algorithms exemplifies this trend. The proposed test environment can be used to increase openness by allowing establishment of open standards for implementation attack resistance and security validation. For example, a standard test script to determine the implementation attack security of an algorithm or protocol could be established and openly distributed.

III. Modular Testing Environment

Our test environment is implemented using a modular approach. Each module of the proposed test environment has been carefully designed to achieve the overall goals of separation between DUT and test script, portability, and openness. In addition to these modules, the test environment

includes a user-defined DUT and test script. The user-defined components are differentiated from test environment modules in Figure 1 using gray shading. In this section, we explain the design of the test environment modules and their interaction with user-defined components. Additional details on the design and functionality of the proposed test environment can be found at http://rijndael.ece.vt.edu/iameter.

A. Usage Model

The proposed modular test environment is intended to be used for evaluation of the implementation attack resistance of a design. Due to the broad scope of implementation attacks, we propose that implementation attack resistance be integrated into the system design process as follows. The initial system requirements document should define both functional and security requirements for the design. Implementation attack security goals should be included through specification of relevant types of attacks and required resistance levels. The system designer then constructs the DUT to incorporate countermeasures intended to reduce the risk of the specified implementation attacks to acceptable levels. Meanwhile, the test script designer, ideally an implementation attack expert or security standards body, independently creates a test script based on the requirements document to determine whether the DUT satisfies the defined implementation attack security requirements.

The test procedure used in the test script varies depending on the cryptographic algorithm or protocol implemented by the DUT, as well as the type of implementation attack performed. However, we present a general test sequence as a guideline of a test procedure that is appropriate in most cases. The test script initialization routine should establish a connection with the DUT and attack equipment using their respective interfaces. If necessary, the initialization routine should also configure the attack equipment with the desired settings. The test script then enters a data collection loop where it sends a message to the DUT to start operation, observes or manipulates DUT operation using the attack equipment, and creates a trace based on the DUT input, output, and the observations or manipulations performed by attack equipment. The number of loop iterations depends on the required security level of the DUT, where a large number of iterations can be used to verify a high security level. Analysis of the traces to determine implementation attack vulnerability can either be performed in real-time during data collection or in post-processing after all traces have been collected. The primary advantage of real-time analysis is that a test script can terminate early when the DUT reaches a failure condition before trace collection is complete.

B. DUT Interface

We define the DUT interface as a module that interacts directly with the DUT and manages data transfer between the test script and the DUT. The primary goal of the DUT interface is to create an abstraction layer over the DUT itself that allows communication with the test script, regardless of the data transfer protocol used by the DUT. To achieve this, the DUT interface defines a message passing protocol that must be supported by both the DUT and the test script. To support the message passing protocol, the DUT must specify its number of input/output ports as well as the size of each port. The message passing protocol defines a message as containing two fields: address and data. The size of the data field is equal to the size of each DUT port and the address field gives the specific port associated with the message data. The DUT interface API is shown in Table I. The API allows the DUT interface to be platform-independent by ensuring that all aspects of data transfer between the DUT and test script below the abstraction level of the message passing protocol are confined to the implementation of the DUT interface itself. In addition, the functions defined in the API are general enough that they can apply to to any DUT and implementation attack scenario.

TABLE I
DUT INTERFACE API

Function	Description
init(addr_len, data_len)	Establish a connection between the DUT and the test script. Transfers size of each message field, as defined in the DUT, to the test script to ensure synchronization
close()	Close the connection between DUT and test script.
read(addr, data)	Reads a message from the DUT and returns it to the test script. The address returned is the address of the DUT port from which the data was received. Returns immediately with value of 0 if a complete message is not available from DUT.
write(addr, data)	Writes a message to the DUT. The parameters addr and data correspond to the address and data message fields respectively.

The DUT interface can also be used to implement synchronization between the DUT and attack equipment. This is supported by an additional triggering output port in the DUT interface that connects directly to the attack equipment. Assertion of the triggering output can be used to notify the attack equipment of the status of sensitive operations being performed by the DUT. In many cases, the trigger from the DUT asserts a start signal in the attack equipment to begin data collection or manipulation.

C. Attack Equipment

Implementation attacks require specialized devices to observe or manipulate behavior of the DUT. We refer to these devices as the attack equipment and require that the test environment include at least one such device. The specific attack equipment required varies based on the type of implementation attack under evaluation. The test environment defines an attack equipment interface in order to allow the test script to interact with the attack equipment. The API of the attack equipment interface is shown in Table II. As shown, the attack equipment interface allows the test script to access data measured from the DUT, send a configuration string to fix the equipment settings, and send or receive equipment-specific commands.

TABLE II
ATTACK EQUIPMENT INTERFACE API

Function	Description
init(ip_addr, dev_id, dev_link)	Establish a connection between the test script and the attack equipment over the given IP address. Returns the name of the attack equipment as *dev_id* and a unique number specifying the link as *dev_link*.
close(ip_address, dev_link)	Closes communication with the attack equipment by terminating the connection over the specified IP address.
set_parameters(dev_link, device_id, config_str)	Configure the attack equipment with the parameters given in *config_str*.
send_receive (dev_link, cmd)	Send or receive a single command from the attack equipment.
capture(dev_link, dev_id, buf)	Retrieve the latest data captured from the attack equipment.
save_capture(filename, buf, buf_size)	Store data captured by the attack equipment in a file.
save_settings(dev_link, dev_id, filename, buf)	Retrieves current settings from attack equipment and saves in a file.
recall_settings(dev_link, dev_id, filename, buf)	Configure the attack equipment with the parameters stored in a file.

Connection between the test script and the attack equipment is established over IP, which has the benefit of compatibility with a wide variety of instrumentation and support for remote interaction.

D. Database

A unique feature of our proposed test environment is the inclusion of a database to store data collected from the attack equipment. We define a trace as a set of data collected by the attack equipment during a single operation performed by the DUT. Storage of traces in the database allows easy management and retrieval of data collected over multiple iterations of testing. In addition, use of the database allows physical separation between trace collection and analysis. Once a set of traces has been collected and stored in the database, test scripts can access them via the database API and perform attacks or analysis remotely.

Each time an implementation attack is performed, hundreds or thousands of measurement traces of DUT behavior must be collected using the attack equipment. Implementation attacks are defined not only by the traces collected from the DUT, but also by the configuration settings of the test environment. Each trace represents an execution of some operation by the DUT and is characterized by the input and output data of the DUT during collection by the trace. The database is organized into two tables: the traces table, which stores measurement traces collected by attack equipment, and the experiments table, which stores data regarding test environment settings during trace collection. The columns of each database table are shown in Table III and Table IV, respectively. The information contained in these tables can be used to analyze a set of attack traces or reproduce a set of traces by using an identical DUT, platform, and test environment configuration. All columns of each database table can be easily accessed within a test script using the provided database API. For ease of the use, the database API provides an additional function that allows storage of multiple traces from one experiment with a single function call.

TABLE III
DATABASE TRACES TABLE

Column	Description
TRACEID	32-bit number uniquely identifying the trace.
ASSOCIATED	1024 character STRING. The string represents associated data for the trace, such as the input/ output values of the cipher DUT (plaintext, ciphertext, key).
CAMPAIGNID	Reference (32-bit) number to the CAMPAIGN record that describes the overall setup used to collect the trace.
DATA	20kb raw binary array. This represents the trace data, in binary format.

TABLE IV
DATABASE EXPERIMENT TABLE

Column	Description
CAMPAIGNID	32-bit number uniquely identifying the experiment.
EQUIPSETTINGS	64 kb STRING. This field contains the complete attack equipment settings associated with the setup in this experiment.
DUTREPOURL	1024 character STRING. This represents a URL to a repository that contains the full DUT definition (source code, bit streams, binaries).
DUTREPOREV	32-bit integer. This represents the repository version of the DUT for this campaign.
CAMPAIGNCOMMENT	User-defined 1024 character string. This field contains a brief description of the experiment, the attack equipment used, the DUT platform used, the DUT, etc. Note that the DUT repository can provide additional detailed documentation, if needed.
SAMPLEBYTES	8-bit number that tells how many bytes per sample the capture contains for the campaign.

E. Script Interface

Test scripts execute in a Python scripting environment installed on a control PC and interact with the DUT, attack equipment, and database. We define the test script interface as the control PC software required to allow communication from the Python scripting environment to the DUT and attack equipment via their respective interfaces. The control PC software is accessed directly from the scripting environment via the test script and operates according to the specified APIs for the DUT interface, attack equipment interface, and database.

IV. CORRELATION POWER ANALYSIS EXAMPLE

In this section, we present an example of an application of the proposed test environment to assess the vulnerability of an

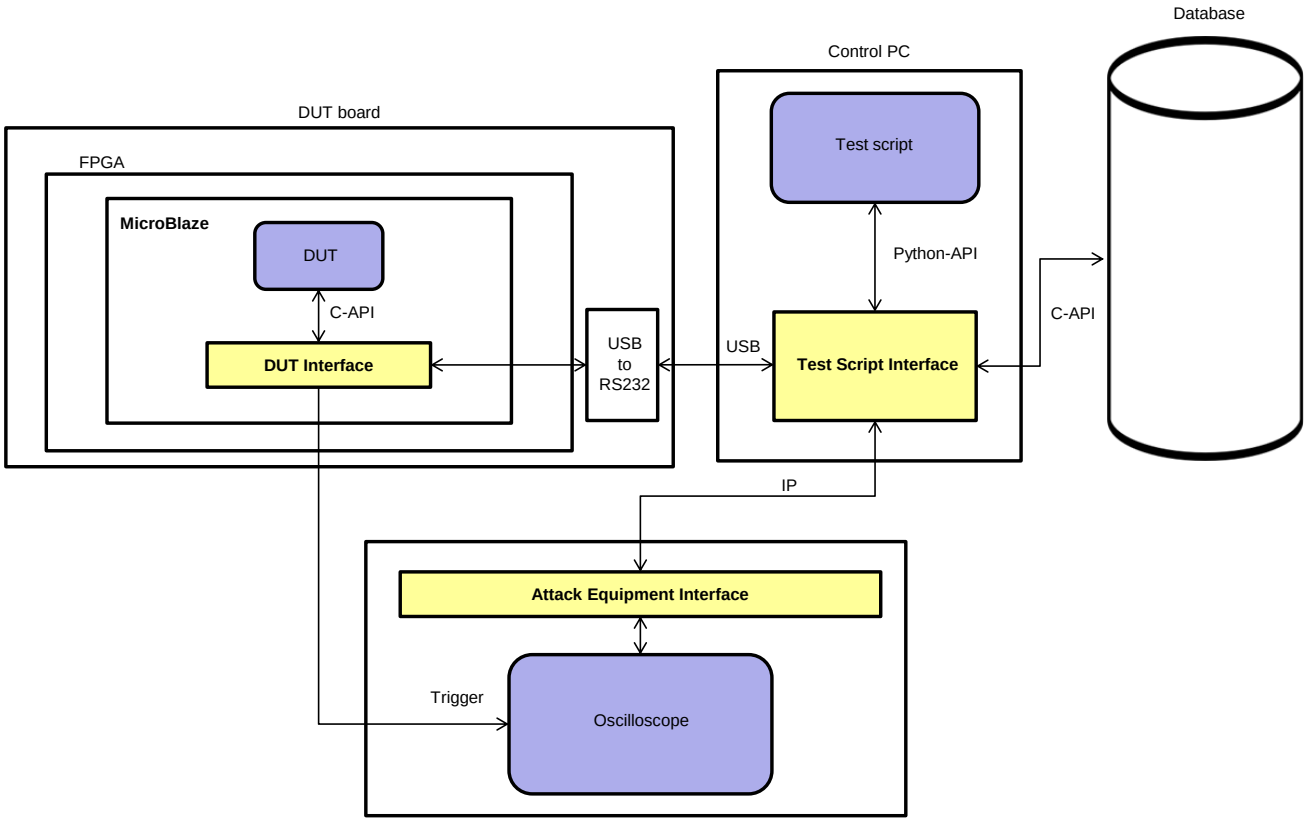

Fig. 2. Test Environment Setup for CPA Experiment: Block Diagram

embedded system to power analysis. In this example, we use a software implementation of AES, without countermeasures against implementation attacks, on a MicroBlaze soft-core microprocessor configured on a Spartan-3 FPGA. We use a Tektronix DPO3034 oscilloscope to collect power traces from the device and each power trace represents the power consumption during one AES encryption. Our analysis uses the correlation power analysis (CPA) technique to determine the secret key used for AES encryption.

A. Background

Correlation power analysis [3] relates variations in power consumption measurements during a secure operation to a leakage model defined based on the internal state of algorithm under attack. Common leakage models include Hamming weight of a state variable and Hamming distance between a state variable and its prior value. CPA can attack any operation that combines a fixed secret value with a known variable. The output of the target operation is assumed to be related to the power consumption of the device based on the selected leakage model. To perform CPA, the statistical correlation between the actual device power consumption and estimated power consumption given by the leakage model is compared for each possible value of the fixed secret. This allows determination of the actual value of the fixed secret because the correct value will show high correlation between the actual and estimated power consumption. To reduce the impact of noise on the results, actual power consumption is replaced with an average of actual power consumption over many traces before calculation of the correlation with estimated power consumption.

We focus our CPA attack on the add round key and S-box lookup operations in the first round of AES encryption. During these operations, the plaintext is the known input to the encryption and the internal state of the algorithm is given by output of the S-box lookup for byte i as

$$state_i = \text{SBOX}(key_i \text{ XOR } plaintext_i).$$

Therefore, it is possible to compute the resulting internal state byte for each possible value of key byte i and estimate the power consumption of the operation as HW($state_i$) using the Hamming weight model. The correct value of the key byte is the one that shows highest correlation with the actual power consumption.

B. DUT

The DUT in this experiment is a standard software implementation of AES encryption. The DUT receives a key and plaintext from the test environment and responds with the encrypted ciphertext. We define the DUT with a total

of three input/output ports, one each for the key, plaintext, and ciphertext, and set the size of each port as sixteen bytes. This allows the complete key, plaintext, or ciphertext to be transferred using a single message.

The DUT is implemented on a Xilinx Spartan 3 XC3S500E FPGA platform configured with a MicroBlaze microprocessor [18]. The MicroBlaze is programmed with a software application that runs the AES encryption in an infinite loop. The DUT is connected to the test script interface using RS-232 serial connection.

The DUT interface handles all RS-232 data transfer operations using a send/receive buffer to implement the message passing protocol for the DUT and test script and ensure that all communications are seen as complete messages by the DUT and test script. The DUT interface also implements assertion of the trigger signal to the attack equipment at the start of each AES encryption. The trigger signal is used directly by the oscilloscope to begin collecting a power trace from the DUT. To reduce the impact of noise on power consumption measurements, each encryption with a single key and plaintext is repeated 32 times and DUT power consumption is averaged over the 32 traces.

C. Test Script

Our complete test environment for this demonstration includes a control PC, DUT board, and oscilloscope. The control PC executes the test script and includes the test script interface for interaction with the other components. The DUT board is a Xilinx Spartan 3 starter kit FPGA board configured with the DUT, as well as the DUT interface. The oscilloscope is used as the attack equipment and measures power consumption of the DUT during AES encryption. A block diagram of the complete test setup is shown in Figure 2 and a photo of the physical setup is shown in Figure 3.

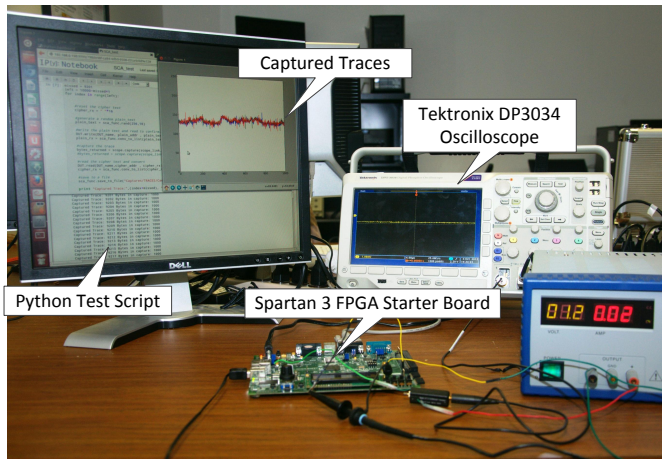

Fig. 3. Test Environment Setup for CPA Experiment: Photograph

In this demonstration, we use the test environment to collect traces from the DUT and store them in the database. After completing trace collection, we read traces from the database and attempt to extract the encryption key using CPA. This requires two separate test scripts; one for data collection and one for data analysis. The data collection test script provides a straightforward example of the interaction between all components of the test environment.

The data collection test script follows the general basic test procedure described in Section III A. The test script first initiates a connection with the DUT over USB and the oscilloscope over IP. Then, the script sets the parameters of the oscilloscope to capture traces in binary format averaging over 32 points and using one byte per sample and one thousand samples per trace. After this initialization process is complete, the test script sends a random encryption key to the DUT. This key remains constant for all encryptions used in this experiment. The test script then enters the data collection loop, where it performs a large number of iterations of the following operations:

- Send random plaintext message to the DUT
- Read captured power trace from the oscilloscope
- Append captured trace data to a file containing all previously collected traces
- Read ciphertext from the DUT
- Append plaintext and ciphertext to a file containing all previous messages

After completion of the data collection loop, the test script stores the collected trace data, along with general information about the experiment, in the SQL database. The traces are stored using the database API function that accepts multiple traces in a single buffer and separates them into a trace table entry corresponding to each measurement. This allows the test script to store all collected traces with a single call to the API by providing the file containing all collected traces and the file containing all plaintext and ciphertext pairs used. The test script stores general data about the experiement and test environment setup, such as oscilloscope setting read directly from the scope and a link to the DUT source code, in the database experiment table.

Python source code of the test script is shown in Listing 1. As shown, the test script uses high-level functions to perform the required operations and relies on the test script interface to implement low-level communication with the DUT and attack equipment interfaces.

```
#SCRIPT PARAMETERS
TOTAL_TRACES = 2000
SAMPLES_PER_TRACE = 1000
BYTES_PER_SAMPLE= 1

#INIT
#Initialize connection to oscilloscope
IP = "192.168.0.98"
scope_id = " " * 45
scope_link = scope.init(IP,scope_id)
#Set scope parameters
scope.set_parameters(scope_link, scope_id,
    num_avg=32, encdg=RPBINARY, recordlength=1000,
    byt_nr=1, hor_scale=400.0000E-6, vert_scale=
    .001)
#Initialize connection to DUT
DUT_name = "Digilent Spartan 3E"
```

```
DUT.init(DUT_name, addr_length, data_length)
#Send key to DUT
key = gen_rand_bytes(16)
DUT.write(DUT_name, key_addr , key)

#DATA COLLECTION LOOP
for i in range(TOTAL_TRACES):
  #Send plaintext to DUT
  plaintext = gen_rand_bytes(16)
  DUT.write(DUT_name, plain_addr , plaintext)
  #Read captured trace from oscilloscope
  bytes_returned = scope.capture(scope_link,
      scope_id, capture_buf)
  #Add trace to aggregated trace file
  append_to_file("Trace.txt", bytes_returned,
      capture_buf)
  #Read ciphertext from DUT
  DUT.read(DUT_name, cipher_addr, cipher_rx)
  #Add plaintext and ciphertext to aggregated
      message file
  append_to_file("Campaign.txt", (i), key,
      plaintext, cipher_rx)

#DATA STORAGE
#Create experiment table entry in database
scope.save_settings(scope_link, scope_id,
    "scopesettings.txt", scopeSettings)
expComment = open('expcomment.txt').read()
db.setCampaign(scopeSettings, DUT_URL, DUT_REV,
    expComment, SAMPLES_PER_PT)
#Create traces table entries in database
db.storeTraces(traceFile, messageFile,
    SAMPLES_PER_TRACE, TOTAL_TRACES)

#Close connection to DUT and oscilloscope
scope.close(IP,scope_link);
DUT.close(DUT_name);
```

Listing 1. Data Collection Test Script for CPA on AES

D. Results

We use a simple analysis script to retrieve the power traces collected from the database and perform the CPA attack. The attack targets each key byte separately and uses a Hamming weight power model to compute the correlation between the measured power traces and the Hamming weight of the first round S-box output for each possible value of the key byte. For each key byte, we choose the key value guess as the value showing the highest correlation with the power traces. The success of the attack is measured by the number of bytes for which the key value guess matches the actual encryption key used during data collection.

In this experiement, we were able to correctly guess all key bytes after analyzing 2000 power traces from the DUT, which indicates that the DUT is not secure against CPA side-channel attacks. This result demonstrates that the proposed test environment is well-suited for practical use in implementation attack resistance testing. An important feature of this demonstration is that the analysis script is completely independent of the data collection script and neither depends on the implementation details of the DUT or its platform.

V. Discussion

Implementation attack strategies vary widely and new attacks are constantly being proposed. An important feature of our proposed test environment is its flexibility to allow use with any kind of implementation attack, secure device, and cryptographic algorithm. In this section, we present examples of well-known implementation attacks and describe how the proposed test environment can be used to evaluate device vulnerability to each attack.

A. Power and Electromagnetic Attack

Side-channel attacks using power consumption or electromagnetic radiation are the most common implementation attacks against embedded systems. These attacks are based on the observation that small fluctuations in device power consumption or electromagnetic radiation during an encryption can reveal information about the internal state of the algorithm that can be used to derive the secret key. Vulnerability to power and electromagnetic attacks can be evaluated in terms of the number of measurement traces required to find the full key. In the previous section, we demonstrated the use of the proposed test environment to perform CPA. Test setup for other forms of power and electromagnetic attacks is identical to the setup used for CPA; the difference in attack strategy is manifested in the analysis script, rather than the test setup.

B. Timing Attack

Timing attacks are a form of side-channel attacks that exploit variations in execution time of secure operations. Timing variations occur due to data-dependent software branching, as well as system architecture. During the AES selection process, analysis by Daemen and Rijmen concluded that Rijndael, later selected as AES, was not vulnerable to timing attacks [7]. However, Bernstein has since demonstrated a simple, yet effective timing attack against a server using AES encryption [1]. Bernstein's attack exploits the timing dependence of table lookups, particularly the data cache used by general purpose CPUs. Table lookups resulting in a cache miss will take measurably longer than lookups resulting in a cache hit. Therefore, detailed analysis of encryption timing and its relationship with plaintext data values allows extraction of the complete encryption key. With improvements to this attack by Bonneau and Mironov, a server's key can be found with 2^{13} encryptions [2].

The AES cache timing attack requires precision timing measurements, a general purpose CPU with cached memory, and knowledge of plaintext and ciphertext. The proposed test environment can evaluate vulnerability to this attack as follows. The DUT used by both Bernstein [1] and Bonneau and Mironov [2] is a commercial CPU performing AES encryption using the OpenSSL library. Required attack equipment is a high-precision clock for measuring execution time of each encryption performed by the DUT. The test script has the DUT perform a number of encryptions and stores the plaintext, ciphertext, and execution time for each. This information can be used directly to perform analysis according to specific techniques described in the literature [1] [2].

C. Cold Boot Attack

Cold boot attacks are a form of side-channel attack against desktop computers introduced by Halderman et al [8]. This attack allows observation of sensitive data in RAM immediately after shut down and reboot of a system. Due to the physical properties of RAM hardware, data remnants are preserved for a short time after power removal, which allows an attacker to extract sensitive data after an incomplete shutdown. The attack is successful because encryption keys are stored in RAM during execution of sensitive operations. Results in [8] demonstrate that this attack can be used to defeat widely used full disk encryption schemes and suggest that any sensitive data in memory is vulnerable.

Test setup to perform a cold boot attack requires physical access to desktop computer with RAM and software to dump RAM contents to a file for analysis. In some cases, cooling equipment may also be necessary to prevent RAM contents from fading before they can be recorded by the attack software. This attack can be incorporated into the proposed test environment by treating the RAM as the DUT with the desktop computer as the DUT platform. This attack requires multiple pieces of attack equipment including attack software that records RAM contents, power supply interrupter for the DUT, and cooling equipment. All communication with each piece of attack equipment is incorporated into the attack equipment interface in the test environment. In this setup, a trace is a file dump of all RAM contents at the time of power removal. To implement the cold boot attack, the test script would use the attack equipment to remove and quickly restore power supply from the DUT, then the test script would direct the attack equipment to execute the attack software and record the contents of the RAM.

D. Optical Fault Attack

In general, fault attacks attempt to reveal secret values by manipulating the behavior of the target device. Techniques used to generate faults vary greatly and are typically classified based on invasiveness. Optical fault induction is a semi-invasive fault attack proposed by Skorobogatov and Anderson [15]. This attack allows manipulation of any individual SRAM bit using a laser. The practical significance of the attack is that it can allow the attacker to change the control flow of cryptographic operations. This technique can be used to make significant changes to computations performed during the cryptographic operation and allow recovery of the secret key. Although it requires depackaging the chip, the semi-invasive optical fault induction attack, unlike invasive attacks, which are typically very expensive to implement, does not require mechanical manipulation of the chip silicon and can be performed with low-cost off-the-shelf components.

Test setup for an optical fault induction attack requires a depackaged chip, laser, and knowledge of chip input and output. The proposed test environment can accommodate this attack by defining the depackaged chip as the DUT and the laser as the attack equipment. The attack equipment interface allows control of the exact chip location exposed to the laser and the DUT interface allows transmission of input and output data to and from the DUT. Without direct knowledge of the device design, optical fault induction attacks require some exploration to determine the impact of different SRAM cells on the cryptographic module under attack. The proposed test environment can also be used by incorporating a test script that induces faults in different SRAM cells and analyzes the impact on the final output of the device. Using this method, a complete attack can be mounted once the attacker determines the SRAM manipulations required to produce output that reveals the secret key.

VI. Conclusion

Implementation attacks are a growing threat to secure systems that cannot be ignored. Vulnerability arises from interaction between the system design and its implementation platform. This makes is impossible to test implementation attack vulnerability using traditional testing approaches, such as unit testing and source code analysis. We have proposed an open-source implementation attack test environment that can be used to determine vulnerability to these attacks. The proposed test environment separates the DUT from the implementation attack procedure to allow each to be designed independently. In addition, our design is focused on preserving portability and openness in the test environment, so we ensure that the environment is applicable to any type of implementation attack, regardless of the DUT platform, implementation details, or cryptographic algorithm used. We contend that the proposed test environment is suitable for use in conjunction with implementation attack security standards to allow security validation that provides uniform, repeatable, and comparable results.

References

[1] D. J. Bernstein, "Cache-timing attacks on AES," Tech. Rep., Apr. 14 2005. [Online]. Available: http://cr.yp.to/antiforgery/cachetiming-20050414.pdf

[2] J. Bonneau and I. Mironov, "Cache-collision timing attacks against AES," in *Cryptographic Hardware and Embedded Systems - CHES 2006*, ser. Lecture Notes in Computer Science, L. Goubin and M. Matsui, Eds. Springer Berlin / Heidelberg, 2006, vol. 4249, pp. 201–215, 10.1007/11894063_16. [Online]. Available: http://dx.doi.org/10.1007/11894063_16

[3] E. Brier, C. Clavier, and F. Olivier, "Correlation power analysis with a leakage model," in *Cryptographic Hardware and Embedded Systems - CHES 2004*, ser. Lecture Notes in Computer Science, M. Joye and J.-J. Quisquater, Eds. Springer Berlin / Heidelberg, 2004, vol. 3156, pp. 135–152, 10.1007/978-3-540-28632-5_2. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-28632-5_2

[4] Brightsight, "Unique tools from the security lab." [Online]. Available: http://www.brightsight.com/documents/marcom-materials/Brightsight_Tools.pdf

[5] C. Clavier, "Side Channel Analysis for Reverse Engineering (SCARE) - an improved attack against a secret A3/A8 GSM algorithm," Cryptology ePrint Archive, Report 2004/049, 2004, http://eprint.iacr.org/.

[6] Cryptography Research, "DPA workstation." [Online]. Available: http://www.cryptography.com/technology/dpa-workstation.html

[7] J. Daemen and V. Rijmen, "Resistance against implementation attacks: a comparative study of the AES proposals," in *Second AES Candidate Conference (AES2)*. National Institute of Standards and Technology (NIST), Mar. 22-23 1999. [Online]. Available: http://csrc.nist.gov/archive/aes/round1/conf2/papers/daemen.pdf

[8] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten, "Lest we remember: Cold-boot attacks on encryption keys." *Communications of the ACM*, vol. 52, no. 5, pp. 91 – 98, 2009. [Online]. Available: http://dx.doi.org/10.1145/1506409.1506429

[9] J. Kim, K. Oh, D. Choi, and H. Kim, "SCARF: profile-based side channel analysis resistant framework," in *Security and Management (SAM'12), 2012 International Conference on*, Jul. 16-19 2012.

[10] National Institute of Standards and Technology (NIST), "FIPS 140-3 (second draft) sections submitted for comments," Aug. 30 2012. [Online]. Available: http://csrc.nist.gov/groups/ST/FIPS140_3/documents/FIPS_140-3_sections_submitted_for_comments.pdf

[11] *Cryptographic Hash Algorithm Competition*, National Institute of Standards and Technology (NIST), 2007-2012. [Online]. Available: http://csrc.nist.gov/groups/ST/hash/sha-3/index.html

[12] *Security Requirements for Cryptographic Modules (Revised Draft)*, National Institute of Standards and Technology (NIST), 2009. [Online]. Available: http://csrc.nist.gov/publications/PubsDrafts.html#FIPS-140--3

[13] J. Nechvatal, E. B. L. Bassham, M. Dworkin, J. Foti, and E. Roback, *Report on the Development of the Advanced Encryption Standard (AES)*, National Institute of Standards and Technology (NIST), Oct. 2 2000.

[14] Riscure, "Inspector." [Online]. Available: http://www.riscure.com/tools/inspector

[15] S. Skorobogatov and R. Anderson, "Optical fault induction attacks," in *Cryptographic Hardware and Embedded Systems - CHES 2002*, ser. Lecture Notes in Computer Science, B. Kaliski, e. Ko, and C. Paar, Eds. Springer Berlin / Heidelberg, 2003, vol. 2523, pp. 31–48, 10.1007/3-540-36400-5_2. [Online]. Available: http://dx.doi.org/10.1007/3-540-36400-5_2

[16] *DPA contest*, Télécom ParisTech, 2008-2012. [Online]. Available: http://www.dpacontest.org

[17] R. Velegalati and J.-P. Kaps, "Introducing FOBOS: Flexible Open-source BOard for Side-channel analysis," in *Constructive Side-Channel Analysis and Secure Design (COSADE), Third International Workshop on: Work in Progress Session*, May 3 2012.

[18] Xilinx, "MicroBlaze soft processor core." [Online]. Available: http://www.xilinx.com/tools/microblaze.htm